Formal Verification for Real-World Cryptographic Protocols and Implementations

Nadim Kobeissi

INSTITUT NATIONAL DE RECHERCHE EN
INFORMATIQUE ET AUTOMATIQUE
ACCREDITED BY ÉCOLE NORMALE SUPÉRIEURE

DOCTORAL THESIS

Formal Verification for Real-World Cryptographic Protocols and Implementations

---

Author:
Nadim Kobeissi

Supervisors:
Karthikeyan Bhargavan
Bruno Blanchet

Referees:
Stéphanie Delaune – IRISA, RENNES
Tamara Rezk – INRIA, SOPHIA ANTIPOLIS

Examiners:
Cas Cremers – CISPA-HELMHOLTZ CENTER, SAARBRUECKEN
Stéphanie Delaune – IRISA, RENNES
Antoine Delignat-Lavaud – MICROSOFT RESEARCH, CAMBRIDGE
Ralf Küsters – UNIVERSITY OF STUTTGART, STUTTGART
David Pointcheval – ÉCOLE NORMALE SUPÉRIEURE, PARIS

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Computer Science from the

ED386 ÉCOLE DOCTORALE

and defended on

DECEMBER 10TH, 2018


informatiques mathématiques
Inria

There is always some madness in love.
But there is also always some reason
in madness. And those who were seen
dancing were thought to be insane by
those who could not hear the music.

Friedrich Nietzsche

Thanks to God that he gave me
stubbornness when I know I am right.

John Adams

إلى أبي، حسن.

Abstract

Individuals and organizations are increasingly relying on the Web and on user-facing applications for use cases such as online banking, secure messaging, document sharing and electronic voting. To protect the confidentiality and integrity of these communications, these systems depend on authentication and authorization protocols, on application-level cryptographic constructions and on transport-layer cryptographic protocols. However, combining these varied mechanisms to achieve high-level security goals is error-prone and has led to many attacks even on sophisticated and well-studied applications.

This thesis aims to develop methods and techniques for reasoning about, designing and implementing cryptographic protocols and secure application components relevant to some of the most frequently used cryptographic protocols and applications. We investigate and formalize various notions of expected guarantees and evaluate whether some of the most popular secure channel protocols, such as secure messaging protocols and transport protocols often operating at the billion-user scale, are capable of meeting these goals.

In this thesis, we ask: can existing major paradigms for formal protocol verification serve as guidelines for the assessment of existing protocols, the prototyping of protocols being designed, and the conception of entirely new protocols, in a way that is *meaningful* and reflective of their expected real-world properties? And can we develop novel frameworks for formal verification and more secure implementation based on these foundations?

We propose new formal models in both symbolic and computational formal verification frameworks for these protocols and applications. In some of the presented work, we obtain a verification methodology that starts from the protocol and ends at the implementation code. In other parts of our work, we develop a dynamic framework for generating formal verification models for any secure channel protocol described in a lightweight syntax. Ultimately, this thesis presents formal verification approaches for existing protocols reaching towards their implementation as well as methods for prototyping and formally verifying new protocols as they are being drafted and designed.

Acknowledgments

Being able to work on this research at INRIA has been the most essential opportunity that I have been given in my life. Furthermore, no more essential opportunity will ever arise in my life, since my future will undeniably be rooted in what I've been allowed to work on at INRIA, in the researchers of INRIA taking me in and teaching me the posture and adroitness necessary for good research.

Being at INRIA has taught me much more than the proper formal analysis of cryptographic protocols. It taught me just how difficult and uncompromising true scientific rigor can be and the importance of a diligent and forward-looking work ethic. It also taught me patience, helped me expand my perspective with regards to the viewpoints of others and my sense of good faith during teamwork. It helped me mature into someone less sure of his intuition while nevertheless radically strengthening that intuition in the same process. The researchers at INRIA are part of an institution that morally and intellectually is larger than life, where the bottom line, the net product ends up being uncompromising research pushed forward with equal amounts of passion and prudence. I hope that I can carry with me what I've been given at INRIA by imparting the insights of what I've been able to work on and the values that determine how the work is to be done.

Most of the contributions herein that can be considered my own are largely due to my acting as a sort of highly opinionated melting pot for Karthik's decades-long vision for Web security, Antoine's first explaining to me that, in F*, "types are proofs", Bruno's unnerving, laser-like focus and rigor — I could go on to cover every member of the PROSECCO research group in words that seem inflated but aren't really. Being at INRIA taught me how to think. Working with these people taught me more than I could have ever learned on my own and made me into more than I could have hoped to become.

I am also deeply grateful for Cedric and Antoine's hosting me at Microsoft Research for the duration of my internship and most of all for actually letting me call my internship project "QuackyDucky", even keeping that name after integrating it into the Project Everest stack well after my departure. I would have never received this opportunity were it not for the trust and good faith that Graham, Harry and Philippe put in me during my first year. Their confidence was pivotal in my being given a chance to learn how to do actual science.

Regarding the thesis committee, who likely already groaned at the prospect of yet another big thesis to read and are even more apprehensive about it after reading this warm, fuzzy essay, I can promise you cookies at the defense.

As for Karthik, what I owe him is beyond evaluation.

Contents

Introduction

End-to-end encryption is increasingly becoming a staple in the security considerations of user-facing services, applications and communications infrastructure. Even during the period in which the work within this thesis was being prepared, things have changed: almost all of the world's most popular instant messengers adopted end-to-end encryption protocols targeting advanced security goals such as forward secrecy. Endpoint authentication for a majority of the secure web adopted a radically different (and for the first time, automated) protocol with the launch of Let's Encrypt. HTTPS, the framework underlying much of the secure Web, went from being based on a relatively hacked-together version of the Transport Layer Security (TLS) standard to upgrading to a protocol that was, for the first time, designed hand in hand with cryptographers and academics since its inception. And, as we have seen, wide-eyed venture capitalists put millions of dollars into programs running on an unverified state machine on top of an experimental blockchain.

A microcosm within this massive shift into ubiquitous, *real-world* cryptographic systems has been the involvement of formal methods and analysis of the underlying cryptography. When work on this thesis began, Signal, the current de-facto standard for secure messaging, which today encrypts daily communications between more than a billion people worldwide, did not even benefit from a public technical (let alone formal) specification. TLS 1.2, the then-latest production standard for HTTPS connections, continued in the vein of the "code first, verify later" methodology that had underpinned all previous versions of the standard.

Much of this thesis chronicles the involvement of formal verification methods as they apply to cryptographic protocols and implementations: as of 2015, it had become abundantly clear that practical, applied security systems suffered from problems of rigor in both protocol design and implementation that were resulting in all kinds of bugs, from state machine implementation errors and broken cryptographic primitives, all the way up to fundamentally flawed protocol blueprints. Perhaps the most self-contained motivating example for the necessity of practically relevant formal verification came in an increasingly large repertoire[1] of problems with TLS ranging from low-level implementation errors to fundamental protocol design problems, all of which were either detected using, or would have been avoided with, formal verification approaches.

New formal approaches present in TLS 1.3 came hot on the heels of multiple analyses of TLS 1.2 and earlier, showing implementation issues as well as protocol-level design flaws. And as we show in this thesis, TLS 1.3 still itself had issues that became apparent with formal verification, even 18 drafts into the standardization process. The silver lining is that we were able to monitor TLS 1.3 for security regressions draft after draft because our formal verification frameworks had become practical enough to be

[1] `https://mitls.org`

used for *formally describing*, *verifying and thereby prototyping the standard as it evolved*, which is likely a milestone in bridging the gap between formal verification and real-world cryptographic systems.

Bridging this gap is ultimately what this thesis is all about. Can existing major paradigms for formal protocol verification serve as guidelines for the assessment of existing protocols, the prototyping of protocols being designed, and the conception of entirely new protocols, in a way that is *meaningful* and reflective of their expected real-world properties? And can we develop novel frameworks for formal verification and more secure implementation based on these foundations?

As much as I'd like to stay dry and deafen you with the silence of my academicism, I'd rather describe this thesis for what it really is: a *The Hobbit*-esque journey of a Ph.D. student who started off woefully unprepared, most of all for his own thirst for adventure. This thesis is a diplomatic treaty between protocol engineers and mathematicians focused on formal methods. This thesis started off as a shot in the dark, but by the time the dragon was slain, the treasure taken and home was found again, the work within it became something that could contend as a serious treatise on how formal verification can *matter*.

So, join me as I recount my adventure! Together, we will wander through the caves, mountains and valleys of my attempts to apply formal verification to every worthwhile cryptographic protocol with real-world implications over the past three years. Starting with the Signal secure messaging protocol, continuing on to TLS, to the Noise Protocol Framework, to Let's Encrypt and then finally to the unexplored new marshes of a new secure collaborative document editing protocol. What strange new JavaScript-based formal verification techniques will we discover along the way? What surprising results await? And, most importantly, which mathematical notation mistake will escape the eyes of all reviewers and make it into the final thesis draft? Who knows? You can!

Cryptographic Protocols and Systems that We Consider

In developing this work, we have tried to focus on real-world use cases as the context that underlies research efforts. We have decided to focus on user-facing protocols and systems with the highest number of users and which we believe are the most fundamental to secure communication on the modern Web.

Secure Messaging

Secure messaging today occupies a central position in real-world cryptography, primarily due to the deployment of secure messaging systems in Apple iMessage, Google Allo, Microsoft Skype, Telegram, WhatsApp [1], Wire and many other platforms, ultimately meaning that the majority of instant messaging communication is indeed end-to-end encrypted [2]. With the exception of Apple iMessage and Telegram, all of the examples mentioned above use the same foundation for their secure messaging cryptography: the Signal Protocol [3, 4].[2]

[2] Academic study of secure messaging protocols has largely focused on Signal Protocol due to its strong security goals, which include forward secrecy and post-compromise security, its open source implementation and its protocol design with includes a zero-round-trip session establishment. In Chapter 1, we briefly examine (and find attacks) in Telegram's "MTProto" protocol. Other secure messaging protocols have not generally received academic attention. Apple's iMessage protocol was subjected to some scrutiny by Garman et al. [5] but similarly receives little review due to its closed source implementation and lack of any technical specification.

Signal Protocol offered an open source foundation for modern secure messaging with features that, at least in 2015, indicated innovative protocol design with ambitious security guarantees that also addresses new secure messaging corner cases inherent to smartphones, such as initializing a secure session with a party who may be offline at the time the first message is sent (thereby mimicking the SMS use case scenario.)

However, secure messaging had been in development for more than a decade prior to Signal Protocol. Perhaps the earliest serious secure messaging effort that produced a robust, well-studied protocol with a usable implementation was the Off-the-Record protocol, or OTR [6], introduced in 2004. OTR was designed before the advent of smartphones and therefore was not interested in asynchronous session establishment: establishing an OTR session required one and a half round trips. Key rotation was slow and heavy, requiring messages to be exchanged by both parties. Due to relatively large public keys, message size was also not restricted in any meaningful way.

While OTR came with a technical specification and a research paper justifying its design decisions, a 2006 security analysis by Bonneau and Morison [7] still found serious vulnerabilities in OTR version 2, including a simple attack that could break message authenticity simply by delaying when messages are received.

Future versions of OTR fixed these issues and improved on the protocol more generally. In fact, the first version of the Signal messenger[3] used OTR as its foundation for secure messaging, taking care to use more modern primitives in order to facilitate the exchange of public key information over SMS, but otherwise leaving the protocol more or less intact.

However, OTR's porting over to the SMS use case did not go over smoothly. In regular SMS usage, Alice can send a text to Bob while Bob's phone is off, only for him to receive this message a week later when he finally turns on his phone. In initial versions of Signal, this was impossible: a round-trip handshake was required in order to establish a secure session, thereby necessitating that both sides be online and able to send and receive SMS messages for a little while before actually being able to maintain a confidential messaging session.

Ultimately, the less-than-ideal nature of SMS as a transport layer forced Signal to adopt multiple changes and compromises to its original implementation of OTR: SMS also has a 140-character limit per message and users may incur variable charges and experience variable rates of reliability and message delivery delays worldwide. Eventually, this led to the development of what would become Signal Protocol, a protocol which focused on the ability to establish secure sessions, and to rotate encryption keys, with the minimum amount of message exchanges necessary. The X3DH authenticated key exchange [3] and Double Ratchet [4][4] that resulted from these use case demands targeted ambitious security guarantees but often were deployed in production prematurely and with little study: Rösler et al [8] showed that Signal Protocol's group chat deployment was underspecified and exposed to different vulnerabilities across multiple platforms. Signal's desktop application, which was built on top of the Web-based Electron framework, was also the subject of multiple security vulnerabilities [9, 10] largely facilitated by the difficulty inherent to writing secure code for Web applications.

While some of the research cited above occured after we had completed and published our own research on secure messaging, it was that zeitgeist that informed our research approach and our priorities. In this thesis, therefore, we have focused signifi-

[3]Initial versions of Signal were called "TextSecure". However, we will refer to Signal as Signal for the remainder of this work in the interest of simplicity.

[4]Signal Protocol's Double Ratchet was previously known as the "Axolotl" ratchet.

cantly on the formal analysis of secure messaging protocols in different formal models, as well as securely implementing these protocols specifically in the context of high-assurance Web applications.

Transport Layer Security

The Transport Layer Security (TLS) protocol is widely used to establish secure channels on the Internet. It was first proposed under the name SSL [11] in 1994 and has undergone a series of revisions since, leading up to the standardization of TLS 1.2 [12] in 2008. Each version adds new features, deprecates obsolete constructions and introduces countermeasures for weaknesses found in previous versions. The behavior of the protocol can be further customized via *extensions*, some of which are mandatory to prevent known attacks on the protocol.

One may expect that TLS clients and servers would use only the latest version of the protocol with all security-critical extensions enabled. In practice, however, many legacy variants of the protocol continue to be supported for backwards compatibility and the everyday use of TLS depends crucially on clients and servers negotiating the most secure variant that they have in common. Securely composing and implementing the many different versions and features of TLS has proved to be surprisingly hard, leading to the continued discovery of high-profile vulnerabilities in the protocol.

A History of TLS Vulnerabilities

TLS has traditionally suffered from mainly four different types of weaknesses and vulnerabilities:

- **Downgrade attacks** enable a network adversary to fool a TLS client and server into using a weaker variant of the protocol than they would normally use with each other. In particular, version downgrade attacks were first demonstrated from SSL 3 to SSL 2 [13] and continue to be exploited in recent attacks like POODLE [14] and DROWN [15].

- **Cryptographic vulnerabilities** rely on weaknesses in the protocol constructions used by TLS. Recent attacks have exploited key biases in RC4 [16, 17], padding oracles in MAC-then-Encrypt [18, 14], padding oracles in RSA PKCS#1 v1.5 [15], weak Diffie-Hellman groups [19] and weak hash functions [20].

- **Protocol composition** flaws appear when multiple modes of the protocol interact in unexpected ways if enabled in parallel. For example, the renegotiation attack [21] exploits the sequential composition of two TLS handshakes, the Triple Handshake attack [22] composes three handshakes and cross-protocol attacks [23, 13] use one kind of TLS handshake to attack another.

- **Implementation bugs** contribute to the fourth category of attacks on TLS, and are perhaps the hardest to avoid. They range from memory safety bugs like HeartBleed and coding errors like GotoFail to complex state machine flaws. Such bugs can be exploited to bypass all the security guarantees of TLS and their prevalence, even in widely-vetted code, indicates the challenges of implementing TLS securely.

Security Proofs

Historically, when an attack is found on TLS, practitioners propose a temporary fix that is implemented in all mainstream TLS libraries, then a longer-term countermeasure is incorporated into a protocol extension or in the next version of the protocol. This has led to a attack-patch-attack cycle that does not provide much assurance in any single version of the protocol, let alone its implementations.

An attractive alternative would have been to develop security proofs that systematically demonstrated the absence of large classes of attacks in TLS. However, developing proofs for an existing standard that was not designed with security models in mind is exceedingly hard [24]. After years of effort, the cryptographic community only recently published proofs for the two main components of TLS: the *record* layer that implements authenticated encryption [25, 26] and the *handshake* layer that composes negotiation and key-exchange [27, 28]. These proofs required new security definitions and custom cryptographic assumptions and even so, they apply only to abstract models of certain modes of the protocol. For example, the proofs do not account for low-level details of message formats, downgrade attacks, or composition flaws. Since such cryptographic proofs are typically carried out by hand, extending the proofs to cover all these details would require a prohibitive amount of work and the resulting large proofs themselves would need to be carefully checked.

A different approach taken by the protocol verification community is to *symbolically* analyze cryptographic protocols using simpler, stronger assumptions on the underlying cryptography, commonly referred to as the Dolev-Yao model [29]. Such methods are easy to automate and can tackle large protocols like TLS in all their gory detail, and even aspects of TLS implementations [30, 31]. Symbolic protocol analyzers are better at finding attacks, but since they treat cryptographic constructions as perfect black boxes, they provide weaker security guarantees than classic cryptographic proofs that account for probabilistic and computational attacks.

The most advanced example of mechanized verification for TLS is the ongoing miTLS project [32], which uses dependent types to prove both the symbolic and cryptographic security of a TLS implementation that supports TLS 1.0-1.2, multiple key exchanges and encryption modes, session resumption, and renegotiation.

Noise Protocol Framework

IK :
$\leftarrow s$
$\cdots$
$\rightarrow e, es, s, ss$
$\leftarrow e, ee, se$
$\rightarrow$
$\leftarrow$

Figure 1: An example Noise Handshake Pattern, `IK`.

Popular Internet protocols such as SSH and TLS use similar cryptographic primitives: symmetric primitives, public key primitives, one-way hash functions and so forth. Protocol stages are also similarly organized, usually beginning with a authenticated key exchange (AKE) stage followed by a messaging stage. And yet, the design methodology, underlying state machine transitions and key derivation logic tend to be entirely different between protocols with nevertheless similar building blocks. The targeted effective security goals tend to be similar, so why can't the same methodology be followed for everything else?

Standard protocols such as those mentioned above choose a specific set of key exchange protocols to satisfy some stated use-cases while leaving other ele-

ments, such as round trips and (notoriously) cipher suites up to the deployer. Specifications use protocol-specific verbose notation to describe the underlying protocol, to the extent that even extracting the core cryptographic protocol becomes hard, let alone analyzing and comparing different modes for security.

Using completely different methodologies to build protocols that nevertheless often share the same primitives and security goals is not only unnecessary, but provably dangerous. The Triple Handshake attack on TLS published in 2014 [22] is based on the same logic that made the attack [33] on the Needham-Schroeder protocol [34] possible almost two decades earlier. The core protocol in TLS 1.2 was also vulnerable to a similar attack, but since the protocol itself is hidden within layers of packet formats and C-like pseudocode, it was difficult for the attack to be detected. However, upon automated symbolic verification [35], the attack quickly appeared not just in TLS, but also in variants of SSH and IPsec. Flaws underlying more recent attacks such as Logjam [19] were known for years before they were observed when the vulnerable protocol was analyzed. Had these protocols differed only in terms of network messages while still using a uniform, formalized logic for internal key derivation and state machine transitioning designed based on the state of the art of protocol analysis, these attacks could have been avoided.

The Noise Protocol Framework [36], recently introduced by Trevor Perrin, aims to avert this problem by presenting a simple language for describing cryptographic network protocols. In turn, a large number of semantic rules extend this simple protocol description to provide state machine transitions, key derivation logic and so on. The goal is to obtain the strongest possible effective security guarantees for a given protocol based on its description as a series of network messages by deriving its other elements from a uniform, formally specified logic followed by all protocol designs.

In designing a new secure channel protocol using the Noise Protocol Framework, one only provides an input using the simple language shown in Fig. 3.1. As such, from the viewpoint of the protocol designer, Noise protocols can only differ in the number of messages, the types of keys exchanged and the sequence or occurrence of public key transmissions and Diffie-Hellman operations. Despite the markedly non-expressive syntax, however, the occurence and position of the "tokens" in each message pattern can trigger complex state machine evolutions for both parties, which include operations such as key derivation and transcript hash mixing.

Let's examine Fig. 1. Before the AKE begins, the responder shares his static public key. Then in the first protocol message, the initiator sends a fresh ephemeral key, calculates a Diffie-Hellman shared secret between her ephemeral key and the recipient's public static key (*es*), sends her public static key and finally calculates a Diffie-Hellman shared secret between her static key and the responder's public static key (*ss*). The responder then answers by generating an ephemeral key pair and sending his ephemeral public key, deriving a Diffie-Hellman shared secret between his ephemeral key and the ephemeral key of the initiator (*ee*) and another Diffie-Hellman shared secret between his static key and the ephemeral key of the initiator (*es*). Both of these AKE messages can also contain message payloads, which, depending on the availability of sufficient key material, could be AEAD-encrypted (in this particular Noise Handshake Pattern, this is indeed the case.)

As we can see, quite a few operations have occurred in what would at first glance appear to be simple syntax for a simple protocol. Indeed, underlying these operations is a sophisticated state machine logic tasked with mixing all of the derived keys together, determining when it is safe (or possible) to send encrypted payloads and ensuring transcript consistency, among other things. This is the value of the Noise Protocol Framework: allowing the protocol designer to describe what they need their protocol

to do fairly effectively using this simple syntax and leaving the rest to a sturdy set of underlying rules.

Automated Domain Name Validation

Since the dawn of HTTPS, being able to secure a public website with SSL or TLS requires obtaining a signature for the website's public certificate from a certificate authority [37] (CA). All major operating system and browser vendors maintain lists of trusted CAs (represented by their root certificates) that can legitimately attest for a reasonable link between a certificate and the identity of the server or domain it claims to represent.

For example, all major operating systems and browsers include and trust Symantec's root certificates, which allows Alice to ask Symantec to attest that the certificate she uses on her website `AliceShop.com` has indeed been issued to her, rather than to an attacker trying to impersonate her website. After Alice pays Symantec some verification fee, Symantec performs some check to verify that Alice and her web server indeed have the authority over `AliceShop.com`. If successful, Symantec then signs a certificate intended for that domain. Since the aforementioned operating systems already trust Symantec, this trust now extends towards Alice's certificate being representative of `AliceShop.com`.

The security of this trust model has always relied on the responsibility and trustworthiness of the CAs themselves, since a single malicious CA can issue arbitrary valid certificates for any website on the Internet. Each certificate authority is free to engineer different user sign-up, domain validation, certificate issuance and certificate renewal protocols of its own design. Since these ad-hoc protocols often operate over weak channels such as HTTP and DNS without strong cryptographic authentication of entities, most of them can be considered secure only under relatively weak threat models, reducing user credentials to a web login and domain validation to an email exchange.

The main guidelines controlling what type of domain validation CAs are allowed to apply are the recommendations in the CA/Browser Forum Baseline Requirements [38]. These requirements, which are adopted by ballot vote between the participating organizations, cover the definition of common notions such as domain common names (CNs), registration authorities (RAs) and differences between regular domain validation (DV) and extended validation (EV).

These guidelines have not proven sufficient for a well-regulated and well specified approach for domain validation: Mozilla was recently forced to remove WoSign [39] (and its subsidiary StartSSL, both major certificate authorities) from the certificate store of Firefox and all other Mozilla products due to a series of documented instances that range from the CA intentionally ignoring security best-practices for certificate issuance, to vulnerabilities allowing attackers to obtain signed certificates for arbitrary unauthorized websites.

The lack of a standardized protocol operating under a well-defined threat model and with clear security goals for certificate issuance has so far prevented a systematic treatment of certificate issuance using well-established formal methods. Instead, academic efforts to improve PKI security focus on measurement studies [40, 41] and transparency and public auditability mechanisms [42, 43] for already-issued certificates.

In 2015, a consortium of high-profile organizations including Mozilla and the Electronic Frontier Foundation launched "Let's Encrypt" [44], a non-profit effort to specify, standardize and automate certificate issuance between web servers and certificate authorities and to provide certificate issuance itself as a free-of-charge service. Since its

launch in April 2016, Let's Encrypt has issued more than 27 million certificates [45] and has been linked to a general increase in HTTPS adoption across the Internet.

Let's Encrypt also introduces ACME [46], an automated domain validation and certificate issuance protocol that gives us for the first time a protocol that can act as a credible target for formal verification in the context of domain validation. ACME also removes almost entirely the human element from the process of domain validation: the subsequently automated validation and issuance of millions of certificates further increases the necessity of a formal approach to the protocol involved.

Secure Collaborative Document Editing

Collaborative document editing software such as Google Docs and Microsoft Office365 has become indispensable in today's work environment. In spite of no confidentiality guarantees, large working groups still find themselves depending on these services even for drafting sensitive documents. Peer pressure, a rush to agree on a working solution and the sheer lack of alternatives have created an ecosystem where a majority of sensitive drafting is currently conducted with no end-to-end encryption guarantees whatsoever. Google and Microsoft servers have full access to all documents being collaborated upon and so do any parties with read access to the internal databases used by these services.

And yet, despite what we see as a clear need for more privacy-preserving collaborative editing framework, this problem space seems largely underaddressed. In the concluding chapter of this thesis, we also contribute a formal approach to the secure collaborative document editing for what we believe is the first time.

Our Approach to Formal Verification

Much of this thesis focuses on the formal verification of what are believed to be some of the most important cryptographic protocols governing the modern Web: Signal Protocol[5] (Chapter 1), TLS 1.3 (Chapter 2), the Noise Protocol Framework (Chapter 3) and ACME (Chapter 4).

The same formal verification paradigms and automated verification tools are adopted across all four chapters: the symbolic formal verification model, for which we use the ProVerif [47, 48] automated verification tool and the computational formal verification model, for which we use the CryptoVerif [49] tool. In this introduction, we explain our motivation behind symbolic and computational verification in ProVerif and CryptoVerif and then provide an informal introduction to the tools and paradigms themselves, equipping the reader with the knowledge necessary to appreciate the methodology and results of the upcoming four chapters.

Why Formal Verification?

Every cryptographic system or protocol we consider in this thesis[6] has followed a "code first, specify later" methodology, where the implementation is fully engineered to suit a particular use case, and sometimes even deployed, before any technical specification is written (usually in the shape of an IETF RFC document or similar.) As described in the

[5] Other secure messaging protocols, such as OTR and Telegram, are also briefly examined.

[6] With the exception of only the latest version of TLS, TLS 1.3 (Chapter 2) and of our own novel secure collaborative document editing protocol (Chapter 6)

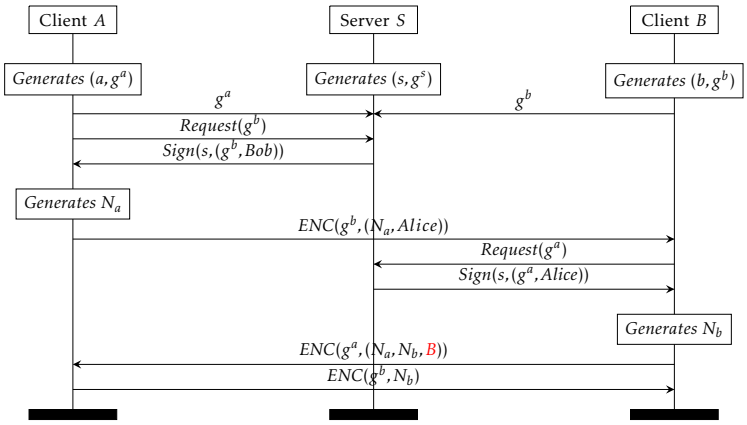

Figure 2: The Needham Schroeder Public-Key Protocol. This simple protocol, first described in 1978, was not broken until 1995 when a man-in-the-middle attack was discovered using automated formal verification in the symbolic model. By adding the component highlighted in red, the vulnerability can be addressed.

Introduction, this trend has coincided with a variety of protocol and implementation errors being detected, sometimes years later.

Many of the protocol issues that were detected in these protocols, including those that constitute some of this thesis's findings, were found using automated verification tools. Automated verification tools are computer software developed around a particular methodology for reasoning about cryptographic primitives, cryptographic protocols and network systems, with the purpose of obtaining answers to questions regarding the execution of these protocols in specific contexts.

When employing automated verification tools, the user is forced, through the first step of modeling the protocol, to specify the protocol to some substantial extent in some well-defined syntax or calculus. Then, based on the constructions, functions and processes within this model as well as the paradigms and network models inherent to the verification framework, answers can be obtained for questions such as *"does this protocol run satisfy forward secrecy for two principals, Alice and Bob?"* or *"is this protocol provable to always satisfy confidentiality on the first message against an active attacker and authenticity of both parties with respect to one another after any successful handshake?"* Generally, the tools respond with a yes-or-no answer built upon a rigorous foundation: either a formal verification result if we are verifying in the *symbolic model*, or a game-based proof, similar to those one may find in academic papers with hand proofs, if we are verifying in the *computational model*. We introduce both verification models in this chapter.

In 1978, Roger Needham and Michael Schroeder published the Needham-Schroeder public-key protocol [34], intended for use in network protocols requiring mutual authentication. While the protocol is considered quite simple relative to today's standards, it was not until 1995 for a man-in-the-middle attack to be documented against it [33, 50]. This attack was discovered using FDR, a then-experimental automated verification tool. Fixes for the attack were also modeled and confirmed using the same automated verification framework [51].

Because automated verification tools can consider an often-unfathomable number of scenarios according to nevertheless rigorous and grounded mathematical notions,

they are useful in unearthing protocol scenarios or errors that can contradict the intuitive assumptions one may have developed for a particular protocol. When work on this thesis began, a dearth existed in the amount of existing work on applying formal verification to the Internet's most used secure protocols and our own intutition was such that if this work were to be done, impactful results could be found.

A common criticism of formal methods is that findings in formal models do not translate to the same security guarantees holding in the target implementation code. This criticism was true before the publication of this thesis and remains true after. However, Chapter 5 of this thesis centers on promising new frameworks and techniques for translating formal security guarantees to real-world implementations targeting the general public. These methodologies have already resulted in stronger real-world integration of formal verification technologies, and establish a clear path forward in that domain.

There are two different styles in which protocols have classically been modeled and in this thesis, we employ both of them. *Symbolic* models were developed by the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles is beyond the scope of this thesis (see e.g. [52]); here we briefly outline their differences in terms of the two tools we will use.

ProVerif analyzes symbolic protocol models, whereas CryptoVerif [53] verifies computational models. The input languages of both tools are similar. For each protocol role (e.g. client or server) we write a *process* that can send and receive messages over public channels, trigger security events and store messages in persistent databases.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically.

In CryptoVerif, messages are concrete bitstrings. Freshly generated nonces and keys are randomly sampled bitstrings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bitstrings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time CryptoVerif process running in parallel.

Authentication goals in both ProVerif and CryptoVerif are written as correspondences between events: for example, if the client triggers a certain event, then the server must have triggered a matching event in the past. Secrecy is treated differently in the two tools; in ProVerif, we typically ask whether the attacker can compute a secret, whereas in CryptoVerif, we ask whether it can distinguish a secret from a random bitstring.

The analysis techniques employed by the two tools are quite different. ProVerif searches for a protocol trace that violates the security goal, whereas CryptoVerif tries to construct a cryptographic proof that the protocol is equivalent (with high probability) to a trivially secure protocol. ProVerif is a push-button tool that may return that the security goal is true in the symbolic model, or that the goal is false with a counterexample, or that it is unable to conclude, or may fail to terminate. CryptoVerif is semi-automated, it can search for proofs but requires human guidance for non-trivial protocols.

We use both ProVerif and CryptoVerif for their complementary strengths. CryptoVerif can prove stronger security properties of the protocol under precise crypto-

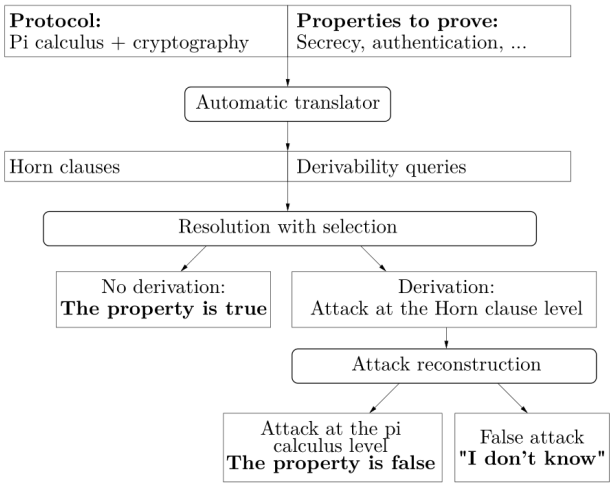

Figure 3: ProVerif's structure, as described by a 2016 technical report on ProVerif [48].

graphic assumptions, but the proofs require more work. ProVerif can quickly analyze large protocols to automatically find attacks, but a positive result does not immediately provide a cryptographic proof of security.

Verification in the Symbolic Model with ProVerif

ProVerif is an automatic symbolic protocol verifier. It supports a wide range of cryptographic primitives, defined by rewrite rules or by equations [48]. In ProVerif, protocols are expressed using the applied-pi calculus [54], a language for modeling security protocols with a syntax inspired by standard ML. Let us examine how each component of a cryptographic protocol ends up being represented in the symbolic model[7].

In the symbolic model, cryptographic primitives are represented as "perfect black-boxes"; a hash function, for example, cannot be modeled to be vulnerable to a length extension attack (which the hash function SHA-1 is vulnerable to, for example.) Encryption primitives are perfect pseudorandom permutations. Hash functions are perfect one-way maps. It remains possible to build complex primitives such as authenticated encryption with associated data (AEAD) and also to model interesting use cases, such as a Diffie-Hellman exponentiation that obtains a shared secret that is outside of the algebraic group. However, in the latter case, such constructions cannot be based on on an algebra that includes the space of integers commonly considered when modeling these scenarios, since all messages are simply a combination of the core terms used to express primitives.

When describing protocols for symbolic verification in ProVerif, the following constructions frequently come into play:

- **Protocol functions.** `let` and `letfun` declarations allow us to chain state transformations or processes in an interface that resembles functions in standard ML.

- **Principals.** Unlike other tools such as AVISPA [56], protocol principals such as "Alice", "Bob", or a client and server are not modeled explicitly. Principals and

[7] Also referred to as the Dolev-Yao model [55].

ProVerif

$M ::=$	terms
v	values
a	names
$f(M_1,\ldots,M_n)$	function application
$E ::=$	enriched terms
M	return value
new $a:\tau; E$	new name a of type τ
let $x = M$ in E	variable definition
if $M = N$ then E_1 else E_2	if-then-else
$P, Q ::=$	processes
0	null process
in$(M, x:\tau); P$	input x from channel M
out$(M, N); P$	output N on channel M
let $x = M$ in P	variable definition
$P \mid Q$	parallel composition
$!P$	replication of P
insert $a(M_1,\ldots,M_n); P$	insert into table a
get $a(=M_1, x_2,\ldots,x_n)$ in P	get table entry specified by M_1
event $M; P$	event M
phase $n; P$	enter phase n
$\Delta ::=$	declaration
type τ	type τ
free $a:\tau$	name a
query q	query q
table $a(\tau_1,\ldots,\tau_n)$	table a
fun $C(\tau_1,\ldots,\tau_n):\tau$	constructor
reduc forall $x_1:\tau_1,\ldots,x_n:\tau_n; f(M_1,\ldots,M_n) = M$	destructor
equation forall $x_1:\tau_1,\ldots,x_n:\tau_n; M = M'$	equation
letfun $f(x_1:\tau_1,\ldots,x_n:\tau_n) = E$	pure function
let $p(x_1:\tau_1,\ldots,x_n:\tau_n) = P$	process
$\Sigma ::= \Delta_1 \ldots \Delta_n$.process P	script

Figure 4: ProVerif syntax, based on the applied-pi calculus.

their roles are inferred from how the protocol messages get constructed and exchanged on the network: after all of the protocol's constructions and functionalities are defined, a "top-level" process expresses the sequence in which operations occur and messages are broadcasted.

Informally, protocol verification in the symbolic model is concerned with modeling every possible execution of the described protocol and ensuring that queries hold in each of the resulting scenarios.

Symbolic verification is useful because it is fully automated whereas verification in the computational model tends to be semi-automated. It analyzes protocol messages

as they are exchanged over the network under a simulated active attacker[8] which will attempt to look for a contradiction to the query by delaying, dropping and injecting messages. That said, ProVerif does not always terminate. ProVerif may sometimes produce an *"I don't know"* answer with more elaborate security queries: however, the effect of this is ProVerif finding false positives. ProVerif cannot miss a contradiction to a query should that contradiction exist. Deriving sound cryptographic proofs using symbolic analysis is still an open problem for real-world protocols [57].

A ProVerif script Σ is divided into two major parts:

1. $\Delta_1 \ldots \Delta_n$, a sequence of declarations which encapsulates all types, free names, queries, constructors, destructors, equations, pure functions and processes. Queries define the security properties to prove. Destructors and equations define the properties of cryptographic primitives.

2. P, the top-level process which then effectively employs $\Delta_1 \ldots \Delta_n$ as its toolkit for constructing a process flow for the protocol.

In processes, the replication $!P$ represents an unbounded number of copies of P in parallel.

Tables store persistent state: The process `insert` $a(M_1,\ldots,M_n);P$ inserts the entry $(M_1,\ldots,M_n)$ in table a and runs P. The process `get` $a(=M_1,x_2,\ldots,x_n)$ `in` P looks for an entry $(N_1,\ldots,N_n)$ in table a such that $N_1 = M_1$. When such an entry is found, it binds $x_2,\ldots,x_n$ to $N_2,\ldots,N_n$ respectively and runs P.

Events are used for recording that certain actions happen (e.g. a message was sent or received), in order to use that information for defining security properties.

Phases model a global synchronization: processes initially run in phase 0; then at some point processes of phase 0 stop and processes of phase 1 run and so on. For instance, the protocol may run in phase 0 and some keys may be compromised after the protocol run by giving them to the adversary in phase 1.

More information on ProVerif is available in the ProVerif manual [58].

Verification in the Computational Model using CryptoVerif

Computational model verification produces *game-based proofs*, while symbolic verification only guarantees that, within the expressed model, no contradiction to a stated security query expressing a specific security goal exists. Given that computational model verification mirrors the style already used by cryptographers for hand proofs, some cryptographers prefer using the computational model protocol verifier CryptoVerif [49].

Traditionally, symbolic cryptographic models have been particularly suitable for automated protocol analysis. They ignore attacks with negligible probability and assume that each cryptographic function is a perfect black-box. For example, in such models, hash functions never collide and encryption is a message constructor that can only be reversed by decryption. In the *computational model*, cryptographic primitives are functions over bitstrings and their security is specified in terms of probabilities. These models are more precise and closer to those used by cryptographers, but usually do not lend themselves to fully automated proofs. Generally, we will use symbolic models when we are trying to find attacks that rely on logical flaws in the protocol and in its use of cryptographic primitives. We will use computational models when we

[8] ProVerif also supports restricting evaluation to a passive attacker, which can be crucial for testing for security goals such as secrecy and authenticity, as shown in Chapter 3.

want to build a cryptographic proof of security, starting from standard cryptographic assumptions.

That being said, computational models are still models. They do not fully match reality: an easy example of this is their ignoring side channels, such as timing attacks or power consumption analysis attacks.

Like ProVerif, CryptoVerif also uses the applied-pi calculus as its syntax for describing protocols, albeit with different underlying semantics. Unlike ProVerif, protocol messages are bitstrings and primitives are functions on those bitstrings. Furthermore, the adversary is not restricted only to apply operations based on described primitives, but is rather a probabilistic polynomial-time Turing machine that can manipulate the bitstrings and apply any algorithm, much like a regular computer. This makes it such that computational modeling in CryptoVerif is considered to be more "realistic" than modeling in ProVerif. This also allows for CryptoVerif to specify probabilistic bounds on attacks and makes security parameters[9] possible to express.

Processes, contexts, adversaries. CryptoVerif mechanizes proofs by sequences of games, similar to those written on paper by cryptographers [59, 60]. It represents protocols and cryptographic games in a probabilistic process calculus. We refer the reader to [53] for details on this process calculus. We will explain the necessary constructs as they appear.

Even though CryptoVerif can evaluate the probability of success of an attack as a function of the number of sessions and the probability of breaking each primitive (exact security), for simplicity, we consider here the asymptotic framework in which we only show that the probability of success of an attack is negligible as a function of the security parameter η. (A function f is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) All processes run in polynomial time in the security parameter and manipulate bitstrings of polynomially bounded length.

A *context* C is a process with one or several holes $[\,]$. We write $C[P_1, \ldots, P_n]$ for the process obtained by replacing the holes of C with $P_1, \ldots, P_n$ respectively. An *evaluation context* is a context with one hole, generated by the following grammar:

$C ::=$	evaluation context
$[\,]$	hole
newChannel $c; C$	channel restriction
$Q \mid C$	parallel composition
$C \mid Q$	parallel composition

The channel restriction **newChannel** $c; Q$ restricts the channel name c, so that communications on this channel can occur only inside Q and cannot be received outside Q or sent from outside Q. The parallel composition $Q_1 \mid Q_2$ makes simultaneously available the processes defined in Q_1 and Q_2. We use evaluation contexts to represent *adversaries*.

Indistinguishability. A process can execute events, by two constructs: **event** $e(M_1, \ldots, M_n)$ executes event e with arguments $M_1, \ldots, M_n$ and **event_abort** e executes event e without argument and aborts the game. After finishing execution of a process, the system produces two results: the sequence of executed events $\mathcal{E}$ and the information whether the game aborted ($a =$ **abort**, that is, executed **event_abort**) or terminated normally ($a = 0$). These events and result can be used to distinguish games, so we introduce an additional algorithm, a *distinguisher* D that takes as input the sequence of events $\mathcal{E}$ and

[9] e.g. "128-bit security", which could be based on a cipher with a key space of 2^{128} and no known weaknesses.

the result a and returns true or false. Distinguishers must run in time polynomial in the security parameter. We write $\Pr[Q \rightsquigarrow_\eta D]$ for the probability that the process Q executes events $\mathcal{E}$ and returns a result a such that $D(\mathcal{E}, a) = \text{true}$, with security parameter η.

Definition 1 (Indistinguishability)**.** *We write $Q \approx^V Q'$ when, for all evaluation contexts C acceptable for Q and Q' with public variables V and all distinguishers D, $|\Pr[C[Q] \rightsquigarrow_\eta D] - \Pr[C[Q'] \rightsquigarrow_\eta D]|$ is a negligible function η.*

Intuitively, $Q \approx^V Q'$ means that an adversary has a negligible probability of distinguishing Q from Q', when it can read the variables in the set V. When V is empty, we omit it.

The condition that C is acceptable for Q and Q' with public variables V is a technical condition that guarantees that $C[Q]$ and $C[Q']$ are well-formed. The public variables V are the variables of Q and Q' that C is allowed to read directly.

The relation $\approx^V Q'$ is an equivalence relation and $Q \approx^V Q'$ implies $C[Q] \approx^{V'} C[Q']$ for all evaluation contexts C acceptable for Q and Q' with public variables V and all $V' \subseteq V \cup \text{var}(C)$, where $\text{var}(C)$ is the set of variables of C.

Security assumptions on cryptographic primitives are given to CryptoVerif as indistinguishability properties that it considers as axioms.

Secrecy. Intuitively, in CryptoVerif, secrecy means that the adversary cannot distinguish between the secrets and independent random values. This definition corresponds to the "real-or-random" definition of security [61]. A formal definition in CryptoVerif can be found in [62].

Correspondences. Correspondences [63] are properties of executed sequences of events, such as "if some event has been executed, then some other event has been executed". They are typically used for formalizing authentication. Given a correspondence *corr*, we define a distinguisher D such that $D(\mathcal{E}, a) = \text{true}$ if and only if the sequence of events $\mathcal{E}$ satisfies the correspondence *corr*. We write this distinguisher simply *corr* and write $\neg corr$ for its negation.

Definition 2 (Correspondence)**.** *The process Q* satisfies the correspondence *corr with public variables V if and only if, for all evaluation contexts C acceptable for Q with public variables V that do not contain events used in corr, $\Pr[C[Q] \rightsquigarrow_\eta \neg corr]$ is negligible.*

We refer the reader to [64] for more details on the verification of correspondences in CryptoVerif.

Outline, Contributions and Related Work

This thesis was written with the intent of bridging the gap between the formal verification and the real-world secure implementation of cryptographic protocols.

Initial chapters of this thesis deal with formal verification research and results, while later chapters deals with robust real-world cryptographic implementations. While the two parts handle distinct features of this thesis, they both center around shared use cases and protocols: Chapters 1 and 2, for example, discusse formal verification approaches to secure messaging and TLS, with Chapter 5 eventually picking up those same protocols again with a focus on secure Web-based implementations.

Chapter 1: Formal Verification for Secure Messaging

Chapter 1 begins with a study of secure messaging protocols. We present security goals and a threat model for secure messaging protocols. We discuss protocol weaknesses and implementation bugs in the messaging protocol underlying the popular Telegram application, as well as issues with the Off-the-Record secure messaging protocol. We formalize and analyze a variant of Signal Protocol for a series of security goals, including confidentiality, authenticity, forward secrecy and future secrecy, against a classic symbolic adversary. Our analysis uncovers several weaknesses, including previously unreported replay and key compromise impersonation attacks and we propose and implement fixes which we then also verify. We present proofs of message authenticity, secrecy and forward secrecy for SP obtained using the CryptoVerif computational model prover [53].

Chapter 2: Formal Verification for Transport Layer Security

In Chapter 2, we follow with a similar treatment to Chapter 1 of TLS 1.3. We present work that was completed admist the effort to specify TLS 1.3: after 18 drafts, the protocol was nearing completion, and the working group had appealed to researchers to analyze the protocol before publication. We responded by presenting a comprehensive analysis of the TLS 1.3 Draft-18 protocol. We seek to answer three questions that had not been fully addressed in previous work on TLS 1.3: (1) Does TLS 1.3 prevent well-known attacks on TLS 1.2, such as Logjam or the Triple Handshake, even if it is run in parallel with TLS 1.2? (2) Can we mechanically verify the computational security of TLS 1.3 under standard (strong) assumptions on its cryptographic primitives? (3) How can we extend the guarantees of the TLS 1.3 protocol to the details of its implementations?

To answer these questions, we propose a methodology for developing verified symbolic and computational models of TLS 1.3 hand-in-hand with a high-assurance reference implementation of the protocol. We present symbolic ProVerif models for various intermediate versions of TLS 1.3 and evaluate them against a rich class of attacks to reconstruct both known and previously unpublished vulnerabilities that influenced the current design of the protocol. We present a computational CryptoVerif model for TLS 1.3 Draft-18 and prove its security.

In comparison with previous cryptographic proofs of draft versions of TLS 1.3 [65, 66, 67], our cryptographic assumptions and proof structure is similar. The main difference in this work is that our proof is mechanized, so we can easily adapt and recheck our proofs as the protocol evolves. Shortly after the publication of our work, other analyses of TLS 1.3 in the symbolic model were published [68], written using the Tamarin proverif instead of ProVerif.

Our full CryptoVerif development consists of 2270 lines, including new definitions and lemmas for the key schedule (590 lines), a model of the initial handshake (710 lines), a model of PSK-based handshakes (820 lines), and a model of the record protocol (150 lines). For different proofs, we sometimes wrote small variations of these files, and we do not count all those variations here. All proofs completed in about 11 minutes. The total verification effort took about 6 person-weeks for a CryptoVerif expert.

Chapter 3: Formal Verification of Arbitrary Noise Protocols

In Chapter 3, we target the Noise Protocol Framework and present Noise Explorer, an online engine for designing, reasoning about and formally verifying arbitrary Noise Handshake Patterns. Based on our formal treatment of the Noise Protocol Framework, Noise Explorer can validate any Noise Handshake Pattern and then translate it into a model ready for automated verification. We use Noise Explorer to analyze 50 Noise Handshake Patterns. We confirm the stated security goals for 12 fundamental patterns and provide precise properties for the rest. We also analyze unsafe Noise Handshake Patterns and document weaknesses that occur when validity rules are not followed. All of this work is consolidated into a usable online tool that presents a compendium of results and can parse formal verification results to generate detailed- but-pedagogical reports regarding the exact security guarantees of each message of a Noise Handshake Pattern with respect to each party, under an active attacker and including malicious principals. Noise Explorer evolves alongside the standard Noise Protocol Framework, having already contributed new security goal verification results and stronger definitions for pattern validation and security parameters.

This work represents the first comprehensive formal analysis of the Noise Protocol Framework. However, substantial tangential work has occured centering on the WireGuard [69] VPN protocol, which employs the `IKpsk2` Noise Handshake Pattern: Lipp [70] presented an automated computational proof of WireGuard, Donenfeld et al [71] presented an automated symbolic verification of WireGuard and Dowling et al [72] presented a hand proof of WireGuard. These analyses' results on the `IKpsk2` handshake pattern were in line with those we found in our own symbolic analysis. Other work exists centering on the automated verification of modern protocols [73, 74].

Chapter 4: Formal Verification for Automated Domain Name Validation

In Chapter 4, we survey the existing process for issuing domain-validated certificates in major certification authorities. Based on our findings, we build a security model of domain-validated certificate issuance. We then model the ACME protocol in the applied pi-calculus and verify its stated security goals against our security model. We compare the effective security of different domain validation methods and show that ACME can be secure under a stronger threat model than that of traditional CAs. We also uncover weaknesses in some flows of ACME 1.0 and propose verified improvements that have been adopted in the latest protocol draft submitted to the IETF.

Chapter 5: Formally Verified Secure Channel Implementations in Web Applications

Chapter 5 mirrors the findings of Chapters 1 and 2 and extends them towards the obtention of robust real-world cryptographic implementations. We present ProScript, a new language subset of JavaScript geared towards secure protocol implementation and that allows for the automated extraction of protocol models straight from protocol implementation code. Using ProScript, our subset of JavaScript geared towards secure protocol implementation, we are able to implement a variant of Signal Protocol which we then automatically translate to ProVerif and verify against a battery of formalized queries, which include tests for forward secrecy and key compromise impersonation. We similarly implement TLS 1.3 (draft-18) using ProScript and verify the extracted

symbolic model against TLS-specific queries. We introduce Cryptocat, the first user-friendly secure messenger to deploy formally verified protocol logic at its core, and RefTLS, an interoperable implementation of TLS 1.0-1.3 and automatically analyze its protocol core by extracting a ProVerif model from its typed JavaScript code.

The symbolic security guarantees of RefTLS are weaker than those of computationally-verified implementations like miTLS [32]. However, unlike miTLS, our analysis is fully automated and it can quickly find attacks. The type-based technique of miTLS requires significant user intervention and is better suited to building proofs than finding attacks.

Our ProScript-to-ProVerif compiler is inspired by previous works on deriving ProVerif models from F# [75], Java [76], and JavaScript [77]. Such translations have been used to symbolically and computationally analyze TLS implementations [31]. An alternative to model extraction is to synthesize a verified implementation from a verified model; [78] shows how to compile CryptoVerif models to OCaml and uses it to derive a verified SSH implementation.

The most advanced case studies for verified protocol implementations use dependent type systems, because they scale well to large codebases. Refinement types for F# have been used to prove both symbolic [79] and cryptographic security properties, with applications to TLS [32]. The F* programming language [80] has been used to verify small protocols and cryptographic libraries [81]. Similar techniques have been applied to the cryptographic verification of Java programs [82].

Chapter 6: Formally Verified Secure Collaborative Document Editing

Chapter 6 concludes the thesis by moving towards an unaddressed protocal use case. We present Capsule, the first formalized and formally verified protocol standard that addresses secure collaborative document editing. Capsule provides confidentiality and integrity on encrypted document data, while also guaranteeing the ephemeral identity of collaborators and preventing the server from adding new collaborators to the document. Capsule also, to an extent, prevents the server from serving different versions of the document being collaborated on. We provide a full protocol description of Capsule as well as formal verification results on the Capsule protocol in the symbolic model. Finally, we present a full software implementation of Capsule, which includes a novel formally verified signing primitive implementation.

To the best of our knowledge, the only prior existing work regarding collaborative document encryption is CryptPad [83], an open source web client. CryptPad shares similarities with Capsule especially in that both use a hash chain of encrypted diffs in order to manage document collaboration and to reconstruct the document. However, CryptPad adopts a more relaxed threat model of an "honest but curious cloud server" and does not appear to guard against a server interfering with the document's list of participants or its history. Meanwhile, Capsule explicitly guards against a server injecting false participants by requiring a certain proof from all participants. CryptPad's software implementation is also limited within a web browser and unlike Capsule's, does not employ formally verified cryptographic primitives.

Chapter 1

Formal Verification for Secure Messaging

This work was published in the proceedings for the 3rd IEEE European Symposium on Security and Privacy, 2017. It was completed with co-authors Karthikeyan Bhargavan and Bruno Blanchet.

Designing new cryptographic protocols is highly error-prone; even well-studied protocols, such as Transport Layer Security (TLS), have been shown to contain serious protocol flaws found years after their deployment (see e.g. [84]). Despite these dangers, modern web applications often embed custom cryptographic protocols that evolve with each release. The design goal of these protocols is typically to protect user data as it is exchanged over the web and synchronised across devices, while optimizing performance for application-specific messaging patterns and deployment constraints. Such custom protocols deserve close scrutiny, but their formal security analysis faces several challenges.

First, web applications often evolve incrementally in an ad hoc manner, so the embedded cryptographic protocol is only ever fully documented in source code. Even when protocol designers or security researchers take the time to create a clear specification or formal model for the protocol, these documents are typically incomplete and quickly go out-of-date. Second, even if the protocol itself is proved to be secure, bugs in its implementation can often bypass the intended security guarantees. Hence, it is not only important to extract a model of the protocol from the source code and analyze its security, it is essential to do so in a way that the model can evolve as the application is modified.

In this chapter, we study the protocol underlying the Signal messaging application developed by Open Whisper Systems. Variants of this protocol have also been deployed within WhatsApp, Facebook Messenger, Viber and many other popular applications, reaching over a billion devices. The protocol has been known by other names in the past, including Axolotl, TextSecure (versions 1, 2, and 3) and it continues to evolve within the Signal application under the name Signal Protocol. Until recently, the main documentation for the protocol was its source code, but new specifications for key components of the protocol have now been publicly released [3, 4].

Signal Protocol has ambitious security goals; it enables asynchronous (zero round-trip) authenticated messaging between users with end-to-end confidentiality. Each message is kept secret even if the messaging server is compromised and even if the

user's long term keys are compromised, as long as these keys are not used by the attacker before the target message is sent (forward and future secrecy.) To achieve these goals, Signal uses a novel authenticated key exchange protocol (based on mixing multiple Diffie-Hellman shared secrets) and a key refresh mechanism (called double ratcheting). The design of these core mechanisms in TextSecure version 2 was cryptographically analyzed in [85] but the protocol has evolved since then and the security of Signal as it is currently implemented and deployed remains an open question.

Although they all implement the same core protocol, different implementations of the Signal protocol vary in important details, such as how users are identified and authenticated, how messages are synchronised across devices, etc.

We call this variant SP in the rest of this chapter. We develop a detailed model for SP in the applied pi calculus and verify it using the ProVerif protocol analyzer [48] for these security goals against adversaries in a classic Dolev-Yao model [29]. We also develop a computational proof for SP using the CryptoVerif prover [53].

Contributions. We present an outline of our contributions in this chapter:

A Security Model and New Attacks. We present security goals and a threat model for secure messaging. As a motivation for our verification approach, we discuss protocol weaknesses and implementation bugs in the messaging protocol underlying the popular Telegram application, as well as issues with the Off-the-Record secure messaging protocol.

A Symbolic Security Analysis of SP. We formalize and analyze a variant of Signal Protocol for a series of security goals, including confidentiality, authenticity, forward secrecy and future secrecy, against a classic symbolic adversary. Our analysis uncovers several weaknesses, including previously unreported replay and key compromise impersonation attacks and we propose and implement fixes which we then also verify.

A Computational Cryptographic Proof for SP. We present proofs of message authenticity, secrecy and forward secrecy for SP obtained using the CryptoVerif computational model prover [53].

In Chapter 5, we extend these contributions to target robust real-world protocol implementations.

Online materials related to this work are available at this online repository: `https://github.com/Inria-Prosecco/proscript-messaging/`.

1.1 A Security Model for Encrypted Messaging

We consider a simple messaging API as depicted below. An initiator A can start a conversation with B by calling startSession with long-term secrets for A and any identity credentials it has for B. This function returns the initial conversation state T_0. Thereafter, the initiator can call send with a plaintext message M_1 to obtain the encrypted message E_1 that it needs to send on the network. Or it can call recv with an encrypted message E_2 it received (supposedly from B) to obtain the plaintext message M_2.

```
1  T_0^ab = startSession(secrets_A, identity_B)
2  T_1^ab, E_1 = send(T_0^ab, M_1)
3  T_2^ab, M_2 = recv(T_1^ab, E_2)
```

The responder B uses a similar API to accept sessions and receive and send messages:

```
1  T_0^ba = acceptSession(secrets_B, identity_A)
2  T_1^ba, M_1 = recv(T_0^ba, E_1)
```

3 $T_2^{ba}, E_2 = \text{send}(T_1^{ba}, M_2)$

We deliberately chose a functional state-passing API with no side-effects in order to focus on cryptographic protocol computations, rather than the concrete details of how these messages are sent over the network.

1.1.1 Threat Model

While threat models vary for different protocols, we consider the following threats in this work:

- **Untrusted Network** We assume that the attacker controls the network and so can intercept, tamper with and inject network messages (e.g. E_1, E_2). Moreover, if two messaging clients communicate via a server, we typically treat that server as untrusted.
- **Malicious Principals** The attacker controls a set of valid protocol participants (e.g. M), for whom it knows the long-term secrets. The attacker may advertise any identity key for its controlled principals; it may even pretend to own someone else's identity keys.
- **Long-term Key Compromise** The attacker may compromise a particular principal (e.g. A) during or after the protocol, to obtain her long-term secrets.
- **Session State Compromise** The attacker may compromise a principal to obtain the full session state at some intermediate stage of the protocol (e.g. T_1^{ab}).

1.1.2 Security Goals

We state a series of semi-formal security goals in terms of our generic messaging API. We use the phrase "A sends a message M to B" to mean that A calls $\mathsf{Send}(T, M)$ with a session state T that represents a conversation between A and B. Similarly, we say that "B receives a message M from A" to mean that B obtained M as a result of calling $\mathsf{Recv}(T, E)$ with a session T with A.

Unless otherwise specified, the following security properties assume that both A and B are honest, that is, their long-term secrets have not been compromised. We begin with several variants of authenticity goals:

- **Message Authenticity** If B receives a message M from A, then A must have sent M to B.
- **No Replays** Each message received by B from A corresponds to a unique message sent by A. That is, the attacker must not be able to get a single message sent by A to be accepted twice at B.
- **No Key Compromise Impersonation** Even if the long-term secrets of B are compromised, message authenticity must hold at B. That is, the attacker must not be able to forge a message from A to B.

Our definition of message authenticity covers integrity as well as sender and recipient authentication. Obtaining message authenticity also helps prevent *unknown key share* attacks, where B receives a message M from A, but A sent that message to a different intended recipient C. We define four confidentiality goals:

- **Secrecy** If A sends some secret message M to B, then nobody except A and B can obtain M.

- **Indistinguishability** If A randomly chooses between two messages M_0, M_1 (of the same size) and sends one of them to B, the attacker cannot distinguish (within the constraints of the cryptographic model) which message was sent.

- **Forward Secrecy** If A sends a secret message M to B and if A's and B's long-term secrets are subsequently compromised, the message M remains secret.

- **Future Secrecy** Suppose A sends M in a session state T, then receives N, then sends M'. If the session state T is subsequently compromised, the message M' remains secret.

Some protocols satisfy a weaker notion of forward secrecy, sometimes called *weak* forward secrecy, where an attacker is not allowed to actively tamper with the protocol until they have compromised the long-term keys [86]. Some messaging protocols also seek to satisfy more specific authenticity and confidentiality goals, such as non-repudiation and plausible deniability. We will ignore them in this work.

In the next section, we evaluate two secure messaging applications against these goals, we find that they fail some of these goals due to subtle implementation bugs and protocol flaws. Hence, we advocate the use of automated verification tools to find such attacks and to prevent their occurrence.

1.2 Symbolic Verification with ProVerif

Modern messaging and transport protocols share several distinctive features [2]: for example, Signal Protocol, SCIMP, QUIC and TLS 1.3 share a strong focus on asynchronous key agreement with a minimum of round trips. Some also guarantee new security goals such as future secrecy. The protocols also assume non-standard (but arguably more user-friendly) authentication infrastructures such as Trust-on-First-Use (TOFU). Modern messaging protocols have several interesting features and properties that set them apart from classic cryptographic protocols and upon which we focus our formal verification efforts:

New Messaging Patterns. In contrast to connection-oriented protocols, modern cryptographic protocols are constrained by new communication flows such as zero-round-trip connections and asynchronous messaging, where the peer may not even be present.

Confidentiality Against Strong Adversaries. Web messaging protocols need to be robust against server compromise and device theft and so seek to provide strong and novel forward secrecy guarantees.

Weak Authentication Frameworks. Many of these protocols do not rely on public key infrastructures. Instead they may authenticate peers on a TOFU basis or even let peers remain anonymous, authenticating only the shared connection parameters.

Code First, Specify Later. Unlike Internet protocols, which are designed in committee, these protocols are first deployed in code and hand-tuned for performance on a particular platform. The code often remains the definitive protocol specification.

Before outlining our verification approach for such protocols, we take a closer look at Telegram, a secure messaging application.

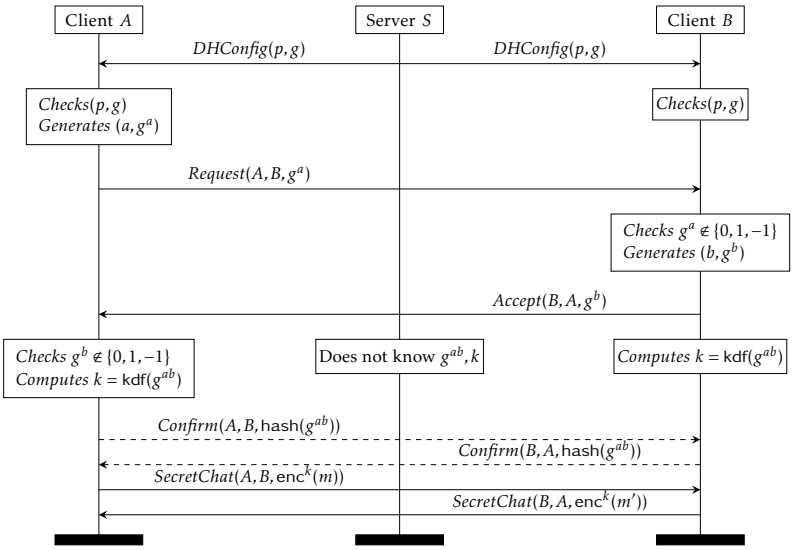

Figure 1.1: Telegram's MTProto Protocol for Secret Chats.

1.2.1 Secret Chats in Telegram

We study the "MTProto" [87] secure messaging protocol used in the Telegram messaging application. We focus on the "secret chat" feature that allows two Telegram clients who have already authenticated themselves to the server to start an encrypted conversation with each other. Although all messages pass through the Telegram server, the server is untrusted and should not be able to decrypt these messages. The two clients A and B download Diffie-Hellman parameters from the Telegram server and then generate and send their public values to each other.

The key exchange is not authenticated with long-term credentials. Instead, the two clients are expected to communicate out-of-band and compare a SHA-1 hash (truncated to 128-bits) of the Diffie-Hellman shared secret. If two users perform this authentication step, the protocol promises that messages between them are authentic, confidential and forward secret, even if the Telegram server is compromised. However this guarantee crucially relies on several cryptographic assumptions, which may be broken either due to implementation bugs or computationally powerful adversaries, as we describe below.

Malicious Primes. MTProto relies on clients checking that the Diffie-Hellman configuration (p, g) that they received from the server is suitable for cryptographic use. The specification requires that p be a large safe prime; hence the client must check that it has exactly 2048 bits and that both p and $(p-1)/2$ are prime, using 15 rounds of the Miller-Rabin primality test. There are several problems with this check. First, the server may be able to carefully craft a non-prime that passes 15 rounds of Miller-Rabin. Second, checking primality is not enough to guarantee that the discrete log problem will be hard. If the prime is chosen such that it has "low weight", the SNFS algorithm applies, making discrete logs significantly more practical [88]. Even if we accept that primality checking may be adequate, it is unnecessary for an application like Telegram, which could simply mandate the use of well-known large primes instead [89].

Public Values in Small Subgroups. A man-in-the-middle can send to both A and B

public Diffie-Hellman values g^b and g^a equal to 1 (resp. 0, resp. $p-1$). Both A and B would then compute the shared secret as $g^{ab} = 1$ (resp. 0, resp. 1 or -1). Since their key hashes match, A and B think they have a confidential channel. However, the attacker can read and tamper with all of their messages. More generally, MTProto relies on both peers verifying that the received Diffie-Hellman public values do not fall in small subgroups. This check is adequate to prevent the above attack but could be made unnecessary if the two public values were to be authenticated along with the shared secret in the hash compared by the two peers.

Implementation Bugs in Telegram for Windows. The above two weaknesses, reported for the first time in this work, can result in attacks if the protocol is not implemented correctly. We inspected the source code for Telegram on different platforms; while most versions perform the required checks, we found that the source code for Telegram for Windows Phone did not check the size of the received prime, nor did it validate the received Diffie-Hellman values against 1, 0 or $p-1$. We reported both bugs to the developers, who acknowledged them and awarded us a bug bounty.

Such bugs and their consequent attacks are due to missed security-relevant checks and they can be found automatically by symbolic analysis. For example, [35] shows how to model unsafe (malicious) primes and invalid public keys in ProVerif and uses this model to find vulnerabilities in several protocols that fail to validate Diffie-Hellman groups or public keys.

Other Cryptographic Weaknesses. MTProto is also known to be vulnerable to an authentication attack if an adversary can compute 2^{64} SHA-1 hashes [90] and to chosen-ciphertext attacks on its unusual AES-IGE encryption scheme [91]. How can we be sure that there are no other protocol flaws or implementation bugs hiding in MTProto? Any such guarantee would require a systematic security analysis of both the protocol and the source code against both symbolic and computational adversaries.

1.2.2 Towards Automated Verification

The innovative designs and unusual security guarantees of secure messaging protocols demand formal security analysis. Hand-written models with detailed cryptographic proofs can be useful as a reference, but we observe that the most recent analysis of Signal Protocol [85] is already out of date, as the protocols have moved on to new versions. Furthermore, manual cryptographic proofs often leave out details of the protocol for simplicity and some of these details (e.g. client authentication) may lead to new attacks.

In this work, we advocate the use of automated verification tools to enable the analysis of complex protocols as they evolve and incorporate new features. Moreover, we would also like to find protocol implementation bugs (like the ones in previous versions of Telegram) automatically.

1.3 Formally Verifying SP

We describe SP, a variant of Signal Protocol that closely follows TextSecure version 3. We show how we implement and verify this protocol in our framework.

1.3.1 Protocol Overview

In SP, as illustrated in Figure 1.2, each client publishes a long-term Diffie-Hellman public key and a set of ephemeral Diffie-Hellman public keys (called *"pre-keys"*). These

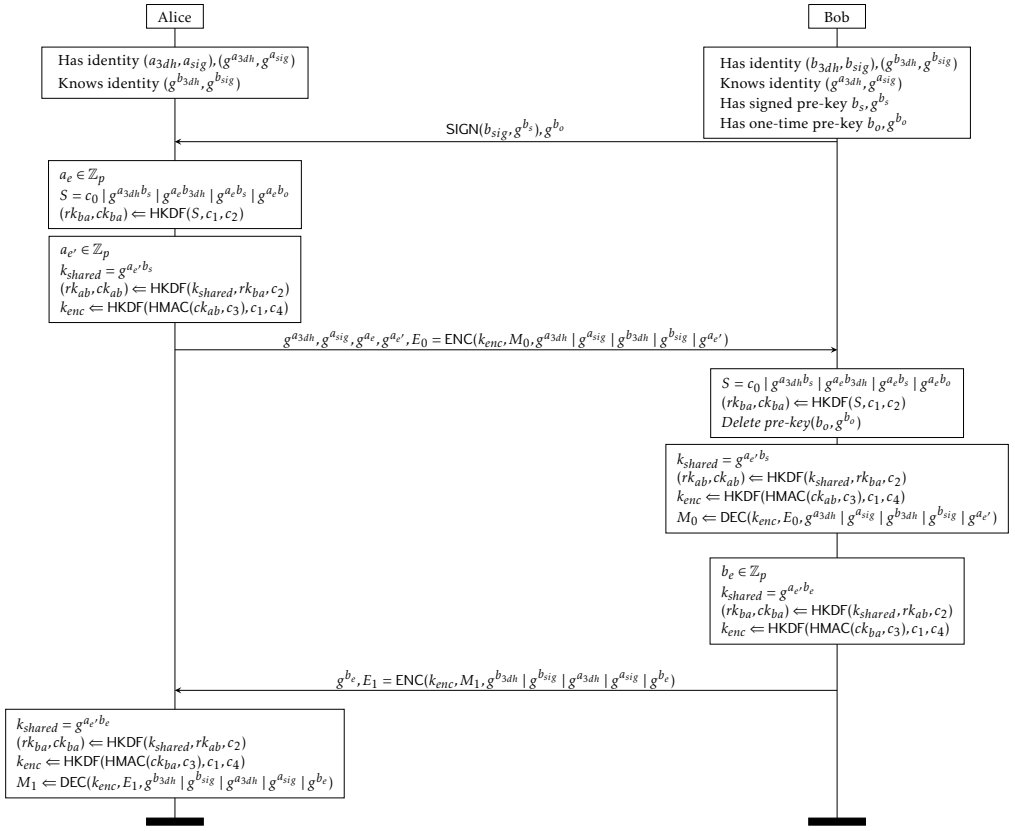

Figure 1.2: SP, a variant of Signal with minor differences. Alice requests a signed pre-key from Bob (via the server) and sends an initial message M_0. Bob accomplishes his side of the key exchange and obtains M_0. Bob later sends his reply M_1, illustrating the Axolotl ratchet post-AKE. We ignore the hash-based ratchet that occurs when two consecutive messages are sent in the same direction. c_i refers to various constants found throughout the protocol.

keys include both signed pre-keys, which can be reused for some period of time and non-signed, one-time pre-keys, which are fresh at each session. To send a message to Bob, Alice retrieves Bob's long-term keys $(g^{b_{3dh}}, g^{b_{sig}})$, a signed pre-key g^{b_s} and a one-time pre-key g^{b_o}. She then chooses her own ephemeral g^{a_e}. A four-way Diffie-Hellman handshake is accomplished using Alice and Bob's long-term identity keys and their short-term ephemeral keys in order to derive the session secret S. The one-time pre-key is optional: when there remains no available one-time pre-key, the exchange is performed with a triple Diffie-Hellman handshake. An encryption key, k_{enc}, is then derived from S by Hash-Based Key Derivation (HKDF) [92] and the message M_0 is sent encrypted under the authenticated encryption scheme AES-GCM, with public and ephemeral keys as associated data: $\mathsf{ENC}(k, m, ad)$ means that m is encrypted with k and both the message m and the associated data ad are authenticated. Subsequent messages in the conversation obtain authentication by chaining to S via a forward-ratcheting construction that also employs HKDF. Each sent message includes its own

newly generated ephemeral public key and the protocol's double ratchet key refresh mechanism manages the key state by advancing key chaining with every message.

SP's forward and future secrecy goals are intended to make it so that the compromise of Alice or Bob's long-term keys allows for their impersonation but not for the decryption of their messages. The use of a signed initial ephemeral pre-key results in weaker forward secrecy guarantees for the first flight of messages from A to B: no forward secrecy is provided if both the long-term keys and pre-keys are leaked, although the guarantees for subsequent flights remain strong. If pre-keys are not signed, then the protocol only offers weak forward secrecy with respect to long-term key leakage. We note that the term "forward secrecy" can be confusing in a protocol like Signal, because a number of keys are at stake: long-term keys $((a_{3dh}, a_{sig}), (b_{3dh}, b_{sig}))$, signed pre-key b_s, one-time pre-key b_o, ephemeral keys $(a_e, a_{e'}, b_{e'})$, root keys $(rk_{ab}, ck_{ab}, rk_{ba}, ck_{ba})$ and message keys (k_{enc}). Any formal analysis of the protocol must precisely state which of these keys can be compromised and when.

Differences from other versions of Signal

An earlier version of the protocol, TextSecure Version 2, was cryptographically analyzed in previous work [85]. There are two key differences between SP and TextSecure Version 2.

Signed, Time-Based Pre-Keys. Version 2 uses a triple Diffie-Hellman handshake with one of a hundred pre-keys that Bob stores on the server (including a "last-resort" pre-key). TextSecure Version 3 and all subsequent versions of Signal, including SP, use a signed time-based pre-key, used in conjunction with an unsigned one-time pre-key in case it is available. Bob periodically replaces his signed pre-key (for example, once every week), which may be re-used until its replacement and refreshes his collection of unsigned one-time pre-keys. In SP, when one-time pre-keys are exhausted, no "last-resort" pre-key is used.

Stronger Identity Protection. Since Version 3, tag_n is expanded to include the long-term identities of the sender and recipient, which is not the case in Version 2. This provides a slightly stronger authentication guarantee in the rare case that the encryption keys for different pairs of users turns out to be the same.

In addition to these differences with Version 2, SP also differs from other variants of Signal in one key aspect. In SP, long-term identities are split into one Diffie-Hellman key pair and one signing key pair. In Signal, the same key pair is used for both operations, by applying an elliptic curve conversion from the Montgomery curve-based Diffie-Hellman key pair to obtain its twisted Edwards curve-based signing key pair equivalent. We choose to use separate keys instead, because in our cryptographic proof, we do not want to add a non-standard cryptographic assumption about the use of the same key in two independent cryptographic operations. In exchange for using standard cryptographic assumptions, we consider the cost of adding an extra 32 byte key to the protocol to be acceptable.

1.3.2 Protocol Verification

We use ProVerif to verify the security goals of our extracted model by defining queries that accurately test the resilience of security properties against an active adversary. Under an active Dolev-Yao adversary, ProVerif was able to verify confidentiality, authenticity, forward secrecy and future secrecy for Alice and Bob initializing a session and exchanging two secret messages, with a compromised participant, Mallory, also

Goals	Messages	Parties	Roles	Time
Secrecy	1	A, B	One	00h.04m.07s.
Secrecy	1	A, B	Two	00h.11m.17s.
Indist.	1	A, B	One	02h.06m.15s.
Authen.	1	A, B, M	One	00h.58m.19s.
Authen.	1	A, B, M	Two	29h.17m.39s.
Fo. Se.	1	A, B	One	00h.04m.14s.
KCI	1	A, B	One	00h.19m.20s.

Figure 1.3: Verification times for SP ProVerif models.

being allowed to initialize sessions and exchange non-secret messages with Alice and Bob. Our analysis revealed two novel attacks: a key compromise impersonation attack and a replay attack, for which we propose a fix. Aside from these attacks, we were also able to model the previously documented Unknown Keyshare Attack [85].

Extracts of our compiled SP implementation are available online [93]. Models begin with type declarations followed by public constant declarations, equational relationships for cryptographic primitives, protocol functions, queries and relevant names and finally the top-level process with its associated queries.

The top-level process queries for security properties such as confidentiality, authenticity and forward secrecy between two roles: an initiator (e.g. Alice) who sends an initial message and thereby initializes an authenticated key exchange and a responder (e.g. Bob) who receives the message and who may send a response. Some models include a third compromised identity, Mallory, who also communicates with Alice and Bob but while leaking her private keys to the attacker beforehand. In some instances, we also model parallel process executions where each identity (Alice, Bob and optionally Mallory) assumes both the role of the initiator and the responder. We informally call this latter scenario a "two-way role" model.

Secrecy and Indistinguishability. For every message considered in our protocol model, we define a secret constant M_n where M_1 is the initial message in a session. These secret values are then used as the plaintext for the encrypted messages sent by the principals. We show that an active attacker cannot retrieve a message's plaintext M_n using the query:

$$\mathsf{query}(\mathsf{attacker}(M_n)) \tag{1.1}$$

Similarly, we show indistinguishability using the query $\mathsf{query}(\mathsf{noninterf}(M_n))$.

Forward and Future Secrecy. We examine forward and future secrecy in Signal Protocol in multiple scenarios: the compromise of long-term keys and the compromise of message keys in two different types of message flights. In these scenarios, we need to model that keys are leaked after sending or receiving certain messages. We rely on ProVerif *phases* for that: intuitively, t represents a global clock and processes occurring after the declaration of a phase t are active only during this phase.

We show that message M_1 remains secret by query (1.1) even if the long-term keys $(a_{3dh}, a_{sig}, b_{3dh}, b_{sig})$ are leaked after sending M_1. Furthermore, we can modify our ProVerif model to produce a sanity check: if responder Bob skips the signature check on g^{a_s}, ProVerif shows that an active attacker becomes capable of violating this forward secrecy property.

Next, we examine two different messaging patterns in the Double Ratchet algorithm and find that they approach forward and future secrecy differently:

- **Single-Flight Pattern.** In this scenario, Alice sends Bob a number of messages M_n and M_{n+1} where $n > 1$ and does not receive a response. In this scenario, Bob's lack of response does not allow Alice to obtain a fresh ephemeral key share g^{b_e} required to establish a new k_{shared} in T^{ab}_{n+1} to be used for M_{n+1}, so Alice just updates the key ck_{ab} by hashing it. If Alice's session state T^{ab}_{n+1}, (which, recall, contains ${a_e}^{n+1}$ and (rk_{ab}, ck_{ab}) for M_{n+1}), is leaked, then M_n remains secret (forward secrecy). Obviously, to take advantage of this property in case of compromise, the keys (rk_{ab}, ck_{ab}) for M_n must have been appropriately deleted, which is delicate when messages are received out-of-order: if $M_{n_1}, \ldots, M_{n_k}$ $(n_1 < \ldots < n_k)$ have been received, the receiver should keep the chaining key ck_{ab} for M_{n_k+1} and the encryption keys k_{enc} for the messages M_i not received yet with $i < n_k$. If T^{ab}_n is leaked, then M_{n+1} is not secret, so no future secrecy is obtained.

- **Message-Response Pattern.** In this scenario, Alice sends Bob a single message M_n where $n > 1$ and receives a response M_{n+1} before sending M_{n+2}. Upon receiving M_{n+1}, Alice will be able to derive a fresh $k_{shared} = g^{{a_e}^{n+2}{b_e}^{n+1}}$. As a result, if T^{ab}_{n+2} is leaked, then M_n remains secret (forward secrecy) and if T^{ab}_n is leaked after M_{n+1} is received, then M_{n+2} remains secret (future secrecy).

Message Authenticity. Signal Protocol relies on a Trust-on-First-Use (TOFU) authentication model: Alice assumes that Bob's advertised identity key is authenticated and untampered with and employs it as such until an event causes the trust of the key to be put in question, such as a sudden identity key change or an out of band verification failure. We model TOFU by embedding Alice and Bob's identity keys into each other's initial states. We are then free to model for message authenticity: informally, if B receives a message M from A, we want A to have sent M to B. In ProVerif, we can specify two events: $\mathsf{Send}(A, B, M)$, which means that A sends M to B and $\mathsf{Recv}(A, B, M)$, which means that B receives M from A. We can then formalize the correspondence

$$\mathsf{event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{event}(\mathsf{Send}(A, B, M)) \tag{1.2}$$

which checks if for all $\mathsf{Recv}(A, B, M)$ events, it must be the case that a $\mathsf{Send}(A, B, M)$ event has also been executed.

ProVerif succeeds in proving correspondence (1.2) using public keys A and B. While this implies the desired property when the relation between the public keys and the identity of the principals is bijective, a limitation of this approach is that the identities of the principals are only expressed in terms of keys and not as a more personally-linked element, such as for example a phone number. Therefore, we cannot formally express stronger identity binding as part of the protocol model. This point leads to the Unknown Key Share Attack first reported for Signal Protocol Version 2 [85]: if an adversary can register the public keys $(g^{b_{3dh}}, g^{b_{sig}})$ of B as public keys of C and A sends a message to C, then C can forward this message to B and B will accept it as coming from A, since B and C have the same public keys.

No Replays. This property is similar to message authenticity, but uses an injective correspondence instead, which means that each execution of $\mathsf{Recv}(A, B, M)$ corresponds to a distinct execution of $\mathsf{Send}(A, B, M)$:

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A, B, M))$$

When a optional one-time pre-key is involved in the initial session handshake, ProVerif shows that the injective correspondence holds for the first message in the conversation. However, when this optional one-time pre-key is not used, a replay attack is detected. Signal Protocol Version 3 will accept a Diffie-Hellman handshake that only employs identity keys and signed pre-keys, both of which are allowed to be reused across sessions. This reuse is what makes a replay attack possible. We propose a fix for this issue by having clients keep a cache of the ephemeral keys used by the sender of received messages, associated with that sender's identity key. We are able to expand our event queries in ProVerif to account for this fix by showing the non-injective correspondence of the Send and Recv events with added ephemeral keys. Coupled with a caching of ephemeral keys, we can ensure that the Recv event is only executed once per ephemeral key. Hence, the injective correspondence is implied by the non-injective correspondence.

Key Compromise Impersonation (KCI). We present a novel key compromise impersonation attack: to detect KCI, we consider a scenario in which Alice or Bob's keys are compromised and test again for authenticity of messages received by the compromised principal. When Alice or Bob's long-term secret key is compromised, ProVerif shows that message authenticity still holds. However, when Bob's signed pre-key is also compromised, ProVerif finds an attack against message authenticity. This is a novel key compromise impersonation attack: when the adversary has Bob's signed pre-key s, he can choose x and x' and compute the session keys using Alice's and Bob' public keys $(g^{a_{3dh}}, g^{a_{sig}})$ and $(g^{b_{3dh}}, g^{b_{sig}})$ and Bob's one time pre-key g^o and send his own message in Alice's name. This message is accepted by Bob as if it came from Alice: the event $\mathsf{Recv}(A, B, M)$ is executed without having executed $\mathsf{Send}(A, B, M)$.

1.3.3 Other Protocols: OTR

As a preliminary result, we also analyzed Off-the-Record (OTR) Messaging Version 2, which has been shown to include various vulnerabilities discovered under finite-state analysis by Bonneau and Morrison [7]. OTR [6] uses a SIGMA-based key exchange [94] and a ratcheting scheme in order to achieve confidentiality, authentication, perfect forward secrecy and deniability. During model verification, known vulnerabilities in OTR Version 2 were automatically detected:

Version Rollback Attack. Since the communication of which OTR versions are supported by a client was performed before authentication occurred, it is possible to maliciously force users to adopt an outdated version of OTR.

Unknown Key-Share Attack. Since OTR's authenticated key exchange lacks sufficient information identifying Alice's intended recipient, Mallory can forward handshake messages in order to successfully trick Alice into establishing a session with Bob, whereas Alice intended to establish a session with Mallory.

Message Integrity Attack. If Alice receives a message from Bob where Bob uses new ratchet keys, she will publish her MAC keys for her previous message. This is an intentional feature meant to provide deniability once both parties are certain that a certain step of ratchet keys are obsolete. However, Mallory can simply block this outgoing message, use it to learn the MAC key for Alice's previous message and commit a forgery under those keys. From Bob's perspective, it would appear that he has received a delayed but nevertheless valid message from Alice.

Integrating Symbolic Verification into the Development Cycle. Human-readability of the automatically compiled ProVerif model is key to our verification methodology.

Goals	Parties	Running Time
Forward Secrecy	A, B, M	3 min. 58 sec.
Forward Secrecy	A, M	7 min. 04 sec.
KCI	A, B, M	3 min. 15 sec.
Others	A, B, M	4 min. 15 sec.
Others	A, M	3 min. 35 sec.

Figure 1.4: Verification times for SP CryptoVerif models, without anti-replay countermeasure. The runtimes with the anti-replay countermeasure are of the same order of magnitude. Tested using CryptoVerif 1.24.

In the case of a query failure, users can opt to modify their implementation and recompile into a new model, or they can immediately modify the model itself and re-test for security queries within reasonable model verification times. For example, if an implementer wants to test the robustness of a passing forward secrecy query, they can disable the signature verification of signed pre-keys by changing a single line in the model, causing the client to accept any pre-key signature.

1.4 Cryptographic Proofs with CryptoVerif

To complement the results obtained in the symbolic model using ProVerif, we use the tool CryptoVerif [53] in order to obtain security proofs in the computational model. This model is much more realistic: messages are bitstrings; cryptographic primitives are functions from bitstrings to bitstrings; the adversary is a probabilistic Turing machine. CryptoVerif generates proofs by sequences of games [59, 60], like those written manually by cryptographers, automatically or with guidance of the user.

The computational model is more realistic, but it also makes it more difficult to mechanize proofs. For this reason, CryptoVerif is less flexible and more difficult to use than ProVerif and our results in the computational model are more limited. We model only one message of the protocol (in addition to the pre-keys), so we do not prove properties of the ratcheting algorithm. Considering several data messages exceeds the current capabilities of CryptoVerif—the games become too big.

1.4.1 Assumptions

We make the following assumptions on the cryptographic primitives:

- The elliptic curve Curve25519 satisfies the gap Diffie-Hellman (GDH) assumption [95]. This assumption means that given g, g^a and g^b for random a, b, the adversary has a negligible probability to compute g^{ab} (computational Diffie-Hellman assumption), even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$ and $Z = G^{xy}$. When we consider sessions between a participant and himself, we need the square gap Diffie-Hellman variant, which additionally says that given g and g^a for random a, the adversary has a negligible probability to compute g^{a^2}. This assumption is equivalent to the GDH assumption when the group has prime order [96], which is true for Ec25519 [97]. We also added that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$

implies $y = y'$, which hold when the considered Diffie-Hellman group is of prime order.

- Ed25519 signatures, used for signing pre-keys, are unforgeable under chosen-message attacks (UF-CMA) [98].
- The functions

$$x_1, x_2, x_3, x_4 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3 \| x_4, c_1, c_2)$$
$$x_1, x_2, x_3 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3, c_1, c_2)$$
$$x, y \mapsto \mathsf{HKDF}(x, y, c_2)$$
$$x \mapsto \mathsf{HKDF}(x, c_1, c_4)$$

  are independent random oracles, where x, y, x_1, x_2, x_3, x_4 and c_1 are 256-bit long. We further justify this assumption in §1.4.2: there, we show that these functions are indifferentiable [99] from independent random oracles, assuming that the compression function underlying SHA256 is a random oracle. (The considered HKDF function [92] is defined from HMAC-SHA256, which is itself defined from SHA256.)
- HMAC-SHA256 is a pseudo-random function (PRF) [100]. This assumption is used for $\mathsf{HMAC}(ck_{ab}, \cdot)$ and $\mathsf{HMAC}(ck_{ba}, \cdot)$.
- The encryption scheme ENC, which is AES-GCM, is a secure authenticated encryption with associated data (AEAD). More precisely, it is indistinguishable under chosen plaintext attacks (IND-CPA) and satisfies ciphertext integrity (INT-CTXT) [101, 102].

CryptoVerif provides a library that predefines the most common cryptographic assumptions, so that the user does not have to write them for each protocol. In our work, we had to adapt these predefined assumptions to our specific needs: the GDH assumption is predefined, but the square GDH variant is not; unary random oracles are predefined, but we also needed binary, ternary and 4-ary ones; predefined PRFs, SUF-CMA MACs and IND-CPA encryption schemes use a key generation function, while in our schemes the key is a plain random bitstring, without a key generation function. Adapting the definition of primitives did not present any major difficulty. As previously mentioned, we made one minor modification to the original Signal Protocol, for it to be provable in the computational model: we use different keys for the elliptic curve Diffie-Hellman and elliptic curve signatures. It is well-known that using the same keys for several cryptographic primitives is undesirable, as proving security requires a joint security assumption on the two primitives in this case. Therefore, we assume each protocol participant to have two key pairs, one for Diffie-Hellman and one for signatures. This problem remains undetected in a symbolic analysis of the protocol.

1.4.2 Indifferentiability of HKDF

We start from the assumption that the compression function underlying SHA256 is a random oracle and show that the functions

$$x_1, x_2, x_3, x_4 \mapsto \mathsf{HKDF}(c_0 \| x_1 \| x_2 \| x_3 \| x_4, c_1, c_2)$$
$$x_1, x_2, x_3 \mapsto \mathsf{HKDF}(c_0 \| x_1 \| x_2 \| x_3, c_1, c_2)$$
$$x, y \mapsto \mathsf{HKDF}(x, y, c_2)$$
$$x \mapsto \mathsf{HKDF}(x, c_1, c_4)$$

where $c_1 = 0$ (256-bits), c_2 = "WhisperRatchet", and c_4 = "WhisperMessageKeys", can be considered as independent random oracles; more formally, we show that they are indifferentiable [99] from independent random oracles.

Definition 3 (Indifferentiability)**.** *A function F with oracle access to a random oracle H is (t_D, t_S, q, ϵ)-indifferentiable from a random oracle H' if there exists a simulator S such that for any distinguisher D*

$$|\Pr[D^{F,H} = 1] - \Pr[D^{H',S} = 1]| \leq \epsilon$$

The simulator S has oracle access to H' and runs in time t_S. The distinguisher D runs in time t_D and makes at most q queries.

In the game $G_0 = D^{F,H}$, the distinguisher interacts with the real function F and the random oracle H from which F is defined. In the game $G_1 = D^{H',S}$, the distinguisher interacts with a random oracle H' instead of F, and with a simulator S, which simulates the behavior of the random oracle H using calls to H'. Indifferentiability means that these two games are indistinguishable.

Theorem 4.4 in [103] shows that HMAC-SHA256 is then indifferentiable from a random oracle, provided the MAC keys are less than the block size of the hash function minus one, which is true here: the block size of SHA256 is 512 bits and the MAC keys are 256-bit long.

In the four calls to HKDF that we consider, we did not make explicit the length of the returned key material. This length is at most 512 bits, so we can consider that HKDF is defined by truncation of the following function HKDF_2 to the desired length:

$$\begin{aligned}
&\mathsf{HKDF}_2(salt, key, info) = K_1 \| K_2 \text{ where} \\
&\quad prk = \mathsf{HMAC\text{-}H}^{salt}((), key) \\
&\quad K_1 = \mathsf{HMAC\text{-}H}^{prk}((), info \| 0x00) \\
&\quad K_2 = \mathsf{HMAC\text{-}H}^{prk}((), K_1 \| info \| 0x01)
\end{aligned} \tag{1.3}$$

and $\mathsf{HKDF}(key, salt, info)$ is a truncation of $\mathsf{HKDF}_2(salt, key, info)$. Much like for HMAC in [103], this function is not indifferentiable from a random oracle in general. Intuitively, the problem comes from a confusion between the first and the second (or third) call to $\mathsf{HMAC\text{-}H}^{\cdot}()$ which makes it possible to generate prk by calling HKDF_2 rather than $\mathsf{HMAC\text{-}H}^{\cdot}()$ In more detail, let

$$\begin{aligned}
&prk \| _ = \mathsf{HKDF}_2(s, k, i) \\
&salt = \mathsf{HMAC\text{-}H}^{s}((), k) \\
&x = \mathsf{HMAC\text{-}H}^{prk}((), info \| 0x00) \\
&x' \| _ = \mathsf{HKDF}_2(salt, i \| 0x00, info)
\end{aligned}$$

where the notation $x_1 \| x_2 = \mathsf{HKDF}_2(s, k, i)$ denotes that x_1 consists of the first 256 bits of $\mathsf{HKDF}_2(s, k, i)$ and x_2 its last 256 bits.

When HKDF_2 is defined from $\mathsf{HMAC\text{-}H}^{a}()$s above, we have $prk = \mathsf{HMAC\text{-}H}^{prk}()', i \| 0x00)$ where $prk' = \mathsf{HMAC\text{-}H}^{s}((), k) = salt$, so $prk = \mathsf{HMAC\text{-}H}^{salt}((), i \| 0x00)$. Hence, $x' = \mathsf{HMAC\text{-}H}^{prk}((), info \| 0) = x$. However, when HKDF_2 is a random oracle and $\mathsf{HMAC\text{-}H}^{i}()$s defined from HKDF_2, the simulator that computes $\mathsf{HMAC\text{-}H}^{s}()$ees what seems to be two unrelated calls to $\mathsf{HMAC\text{-}H}^{\cdot}()$ (It is unable to see that prk is in fact related to the previous call $salt = \mathsf{HMAC\text{-}H}^{s}((), k)$: we have $prk \| \_ = \mathsf{HKDF}_2(s, k, i)$ but the simulator does not know which value of i it should use.) Therefore, the simulator can only return fresh random values for $salt$ and x and $x \neq x'$ in general.

We can however recover the indifferentiability of HKDF_2 under the additional assumption that the three calls to HMAC-H^u()se disjoint domains. Let $\mathcal{S}$, $\mathcal{K}$ and $\mathcal{I}$ be the sets of possible values of *salt*, *key* and *info* respectively and $\mathcal{M}$ the set of 256-bit bitstrings, output of HMAC-H$^\cdot$()

Lemma 1. *If $\mathcal{K} \cap (\mathcal{I}\|0x00 \cup \mathcal{M}\|\mathcal{I}\|0x01) = \emptyset$ and $\mathcal{S}$ consists of bitstrings of 256 bits, then HKDF_2 with domain $\mathcal{S} \times \mathcal{K} \times \mathcal{I}$ is (t_D, t_S, q, ϵ)-indifferentiable from a random oracle, where $\epsilon = \mathcal{O}(q^2/|\mathcal{M}|)$ and $t_S = \mathcal{O}(q^2)$ and $\mathcal{O}$ just hides small constants.*

Proof. Consider

- the game G_0 in which HMAC-H^i()s a random oracle and HKDF_2 is defined from HMAC-H^b()y (1.3) and
- the game G_1 in which HKDF_2 is a random oracle and HMAC-H^i()s defined as follows.

  Let L be a list of pairs $((k, m), r)$ such that r is the result of a previous call to HMAC-H^k$((), m)$. The list L is initially empty.

  HMAC-H^k$((), m)$ =

  1. if $((k, m), r) \in L$ for some r, then return r, else
  2. if $((k_0, m_0), k) \in L$ for some $k_0 \in \mathcal{S}$ and $m_0 \in \mathcal{K}$ and $m = info\|0x00$ for some $info \in \mathcal{I}$, then let $r\|\_ = \mathsf{HKDF}_2(k_0, m_0, info)$, else
  3. if $((k_0, m_0), k) \in L$ for some $k_0 \in \mathcal{S}$ and $m_0 \in \mathcal{K}$ and $m = k_1\|info\|0x01$ for some $k_1 \in \mathcal{M}$ and $info \in \mathcal{I}$, then let $k_1'\|k_2' = \mathsf{HKDF}_2(k_0, m_0, info)$; if $k_1' = k_1$, then $r = k_2'$;
  4. otherwise, let r be a fresh random element of $\mathcal{M}$;
  5. add $((k, m), r)$ to L;
  6. return r.

  We name *direct* oracle calls to HKDF_2 or HMAC-H^c()alls that are done directly by the distinguisher and *indirect* oracle calls the calls to HMAC-H^d()one from inside HKDF_2 (in G_0) and the calls to HKDF_2 done from inside HMAC-H$^($()in G_1).

Let us show that these two games are indistinguishable as long as, in G_0,

H1. HMAC-H^n()ever returns the same result for different arguments, HYP2.1

H2. no fresh result of HMAC-H^i()s equal to the first argument of a previous call to HMAC-H$^{i'}$() HYP2.2

H3. the distinguisher never calls HMAC-H^k$((), m)$ where k = HMAC-H^{salt}$((), key)$ has been called from inside HKDF_2 but not directly by the distinguisher,

H4. and HMAC-H^{prk}$((), info\|0x00)$ never returns a fresh k_1 such that HMAC-H^{prk}$((), k_1\|info\|0x01)$ has been called (directly or indirectly) before, HYP2.4

and in G_1,

H5. there are no two elements $((k, m), r)$ and $((k', m'), r)$ in L with $(k, m) \neq (k', m')$,

H6. no returned r is equal to a previous k,

H7. if the distinguisher calls HMAC-H^{prk}$((), k_1\|info\|0x01)$ with $((salt, key), prk) \in L$ and $k_1\|\_ = \mathsf{HKDF}_2(salt, key, info)$, then $\mathsf{HKDF}_2(salt, key, info)$ has been called (directly or indirectly at step 2) before the call to HMAC-H^{prk}$((), k_1\|info\|0x01)$.

We have the following invariant:

P1. Given $salt, key$, there is at most one prk such that $((salt, key), prk) \in L$.

Indeed, when L contains such an element, calls to HMAC-H$^{salt}((), key)$ immediately return prk at step 1 and never add another element $((salt, key), prk')$ to L.

Case 1. Suppose the distinguisher makes a direct oracle call to HKDF$_2$ or HMAC-H^w()ith the same arguments as a previous direct call to the same oracle. Both G_0 and G_1 return the same result as in the previous call.

Case 2. Suppose the distinguisher makes a direct call to HMAC-H$^k((), m)$ with arguments that do not occur in a previous direct call to HMAC-H$^{\cdot}$()

Case 2.a) In G_0, this HMAC-H^c()all has already been done as HMAC-H$^{salt}((), key)$ from inside HKDF$_2$. In G_0, the result is prk = HMAC-H$^{salt}((), key)$, which is independent from previously returned values, so it looks like a fresh random value to the distinguisher. In G_1, we cannot have $m = info\|0x00$ nor $m = k_1\|info\|0x01$ because $m = key \in \mathcal{K}$ which is disjoint from $\mathcal{I}\|0x00$ and from $\mathcal{M}\|\mathcal{I}\|0x01$, so HMAC-H^r()eturns a fresh random value.

Case 2.b) In G_0, this HMAC-H^c()all has already been done as HMAC-H$^{prk}((), info\|0x00)$ from inside HKDF$_2(salt, key, info)$. Hence HMAC-H$^k((), m)$ = HMAC-H$^{prk}((), info\|0x00)$ is the first 256 bits of HKDF$_2(salt, key, info)$ and prk = HMAC-H$^{salt}((), key)$. Since by H3, the distinguisher never calls HMAC-H$^k((), m)$ where k = HMAC-H$^{salt}((), key)$ has been called from inside HKDF$_2$ but not directly by the distinguisher, HMAC-H$^{salt}((), key)$ has been called directly by the distinguisher. In G_1, since HMAC-H$^{salt}((), key)$ has been called, $((salt, key), prk) \in L$, so HMAC-H$^k((), m)$ = HMAC-H$^{prk}((), info\|0x00)$ returns the first 256 bits of HKDF$_2(salt, key, info)$ (step 2), as in G_0.

Case 2.c) In G_0, this HMAC-H^c()all has already been done as HMAC-H$^{prk}((), K_1\|info\|0x01)$ from inside HKDF$_2(salt, key, info)$. Hence HMAC-H$^k((), m)$ = HMAC-H$^{prk}((), K_1\|info\|0x01)$ is the last 256 bits of HKDF$_2(salt, key, info)$, prk = HMAC-H$^{salt}((), key)$ and K_1 = HMAC-H$^{prk}((), info\|0x00)$ is the first 256 bits of HKDF$_2(salt, key, info)$. As above, HMAC-H$^{salt}((), key)$ has been called directly by the distinguisher. In G_1, since HMAC-H$^{salt}((), key)$ has been called, $((salt, key), prk) \in L$, so, since K_1 is the first 256 bits of HKDF$_2(salt, key, info)$, HMAC-H$^k((), m)$ = HMAC-H$^{prk}((), K_1\|info\|0x01)$ returns the last 256 bits of HKDF$_2(salt, key, info)$ (step 3), as in G_0.

Case 2.d) In G_0, this HMAC-H^c()all has never been done, directly or indirectly. Hence, HMAC-H^r()eturns a fresh random value. In G_1, if $((salt, key), k) \in L$, then HMAC-H^m()ay return the first or last 256 bits of HKDF$_2(salt, key, info)$. However, since HMAC-H$^k((), m)$ has not been called from HKDF$_2$ in G_0, HKDF$_2(salt, key, info)$ has not been called directly by the distinguisher, so the result of HMAC-H^a()lways looks like a fresh random value to the distinguisher.

Case 3. Suppose the distinguisher makes a direct call to HKDF$_2(salt, key, info)$ with arguments that do not occur in a previous direct call to HKDF$_2$.

Case 3.a) In G_1, this call to HKDF$_2$ has already been done from HMAC-H$^{\cdot}$() Hence $((salt, key), prk) \in L$ and HMAC-H$^{prk}((), info\|0x00)$ or HMAC-H$^{prk}((), k_1\|info\|0x01)$ has been called. Since $((salt, key), prk) \in L$, HMAC-H$^{salt}((), key)$ has been called before the call to HMAC-H$^{prk}((), info\|0x00)$ or HMAC-H$^{prk}((), k_1\|info\|0x01)$ and it has returned prk.

Case 3.a) i) Suppose that HMAC-H$^{prk}((), info\|0x00)$ has been called and it returned k_1' and HMAC-H$^{prk}((), k_1'\|info\|0x01)$ has not been called. By step 2 of the definition of HMAC-H^i()n G_1, since by H5, the only element of L of the form $(\_, prk)$ is $((salt, key), prk)$, of a previous call to HKDF$_2(salt, key, info)$. The current call to HKDF$_2(salt, key, info)$ returns the same result, to its first 256 bits are HMAC-H$^{prk}((), info\|0x00)$. Its last 256 bits

are independent from returned random values. Indeed, if a call to HMAC-H^r()eturns the last 256 bits of HKDF$_2(salt, key, info)$, then this call occurs in step 3 of HMAC-H^a()nd it is HMAC-H$^{prk}((\,)', m)$ with $((salt, key), prk') \in L$, $m = k_1''\|info\|0x01$ and k_1'' is the first 256 bits of HKDF$_2(salt, key, info)$. By P1, $prk' = prk$. We have $k_1' = k_1''$, so HMAC-H$^{prk}((\,)', m)$ is HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$. But HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$ has not been called by hypothesis, so no previous call to HMAC-H^r()eturns the last 256 bits of HKDF$_2(salt, key, info)$. So the last 256 bits of HKDF$_2(salt, key, info)$ look like a fresh random value.

In G_0, the first 256 bits of HKDF$_2(salt, key, info)$ are also HMAC-H$^{prk}((\,), info\|0x00)$, where prk = HMAC-H$^{salt}((\,), key)$. Furthermore, the last 256 bits of HKDF$_2(salt, key, info)$ are independent of previously returned values. Indeed, HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$ has not been called directly. Furthermore, it has not been called from previous calls to HKDF$_2$, because, if HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$ had been called from HKDF$_2(salt', key', info')$, then by $\mathcal{K} \cap (\mathcal{I}\|0x00 \cup \mathcal{M}\|\mathcal{I}\|0x01) = \emptyset$, this call would be the last of the three calls to HMAC-H^i()n HKDF$_2(salt', key', info')$, prk = HMAC-H$^{salt}((\,)', key')$ and $info' = info$. Since by H1, HMAC-H^n()ever returns the same result for different arguments, $salt' = salt$ and $key' = key$, contradicting that HKDF$_2(salt, key, info)$ has not been called before. Therefore, the last 256 bits of HKDF$_2(salt, key, info)$ look like a fresh random value.

Case 3.a) ii) Suppose that HMAC-H$^{prk}((\,), info\|0x00)$ has been called and it returned k_1' and HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$ has been called. By definition of HMAC-H^i()n G_1, HMAC-H$^{prk}((\,), info\|0x00)$ is the first 256 bits of HKDF$_2(salt, key, info)$ (step 2) and HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$ is its last 256 bits (step 3), so HKDF$_2(salt, key, info) = k_1'\|k_2'$ where k_1' = HMAC-H$^{prk}((\,), info\|0x00)$ and k_2' = HMAC-H$^{prk}((\,), k_1'\|info\|0x01)$. In G_0, we have the same property by definition of HKDF$_2$.

Case 3.a) iii) Otherwise, HMAC-H$^{prk}((\,), info\|0x00)$ has not been called.

If HKDF$_2(salt, key, info)$ had been called from step 2 of HMAC-H$^{\cdot}$() then we would have called HMAC-H$^{prk}((\,)', m)$ with $((salt, key), prk') \in L$ and $m = info\|0x00$. Furthermore, by P1, $prk' = prk$, so we would have called HMAC-H$^{prk}((\,), info\|0x00)$. Contradiction. So HKDF$_2(salt, key, info)$ has not been called from step 2 of HMAC-H$^{\cdot}$()

Therefore, HKDF$_2(salt, key, info)$ has been called from step 3 of HMAC-H$^{\cdot}$() If HKDF$_2(salt, key, info)$ had been called at step 3 of HMAC-H^a()nd its last 256 bits were returned, then the distinguisher would have called HMAC-H$^{prk}((\,)', k_1'\|info\|0x01)$ with $((salt, key), prk') \in L$ and $k_1'\|\_$ = HKDF$_2(salt, key, info)$. By H7, HKDF$_2(salt, key, info)$ would have been called before, either directly (excluded by hypothesis) or indirectly at step 2. Then the distinguisher would have called HMAC-H$^{prk}((\,)'', info\|0x00)$ with $((salt, key), prk') \in L$, so by P1, $prk'' = prk$, so this is excluded by hypothesis. Therefore, the last 256 bits of HKDF$_2(salt, key, info)$ were not returned at step 3.

We can then conclude that in G_1, the value of HKDF$_2(salt, key, info)$ is independent from previously returned values, so it looks like a fresh random value.

In G_0, HMAC-H$^{salt}((\,), key)$ has been called directly and returned prk, HMAC-H$^{prk}((\,), info\|0x00)$ has not been called directly. If a previous call to HKDF$_2(salt', key', info')$ called HMAC-H$^{prk}((\,), info\|0x00)$, then we would have $info' = info$ and prk = HMAC-H$^{salt}((\,)', key')$. By H1, this would imply $salt' = salt$ and $key' = key$, so HKDF$_2(salt, key, info)$ would have been called before, which is excluded by hypothesis. Therefore, HMAC-H$^{prk}((\,), info\|0x00)$ has not been called before, directly or indirectly. By H4, HMAC-H$^{prk}((\,), k_1\|info\|0x01)$ has not been called before, with k_1 = HMAC-H$^{prk}((\,), info\|0x00)$. Therefore, HMAC-H$^{prk}((\,), info\|0x00)$ and HMAC-H$^{prk}((\,), k_1\|info\|0x01)$ have not been called before, so their result is independent from previously returned values. Hence HKDF$_2(salt, key, info)$ is independent from previously returned values, as in G_1.

Case 3.b) In G_1, this HKDF$_2$ call has never been done, directly or indirectly. Hence

HKDF_2 returns a fresh random value. In G_0, the result is obtained from calls to $\mathsf{HMAC\text{-}H}^{\cdot}()$ The distinguisher has not made these calls to $\mathsf{HMAC\text{-}H}^{d}()$irectly calling $\mathsf{HMAC\text{-}H}^{salt}((), key)$ first, because otherwise the simulator for $\mathsf{HMAC\text{-}H}^{i}()$n G_1 would have called $\mathsf{HKDF}_2(salt, key, info)$. Furthermore, it cannot call $\mathsf{HMAC\text{-}H}^{salt}((), key)$ with result prk after calling $\mathsf{HMAC\text{-}H}^{prk}((), info \| 0x00)$ or $\mathsf{HMAC\text{-}H}^{prk}((), k_1 \| info \| 0x01)$ by H2. So the result of HKDF_2 is independent of the result of direct $\mathsf{HMAC\text{-}H}^{c}()$alls made by the distinguisher. Moreover, other calls to HKDF_2 did not generate the same last two calls to $\mathsf{HMAC\text{-}H}^{\cdot}()$ because by H1, the first call to $\mathsf{HMAC\text{-}H}^{r}()$ $\mathsf{HMAC\text{-}H}^{salt}((), key)$, never returns the same result for different arguments. random value to the distinguisher.

The previous proof shows that the games G_0 and G_1 are indistinguishable assuming the hypotheses H1–H7 hold. Let us bound the probability that they do not hold. Suppose that there are at most q (direct or indirect) queries to $\mathsf{HMAC\text{-}H}^{\cdot}()$

- The probability that H1 does not hold is at most the probability that among q random values in $\mathcal{M}$, two of them collide, so it is at most $q^2/|\mathcal{M}|$.
- The probability that H2 does not hold is at most the probability that among q random values in $\mathcal{M}$, one of them is equal to one among the q first arguments of $\mathsf{HMAC\text{-}H}^{q}()$ueries, so it is also at most $q^2/|\mathcal{M}|$.
- When H3 does not hold, the distinguisher calls $\mathsf{HMAC\text{-}H}^{k}((), m)$ for a value k that happens to be equal to $\mathsf{HMAC\text{-}H}^{salt}((), key)$, which is independent of the values the distinguisher has seen, since $\mathsf{HMAC\text{-}H}^{salt}((), key)$ has not been called directly by the distinguisher. There are at most q values $\mathsf{HMAC\text{-}H}^{salt}((), key)$ and the distinguisher has q attempts, so the probability that H3 does not hold is at most $q^2/|\mathcal{M}|$.
- Similarly, when H4 does not hold, the fresh random value $\mathsf{HMAC\text{-}H}^{prk}((), info \| 0x00)$ collides with a previously fixed k_1. There are at most q values $\mathsf{HMAC\text{-}H}^{prk}((), info \| 0x00)$ and at most q values k_1, so the probability that H4 does not hold is at most $q^2/|\mathcal{M}|$.
- Let us show that, if the random values r chosen at step 4 are all distinct and distinct from first and second halves of HKDF_2 results used in $\mathsf{HMAC\text{-}H}^{\cdot}()$ then H5 holds. The proof is by induction on the sequence of calls of $\mathsf{HMAC\text{-}H}^{\cdot}()$ If $((k, m), r)$ is added to L and r comes from a result of HKDF_2 at step 2 or 3, then k determines k_0, m_0 uniquely by induction hypothesis and m determines $info$ as well as which half of the result of HKDF_2 is r, hence r is uniquely determined from k, m and distinct from elements chosen at step 4 by hypothesis. If $((k, m), r)$ is added to L and r is chosen at step 4, then r is always distinct from elements already in L by hypothesis. This concludes the proof of our claim.

  From this claim, we can easily see that the probability that H5 does not hold is at most $q^2/|\mathcal{M}|$.
- When H7 does not hold, the distinguisher calls $\mathsf{HMAC\text{-}H}^{prk}((), k_1 \| info \| 0x01)$ and k_1 happens to be equal to the first 256 bits of $\mathsf{HKDF}_2(salt, key, info)$ which is independent from values returned to the distinguisher. So the probability that H7 does not hold is at most $q^2/|\mathcal{M}|$.

Hence, the probability that the distinguisher distinguishes G_0 from G_1 is at most $6q^2/|\mathcal{M}|$. □

The hypothesis of Lemma 1 is satisfied in our case because $\mathcal{K}$ consists of bitstrings of length 256 bits = 32 bytes or $3 \times 32 = 96$ bytes, $\mathcal{I} \| 0x00$ consists of bitstrings of length at most 31 bytes and $\mathcal{M} \| \mathcal{I} \| 0x01$ consists of bitstrings of length between 33 and 63 bytes.

Lemma 2. *If H is a random oracle, then the functions $H_1,\ldots,H_n$ defined as H on disjoint subsets $D_1,\ldots,D_n$ of the domain D of H are $(t_D,t_S,q,0)$-indifferentiable from independent random oracles, where $t_S = \mathcal{O}(q)$ assuming one can determine in constant time to which subset D_i an element belongs.*

Proof. Consider

- the game G_0 in which H is a random oracle and $H_i(x) = H(x)$ for each $x \in D_i$ and $i \leq n$ and
- the game G_1 in which $H_1,\ldots H_n$ are independent random oracles defined on $D_1,\ldots,D_n$ respectively and $H(x) = H_i(x)$ if $x \in D_i$ for some $i \leq n$ and $H(x) = H_0(x)$ otherwise, where H_0 is a random oracle of domain $D \setminus (D_1 \cup \cdots \cup D_n)$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

By combining Lemmas 1 and 2, we obtain that

$$\begin{aligned}
&x_1,x_2,x_3,x_4 \mapsto \mathsf{HKDF}_2(0,c_0\|x_1\|x_2\|x_3\|x_4,c_2)\\
&x_1,x_2,x_3 \mapsto \mathsf{HKDF}_2(0,c_0\|x_1\|x_2\|x_3,c_2)\\
&x,y \mapsto \mathsf{HKDF}_2(x,y,c_2)\\
&x \mapsto \mathsf{HKDF}_2(0,x,c_4)
\end{aligned}$$

are indifferentiable from independent random oracles. The domains are disjoint because different constants c_2 and c_4 are used and furthermore, the three cases that use c_2 differ by the length of their second argument ($4 \times 256 = 1024$ bits plus the length of c_0 for $x_1,x_2,x_3,x_4 \mapsto \mathsf{HKDF}_2(0,c_0\|x_1\|x_2\|x_3\|x_4,c_2)$, $3 \times 256 = 768$ bits plus the length of c_0 for $x_1,x_2,x_3 \mapsto \mathsf{HKDF}_2(0,c_0\|x_1\|x_2\|x_3,c_2)$ and 256 bits for $x,y \mapsto \mathsf{HKDF}_2(x,y,c_2)$).

Lemma 3. *If H is a random oracle that returns bitstrings of length l, then the truncation of H to length $l' < l$ is $(t_D,t_S,q,0)$-indifferentiable from a random oracle, where $t_S = \mathcal{O}(q)$.*

Proof. Consider

- the game G_0 in which H is a random oracle and $H'(x)$ is $H(x)$ truncated to length l' and
- the game G_1 in which H' is a random oracle that returns bitstrings of length l' and $H(x) = H'(x)\|H''(x)$ where H'' is a random oracle that returns bitstrings of length $l - l'$.

It is easy to see that these two games are perfectly indistinguishable, which proves indifferentiability. □

By combining Lemma 3 with the previous results, we conclude that

$$\begin{aligned}
&x_1,x_2,x_3,x_4 \mapsto \mathsf{HKDF}(c_0\|x_1\|x_2\|x_3\|x_4,c_1,c_2)\\
&x_1,x_2,x_3 \mapsto \mathsf{HKDF}(c_0\|x_1\|x_2\|x_3,c_1,c_2)\\
&x,y \mapsto \mathsf{HKDF}(x,y,c_2)\\
&x \mapsto \mathsf{HKDF}(x,c_1,c_4)
\end{aligned}$$

are indifferentiable from independent random oracles.

1.4.3 Protocol Model

We model SP as a process in the input language of CryptoVerif, which is similar to the one of ProVerif. We consider simultaneously the protocol of Figure 1.2 and the version without the optional one-time pre-key b_o. As mentioned above, we consider only one message in each session. Our threat model includes an untrusted network, malicious principals and long-term key compromise. It does not include session state compromise, which is less useful with a single message.

To make verification easier for CryptoVerif, we specify a lower-level interface. We consider two honest principals Alice and Bob and define separate processes for Alice interacting with Bob, with herself, or with a malicious participant, Bob interacting with Alice and Bob interacting with himself or a malicious participant, as well as similar processes with the roles of Alice and Bob reversed. The adversary can then implement the high-level interface of § 4.2 from this lower-level interface: the adversary is supposed to implement the malicious principals (including defining keys for them) and to call the low-level interface processes to run sessions that involve the honest principals Alice and Bob.

We make two separate proofs: In the first one, we prove the security properties for sessions in which Bob generates pre-keys and runs the protocol with Alice. (Other protocol sessions exist in parallel as described above; we do not prove security properties for them. For sessions for which we do not prove security properties, we give to the adversary the ephemeral a'_e and the key rk_{ba} or rk_{ba} and let the adversary encrypt and MAC the message himself, to reduce the size of our processes.) In the second one, we prove the security properties for sessions in which Alice generates pre-keys and runs the protocol with herself. Bob is included in the adversary in this proof. The security for sessions in which Alice generates pre-keys and runs the protocol with Bob follows from the first proof by symmetry. The security for sessions in which Bob generates pre-keys and runs the protocol with himself follows from the second proof. The other sessions do not satisfy security properties since they involve the adversary. (They must still be modeled, as they could break the protocol if it were badly designed.) Therefore, these two proofs provide all desired security properties.

1.4.4 Security Goals

We consider the following security goals from § 4.2:

Message Authenticity, No Replays and Key Compromise Impersonation (KCI). These properties are modeled by correspondences as in ProVerif (§ 1.3.2). For key compromise impersonation, we consider the compromise of the long-term Diffie-Hellman and signature keys of Bob and prove again message authenticity. We do not consider the compromise of the signed pre-key since we already know from the symbolic analysis that there is an attack in this case.

Computational Indistinguishability. If A randomly chooses between two messages M_0, M_1 of the same length and sends one of them to B, then the adversary has a negligible probability of guessing which of the two messages was sent. In our model, this is formalized by choosing a random bit $secb \in \{0,1\}$; then A sends message M_b to B and we show that the bit $secb$ remains secret, with the query **secret** $secb$.

Forward Secrecy. This is proved exactly like indistinguishability, but with an additional oracle that allows the adversary to obtain the secret keys of the principals, thus compromising them.

We do not consider future secrecy since we have a single message. We do not consider secrecy since we directly deal with the stronger property of indistinguishability.

1.4.5 Results

CryptoVerif proves message authenticity, absence of key compromise impersonation attacks (when the long-term keys of Bob are compromised), indistinguishability and forward secrecy, but cannot prove absence of replays. This is due to the replay attack mentioned in § 1.3.2. Since this attack appears only when the optional one-time pre-key is omitted, we separate our property into two: we use events $\mathsf{Send}(A,B,M)$ and $\mathsf{Recv}(A,B,M)$ for the protocol with optional pre-key and events $\mathsf{Send3}(A,B,M)$ and $\mathsf{Recv3}(A,B,M)$ for the protocol without optional pre-key. CryptoVerif then proves

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A,B,M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A,B,M))$$
$$\mathsf{event}(\mathsf{Recv3}(A,B,M)) \Longrightarrow \mathsf{event}(\mathsf{Send3}(A,B,M))$$

which proves message authenticity and no replays when the one-time pre-key is present and only message authenticity when it is absent. This is the strongest we can hope for the protocol without anti-replay countermeasure.

With our anti-replay countermeasure (§1.3.2), CryptoVerif can prove the absence of replays, thanks to a recent extension that allows CryptoVerif to take into account the replay cache in the proof of injective correspondences, implemented in CryptoVerif version 1.24. Our CryptoVerif proofs have been obtained with some manual guidance: we indicated the main security assumptions to apply, instructed CryptoVerif to simplify the games or to replace some variables with their values, to make terms such as $m^a = m^b$ appear. The proofs were similar for all properties.

1.5 Conclusion and Related Work

While the ambition of cryptographic protocols found in web applications continues to grow, there remains a large disconnect between implementation and formal guarantees.

Drawing from existing design trends in modern cryptographic web application, we have presented a framework that supports the incremental development of custom cryptographic protocols hand-in-hand with formal security analysis. By leveraging state-of-the-art protocol verification tools and building new tools, we showed how many routine tasks can be automated, allowing the protocol designer to focus on the important task of analyzing her protocol for sophisticated security goals against powerful adversaries.

Unger et al. survey previous work on secure messaging [2]. We discuss three recent closely-related works here. Future secrecy was formalized by Cohn-Gordon et al. as "post-compromise security" [104]. Our symbolic formulation is slightly different since it relies on the definition of protocol phases in ProVerif.

Cryptographic security theorems and potential unknown key-share attacks on TextSecure Version 2 were presented by Frosch et al. [85]. In comparison to that work, our analysis covers a variant of TextSecure Version 3 called SP. Our analysis is also fully mechanized, and we address implementation details. Our CryptoVerif model only covers a single message, but we consider the whole protocol at once, while they prove pieces of the protocol separately. Like we do, they consider that HKDF is a random oracle. We further justify this assumption by an indifferentiability proof.

In parallel with this work, Cohn-Gordon et al. [105] proved, by hand, that the message encryption keys of Signal are secret in the computational model, in a rich compromise scenario, under assumptions similar to ours. Thereby, they provide a detailed proof of the properties of the double ratcheting mechanism. However, they do not model the signatures of the signed pre-keys, and they do not consider key compromise impersonation attacks or replay attacks or other implementation-level details. In contrast to their work, our computational proof is mechanized, but limited to only one message.

After more recent work outlined serious shortcomings in secure group messaging protocols [8], largely due to underspecification, efforts have sufaced to standardize group secure messaging as "Message Layer Security" (MLS) through the IETF, in a fashion similar to TLS. Initially, the foundation for MLS's session management algorithm was a new concept by Cohn-Gordon et al called "Asynchronous Ratcheting Trees" [106] (ART). Since then, competing designs for MLS's core session management logic have been proposed, most notably TreeKEM [107].

Chapter 2

Formal Verification for Transport Layer Security

This work was published in the proceedings for the 38th IEEE Symposium on Security and Privacy, 2017. It was completed with co-authors Karthikeyan Bhargavan and Bruno Blanchet. In this particular work, my role was largely assistive, with Bhargavan and Blanchet leading the way.

In 2014, the TLS working group at the IETF commenced work on TLS 1.3, with the goal of designing a faster protocol inspired by the success of Google's QUIC protocol [108]. Learning from the pitfalls of TLS 1.2, the working group invited the research community to contribute to the design of the protocol and help analyze its security even before the standard is published. A number of researchers, including the authors of this chapter, responded by developing new security models and cryptographic proofs for various draft versions and using their analyses to propose protocol changes. Cryptographic proofs were developed for Draft-5 [65], Draft-9 [66], and Draft-10 [67], which justified the core design of the protocol. A detailed symbolic model in Tamarin was developed for Draft-10 [109]. Other works studied specific aspects of TLS 1.3, such as key confirmation [110], client authentication [111] and downgrade resilience [112].

Some of these analyses also found attacks. The Tamarin analysis [109] uncovered a potential attack on the composition of pre-shared keys and certificate-based authentication and this attack was prevented in Draft-11. A version downgrade attack was found in Draft-12 and its countermeasure in Draft-13 was proved secure [112]. A cross-protocol attack on RSA signatures was described in [113]. Even in this work, we describe two vulnerabilities in 0-RTT client authentication that we discovered and reported, which influenced the subsequent designs of Draft-7 and -13.

After 18 drafts, TLS 1.3 is entering the final phase of standardization. Although many of its design decisions have now been vetted by multiple security analyses, several unanswered questions remain. First, the protocol has continued to evolve rapidly with every draft version, so many of the cryptographic proofs cited above are already obsolete and do not apply to Draft-18. Since many of these are manual proofs, it is not easy to update them and check all the proof steps. Second, none of these symbolic or cryptographic analyses, with the exception of [112], consider the composition of TLS 1.3 with legacy versions like TLS 1.2. Hence, they do not account for attacks like [113] that exploit weak legacy crypto in TLS 1.2 to break the modern cryptographic constructions of TLS 1.3. Third, none of these works addresses TLS 1.3 implementa-

tions. In this work, we seek to cover these gaps with a new comprehensive analysis of TLS 1.3 Draft-18.

2.0.1 Our Contributions

We propose a methodology for developing mechanically verified models of TLS 1.3 alongside a high-assurance reference implementation of the protocol.

We present symbolic protocol models for TLS 1.3 written in ProVerif [48]. They incorporate a novel security model (described in §2.1) that accounts for all recent attacks on TLS, including those relying on weak cryptographic algorithms. In §2.2-2.4, we use ProVerif to evaluate various modes and drafts of TLS 1.3 culminating in the first symbolic analysis of Draft-18 and the first composite analysis of TLS 1.3+1.2. Our analyses uncover known and new vulnerabilities that influenced the final design of Draft-18. Some of the features we study no longer appear in the protocol, but our analysis is still useful for posterity, to warn protocol designers and developers who may be tempted to reintroduce these problematic features in the future.

In §2.5, we develop the first machine-checked cryptographic proof for TLS 1.3 using the verification tool CryptoVerif [53]. Our proof reduces the security of TLS 1.3 Draft-18 to standard cryptographic assumptions over its primitives. In contrast to manual proofs, our CryptoVerif script can be more easily updated from draft-to-draft and as the protocol evolves.

Our ProVerif and CryptoVerif models capture the protocol core of TLS 1.3, but they elide many implementation details such as the protocol API and state machine. To demonstrate that our security results apply to carefully-written implementations of TLS 1.3, we present RefTLS (§5.3) in Chapter 6 of this thesis, the first reference implementation of TLS 1.0-1.3 whose core protocol code has been formally analyzed for security. RefTLS is written in Flow, a statically typed variant of JavaScript and is structured so that all its protocol code is isolated in a single module that can be automatically translated to ProVerif and symbolically analyzed against our rich threat model.

Our models and code are available at:

`https://github.com/inria-prosecco/reftls`

2.1 A Security Model for TLS

Figure 2.1 depicts the progression of a typical TLS connection: negotiation, then Authenticated Key Exchange (AKE), then Authenticated Encryption (AE) for application data streams. The server chooses $mode_S$, including the protocol version, the AKE mode and the AE algorithm. In TLS 1.2, the AKE may use RSA, (EC)DHE, PSK, etc. and AE may use AES-CBC MAC-Encode-Encrypt, RC4, or AES-GCM. In TLS 1.3, the AKE may use (EC)-DHE, PSK, or PSK-(EC)DHE and AE may use AES-GCM, AES-CCM, or ChaCha20-Poly1305. The use of one or more of (pk_C,pk_S,psk) in the session depends on the AKE mode.

Since a client and server may support different sets of features, they first *negotiate* a protocol mode that they have in common. In TLS, the client C makes an $offer_C$ and the server chooses its preferred $mode_S$, which includes the protocol version, the key exchange protocol, the authenticated encryption scheme, the Diffie-Hellman group (if applicable) and the signature and hash algorithms.

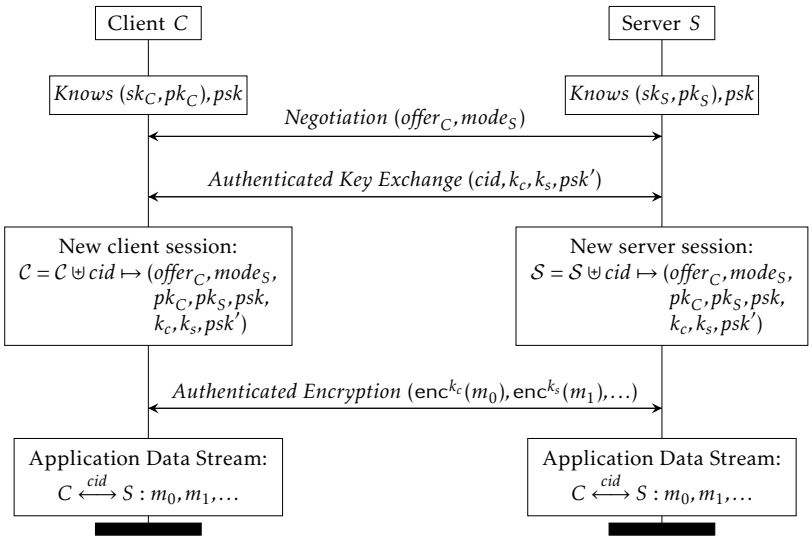

Figure 2.1: TLS Protocol Structure.

2.2 TLS 1.3 1-RTT: Simpler, Faster Handshakes

Then, C and S execute the negotiated *authenticated key exchange* protocol (e.g. Ephemeral Elliptic-Curve Diffie Hellman), which may use some combination of the long-term keys (e.g. public/private key pairs, symmetric pre-shared keys) known to the client and server. The key exchange ends by computing fresh symmetric keys (k_c, k_s) for a new session (with identifier *cid*) between C and S and potentially a new pre-shared key (psk') that can be used to authenticate future connections between them.

In TLS, the negotiation and key exchange phases are together called the *handshake* protocol. Once the handshake is complete, C and S can start exchanging application data, protected by an authenticated encryption scheme (e.g. AES-GCM) with the session keys (k_c, k_s). The TLS protocol layer that handles authenticated encryption for application data is called the *record* protocol.

2.2.1 Security Goals for TLS

Each phase of a TLS connection has its own correctness and security goals. For example, during negotiation, the server must choose a $mode_S$ that is consistent with the client's $offer_C$; the key exchange must produce a secret session key and so on. Although these intermediate security goals are important building blocks towards the security of the full TLS protocol, they are less meaningful to applications that typically use TLS via a TCP-socket-like API and are unaware of the protocol's internal structure. Consequently, we state the security goals of TLS from the viewpoint of the application, in terms of messages it sends and receives over a protocol session.

All goals are for messages between honest and authenticated clients and servers, that is, for those whose long-term keys (sk_C, sk_S, psk) are unknown to the attacker. If only the server is authenticated, then the goals are stated solely from the viewpoint of the client, since the server does not know whether it is talking to an honest client or the attacker.

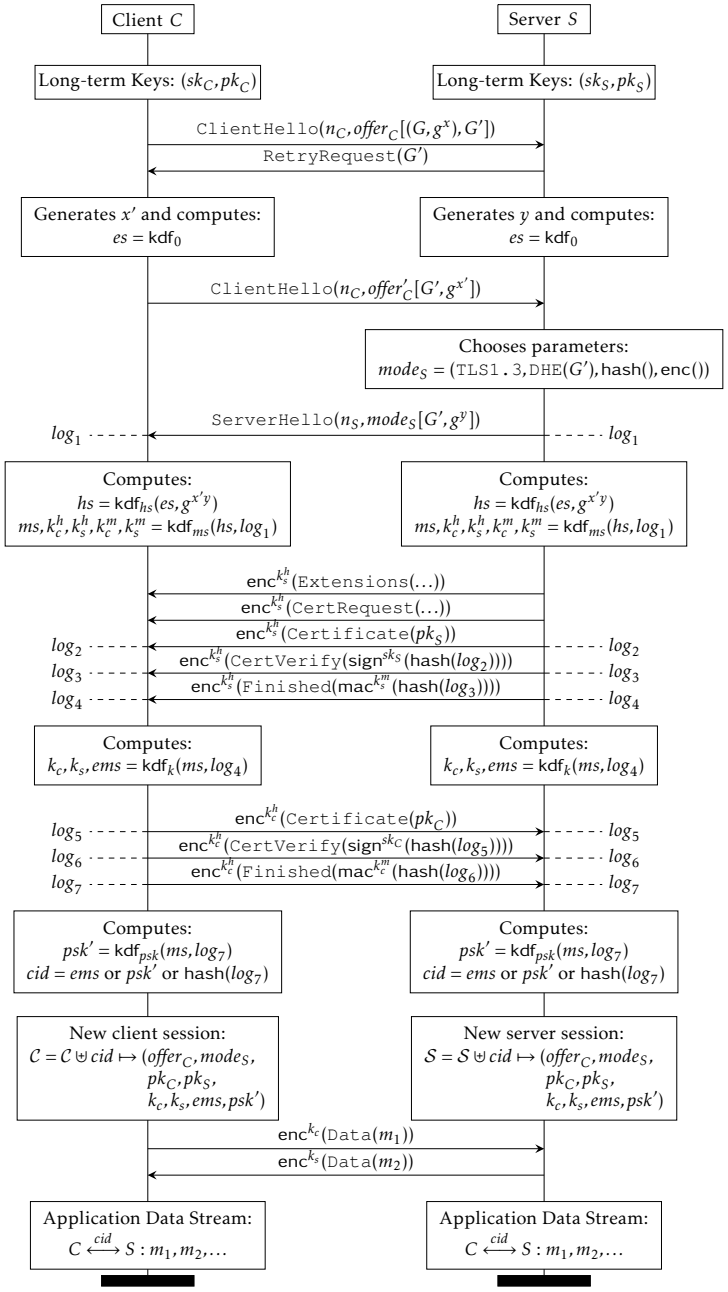

Figure 2.2: TLS 1.3 Draft-18 1-RTT Protocol. The protocol uses an (EC)DHE key exchange with server certificate authentication: client authentication and the `RetryRequest` negotiation steps are optional.

Secrecy: If an application data message m is sent over a session *cid* between an honest client C and honest server S, then this message is kept confidential from an attacker who cannot break the cryptographic constructions used in the session *cid*.

Forward Secrecy: Secrecy (above) holds even if the long-term keys of the client and server (sk_C, pk_C, psk) are given to the adversary after the session *cid* has been completed and the session keys k_c, k_s are deleted by C and S.

Authentication: If an application data message m is received over a session *cid* from an honest and authenticated peer, then the peer must have sent the same application data m in a matching session (with the same parameters $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

Replay Prevention: Any application data m sent over a session *cid* may be accepted at most once by the peer.

Unique Channel Identifier: If a client session and a server session have the same identifier *cid*, then all other parameters in these sessions must match (same $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

These security goals encompass most of the standard security goals for secure channel protocols such as TLS. For example, secrecy for application data implicitly requires that the authenticated key exchange must generate secret keys. Authentication incorporates the requirement that the client and server must have matching sessions and in particular, that they agree on each others' identities as well as the inputs and outputs of negotiation. Hence, it prohibits client and server impersonation and man-in-the-middle downgrade attacks.

The requirement for a unique channel identifier is a bit more unusual, but it allows multiple TLS sessions to be securely composed, for example via session resumption or renegotiation, without exposing them to credential forwarding attacks like Triple Handshake [22]. The channel identifier could itself be a session key or a value generated from it, but is more usually a public value that is derived from session data contributed by both the client and server [35].

2.2.2 A Realistic Threat Model for TLS

We seek to analyze TLS 1.3 for the above security goals against a rich threat model that includes both classic protocol adversaries as well as new ones that apply specifically to multi-mode protocols like TLS. In particular, we model recent downgrade attacks on TLS by allowing the use of weak cryptographic algorithms in older versions of TLS. In our analyses, the attacker can use any of the following attack vectors to disrupt the protocol.

- **Network Adversary:** As usual, we assume that the attacker can intercept, modify and send all messages sent on public network channels.
- **Compromised Principals:** The attacker can compromise any client or server principal P by asking for its long-term secrets, such as its private key (sk_P) or pre-shared key (psk). We do not restrict which principals can be compromised, but whenever such a compromise occurs, we mark it with a security event: `Compromised`(pk_P) or `CompromisedPSK`(psk). If the compromise event occurs after a session is complete, we issue a different security event: `PostSessionCompromise`(`cid`,pk_P).
- **Weak Long-term Keys:** If the client or server has a weak key that the attacker may be able to break with sufficient computation, we treat such keys the same way as compromised keys and we issue a more general event:`WeakOrCompromised`(pk_P). This conservative model of weak keys is enough to uncover attacks that rely on the use of 512-bit RSA keys by TLS servers.

- **RSA Decryption Oracles:** TLS versions up to 1.2 use RSA PKCS#1 v1.5 encryption, which is known to be vulnerable to a form of padding oracle attack on decryption originally discovered by Bleichenbacher [114]. Although countermeasures to this attack have been incorporated into TLS, they remains hard to implement securely [115] resulting in continued attacks such as DROWN [15]. Furthermore, such padding oracles can sometimes even be converted to signature oracles for the corresponding private key [113].

  We assume that any TLS server (at any version) that enables RSA decryption may potentially be vulnerable to such attacks. We distinguish between two kinds of RSA key exchange: `RSA(StrongRSADecryption)` and `RSA(WeakRSADecryption)`. In any session, if the server chooses the latter, we provide the attacker with a decryption and signature oracle for that private key.

- **Weak Diffie-Hellman Groups:** To account for attacks like Logjam [19], we allow servers to choose between strong and weak Diffie-Hellman groups (or elliptic curves) and mark the corresponding key exchange mode as `DHE(StrongDH)` or `DHE(WeakDH)`. We conservatively assume that weak groups have size 1, so all Diffie-Hellman exponentiations in these groups return the same distinguished element `BadElement`.

  Even strong Diffie-Hellman groups typically have small subgroups that should be avoided. We model these subgroups by allowing a weak subgroup (of size 1) even within a strong group. A malicious client or server may choose `BadElement` as its public value and then all exponentiations with this element as the base will also return `BadElement`. To avoid generating keys in this subgroup, clients and servers must validate the received public value.

- **Weak Hash Functions:** TLS uses hash functions for key derivation, HMAC and for signatures. Versions up to TLS 1.2 use various combinations of MD5 and SHA-1, both of which are considered weak today, leading to exploitable attacks on TLS such as SLOTH [20].

  We model both strong and weak hash functions and the client and server get to negotiate which function they will use in signatures. Strong hash functions are treated as one-way functions in our symbolic model, whereas weak hash functions are treated as point functions that map all inputs to a constant value: `Collision`. Hence, in our model, it is trivial for the attacker to find collisions as well as second preimages for weak hash functions.

- **Weak Authenticated Encryption:** To model recent attacks on RC4 [16, 17] and TripleDES [116], we allow both weak and strong authenticated encryption schemes. For data encrypted with a weak scheme, irrespective of the key, we provide the adversary with a decryption oracle.

  A number of attacks on the TLS Record protocol stem from its use of a MAC-Encode-Encrypt construction for CBC-mode ciphersuites. This construction is known to be vulnerable to padding oracle attacks such as POODLE [14] and Lucky13 [18], and countermeasures have proved hard to implement correctly [117]. We model such attacks using a leaky decryption function. Whenever a client or server decrypts a message with this function, the function returns the right result but also leaks the plaintext to the adversary.

The series of threats described above comprise our conservative threat model for TLS 1.3 and incorporates entire classes of attacks that have been shown to be effective against older versions of the protocol, including Triple Handshake, POODLE,

```
1  type group.
2  const StrongDH: group [data].
3  const WeakDH: group [data].
4
5  type element.
6  fun e2b(element): bitstring [data].
7  const BadElement: element [data].
8  const G: element [data].
9
10 fun dh_ideal(element,bitstring):element.
11 equation forall x:bitstring, y:bitstring; dh_ideal(dh_ideal(G,x),y) = dh_ideal(dh_ideal(G,y),x).
12
13 fun dh_exp(group,element,bitstring):element reduc forall g:group, e:element,
14 x:bitstring;
15 dh_exp(WeakDH,e,x) = BadElement
16 otherwise forall g:group, e:element, x:bitstring; dh_exp(StrongDH,BadElement,x) =
       BadElement
17 otherwise forall g:group, e:element, x:bitstring; dh_exp(StrongDH,e,x) = dh_ideal(e,x).
18
19 letfun dh_keygen(g:group) = new x:bitstring; let gx = dh_exp(g,G,x) in
20 (x,gx).
```

Figure 2.3: A Model of Diffie-Hellman in ProVerif that allows the weak groups and allows elements in small subgroups.

Lucky 13, RC4 NOMORE, FREAK, Logjam, SLOTH, DROWN. In most cases, we assume strictly stronger adversaries than have been demonstrated in practice, but since attacks only get better over time, our model seeks to be defensive against future attacks. It is worth noting that, even though TLS 1.3 does not itself support any weak ciphers, TLS 1.3 clients and servers will need to support legacy protocol versions for backwards compatibility. Our model enables a fine-grained analysis of vulnerabilities: we can ask whether TLS 1.3 connections between a client and a server are secure even if TLS 1.2 connections between them are broken.

2.2.3 Modeling the Threat Model in ProVerif

We encode our threat model as a generic ProVerif crypto library that can be used with any protocol. For each cryptographic primitive, our library contains constructors and destructors, that not only model the ideal behavior of strong cryptographic algorithms but also incorporates the possibility that honest protocol participants may support weak cryptographic algorithms.

Figure 2.3 displays our ProVerif model for Diffie-Hellman. We start by defining a type for groups; for simplicity, we only allow two groups (one strong, one weak). We then define a type for group elements and identify two distinguished elements that occur in every group, a basepoint `G` and an element `BadElement` that belongs to a trivial subgroup. The function `dh_ideal` exponentiates a (public) element with a (secret) scalar; the equation describing its behavior encode the expected, ideal behaviour of Diffie-Hellman exponentiation. However, our protocol processes do not use `dh_ideal`; instead they call `dh_exp` which is parameterized by the group and behaves differently depending on the strength of the group and the validity of the element. If the group is strong and the element is a valid member of the group (that is, it does not belong

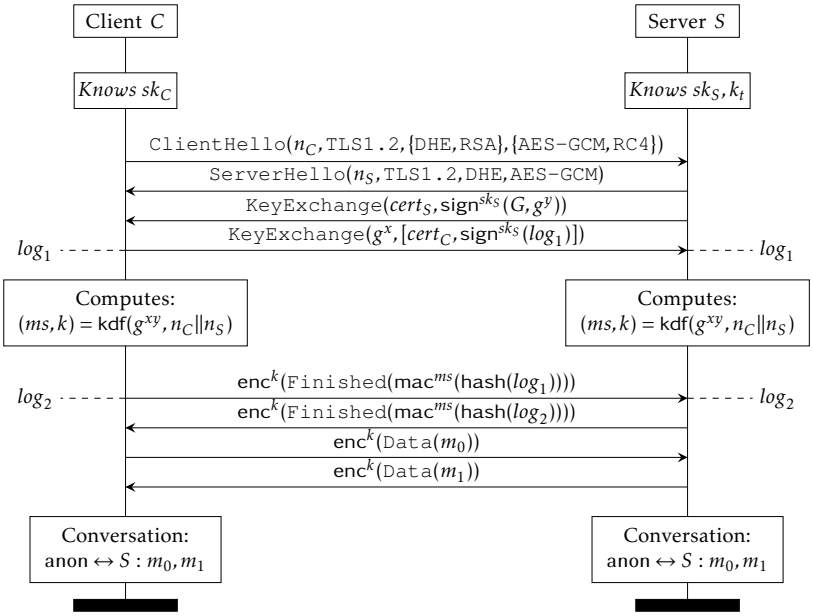

Figure 2.4: TLS 1.2 (EC)DHE handshake, no client authentication.

to a small subgroup), then `dh_exp` behaves like `dh_ideal`. In all other cases, the result of `dh_exp` is the trivial element `BadElement`; that is, it is known to the attacker. The final `letfun` in the figure shows how Diffie-Hellman keys are generated. We note that our model of weak groups and bad elements is conservative and simplistic; we aim to verify the security of protocols that use strong groups, even if weak groups are catastrophically broken.

Similarly to our model of Diffie-Hellman, we write models for AEAD, hash functions, HMAC, RSA signatures and encryption. Using these primitives, all the cryptographic constructions of TLS 1.2 and 1.3 are built as derived functions. To understand our security theorems, it is important for the analyst to carefully inspect our cryptographic library and agree with its implicit and explicit assumptions.

2.2.4 Modeling and Verifying TLS 1.2 in ProVerif

To evaluate our threat model and in preparation for our analysis of TLS 1.3, we symbolically analyze a model of TLS 1.2 using ProVerif. Our model includes TLS 1.2 clients and servers that support both RSA and Diffie-Hellman key exchanges and are willing to use both weak and strong cryptography. We assume that clients are unauthenticated.

For illustration, Figure 2.4 shows the Diffie-Hellman handshake followed by the exchange of two application data fragments m_0, m_1. The handshake has four flights; the client sends a `ClientHello` offering a set of algorithms; the server responds with a `ServerHello` that chooses specific connection parameters and then sends its certificate and (signed) key exchange message. The client responds with its own key exchange message and a `Finished` message with a MAC of the handshake using the connection master secret *ms*. The server completes the handshake by sending its own `Finished` message. These last two messages are meant to guarantee agreement on the keys and the handshake transcript and they already encrypted with the new connection keys *k*. Once both have verified each others' finished messages, the application on

top of TLS can start sending application data in the two directions.

We write ProVerif processes for TLS 1.2 clients and servers that exchange messages according to the protocol flow described above and issue a sequence of events– `ClientOffers`, `ServerChooses`, `ClientFinished`, `ServerFinished`, `ClientSends`, `ServerReceives`– indicating their progress through the protocol. We then compose these processes with our threat model and add queries for message authenticity and secrecy.

For example, a secrecy query may ask whether the attacker can learn some application data message (say m_0) sent by the client over a particular connection (identified by the client and server random values and the server's public key):

When we run ProVerif for this query, it finds a counter-example: the attacker can learn `m` if it can compromise server's private key (`WeakOrCompromised`(pk_S)). To check whether this is the only case in which `m` is leaked, we refine the secrecy query and run ProVerif again. ProVerif again finds a counter-example: the attacker can learn `m` if the server chooses a weak Diffie-Hellman group (`ServerChoosesKex(DHE(WeakDH))`). In this way, we keep refining our queries until we obtain the strongest secrecy query that holds for our TLS 1.2 model, that is, until we stop finding attacks:

```
aek:ae_key, g:group, ra:rsa_alg, cb:bitstring, cr':random, sr':random,
v:version; attacker(m_0(cr,sr,p)) ==> event(WeakOrCompromisedKey(p)) ||
event(ServerChoosesAE(cr,sr,p,TLS12,WeakAE)) ||
event(ServerChoosesKEX(cr,sr,p,TLS12,DHE(WeakDH))) ||
event(ServerChoosesKEX(cr',sr',p,TLS12,RSA(WeakRSADecryption))) ||
event(ServerChoosesHash(cr',sr',p,TLS12,WeakHash)) ||
(event(PostSessionCompromisedKey(p)) &&
event(ServerChoosesKEX(cr,sr,p,TLS12,RSA(ra)))) ||
```

This query is our main symbolic secrecy result for TLS 1.2; we similarly derive our strongest symbolic authenticaiton query. These two goals can be read as follows:

- **TLS 1.2 (Forward) Secrecy:** A message m sent by an honest client in a session *cid* to a server S cannot be known to the adversary unless one of the following conditions holds:
  (1) the server's public key is weak or compromised, or
  (2) the session uses weak authenticated encryption, or
  (3) the session uses a weak Diffie-Hellman group, or
  (4) the server uses weak RSA decryption with the same public key (in this or any other session), or
  (5) the server uses a weak hash function for signing with the same public key (in any session), or
  (6) the session uses an RSA key exchange and the server's public key was compromised after the session was complete.

- **TLS 1.2 Authenticity & Replay Protection:** Every message m accepted by an honest client in a session *cid* with some server S corresponds to a unique message sent by S on a matching session, unless one of the conditions (1)-(5) above holds.

Both these queries are verified by ProVerif in a few seconds. All the disjuncts (1)-(6) in these queries are necessary, removing any of them results in a counterexample discovered by ProVerif, corresponding to some well-known attack on badly configured TLS 1.2 connections.

Interestingly, the conditions (2), (3) are session specific, that is, only the sessions where these weak constructions are used are affected. In contrast, (4) and (5) indicate that the use of weak RSA decryption or a weak hash function in any session affects all other sessions that use the same server public key. As we shall see, this has an impact on the security of TLS 1.3 when it is composed with TLS 1.2. (6) encodes our forward secrecy goal, which does not hold for connections that use RSA key exchange, but holds for (EC)DHE.

We also verify our TLS 1.2 model for more advanced properties, such as unique channel identifiers; we find that using the connection key $cid = k_c$ does not yield a unique identifier. ProVerif finds a variant of the Triple Handshake attack, unless we implement the recommended countermeasure [118].

2.2.5 Verification Effort

The work of verifying TLS 1.2 can be divided into three tasks. We first modeled the threat model as a 400 line ProVerif library, but this library can now be reused for other protocols, including TLS 1.3. We then modeled the TLS 1.2 protocol in about 200 lines of ProVerif. Finally, we wrote about 50 lines of queries, both to validate our model (e.g. checking that the protocol completes in the absence of an attacker) and to prove our desired security goals. Most of the effort is in formalizing, refining, and discovering the right security queries. Although ProVerif is fully automated, verification gets more expensive as the protocol grows more complex. So, as we extend our models to cover multiple modes of TLS 1.3 composed with TLS 1.2, we sometimes need to simplify or restructure our models to aid verification.

In its simplest form, TLS 1.3 consists of a Diffie-Hellman handshake, typically using an elliptic curve, followed by application data encryption using an AEAD scheme like AES-GCM. The essential structure of 1-RTT has remained stable since early drafts of TLS 1.3. It departs from the TLS 1.2 handshake in two ways. First, the key exchange is executed alongside the negotiation protocol so the client can start sending application data along with its second flight of messages (after one round-trip, hence 1-RTT), unlike TLS 1.2 where the client had to wait for two message flights from the server. Second, TLS 1.3 eliminates a number of problematic features in TLS 1.2; it removes RSA key transport, weak encryption schemes (RC4, TripleDES, AES-CBC) and renegotiation; it requires group negotiation with strong standardized Diffie-Hellman groups and it systematically binds session keys to the handshake log to prevent attacks like the Triple Handshake. In this section, we detail the protocol flow, we model it in ProVerif and we analyze it alongside TLS 1.2 in the security model of §2.1.

2.2.6 1-RTT Protocol Flow

A typical 1-RTT connection in Draft 18 proceeds as shown in Figure 2.2. The first four messages form the negotiation phase. The client sends a `ClientHello` message containing a nonce n_C and an $offer_C$ that lists the versions, groups, hash functions, and authenticated encryption algorithms that it supports. For each group G that the client supports, it may include a Diffie-Hellman key share g^x. On receiving this message, the server chooses a $mode_S$ that fixes the version, group and all other session parameters. Typically, the server chooses a group G for which the client already provided a public value and so it can send its `ServerHello` containing a nonce n_S, $mode_S$ and g^y to the client. If none of the client's groups are acceptable, the server may ask the client (via `RetryRequest`) to resend the client hello with a key share $g^{x'}$ for the server's preferred group G'. (In this case, the handshake requires two round trips.)

Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^{k}(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) = \mathsf{HMAC\text{-}H}^{s}(len_{\mathsf{hash()}} || \text{“TLS 1.3,”} || l || h || \mathsf{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{hash}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{hash()}}}, 0^{len_{\mathsf{hash()}}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where

$ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{hash()}}})$

$hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1)$

$hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$

$k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \text{“”})$

$k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \text{“”})$

$k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \text{“”})$

$k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where

$ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4)$

$ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4)$

$ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$

$k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \text{“”})$

$k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where

$psk' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where

$es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{hash()}}}, psk)$

$k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where

$ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$

$k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \text{“”})$

Figure 2.5: TLS 1.3 Draft-18 Key Schedule. The hash function hash() **is typically** `SHA-256`**, which has length** $len_{\mathsf{hash()}} = 32$ **bytes.**

Once the client receives the `ServerHello`, the negotiation is complete and both participants derive handshake encryption keys from g^{xy}, one in each direction (k_c^h, k_s^h),

with which they encrypt all subsequent handshake messages. The client and server also generate two MAC keys (k_c^m, k_s^m) for use in the `Finished` messages described below. The server then sends a flight of up to 5 encrypted messages: `Extensions` contains any protocol extensions that were not sent in the `ServerHello`; `CertRequest` contains an optional request for a client certificate; `Certificate` contains the server's X.509 public-key certificate; `CertVerify` contains a signature with server's private key sk_S over the log of the transcript so far (log_2); `Finished` contains a MAC with k_s^m over the current log (log_3). Then the server computes the 1-RTT traffic keys k_c, k_s and may immediately start using k_s to encrypt application data to the client.

Upon receiving the server's encrypted handshake flight, the client verifies the certificate, the signature and the MAC and if all verifications succeed, the client sends its own second flight consisting of an optional certificate `Certificate` and signature `CertVerify`, followed by a mandatory `Finished` with a MAC over the full handshake log. Then the client starts sending its own application data encrypted under k_c. Once the server receives the client's second flight, we consider the handshake complete and put all the session parameters into the local session databases at both client and server ($\mathcal{C}$, $\mathcal{S}$).

In addition to the traffic keys for the current session, the 1-RTT handshake generates two extra keys: *ems* is an exporter master secret that may be used by the application to bind authentication credentials to the TLS channel; *psk'* is a resumption master secret that may be used as a pre-shared key in future TLS connections between C and S.

The derivation of keys in the protocol follows a linear key schedule, as depicted on the right of Figure 2.2. The first version of this key schedule was inspired by OPTLS [66] and introduced into TLS 1.3 in Draft-7. The key idea in this design is to accumulate key material and handshake context into the derived keys using a series of HKDF invocations as the protocol progresses. For example, in connections that use pre-shared keys (see §2.4), the key schedule begins by deriving *es* from *psk*, but after the `ServerHello`, we add in g^{xy} to obtain the handshake secret *hs*. Whenever we extract encryption keys, we mix in the current handshake log, in order to avoid key synchronization attacks like the Triple Handshake.

Since its introduction in Draft-7, the key schedule has undergone many changes, with a significant round of simplifications in Draft-13. Since all previously published analyses of 1-RTT predate Draft-13, this leaves open the question whether the current Draft-18 1-RTT protocol is still secure.

2.2.7 Modeling 1-RTT in ProVerif

We write client and server processes in ProVerif that implement the message sequence and key schedule of Figure 2.2.

Our models are abstract with respect to the message formats, treating each message (e.g. `ClientHello`$(\cdots)$) as a symbolic constructor, with message parsing modeled as a pattern-match with this constructor. This means that our analysis assumes that message serialization and parsing is correct; it won't find any attacks that rely on parsing ambiguities or bugs. This abstract treatment of protocol messages is typical of symbolic models; the same approach is taken by Tamarin [109]. In contrast, miTLS [32] includes a fully verified parser for TLS messages.

The key schedule is written as a sequence of ProVerif functions built using an HMAC function, `hmac(H,m)`, which takes a hash function `H` as argument and is assumed to be a one-way function as long as `H` = `StrongHash`. All other cryptographic functions are modeled as described in §2.1, with both strong and weak variants.

Persistent state is encoded using tables. To model principals and their long-term keys, we use a global private table that maps principals (A) to their key pairs ((sk_A, pk_A)). To begin with, the adversary does not know any of the private keys in this table, but it can compromise any principal and obtain her private key. As described in §2.1, this compromise is recorded in ProVerif by an event `WeakOrCompromised`(pk_A).

As the client and server proceed through the handshake they record security events indicating their progress. We treat the negotiation logic abstractly; the adversary gets to choose $offer_C$ and $mode_S$ and we record these choices as events (`ClientOffers`, `ServerChooses`) at the client and server. When the handshake is complete, the client and server issue events `ServerFinished`, `ClientFinished` and store their newly established sessions in two private tables `clientSession` and `serverSession` (corresponding to $\mathcal{C}$ and $\mathcal{S}$). These tables are used by the record layer to retrieve the traffic keys k_c, k_s for authenticated encryption. Whenever the client or server sends or receives an application data message, it issues further events (`ClientSends`, `ServerReceives`, etc.) We use all these events along with the client and server session tables to state our security goals.

2.2.8 1-RTT Security Goals

We encode our security goals as ProVerif *queries* as follows:

- **Secrecy** for a message, such as m_1, is encoded using an auxiliary process that asks the adversary to guess the value of m_1; if the adversary succeeds, the process issues an event `MessageLeaked`(cid,$\mathtt{m}_1$). We then write a query to ask ProVerif whether this event is reachable.

- **Forward Secrecy** is encoded using the same query, but we explicitly leak the client and server's long-term keys (sk_C, sk_S) at the end of the session cid. ProVerif separately analyzes pre-compromise and post-compromise sessions as different *phases*; the forward secrecy query asks that messages sent in the first phase are kept secret even from attackers who learn the long-term keys in the second phase.

- **Authentication** for a message m_1 received by the server is written as a query that states that whenever the event `ServerReceives`(cid,$\mathtt{m}_1$) occurs, it must be preceded by three matching events: `ServerFinished`(cid,...), `ClientFinished`(cid,...), and `ClientSends`(cid,$\mathtt{m}_1$), which means that some honest client must have sent m_1 on a matching session. The authentication query for messages received by clients is similar.

- **Replay protection** is written as a stronger variant of the authentication query that requires *injectivity*: each `ServerReceives` event must correspond to a unique, matching, preceding `ClientSends` event.

- **Unique Channel Identifiers** are verified using another auxiliary process that looks up sessions from the `clientSession` and `serverSession` tables and checks that if the cid in both is the same, then all other parameters match. Otherwise it raises an event and we ask ProVerif to prove that this event is not reachable.

In addition to the above queries, our scripts often include auxiliary queries about session keys and other session variables. We do not detail all our queries here; instead, we only summarize the main verification results. When we first ask ProVerif to verify these queries, it fails and provides counterexamples; for example, client message

authentication does not hold if the client is compromised `Compromised`(pk_C) or unauthenticated in the session. We then refine the query by adding this failure condition as a disjunct and run ProVerif again and repeat the process until the query is proved. Consequently, our final verification results are often stated as a long series of disjuncts listing the cases where the desired security goal does not hold.

2.2.9 Verifying 1-RTT in Isolation

For our model of Draft-18 1-RTT, ProVerif can prove the following secrecy query about all messages ($m_{0.5}, m_1, m_2$):

- **1-RTT (Forward) Secrecy**: Messages m sent in a session between C and S are secret as long as the private keys of C and S are not revealed before the end of the session and the server chooses a $mode_S$ with a strong Diffie-Hellman group, a strong hash function, and a strong authenticated encryption algorithm.

If we further assume that TLS 1.3 clients and servers only support strong algorithms, we can simplify the above query to show that all messages sent between uncompromised principals are kept secret. In the rest of this chapter, we assume that TLS 1.3 only enables strong algorithms, but that earlier versions of the protocol may continue to support weak algorithms.

Messages m_1 from the client to the server enjoy strong authentication and protection from replays:

- **1-RTT Authentication (and Replay Prevention)**: If a message m is accepted by S over a session with an honest C, then this message corresponds to a unique message sent by the C over a matching session.

However the authentication guarantee for messages $m_{0.5}, m_1$ received by the client is weaker. Since the client does not know whether the server sent this data before or after receiving the client's second flight, the client and server sessions may disagree about the client's identity. Hence, for these messages, we can only verify a weaker property:

- **0.5-RTT Weak Authentication (and Replay Prevention)**: If a message m is accepted by C over a session with an honest S, then this message corresponds to a unique message sent by S over a server session that matches all values in the client session except (possibly) the client's public key pk_C, the resumption master secret psk' and the channel identifier cid.

We note that by allowing the server to send 0.5-RTT data, Draft-18 has weakened the authentication guarantees for all data received by an authenticated client. For example, if a client requests personal data from the server over a client-authenticated 1-RTT session, a network attacker could delay the client's second flight (`Certificate–Finished`) so that when the client receives the server's 0.5-RTT data, it thinks that it contains personal data, but the server actually sent data intended for an anonymous client.

2.2.10 Verifying TLS 1.3 1-RTT composed with TLS 1.2

We combine our model with the TLS 1.2 model described at the end of §2.1 so that each client and server supports both versions. We then ask the same queries as above, but only for sessions where the server chooses TLS 1.3 as the version in $mode_S$. Surprisingly, ProVerif finds two counterexamples.

First, if a server supports `WeakRSADecryption` with RSA key transport in TLS 1.2, then the attacker can use the RSA decryption oracle to forge TLS 1.3 server signatures and hence break our secrecy and authentication goals. This attack found by ProVerif directly corresponds to the cross-protocol Bleichenbacher attacks described in [113, 15]. It shows that removing RSA key transport from TLS 1.3 is not enough, one must disable the use of TLS 1.2 RSA mode on any server whose certificate may be accepted by a TLS 1.3 client.

Second, if a client or server supports a weak hash function for signatures in TLS 1.2, then ProVerif shows how the attacker can exploit this weakness to forge a TLS 1.2 signature and establish a TLS 1.2 session in our model, hence breaking our security goals. This attack corresponds to the SLOTH transcript collision attack on TLS 1.3 signatures described in [20]. To avoid this attack, TLS 1.3 implementations must disable weak hash functions in all supported versions, not just TLS 1.3.

After disabling these weak algorithms in TLS 1.2, we can indeed prove all our expected security goals about Draft-18 1-RTT, even when it is composed with TLS 1.2.

We may also ask whether TLS 1.3 clients and servers can be downgraded to TLS 1.2. If such a version downgrade takes place, we would end up with a TLS 1.2 session, so we need to state the query in terms of sessions where $mode_S$ contains TLS 1.2. ProVerif finds a version downgrade attack on a TLS 1.3 session, if the client and server support weak Diffie-Hellman groups in TLS 1.2. This attack closely mirrors the flaw described in [112]. Draft-13 introduced a countermeasure in response to this attack and we verify that by adding it to the model, the downgrade attack disappears.

Although our models of TLS 1.3 and 1.2 are individually verified in a few seconds each, their composition takes several minutes to analyze. As we add more features and modes to the protocol, ProVerif takes longer and requires more memory. Our final composite model for all modes of TLS 1.3+1.2 takes hours on a powerful workstation.

2.3 0-RTT with Semi-Static Diffie-Hellman

In earlier versions of TLS, the client would have to wait for two round-trips of handshake messages before sending its request. 1-RTT in TLS 1.3 brings this down to one round trip, but protocols like QUIC use a "zero-round-trip" (0-RTT) mode, by relying on a *semi-static* (long-term) Diffie-Hellman key. This design was adapted for TLS in the OPTLS proposal [66] and incorporated in Draft-7 (along with a fix we proposed, as described below).

2.3.1 Protocol Flow

The protocol is depicted in Figure 2.6. Each server maintains a Diffie-Hellman key pair (s, g^s) and publishes a signed server configuration containing g^s. As usual, a client initiates a connection with a `ClientHello` containing its ephemeral key g^x. If a client has already obtained and cached the server's certificate and signed configuration (in a prior exchange for example), then the client computes a shared secret g^{xs} and uses it to derive an initial set of shared keys which can then immediately be used to send encrypted data. To authenticate its 0-RTT data, the client may optionally send a certificate and a signature over the client's first flight.

The server then responds with a `ServerHello` message that contains a fresh ephemeral public key g^y. Now, the client and server can continue with a regular 1-RTT handshake using the new shared secret g^{xy} in addition to g^{xs}.

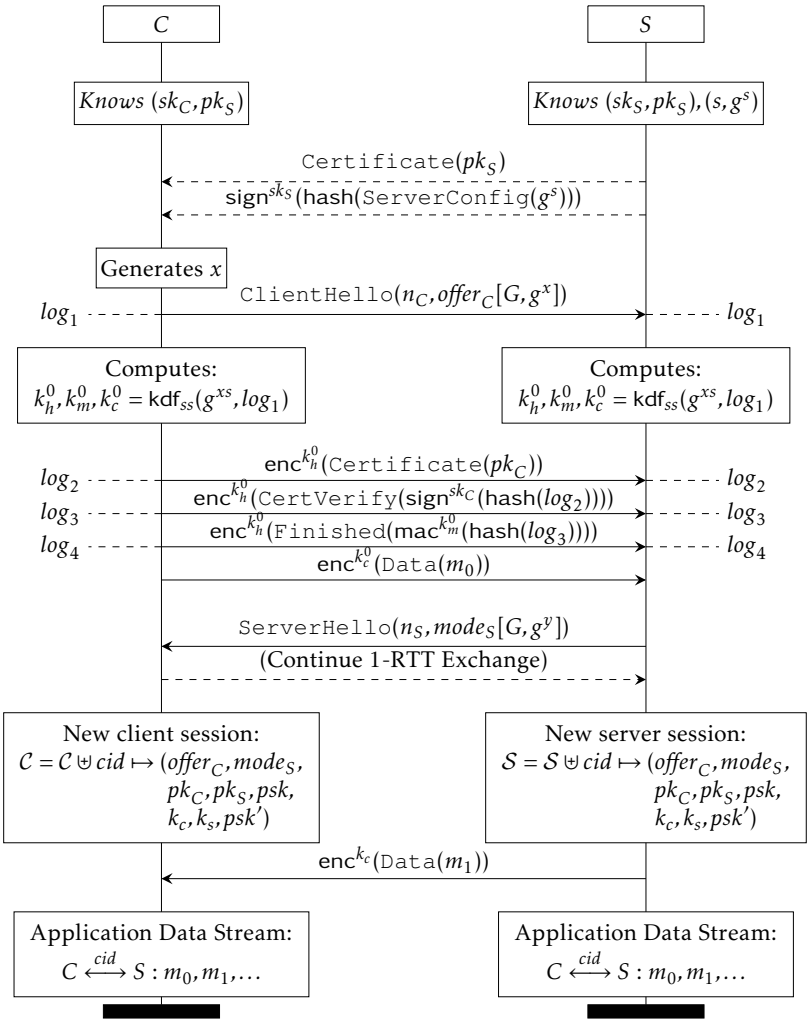

Figure 2.6: DH-based 0-RTT in TLS 1.3 Draft-12, inspired by QUIC and OPTLS.

The 0-RTT protocol continued to evolve from Draft-7 to Draft-12, but in Draft-13, it was removed in favor of a PSK-based 0-RTT mode. Even though Diffie-Hellman-based 0-RTT no longer exists in Draft-18, we analyze its security in this section, both for posterity and to warn protocol designers about the problems they should watch our for if they decide to reintroduce DH-based 0-RTT in a future version of TLS.

2.3.2 Verification with ProVerif

We modeled the protocol in ProVerif and wrote queries to check whether the 0-RTT data m_0 is (forward) secret and authentic. ProVerif is able to prove secrecy but finds that m_0 is not forward secret if the semi-static key s is compromised once the session is over. ProVerif also finds a Key Compromise Impersonation attack on authentication: if g^s is compromised, then an attacker can forge 0-RTT messages from C to S. Furthermore, the 0-RTT flight can be replayed by an attacker and the server will process

it multiple times, thinking that the client has initiated a new connection each time. In addition to these three concerns, which were documented in Draft-7, ProVerif also finds a new attack, explained below, that breaks 0-RTT authentication if the server's certificate is not included in the 0-RTT client signature.

2.3.3 Unknown Key Share Attack on DH-based 0-RTT in QUIC, OPTLS, and TLS 1.3

We observe that in the 0-RTT protocol, the client starts using g^s without having any proof that the server knows s. So a dishonest server M can claim to have the same semi-static key as S by signing g^s under its own key sk_M. Now, suppose a client connects to M and sends its client hello and 0-RTT data; M can simply forward this whole flight to S, which may accept it, because the semi-static keys match. This is an *unknown key share* (UKS) attack where C thinks it is talking to M but it is, in fact, connected to S.

In itself, the UKS attack is difficult to exploit, since M does not know g^{xs} and hence cannot decrypt or tamper with messages between C and S. However, if the client authenticates its 0-RTT flight with a certificate, then M can forward C's certificate (and C's signature) to S, resulting in a *credential forwarding* attack, which is much more serious. Suppose C is a browser that has a page open at website M; from this page M can trigger any authenticated 0-RTT HTTPS request m_0 to its own server, which then uses the credential forwarding attack to forward the request to S, who will process m_0 as if it came from C. For example, M may send a POST request that modifies C's account details at S.

The unknown key share attack described above applies to both QUIC and OPTLS, but remained undiscovered despite several security analyses of these protocols [119, 120, 66], because these works did not consider client authentication and hence did not formulate an authentication goal that exposed the flaw. We informed the authors of QUIC and they acknowledged our attack. They now recommend that users who need client authentication should not use QUIC and should instead move over to TLS 1.3. We also informed the authors of the TLS 1.3 standard and on our suggestion, Draft-7 of TLS 1.3 included a countermeasure for this attack: the client signature and 0-RTT key derivation include not just the handshake log but also the cached server certificate. With this countermeasure in place, ProVerif proves authentication for 0-RTT data.

2.4 Pre-Shared Keys for Resumption and 0-RTT

Aside from the number of round-trips, the main cryptographic cost of a TLS handshake is the use of public-key algorithms for signatures and Diffie-Hellman, which are still significantly slower than symmetric encryption and MACs. So, once a session has already been established between a client and server, it is tempting to reuse the symmetric session key established in this session as a pre-shared symmetric key in new connections. This mechanism is called *session resumption* in TLS 1.2 and is widely used in HTTPS where a single browser typically has many parallel and sequential connections to the same website. In TLS 1.2, pre-shared keys (PSKs) are also used instead of certificates by resource-constrained devices that cannot afford public-key encryption. TLS 1.3 combines both these use-cases in a single PSK-based handshake mode that combines resumption, PSK-only handshakes and 0-RTT.

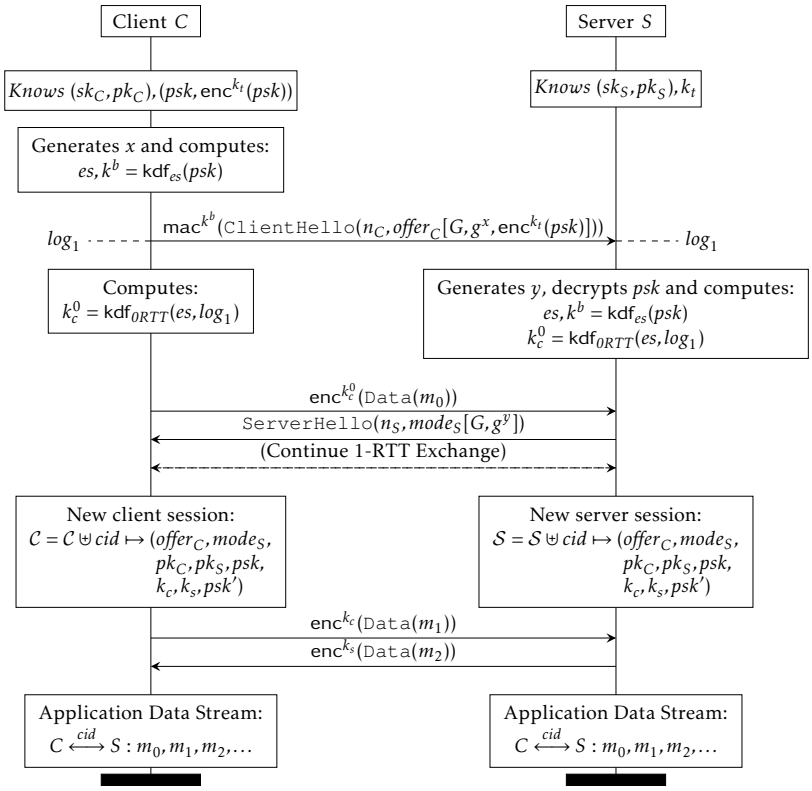

Figure 2.7: TLS 1.3 Draft-18 PSK-based Resumption and 0-RTT.

2.4.1 Protocol Flow

Figure 2.7 shows how this mode extends the regular 1-RTT handshake; in our analysis, we only consider PSKs that are established within TLS handshakes, but similar arguments apply to PSKs that are shared out-of-band. We assume that the client and server have established a pre-shared key *psk* in some earlier session. The client has cached *psk*, but in order to remain state-less, the server has given the client a ticket containing *psk* encrypted under an encryption key k_t. As usual, the client sends a `ClientHello` with its ephemeral key share g^x and indicates that it prefers to use the shared PSK *psk*. To prove its knowledge of *psk* and to avoid certain attacks (described below), it also MACs the `ClientHello` with a *binder* key k^b derived from the *psk*. The client can then use *psk* to already derive an encryption key for 0-RTT data m_0 and start sending data without waiting for the server's response. When the server receives the client's flight, it can choose to accept or reject the offered *psk*. Even if it accepts the *psk*, the server may choose to reject the 0-RTT data, it may choose to skip certificate-based authentication and (if it does not care about forward secrecy) it may choose to skip the Diffie-Hellman exchange altogether. The recommended mode is PSK-DHE, where *psk* and g^{xy} are both mixed into the session keys. The server then sends back a `ServerHello` with its choice and the protocol proceeds with the appropriate 1-RTT handshake and completes the session.

2.4.2 Verifying PSK-based Resumption

We first model the PSK-DHE 1-RTT handshake (without certificate authentication) and verify that it still meets our usual security goals:

- **PSK-DHE 1-RTT (Forward) Secrecy** Any message m sent over a PSK-DHE session in 1-RTT is secret as long as the PSK psk and the ticket encryption key k_t are not compromised until the end of the session.
- **PSK-DHE 1-RTT Authentication and Replay Protection** Any message m received over a PSK-DHE session in 1-RTT corresponds to a unique message sent by a peer over a matching session (notably with the same psk) unless psk or k_t are compromised during the session.
- **PSK-DHE 1-RTT Unique Channel Identifier** The values psk', ems, and $\mathsf{hash}(log_7)$ generated in a DHE or PSK-DHE session are all unique channel identifiers.

Notably, data sent over PSK-DHE is forward secret even if the server's long term ticket encryption key k_t is compromised after the session. In contrast, pure PSK handshakes do not provide this forward secrecy.

The authentication guarantee requires that the client and server must agree on the value of the PSK psk and if this PSK was established in a prior session, then the unique channel identifier property says that the client and server must transitively agree on the prior session as well. An earlier analysis of Draft-10 in Tamarin [109] found a violation of the authentication goal because the 1-RTT client signature in Draft-10 did not include the server's `Finished` or any other value that was bound to the PSK. This flaw was fixed in Draft-11 and hence we are able to prove authentication for Draft-18.

Verifying PSK-based 0-RTT. We extend our model with the 0-RTT exchange and verify that m_0 is authentic and secret. The strongest queries that ProVerif can prove are the following:

- **PSK-based 0-RTT (Forward) Secrecy** A message m_0 sent from C to S in a 0-RTT flight is secret as long as psk and k_t are never compromised.
- **PSK-based 0-RTT Authentication** A message m_0 received by S from C in a 0-RTT flight corresponds to some message sent by C with a matching `ClientHello` and matching psk, unless the psk or k_t are compromised.

In other words, PSK-based 0-RTT data is not forward secret and is vulnerable to replay attacks. As can be expected, it provides a symmetric authentication property: since both C and S know the psk, if either of them is compromised, the attacker can forge 0-RTT messages.

2.4.3 An Attack on 0-RTT Client Authentication

Up to Draft-12, the client could authenticate its 0-RTT data with a client certificate in addition to the PSK. This served the following use case: suppose a client and server establish an initial 1-RTT session (that outputs psk') where the client is unauthenticated. Some time later, the server asks the client to authenticate itself and so they perform a PSK-DHE handshake (using psk') with client authentication. The use of psk' ensures continuity between the two sessions. In the new session, the client wants to start sending messages immediately, and so it would like to use client authentication in 0-RTT.

To be consistent with Draft-12, suppose we remove the outer binder MAC (using k^b) on the `ClientHello` in Figure 2.7 and we allow client authentication in 0-RTT.

Then, if we model this protocol in ProVerif and ask the 0-RTT authentication query again, ProVerif finds a credential forwarding attack, explained next.

Suppose a client C shares *psk* with a malicious server M and M shares a different *psk*$'$ with an honest server S. If C sends an authenticated 0-RTT flight (certificate, signature, data m_0) to M, M can decrypt this flight using *psk*, re-encrypt it using *psk*$'$ and forward the flight to S. S will accept the authenticated data m_0 from C as intended for itself, whereas C intended to send it only to M. In many HTTPS scenarios, as discussed in §2.3, M may be able to control the contents of this data, so this attack allows M to send arbitrary requests authenticated by C to S.

This attack was not discovered in previous analyses of TLS 1.3 since many of them did not consider client authentication; the prior Tamarin analysis [109] found a similar attack on 1-RTT client authentication but did not consider 0-RTT client authentication. The attacks described here and in [109] belong to a general class of *compound authentication* vulnerabilities that appear in protocols that compose multiple authentication credentials [35]. In this case, the composition of interest is between PSK and certificate-based authentication. We found a similar attack on 1-RTT server authentication in pure PSK handshakes.

In response to our attack, Draft-13 included a `resumption_context` value derived from the *psk* in the handshake hash, to ensure that the client's signature over the hash cannot be forwarded on another connection (with a different *psk*$'$). This countermeasure has since evolved to the MAC-based design showed in Figure 2.7, which has now been verified in this work.

2.4.4 The Impact of Replay on 0-RTT and 0.5-RTT

It is now widely accepted that asynchronous messaging protocols like 0-RTT cannot be easily protected from replay, since the recipient has no chance to provide a random nonce that can ensure freshness. QUIC attempted to standardize a replay-prevention mechanism but it has since abandoned this mechanism, since it cannot prevent attackers from forcing the client to resend 0-RTT data over 1-RTT [121].

Instead of preventing replays, TLS 1.3 Draft-18 advises applications that they should only send non-forward-secret and idempotent data over 0-RTT. This recommendation is hard to systematically enforce in flexible protocols like HTTPS, where all requests have secret cookies attached and even GET requests routinely change state.

We argue that replays offer an important attack vector for 0-RTT and 0.5-RTT data. If the client authenticates its 0-RTT flight, then an attacker can replay the entire flight to mount *authenticated replay* attacks. Suppose the (client-authenticated) 0-RTT data asks the server to send a client's bank statement and the server sends this data in a 0.5-RTT response. An attacker who observes the 0-RTT request once, can replay it any number of times to the server from anywhere in the world and the server will send it the user's (encrypted) bank statement. Although the attacker cannot complete the 1-RTT handshake or read this 0.5-RTT response, it may be able to learn a lot from this exchange, such as the length of the bank statement and whether the client is logged in.

In response to these concerns, client authentication has now been removed from 0-RTT. However, we note that similar replay attacks apply to 0-RTT data that contains an authentication cookie or OAuth token. We highly recommend that TLS 1.3 servers should implement a replay cache (based on the client nonce n_C and the ticket age) to detect and reject replayed 0-RTT data. This is less practical in server farms, where time-based replay mitigation may be the only alternative.

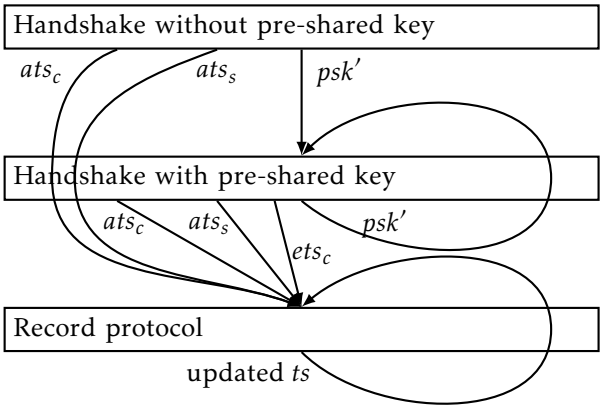

Figure 2.8: Structure of the CryptoVerif proof

2.5 Computational Analysis of TLS 1.3 Draft-18

Our ProVerif analysis of TLS 1.3 Draft-18 identifies the necessary conditions under which the symbolic security guarantees of the protocol hold. We now use the tool CryptoVerif [53] to see whether these conditions are sufficient to obtain cryptographic security proofs for the protocol in a more precise computational model. In particular, under the assumption that the algorithms used in TLS 1.3 Draft-18 satisfy certain strong cryptographic assumptions, we prove that the protocol meets our security goals.

Proofs in the computational model are hard to mechanize and CryptoVerif offers less flexibility and automation than ProVerif. To obtain manageable proofs, we focus only on TLS 1.3 (we do not consider TLS 1.2) and we ignore downgrade attacks. Moreover, we proceed modularly: we first prove some lemmas on primitives and we split the protocol into three pieces and prove them separately using CryptoVerif, before composing them manually to obtain a proof for the full protocol.

2.5.1 Cryptographic Assumptions

We make the following assumptions about the cryptographic algorithms supported by TLS 1.3 clients and servers.

Diffie-Hellman. We assume that the Diffie-Hellman groups used in TLS 1.3 satisfy the gap Diffie-Hellman (GDH) assumption [95]. This assumption means that given g, g^a and g^b for random a, b, the adversary has a negligible probability to compute g^{ab}, even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$.

In our proof, we require GDH rather than the weaker decisional Diffie-Hellman (DDH) assumption, in order to prove secrecy of keys on the server side as soon as the server sends its `Finished` message: at this point, if the adversary controls a certificate accepted by the client, he can send its own key share y' to the client to learn information on $g^{x'y'}$ and that would be forbidden under DDH. We also require that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$ implies $y = y'$, which holds when the considered Diffie-Hellman group is of prime order. This is true for all groups currently specified in TLS 1.3 [122, 97, 123, 124] and our proof requires it for all groups included in the future.

We also assume that all Diffie-Hellman group elements have a binary representation different from $0^{len_{\mathsf{hash}()}}$. This assumption simplifies the proof by avoiding a

possible confusion between handshakes with and without Diffie-Hellman exchange. Curve25519 does have a 32-byte zero element, but excluding zero Diffie-Hellman shared values is already recommended to avoid points of small order [125].

Finally, we assume that all Diffie-Hellman group elements have a binary representation different from $len_{\mathsf{hash()}}||\text{“TLS 1.3,”}||l||h||0\text{x}01$. This helps ease our proofs by avoiding a collision between $\mathsf{hkdf\text{-}extract}(es, e)$ and $\mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ or $\mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$. This assumption holds with the currently specified groups and labels, since group elements have a different length than the bitstring above. The technical problem identified by our assumption was independently discovered and discussed on the TLS mailing list [126] and has led to a change in Draft-19 which makes this assumption unnecessary: in the key schedule, hkdf-extract is applied to the result of $\mathsf{derive\text{-}secret}(es, \mathsf{ds}, \text{“”})$ instead of applying it to es.

Signatures. We assume that the function sign is unforgeable under chosen-message attacks (UF-CMA) [98]. This means that an adversary with access to a signature oracle has a negligible probability of forging a signature for a message not signed by the signature oracle. Only the oracle has access to the signing key; the adversary has the public key.

Hash Functions. We assume that the function H is collision-resistant [127]: the adversary has a negligible probability of finding two different messages with the same hash.

HMAC. We need two assumptions on HMAC-H:

We require that the functions $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{hash()}}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are independent random oracles, in order to justify the use of HMAC-H as a randomness extractor in the HKDF construct. This assumption can itself be justified as follows. Assuming that the compression function underlying the hash function is a random oracle, Theorem 4.4 in [103] shows that HMAC is indifferentiable [99] from a random oracle, provided the MAC keys are less than the block size of the hash function minus one, which is true for HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. It is then easy to show that $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{hash()}}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are indifferentiable from independent random oracles in this case.

We assume that HMAC-H is a pseudo-random function (PRF) [100], that is, HMAC-H is indistinguishable from a random function provided its key is random and used only in HMAC-H, when the key is different from $0^{len_{\mathsf{hash()}}}$ and kdf_0. We avoid these two keys to avoid confusion with the two random oracles above. Since keys are chosen randomly with uniform probability from a set key (with cardinality |key|), the only consequence of avoiding these keys is that $\frac{2}{|\mathsf{key}|}$ is added to the probability of breaking the PRF assumption.

Authenticated Encryption. The authenticated encryption scheme is IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) [128], provided the same nonce is never used twice with the same key. IND-CPA means that the adversary has a negligible probability of distinguishing encryptions of two distinct messages of the same length that it has chosen. INT-CTXT means that an adversary with access to encryption and decryption oracles has a negligible probability of forging a ciphertext that decrypts successfully and has not been returned by the encryption oracle.

2.5.2 Lemmas on Primitives and on the Key Schedule

We show the following properties:

- $\mathsf{mac}_{\mathsf{H}}^{k}(m) = \mathsf{mac}^{k}(\mathsf{hash}(m))$ is an SUF-CMA (strongly unforgeable under chosen message attacks) MAC. Indeed, since mac = HMAC-H is a PRF, it is an SUF-CMA MAC as shown in [129] and this property is preserved by composition with a collision-resistant hash function.
- $\mathsf{sign}_{\mathsf{H}}^{sk}(m) = \mathsf{sign}^{sk}(\mathsf{hash}(m))$ is an UF-CMA signature. Indeed, sign is an UF-CMA signature and this property is preserved by composition with a collision-resistant hash function.

We also prove several lemmas on the key schedule of TLS 1.3, using CryptoVerif.

- When es is a fresh random value, $e \mapsto \mathsf{hkdf\text{-}extract}(es, e)$ and $log_1 \mapsto \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$ are indistinguishable from independent random functions and $k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ and $\mathsf{hkdf\text{-}extract}(es, 0^{len_{\mathsf{hash}()}})$ are indistinguishable from independent fresh random values independent from these random functions.
- When hs is a fresh random value, $log_1 \mapsto \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1) \| \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$ is indistinguishable from a random function and $\mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{hash}()}})$ is indistinguishable from a fresh random value independent from this random function.
- When ms is a fresh random value, $log_4 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$ and $log_7 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$ are indistinguishable from independent random functions.
- When l_1, l_2, l_3 are pairwise distinct labels and s is a fresh random value, we have that the keys $\mathsf{hkdf\text{-}expand\text{-}label}(s, l_i, \text{“”})$ for $i = 1, 2, 3$ are indistinguishable from independent fresh random values.

All random values considered above are uniformly distributed. We use these properties as assumptions in our proof of the protocol. This modular approach considerably reduces the complexity of the games that CryptoVerif has to consider.

These results suggest that the key schedule could be simplified by replacing groups of calls to derive-secret that use the same key and log with a single call to derive-secret that would output the concatenation of several keys. The same remark also holds for calls to hkdf-expand-label that use the same key. This approach corresponds to the usage of expansion recommended in the formalization of HKDF [92] and would simplify the proof: some lemmas above would no longer be needed. We would also recommend replacing $ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{hash}()}})$ with $ms = \mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$: that would be more natural since we use the PRF property of HMAC-H for this computation and not the randomness extraction. If the argument $0^{len_{\mathsf{hash}()}}$ may change in the future, then we would support the recommendation of applying hkdf-extract to the result of $\mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$, discussed on the TLS mailing list [126] and implemented in Draft-19.

2.5.3 Verifying 1-RTT Handshakes without Pre-Shared Keys

To prove the security of TLS 1.3 in CryptoVerif, we split the protocol into three parts, as shown in Figure 2.8 and verify them in sequence, before composing them by hand into a proof for the full protocol. This modular hybrid approach allows us to have proofs of manageable complexity and to obtain results even when keys are reused many times, such as when several PSK-based resumptions are performed, which would otherwise be out of scope of CryptoVerif.

We first consider the initial 1-RTT handshake shown in Figure 2.2, until the new client and server session boxes. We model a honest client and a honest server, which are willing to interact with each other, but also with dishonest clients and servers included in the adversary. We do not consider details of the negotiation (or the `RetryRequest` message). We give the handshake keys (k_c^h and k_s^h) to the adversary, and let it encrypt and decrypt the handshake messages, so our security proof does not rely on the encryption of the handshake.

We assume that the server is always authenticated and consider both the handshake with and without client authentication. The honest client and server may be compromised at any time: the secret key of the compromised participant is then sent to the adversary and the compromise is recorded by defining a variable `corruptedClient` or `corruptedServer`.

The outputs of this protocol are the application traffic secrets ats_c and ats_s (the derivation of the keys k_c and k_s from these secrets is left for the record protocol), the exporter master secret *ems* and the resumption master secret psk' (later used as pre-shared key).

This protocol is modeled in CryptoVerif as shown in Figure 2.9. The context C_{h} first chooses randomly a key *hk* that models the choice of the random oracle HKDF_extract_zero_salt, that is, $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{hash}()}}}(x)$. Then, it provides the process Q_{h}, which allows the adversary to query this random oracle by sending its query on channel $c_{\mathsf{h}3}[i_{\mathsf{h}}]$ and receiving the result on channel $c_{\mathsf{h}4}[i_{\mathsf{h}}]$. This random oracle is actually not used in the initial handshake; adding it simplifies the composition with the handshake with pre-shared keys which uses it. The context C generates keys, defines the random oracle $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ in the same style as C_{h}, deals with key compromise, and runs client and server processes. We omit the details of this context to focus on the way we specify security properties.

At the end of the client code, the context C runs the process $P_{\mathsf{ClientFinal}}$. This process executes event ClientTerm1 with three arguments: the messages until the server `Finished` message (log_4), the messages after the server `Finished` message until the client `Finished` message (messages in log_7 but not in log_4) and the session keys. This event means that the client terminates. It is used for the unique channel identifier property. Next, the process distinguishes two cases, depending on whether the client is in a honest session or not. We say that the *client is in a honest session* when the certificate it received is the one of the honest server and either this server is not corrupted or the messages received by the client come from the honest server. Intuitively, the client is in a honest session when it talks to the honest server. In this case, the client executes event ClientTerm with arguments the messages until the server `Finished` message (log_4) and the session keys (psk' excluded). It also executes event ClientAccept with arguments all messages (log_7), all session keys and the replication index of the client i_C. These events are used for key authentication. It stores the keys ats_c (that is, *cats*), *ems*, psk' (that is, *resumption_secret*) in variables *c_cats*, *c_ems*, *c_resumption_secret* respectively. We shall prove the secrecy of these variables. Finally, it outputs the final message (client `Certificate`, `CertVerify`, `Finished`). When the client is not in a honest session, it just outputs the final message and the keys ats_c, ats_s, *ems*, psk', so that the adversary can continue the protocol. No security property is proved in this case.

On the server side, the context C runs the server until it is ready to send the server `Finished` message and 0.5-RTT data. Then it executes the process $P_{\mathsf{Server0.5\text{-}RTT}}$. This process distinguishes two cases, depending on whether the Diffie-Hellman key share $g^{x'}$ received by the server comes from the honest client. If it does, it executes the event ServerAccept with arguments the messages until the server `Finished` message (log_4),

$Q_{\mathrm{IH}} = C_{\mathsf{h}}[C[P_{\mathrm{ClientFinal}}, P_{\mathrm{Server0.5\text{-}RTT}}]]$

$C_{\mathsf{h}} = c_{\mathsf{h}1}();$**new** $hk : T_{hk}; \overline{c_{\mathsf{h}2}}\langle\rangle;([\,] \mid Q_{\mathsf{h}})$
$Q_{\mathsf{h}} = !^{i_{\mathsf{h}} \leq n_{\mathsf{h}}} c_{\mathsf{h}3}[i_{\mathsf{h}}](x : T_{\mathsf{h}}); \overline{c_{\mathsf{h}4}[i_{\mathsf{h}}]}\langle \mathsf{HKDF\_extract\_zero\_salt}(hk, x)\rangle$

$P_{\mathrm{ClientFinal}} =$
event ClientTerm1((*cr*, *cgx*, *sr*, *cgy*, *log0*, *log1*, *scv*, *m*), *final_log*, (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *cats*, *c_sats*, *ems*, *resumption_secret*));
if *honestsession* **then**
(
1: **event** ClientTerm((*cr*, *cgx*, *sr*, *cgy*, *log0*, *log1*, *scv*, *m*), (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *cats*, *c_sats*, *ems*));
2: **event** ClientAccept((*cr*, *cgx*, *sr*, *cgy*, *log0*, *log1*, *scv*, *m*, *final_log*), (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *cats*, *c_sats*, *ems*, *resumption_secret*), i_C);
let *c_cats* : *key* = *cats* **in**
let *c_ems* : *key* = *ems* **in**
let *c_resumption_secret* : *key* = *resumption_secret* **in**
$\overline{io6[i_C]}\langle final\_mess\rangle$
)
else
3: $\overline{io7[i_C]}\langle(final\_mess, (resumption\_secret, cats, c\_sats, ems))\rangle$.

$P_{\mathrm{Server0.5\text{-}RTT}} =$
find $j \leq N_C$ **suchthat defined**$(cgx[j]) \wedge sgx = cgx[j]$ **then**
(
4: **event** ServerAccept((*cr*, *sgx*, *sr*, *sgy*, *log0*, *log1*, *scv*, *m*), (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *cats*, *sats*, *ems*), i_S);
let *s_sats* : *key* = *sats* **in**
$\overline{io25[i_S]}\langle(\mathsf{ServerCertificateVerifyOut}(scv), \mathsf{ServerFinishedOut}(m))\rangle$;
$Q_{\mathrm{ServerAfter0.5\text{-}RTT1}}$
)
else
5: $\overline{io25'[i_S]}\langle((\mathsf{ServerCertificateVerifyOut}(scv), \mathsf{ServerFinishedOut}(m)), sats)\rangle$;
$Q_{\mathrm{ServerAfter0.5\text{-}RTT2}}$

$Q_{\mathrm{ServerAfter0.5\text{-}RTT1}} = C'[$
event ServerTerm1((*cr*, *sgx*, *sr*, *sgy*, *log0*, *certS*, *log1*, *scv*, *m*), *clientfinished*, (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *s_cats1*, *sats*, *s_ems1*, *s_resumption_secret1*));
if *honestsession* **then**
6: **event** ServerTerm((*cr*, *sgx*, *sr*, *sgy*, *log0*, *certS*, *log1*, *scv*, *m*, *clientfinished*), (*client_hk*, *server_hk*, *client_hiv*, *server_hiv*, *cfk*, *sfk*, *s_cats1*, *sats*, *s_ems1*, *s_resumption_secret1*));
$\overline{io27[i_S]}\langle\rangle$
else
7: $\overline{io28[i_S]}\langle(s\_resumption\_secret1, s\_cats1, sats, s\_ems1)\rangle]$

Figure 2.9: Model of the initial 1-RTT handshake

the session keys (psk' excluded) and the replication index of the server i_S. It stores the key ats_s (that is, *sats*) in *s_sats*. We shall prove secrecy of this key. Finally, it outputs the server `CertVerify` and `Finished` messages. When the Diffie-Hellman key share received by the server does not come from the honest client, the server outputs the `CertVerify` and `Finished` messages and the key ats_s so that the adversary can send 0.5-RTT data by himself. We do not prove security of 0.5-RTT data in this case.

After that, the server continues with $Q_{\mathrm{ServerAfter0.5\text{-}RTT1}}$ or $Q_{\mathrm{ServerAfter0.5\text{-}RTT2}}$. The

process $Q_{\text{ServerAfter0.5-RTT1}}$ executes event ServerTerm1 with three arguments: the messages until the server `Finished` message (log_4), the messages after the server `Finished` message until the client `Finished` message (messages in log_7 but not in log_4) and the session keys. Next, the process distinguishes two cases, depending on whether the server is in a honest session or not. We say that the *server is in a honest session* when

- if the client is authenticated, the certificate received by the server is the one of the honest client and either this client is not corrupted or the messages received by the server come from the honest client;
- if the client is not authenticated, the Diffie-Hellman key share $g^{x'}$ received by the server comes from the honest client.

Intuitively, the server is in a honest session when it talks to the honest client. In this case, the server executes event ServerTerm with arguments all messages (log_7) and all session keys and finally outputs an empty message to return control to the adversary. When the server is not in a honest session, it outputs the keys (psk',ats_c,ats_s,ems), so that the adversary can continue the protocol. No security property is proved in this case.

The process $Q_{\text{ServerAfter0.5-RTT2}}$ is similar to $Q_{\text{ServerAfter0.5-RTT1}}$, but with renamed channels (so that all channels are distinct) and variables:
s_resumption_secret1,*s_cats1*,*s_ems1*
Renamed into:
s_resumption_secret2,*s_cats2*,*s_ems2*.

This renaming is necessary because, when we prove secrecy of a variable, CryptoVerif requires that it is defined at a single location of a game.

Let us define $V_{\text{in}} = \{c\_cats, c\_sats, c\_ems, c\_resumption\_secret, s\_cats1, s\_cats2, s\_sats, s\_ems1, s\_ems2, s\_resumption\_secret1, s\_resumption\_secret2\}$ the set of variables containing the keys ats_c, ats_s, ems, psk' in honest sessions, on the client and server sides.

CryptoVerif proves the following properties:

- **Key Authentication:** If the client terminates and is in a honest session, then the server has accepted a session with the honest client and they share the same parameters: the keys ats_c, ats_s and ems and all messages sent in the protocol until the server `Finished` message. (We can make no claim on the client `Finished` message because it has not been received by the server at this point, nor on psk' because it depends on the client `Finished` message.) Formally, CryptoVerif proves the correspondence

  $$\textbf{inj-event}(\text{ClientTerm}(log_4, keys)) \Longrightarrow \textbf{inj-event}(\text{ServerAccept}(log_4, keys, i)) \quad (2.1)$$

  with public variables V_{in}, which means that, with overwhelming probability, each execution of event ClientTerm corresponds to a distinct execution of event ServerAccept with the same messages until server `Finished` and the same keys including ats_c, ats_s and ems. Event ClientTerm is executed when the client terminates and is in a honest session, event ServerAccept is executed when the server accepts a session with the client.

  Conversely, if a server terminates and is in a honest session, then the client has accepted a session with the honest server and they agree on the established keys and on all messages sent in the protocol. Formally, CryptoVerif proves the correspondence

  $$\textbf{inj-event}(\text{ServerTerm}(log_7, keys)) \Longrightarrow \textbf{inj-event}(\text{ClientAccept}(log_7, keys, i)) \quad (2.2)$$

with public variables V_{in}, which means that, with overwhelming probability, each execution of event ServerTerm corresponds to a distinct execution of event ClientAccept with the same messages and keys.

- **Replay Prevention:** The authentication properties stated above are already injective (because of the presence of **inj-event**), that is, they guarantee that each session of the client (resp. server) corresponds to a distinct session of the server (resp. client), and consequently, they forbid replay attacks.

- **(Forward) Secrecy of Keys:** The keys ats_c, ats_s, ems, and psk' exchanged in several protocol sessions are indistinguishable from independent fresh random values. This property means for instance that the keys ats_s remains secret (indistinguishable from independent fresh random values) even if ats_c, ems, psk' are given to the adversary and similarly for the other keys.

  We prove secrecy of ats_s on the server side when the key share $g^{x'}$ comes from the client as soon as the server sends its `Finished` message. This property allows us to prove security of 0.5-RTT messages by composition with the record protocol. Secrecy holds on the client side as well, when the client is in a honest session, because the client uses the same key as the server by key authentication. Formally, CryptoVerif proves that the protocol preserves the secrecy of *s_sats* with public variables $V_{\text{in}} \setminus \{c\_sats, s\_sats\}$.

  As noted in Section 2.2, the authentication for 0.5-RTT messages is weak: the client is not authenticated yet, so in the proof of secrecy of ats_s, we require that the key share $g^{x'}$ comes from the client. That weakens the authentication guarantees for all data received by an authenticated client. We have also written an alternative model without 0.5-RTT messages, in which we prove secrecy of ats_s on the client side when the client is in a honest session.

  Similarly, we prove secrecy of ats_c, ems and psk' on the client side when the client is in a honest session. Secrecy holds on the server side as well, when the server is in a honest session, because the server uses the same key as the client by key authentication. Formally, CryptoVerif proves that the protocol preserves the secrecy of *c_cats* with public variables $V_{\text{in}} \setminus \{c\_cats, s\_cats1, s\_cats2\}$, of *c_ems* with public variables $V_{\text{in}} \setminus \{c\_ems, s\_ems1, s\_ems2\}$ and of *c_resumption_secret* with public variables $V_{\text{in}} \setminus \{c\_resumption\_secret, s\_resumption\_secret1, s\_resumption\_secret2\}$.

- **Unique Accept:** The server never accepts twice with the honest client and the same messages until the server `Finished` message. Formally, CryptoVerif proves the correspondence

  $$\textbf{event}(\text{ServerAccept}(log_4, s\_keys, i)) \wedge \textbf{event}(\text{ServerAccept}(log_4, s\_keys', i')) \Longrightarrow i = i' \tag{2.3}$$

  with public variables V_{in}: two ServerAccept events with the same messages must have the same replication index, so they are in fact a single event.

  Similarly, the client never accepts twice with the honest server and the same messages until the client `Finished` message. Formally, CryptoVerif proves the correspondence

  $$\textbf{event}(\text{ClientAccept}(log_7, c\_keys, i)) \wedge \textbf{event}(\text{ClientAccept}(log_7, c\_keys', i')) \Longrightarrow i = i' \tag{2.4}$$

  with public variables V_{in}.

The correspondences (2.1) and (2.3) imply the correspondence

$$\textbf{event}(\mathsf{ClientTerm}(log_4, c\_keys)) \wedge \textbf{event}(\mathsf{ServerAccept}(log_4, s\_keys, i)) \Longrightarrow c\_keys = s\_keys$$

Indeed, if $\mathsf{ClientTerm}(log_4, c\_keys)$ and $\mathsf{ServerAccept}(log_4, s\_keys, i)$ are executed, then by (2.1), $\mathsf{ServerAccept}(log_4, c\_keys, i')$ is executed, so by (2.3), $i = i'$, so $\mathsf{ServerAccept}(log_4, s\_keys, i)$ and $\mathsf{ServerAccept}(log_4, c\_keys, i')$ are in fact the same event, hence $c\_keys = s\_keys$. This correspondence means that, if the client terminates and is in a honest session and the server accepts a session with the honest client and with the same messages until the server `Finished` message, then the client and server have the same keys ats_c, ats_s and ems. We name this property "same keys".

Similarly, the correspondences (2.2) and (2.4) imply

$$\textbf{event}(\mathsf{ServerTerm}(log_7, s\_keys)) \wedge \textbf{event}(\mathsf{ClientAccept}(log_7, c\_keys, i)) \Longrightarrow s\_keys = c\_keys$$

If the server terminates and is in a honest session and the client accepts a session with the honest server and with the same messages until the client `Finished` message, then the client and server have the same keys ats_c, ats_s, ems and psk'.

The three properties key authentication with replay prevention, secrecy of keys, and same keys are standard security properties for a key exchange protocol [64]. CryptoVerif could also prove "same keys" directly, but we need (2.3) and (2.4) for composition.

- **Unique Channel Identifier:** When cid is psk' or $\mathsf{hash}(log_7)$, we do not use CryptoVerif as the result is immediate: if a client session and a server session have the same cid, then these sessions have the same log_7 by collision-resistance of H (which implies collision-resistance of HMAC-H), so all their parameters are equal.

  When cid is ems, collision-resistance just yields that the client and server sessions have the same log_4. CryptoVerif proves that, if a client session and a server session both terminate successfully with the same log_4, then they have the same log_7 and the same keys, so all their parameters are equal. Formally, CryptoVerif proves the correspondence

$$\begin{aligned}&\textbf{event}(\mathsf{ClientTerm1}(sfl, c\_cfl, c\_keys)) \wedge \textbf{event}(\mathsf{ServerTerm1}(sfl, s\_cfl, s\_keys))\\ &\quad\Longrightarrow c\_cfl = s\_cfl \wedge c\_keys = s\_keys\end{aligned} \tag{2.5}$$

We need to guide CryptoVerif in order to prove these properties, with the following main steps. We first apply the security of the signature under the server key sk_S. We introduce tests to distinguish cases, depending on whether the Diffie-Hellman share received by the server is a share $g^{x'}$ from the client and whether the Diffie-Hellman share received by the client is the share g^y generated by the server upon receipt of $g^{x'}$. Then we apply the random oracle assumption on $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$, replace variables that contain $g^{x'y}$ with their values to make equality tests $m = g^{x'y}$ appear and apply the gap Diffie-Hellman assumption. At this point, the handshake secret hs is a fresh random value. We use the properties on the key schedule established in Section 2.5.2 to show that the other keys are fresh random values and apply the security of the MAC and of the signature under the client key sk_C.

2.5.4 Verifying Handshakes with Pre-Shared Keys

We now analyze the handshake protocol in Figure 2.7, up until the new client and server sessions are established. The protocol begins with 0-RTT and continues on to 1-

RTT. We consider both variants of PSK-based 1-RTT, with and without Diffie-Hellman exchange.

We ignore the ticket $\mathsf{enc}^{k_t}(psk)$ and consider a honest client and a honest server that initially share the pre-shared key *psk*. Dishonest clients and servers may be included in the adversary. As in the previous section, we give the handshake keys (k_c^h and k_s^h) to the adversary and ignore handshake encryption. Certificates for the client and server are optional, since they are already authenticated via the *psk*; we do not rely on authentication in our proofs and consider that the adversary performs the signature and verification operations on certificates if they occur.

The outputs of this protocol are the client early traffic secret ets_c (the derivation of the key k_c from ets_c is left for the record protocol), the application traffic secrets ats_c and ats_s, the exporter master secret *ems* and the resumption master secret psk'.

This protocol is modeled in CryptoVerif as shown in Figure B.1, after moving some random number generations and assignments. The process first defines the random oracle $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{hash}()}}}(x)$ using the context C_{h} as in Figure 2.9. Then it chooses a fresh pre-shared key *psk*. (The input and output just allow the adversary to schedule this choice.) Finally, it launches processes for the client and the server, for the handshakes with and without Diffie-Hellman exchange.

The process $Q_{\text{PSKOnlyClient}}$ represents the client for a handshake with pre-shared key and without Diffie-Hellman exchange. The context $C_{\text{PSKOnlyClient}}$ builds the `ClientHello` message and computes the early traffic secret ets_c. Then the process inserts the `ClientHello` message (nonce *c_cr*, binder *c_binder* and other elements $varc\_log1, c\_log1'$) in the table c_table. Tables are lists of tuples shared between all honest participants of the protocol. They can be read by the **get** construct: **get** $Tbl(x_1,\ldots,x_l)$ **suchthat** M **in** P **else** P' tries to retrieve an element $(x_1,\ldots,x_l)$ in the table Tbl such that M is true. When such an element is found, it executes P with $x_1,\ldots,x_l$ bound to that element. When no such element is found, P' is executed. Equality tests $= M_i$ are also allowed instead of variables x_i; in this case, the table element must contain the value of M_i at the i-th position. Then, the process executes the event ClientEarlyAccept1 with arguments the `ClientHello` message, ets_c (that is, *cets*) and the replication index of the client i_C and stores *cets* in *c_cets*. These operations are useful to establish security of 0-RTT data. Then it outputs the `ClientHello` message and continues the protocol until the client `Finished` message. It executes the event ClientTerm1 with three arguments: the messages until the server `Finished` message, the messages after the server `Finished` message until the client `Finished` message and the session keys. This event means that the client terminates. It is used for the unique channel identifier property. Next, the process stores ats_s into *c_sats*. It executes event ClientTerm with arguments the messages until the server `Finished` message and the session keys (psk' excluded). It also executes event ClientAccept with arguments all messages (log_7), all session keys, and the replication index of the client i_C. These events are used for key authentication. It stores the keys ats_c (that is, *cats*), *ems*, psk' (that is, *resumption_secret*) in variables *c_cats*, *c_ems*, *c_resumption_secret* respectively. We shall prove the secrecy of these variables. Finally, it outputs the final message (client `Certificate`, `CertVerify`, `Finished`).

The process $Q_{\text{PSKOnlyServer}}$ represents the server for a handshake with pre-shared key and without Diffie-Hellman exchange. The context $C_{\text{PSKOnlyServer}}$ receives the `ClientHello` message and runs two holes in parallel. The first hole deals with the reception of 0-RTT data. It distinguishes several cases. When the received `ClientHello` comes unaltered from the honest client (it is in the table c_table), it stores the early traffic secret in *s_cets2*, executes event ServerEarlyTerm1 with arguments the `ClientHello` message and *s_cets2* and returns control to the adversary. In this case, we are going to show that

0-RTT data is secure: it is authenticated and confidential, but may be replayed. Otherwise, it stores the early traffic secret in *s_cets3*, executes event ServerEarlyTerm2 with arguments the `ClientHello` message and *s_cets2* and further distinguishes two cases: either the received `ClientHello` message has already been received before, and we are going to reuse the previous early traffic secret, or it is a new `ClientHello` message and we store the early traffic secret in *s_cets1*. We shall prove that this variable is secret and that, when the server receives an altered `ClientHello` message, it cannot receive 0-RTT data. The second hole deals with the rest of the protocol. It first executes event ServerAccept with arguments the messages until the server `Finished` message, the session keys (psk' excluded) and the replication index of the server i_S. It stores the key ats_s in *s_sats*. We shall prove secrecy of this key, used for 0.5-RTT data. Finally, it outputs the server `CertVerify` and `Finished` messages. Then the server receives the client `Finished` message, executes event ServerTerm1 with three arguments: the messages until the server `Finished` message, the messages after the server `Finished` message until the client `Finished` message and the session keys. It stores the keys ats_c, *ems*, psk' (that is, *resumption_secret*) in variables *s_cats*, *s_ems*, *s_resumption_secret* respectively. It executes event ServerTerm with arguments all messages and all session keys and finally outputs an empty message.

The processes for the handshake with Diffie-Hellman exchange are similar. Obviously, they additionally perform the Diffie-Hellman exchange. The events ClientEarlyAccept1, ClientTerm, ClientAccept, ServerEarlyTerm1, ServerEarlyTerm2, ServerAccept, and ServerTerm have an additional suffix DHE. The variables and tables have prefix *cdhe_* instead of *c_* and *sdhe_* instead of *s_*.

The set of variables V_{psk} contains all variables with prefixes *c_*, *s_*, *cdhe_* and *sdhe_*.

We run CryptoVerif on our model to obtain the following verification results:

- **Key Authentication, Replay Prevention, Secrecy of Keys, Unique Accept:** CryptoVerif shows the same properties as for the handshake without pre-shared key, with similar queries. The differences are as follows: the key *psk* is never compromised, so the client and server are always in a honest session; the queries are duplicated for the handshake with and without Diffie-Hellman exchange; the variables are not duplicated for the distinction whether the Diffie-Hellman key share received by the server comes from the client or not in 0.5-RTT, but are duplicated for the handshake with and without Diffie-Hellman exchange.

- **0-RTT data:** CryptoVerif cannot prove authentication of ets_c. While the binder $\mathsf{mac}^{k_b}(\cdot)$ authenticates most of the client `ClientHello` message, the client may offer several pre-shared keys and send a binder for each of these keys. Only the binder for the pre-shared key selected by the server is checked. Hence the adversary may alter another of the proposed binders, yielding a different log_1 and a different ets_c on the server side. This is not a serious attack, as the record protocol will fail if ets_c does not match on the client and server sides.

  CryptoVerif cannot prove replay protection for the 0-RTT session key ets_c, and indeed the client `ClientHello` message can be replayed, yielding the same key ets_c for several sessions of the server even though there is a single session of the client.

  Secrecy of ets_c holds on the client side; on the server side, each key ets_c is indistinguishable from random, but the keys ets_c are not independent of each other since an adversary may force the server to accept several times the same key ets_c by replaying the client `ClientHello` message.

CryptoVerif shows that, if the server receives the `ClientHello` message unaltered, then the client sent it (obviously!) and the client and server share the same early traffic secret ets_c. Formally,

$$\textbf{event}(\mathsf{ServerEarlyTerm1}(log1, cets)) \Longrightarrow \textbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets, i)) \tag{2.6}$$

with public variables V_{psk}. It shows that the client never sends twice the same `ClientHello` message. Formally,

$$\textbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets, i)) \wedge \textbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets', i')) \Longrightarrow i = i' \tag{2.7}$$

with public variables V_{psk}. It shows that ets_c is secret on the client side. Formally, it shows the secrecy of *c_cets* with public variables $V_{\text{psk}} \setminus \{c\_cets, s\_cets2\}$. These three properties provide security for a key exchange with one-way non-injective authentication. (The correspondences (2.6) and (2.7) imply a "same key" property:

$$\textbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets, i)) \wedge \textbf{event}(\mathsf{ServerEarlyTerm1}(log1, cets')) \Longrightarrow cets = cets'$$

as in the initial handshake.)

CryptoVerif also shows that, if the server receives twice the same altered `ClientHello` message, then it computes the same early traffic secret ets_c. Formally,

$$\textbf{event}(\mathsf{ServerEarlyTerm2}(log1, cets)) \wedge \textbf{event}(\mathsf{ServerEarlyTerm2}(log1, cets')) \Longrightarrow cets = cets' \tag{2.8}$$

with public variables V_{psk}. Finally, it shows the secrecy of *s_cets1* with public variables $V_{\text{psk}} \setminus \{s\_cets1, s\_cets3\}$. These two properties deal with the case in which the `ClientHello` message is altered. They show that in this case, ets_c is fresh secret random value when that `ClientHello` is received for the first time and that it is the same as the previous ets_c when `ClientHello` is replayed. By composition with the record protocol, they imply that the server fails to receive 0-RTT data in this case.

CryptoVerif shows the same properties for the handshake with Diffie-Hellman exchange.

- **Forward Secrecy:** CryptoVerif is unable to prove secrecy of the keys when *psk* is compromised after the end of the session, even assuming that hkdf-extract is a random oracle. Secrecy obviously does not hold in this case for the handshake without Diffie-Hellman exchange. We believe that it still holds for the handshake with Diffie-Hellman exchange; our failure to prove it in this case is due to the current limitations of CryptoVerif.
- **Unique Channel Identifier:** We proceed as in the handshake without pre-shared key. We additionally notice that, if a client session and a server session have the same log_7, then they have the same *psk*. Indeed, by collision-resistance of mac = HMAC-H, they have the same k^b, so the same *es*, so the same *psk*.

2.5.5 Verifying the Record Protocol

The third component of TLS 1.3 is the record protocol that encrypts and decrypts messages after the new client and server sessions have been established in Figures 2.2 and 2.7.

In our model, we assume that the client and server share a fresh random traffic secret. We generate an encryption key and an initialization vector (IV), and send and receive encrypted messages using those key and IV and a counter that is distinct for each message. (Our model is more detailed than the symbolic presentation given in the figures as we consider the IV and the counter.) We also generate a new traffic secret as specified in the key update mechanism of TLS 1.3 Draft-18 (Section 7.2).

More formally, after moving some assignments, our model of the record protocol in CryptoVerif is of the following form

$$Rec = c_1(); \mathbf{new}\ b : bool; \mathbf{new}\ ts : key; \mathbf{let}\ ts_{upd} : key = \mathsf{HKDF\_expand\_upd\_label}(ts)\ \mathbf{in}\ \overline{c_2}\langle\rangle; \\ (Q_{send}(b) \mid Q_{recv})$$

It chooses a random bit b (false = 0 or true = 1) and a random traffic secret ts. It computes the updated traffic secret ts_{upd} and then provides two processes $Q_{send}(b)$ and Q_{recv}. The process $Q_{send}(b)$ receives two clear messages msg_0 and msg_1 and a counter $count$. Provided the counter has not been used for sending a previous message and the messages msg_0 and msg_1 have the same padded length, it executes the event $\mathsf{sent}_0(count, msg_b)$ and sends the message msg_b encrypted using keys derived from the traffic secret ts. The process Q_{recv} receives an encrypted message and a counter $count$. Provided the counter has not been used for receiving a previous message, it decrypts the message using keys derived from the traffic secret ts and executes event $\mathsf{received}_0(count, msg)$ where msg is the clear message. Both the emission and reception can be executed several times.

CryptoVerif proves the following properties automatically:

- **Key Secrecy:** CryptoVerif proves that the updated traffic secret is indistinguishable from a fresh random value. Formally, CryptoVerif proves that ts_{upd} is secret with public variable b.

- **Message Secrecy:** CryptoVerif proves that, when the adversary provides two sets of plaintexts m_i and m'_i of the same padded length, it is unable to determine which of the two sets is encrypted, even when the updated traffic secret is leaked. Formally, CryptoVerif proves that b is secret with public variable ts_{upd}.

- **Message Authentication:** CryptoVerif proves that, if a message msg is decrypted by the receiver with a counter $count$, then the message msg has been encrypted and sent by an honest sender with the same counter $count$. Formally, CryptoVerif proves the correspondence

  $$\mathbf{event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathbf{event}(\mathsf{sent}_0(count, msg))$$

  with public variables b, ts_{upd}.

- **Replay Prevention:** CryptoVerif shows that any sent application data may be accepted at most once by the receiver. Formally, CryptoVerif proves the correspondence

  $$\mathbf{inj\text{-}event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathbf{inj\text{-}event}(\mathsf{sent}_0(count, msg))$$

  with public variables b, ts_{upd}, which means that each execution of event $\mathsf{received}_0(count, msg)$ corresponds to a distinct execution of $\mathsf{sent}_0(count, msg)$. This correspondence implies message authentication.

We consider two other variants of the record protocol, used for 0-RTT. In the first variant, $Rec_{0\text{-RTT}}$, the receiver process is replicated once more, so that several sessions may have the same traffic secret, thus the receiver accepts messages with the same counter in different sessions with the same traffic secret. It models that the server may receive several times the same `ClientHello` message, yielding the same traffic secret. In this model, CryptoVerif proves key and message secrecy and message authentication but not replay prevention. In the second variant, $Rec_{0\text{-RTT,Bad}}$, the sender process is additionally removed. This model corresponds to the situation in which the `ClientHello` message is altered and thus the server obtains a traffic secret that is not used by any client. In this model, CryptoVerif proves key secrecy and that the $\mathsf{received}_0$ event has a negligible probability of being executed: $\mathbf{event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathsf{false}$ with public variable ts_{upd}.

2.5.6 A Composite Proof for TLS 1.3 Draft-18

We compose these results using a hybrid argument (as in [65]). Figure 2.8 summarizes the structure of the composition. We provide only a brief sketch of the composition. A more detailed proof [130] has since been published independently of this thesis.

First, we compose the record protocol *Rec* with itself recursively, using the secrecy of the updated traffic secret, to show that the security properties of the record protocol are preserved by key updates. We also perform similar compositions for the two variants $Rec_{0\text{-RTT}}$ and $Rec_{0\text{-RTT,Bad}}$ of the record protocol for 0-RTT. We obtain processes Rec^m, ${Rec_{0\text{-RTT}}}^m$ and ${Rec_{0\text{-RTT,Bad}}}^m$ that perform at most m key updates, for any m.

Second, we compose the handshakes with pre-shared key with the record protocol:

- using the secret key *c_cats* as traffic secret in the process Rec^m for 1-RTT client-to-server messages: the sender side is plugged after event ClientAccept at line 3: and the receiver side is plugged after event ServerTerm at line 7: in Figure B.1;
- using the secret key *s_sats* as traffic secret in the process Rec^m for 0.5-RTT and 1-RTT server-to-client messages: the sender side is plugged after event ServerAccept at line 6: and the receiver side is plugged after event ClientTerm at line 2: in Figure B.1;
- using the secret key *c_cets* as traffic secret in the process ${Rec_{0\text{-RTT}}}^m$ for 0-RTT messages when the `ClientHello` message has not been altered: the sender side is plugged after event ClientEarlyAccept1 at line 1: and the receiver side is plugged after event ServerEarlyTerm1 at line 4: in Figure B.1;
- using the secret key *s_cets3* as traffic secret in the process ${Rec_{0\text{-RTT,Bad}}}^m$ for 0-RTT when the `ClientHello` message has been altered: the receiver process is plugged after event ServerEarlyTerm2 at line 5: in Figure B.1. (There is no sender process in this case.)

We perform similar compositions in the handshake with pre-shared key and Diffie-Hellman key agreement. We also compose the obtained process with itself recursively, using the resumption secret *c_resumption_secret* as pre-shared key in the next handshake: the client side is plugged after event ClientAccept at line 3: and the server side is plugged after event ServerTerm at line 7: in Figure B.1. We perform a similar composition with secret *cdhe_resumption_secret*.

For these compositions, we rely the properties of key secrecy, key authentication with replay prevention and unique accept. We use the key secrecy property to replace session keys with independent fresh random values. We rely on authentication with

replay prevention and unique accept to show that the same replacement is performed in matching sessions of the client and server. For 0-RTT, the authentication is weaker (we do not have replay prevention), so we need to adapt our composition theorems to this case. The composed protocol inherits security properties from the components we compose. In particular, we obtain secrecy, authentication and replay prevention for 0.5-RTT and 1-RTT application messages in both directions. For 0-RTT messages, since the handshake does not prevent replays for this key, the composition will not prevent replays for messages sent under this key. These compositions yield processes $Q^{l,m}_{\text{PSKH}}$ that perform at most l successive handshakes with pre-shared key and m key updates.

Third, we compose the initial handshake with the record protocol

- using the secret key *c_cats* as traffic secret in the process Rec^m for 1-RTT client-to-server messages: the sender side is plugged after event ClientAccept at line 2: and the receiver side is plugged after event ServerTerm at line 6: in Figure 2.9 and in the process $Q_{\text{ServerAfter0.5-RTT2}}$;
- using the secret key *s_sats* as traffic secret in the process Rec^m for 0.5-RTT and 1-RTT server-to-client messages: the sender side is plugged after event ServerAccept at line 4: and the receiver side is plugged after event ClientTerm at line 1: in Figure 2.9.

We also compose the initial handshake with the process $Q^{l,m}_{\text{PSKH},s}$ that runs handshakes with pre-shared key, using the secret key *c_resumption_secret* as pre-shared key: the client side is plugged after event ClientAccept at line 2: and the server side is plugged after event ServerTerm at line 6: in Figure 2.9 and in the process $Q_{\text{ServerAfter0.5-RTT2}}$. (The forward secrecy property of the initial handshake allows us to leak the keys sk_S and sk_C, so that the adversary can indeed perform the signature operations related to certificates, as we assumed in our model of handshakes with pre-shared keys.)

Finally, we compose the obtained process with a process that runs the rest of the TLS protocol without any event or security claim, at lines 3:, 5:, 7: of Figure 2.9 and at a line similar to 7: in the process $Q_{\text{ServerAfter0.5-RTT2}}$.

These compositions allow us to infer security properties of the TLS protocol from properties of the handshakes and the record protocol. In particular, we obtain secrecy, forward secrecy (with respect to the compromise of sk_S and sk_C), authentication and replay prevention for 0.5-RTT and 1-RTT application messages in both directions. For 0-RTT messages, we do not obtain replay prevention, but still obtain secrecy, forward secrecy (with respect to the compromise of sk_S and sk_C) and authentication.

2.5.7 Conclusion

Symbolic Analysis of TLS 1.3. We symbolically analyzed a composite model of TLS 1.3 Draft-18 with optional client authentication, PSK-based resumption, and PSK-based 0-RTT, running alongside TLS 1.2 against a rich threat model and we established a series of security goals. In summary, 1-RTT provides forward secrecy, authentication and unique channel identifiers, 0.5-RTT offers weaker authentication and 0-RTT lacks forward secrecy and replay protection.

We discovered potential vulnerabilities in 0-RTT client authentication in earlier draft versions. These attacks were presented at the TLS Ready-Or-Not (TRON) workshop and contributed to the removal of certificate-based 0-RTT client authentication from TLS 1.3. The current design of PSK binders in Draft-18 is also partly inspired by these kinds of authentication attacks.

TLS 1.3 has been symbolically analyzed before, using the Tamarin prover [109]. ProVerif and Tamarin are both state-of-the-art protocol analyzers with different strengths. Tamarin can verify arbitrary compositions of protocols by relying on user-provided lemmas, whereas ProVerif is less expressive but offers more automation. In terms of protocol features, the Tamarin analysis covered PSK and ECDHE handshakes for 0-RTT and 1-RTT in Draft-10, but did not consider 0-RTT client certificate authentication or 0.5-RTT data. On the other hand, they do consider delayed (post-handshake) authentication, which we did not consider here.

The main qualitative improvement in our verification results over theirs is that we consider a richer threat model that allows for downgrade attacks and that we analyze TLS 1.3 in composition with previous versions of the protocol, whereas they verify TLS 1.3 in isolation.

Our full ProVerif development consists of 1030 lines of ProVerif; including a generic library incorporating our threat model (400 lines), processes for TLS 1.2 (200 lines) and TLS 1.3 (250 lines) and security queries for TLS 1.2 (50 lines) and TLS 1.3 (180 lines). All proofs complete in about 70 minutes on a powerful workstation. In terms of manual effort, these models took about 3 person-weeks for a ProVerif expert.

Computational Proofs for TLS 1.3. We presented the first mechanically-checked cryptographic proof for TLS 1.3, developed using the CryptoVerif prover. We prove secrecy, forward secrecy with respect to the compromise of long-term keys, authentication, replay prevention (except for 0-RTT data) and existence of a unique channel identifier for TLS 1.3 draft-18. Our analysis considers PSK modes with and without DHE key exchange, with and without client authentication. It includes 0-RTT and 0.5-RTT data, as well as key updates, but not post-handshake authentication.

Unlike the ProVerif analysis, our CryptoVerif model does not consider compositions of client certificates and pre-shared keys in the same handshake. It also does not account for version or ciphersuite negotiation; instead, we assume that the client and server only support TLS 1.3 with strong cryptographic algorithms. The reason we limit the model in this way is to make the proofs more tractable, since CryptoVerif is not fully automated and requires significant input from the user. With future improvements in the tool, we may be able to remove some of these restrictions.

CryptoVerif is better suited to proofs than finding attacks. In case of attack, the proof fails, but proof failure may also come from other reasons: limitations of the tool, assumptions too weak, inappropriate guidance from the user. Therefore, we only consider in CryptoVerif properties for which ProVerif did not find attacks. Sometimes, proof failures in CryptoVerif might lead us towards computational attacks that do not appear at the symbolic level, for instance attacks that allow an adversary to distinguish between two different data messages but do not allow it to compute these messages, or attacks that rely on the algorithms of cryptographic primitives. However, we did not find such attacks in our model of TLS 1.3. We failed to prove forward secrecy for handshakes that use both pre-shared keys and Diffie-Hellman, but this failure is due to limitations in our tool, not due to an attack. Our proofs required some unusual assumptions on public values in Diffie-Hellman groups to avoid confusions between different key exchange modes; these ambiguities are inherent in Draft-18 but have been fixed in Draft-19, making some of our assumptions unnecessary.

Verifying TLS Implementations. Specifications for protocols like TLS are primarily focused on interoperability; the RFC standard precisely defines message formats, cryptographic computations and expected message sequences. However, it says little about what state machine these protocol implementations should use, or what APIs they should offer to their applications. This specification ambiguity is arguably the

culprit for many implementation bugs and protocol flaws [22] in TLS.

In the absence of a more explicit specification, we advocate the need for verified reference implementations of TLS that can provide exemplary code and design patterns on how to deploy the protocol securely. We proposed one such implementation, RefTLS, for use in JavaScript applications. The core protocol code in RefTLS implements both TLS 1.2 and 1.3 and has been verified using ProVerif. However, RefTLS is a work-in-progress and many of its trusted components remain to verified. For example, we did not verify our message parsing code or cryptographic libraries and our verification results rely on the correctness of the unverified ProScript-to-ProVerif compiler [73].

Other Verification Approaches. In addition to ProVerif and CryptoVerif, there are many symbolic and computational analysis tools that have been used to verify cryptographic protocols like TLS. As discussed above, Tamarin [131] was used to symbolically analyze TLS 1.3 Draft-10 [109]. EasyCrypt [132] has been used to develop cryptographic proofs for various components used in TLS, including the MAC-Encode-Encrypt construction used in the record layer [133].

Chapter 3

Formal Verification of Arbitrary Noise Protocols

This work was completed very shortly before the final draft of this thesis was submitted and is currently under peer review. It was completed with the assistance of Karthikeyan Bhargavan and would not have been possible without the assistance of Bruno Blanchet. I am also grateful towards Trevor Perrin, the author of the Noise Protocol, for his feedback, engagement and encouragement.

Popular Internet protocols such as SSH and TLS use similar cryptographic primitives: symmetric primitives, public key primitives, one-way hash functions and so forth. Protocol stages are also similarly organized, usually beginning with an authenticated key exchange (AKE) stage followed by a messaging stage. And yet, the design methodology, underlying state machine transitions and key derivation logic tend to be entirely different between protocols with nevertheless similar building blocks. The targeted effective security goals tend to be similar, so why can't the same methodology be followed for everything else?

Standard protocols such as those mentioned above choose a specific set of key exchange protocols to satisfy some stated use-cases while leaving other elements, such as round trips and (notoriously) cipher suites up to the deployer. Specifications use protocol-specific verbose notation to describe the underlying protocol, to the extent that even extracting the core cryptographic protocol becomes hard, let alone analyzing and comparing different modes for security.

Using completely different methodologies to build protocols that nevertheless often share the same primitives and security goals is not only unnecessary, but provably dangerous. The Triple Handshake attack on TLS published in 2014 [22] is based on the same logic that made the attack [33] on the Needham-Schroeder protocol [34] possible almost two decades earlier.

The core protocol in TLS 1.2 was also vulnerable to a similar attack, but since the protocol itself is hidden within layers of packet formats and C-like pseudocode, it was difficult for the attack to be detected. However, upon automated symbolic verification [35, 68], the attack quickly appeared not just in TLS, but also in variants of SSH and IPsec. Flaws underlying more recent attacks such as Logjam [19] were known for years before they were observed when the vulnerable protocol was analyzed. Had these protocols differed only in terms of network messages while still using a uniform, formalized logic for internal key derivation and state machine transitioning designed

based on the state of the art of protocol analysis, these attacks could have been avoided.

3.0.1 The Noise Protocol Framework

$$
\begin{aligned}
&IK: \\
&\quad \leftarrow s \\
&\quad \cdots \\
&\quad \rightarrow e, es, s, ss \\
&\quad \leftarrow e, ee, se
\end{aligned}
$$

Figure 3.1: An example Noise Handshake Pattern, `IK`.

The Noise Protocol Framework [36], recently introduced by Trevor Perrin, aims to avert this problem by presenting a simple language for describing cryptographic network protocols. In turn, a large number of semantic rules extend this simple protocol description to provide state machine transitions, key derivation logic and so on. The goal is to obtain the strongest possible effective security guarantees for a given protocol based on its description as a series of network messages by deriving its other elements from a uniform, formally specified logic followed by all protocol designs.

In designing a new secure channel protocol using the Noise Protocol Framework, one only provides an input using the simple language shown in Fig. 3.1. As such, from the viewpoint of the protocol designer, Noise protocols can only differ in the number of messages, the types of keys exchanged and the sequence or occurrence of public key transmissions and Diffie-Hellman operations. Despite the markedly non-expressive syntax, however, the occurence and position of the "tokens" in each message pattern can trigger complex state machine evolutions for both parties, which include operations such as key derivation and transcript hash mixing.

Let's examine Fig. 3.1. Before the AKE begins, the responder shares his static public key. Then in the first protocol message, the initiator sends a fresh ephemeral key, calculates a Diffie-Hellman shared secret between her ephemeral key and the recipient's public static key, sends her public static key and finally calculates a Diffie-Hellman shared secret between her static key and the responder's public static key. The responder then answers by generating an ephemeral key pair and sending his ephemeral public key, deriving a Diffie-Hellman shared secret between his ephemeral key and the ephemeral key of the initiator and another Diffie-Hellman shared secret between his static key and the ephemeral key of the initiator. Both of these AKE messages can also contain message payloads, which, depending on the availability of sufficient key material, could be AEAD-encrypted (in this particular Noise Handshake Pattern, this is indeed the case.)

As we can see, quite a few operations have occured in what would at first glance appear to be simple syntax for a simple protocol. Indeed, underlying these operations is a sophisticated state machine logic tasked with mixing all of the derived keys together, determining when it is safe (or possible) to send encrypted payloads and ensuring transcript consistency, among other things. This is the value of the Noise Protocol Framework: allowing the protocol designer to describe what they need their protocol to do fairly effectively using this simple syntax, and leaving the rest to a sturdy set of underlying rules.

3.0.2 Noise Explorer: Formal Verification for any Noise Handshake Pattern

IN :
$\rightarrow e, s$
$\leftarrow e, ee, se$

Figure 3.2: An example Noise Handshake Pattern, `IN`.

Noise Explorer, the central contribution of this work, capitalizes on the strengths of the Noise Protocol Framework in order to allow for automated protocol verification to no longer be limited only to monolithic, pre-defined protocols with their own notation. In this work, we formalize Noise's syntax, semantics, state transitions and Noise Handshake Pattern validity rules. We then present translation logic to go from Noise Handshake Patterns directly into full symbolic models ready for automated verification using the ProVerif [47, 48] automatic protocol verifier.

This allows us to then construct Noise Explorer, an online engine that allows for designing, validating and subsequently generating cryptographic models for the automated formal verification of any arbitrary Noise Handshake Pattern. Models generated using Noise Explorer allow for the verification of Noise-based secure channel protocols against a battery of comprehensive ProVerif queries. Noise Explorer also comes with the first compendium of formal verification results for Noise Handshake Patterns, browsable online using an interactive web application that presents dynamically generated diagrams indicating every cryptographic operation and security guarantee relevant to every message within the Noise Handshake Pattern.

3.0.3 Contributions

Formal semantics and validity rules for Noise Handshake Patterns. §3.1 introduces formal verification in the symbolic model using ProVerif, setting the stage for §3.2.

Translations from Noise Patterns to processes in the applied-pi calculus. §3.2 discusses automated translations from valid Noise Handshake Patterns into a representation in the applied-pi calculus [54] which includes cryptographic primitives, state machine transitions, message passing and a top-level process illustrating live protocol execution. We present the first formal semantics and validity rules (illustrated as typing inference rules) for Noise Handshake Patterns. This allows Noise Explorer to validate and separate sane Noise Handshake Patterns from invalid ones based on arbitrary input, and is the foundation of further contributions described below.

Noise Handshake Pattern security goals expressed as ProVerif queries. In §3.3, we model all five "confidentiality" security goals from the Noise Protocol Framework specification in the applied-pi calculus and extend the two "authentication" goals to four.

Formal verification results for 50 Noise Handshake Patterns in the Noise Protocol Framework specification. §3.4 sees all of the previous contributions come together to provide formal verification results for 50 Noise Handshake Patterns.[1] We find that while most of the results match those predicted by the specification authors, our extended model for "authentication" queries allows for more nuanced results. Furthermore, in §3.5, we analyze unsafe Noise Handshake Patterns and discover a potential for forgery attacks should Noise Handshake Patterns not be followed properly.

[1] Anyone can use Noise Explorer to increase this number by designing, validating then automatically verifying their own Noise Handshake Pattern.

3.1 Formal Verification in the Symbolic Model

The main goal of this work is to use the ProVerif automated protocol verifier to obtain answers to our formal verification queries. In this section, we describe the parts of ProVerif that are relevant to our analysis.

ProVerif uses the applied-pi calculus, a language geared towards the description of network protocols, as its input language. It analyzes described protocols under a Dolev-Yao model, which effectively mimicks an active network attacker. ProVerif models are comprised of a section in which cryptographic protocol primitives and operations are described as `fun`s or `letfun`s and a "top-level process" section in which the execution of the protocol on the network is outlined.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels and can manipulate them symbolically. Parallel and unbounded numbers of executions of different parts of the protocol are supported.

In the symbolic model, cryptographic primitives are represented as "perfect blackboxes"; a hash function, for example, cannot be modeled to be vulnerable to a length extension attack (which the hash function SHA-1 is vulnerable to, for example.) Encryption primitives are perfect pseudorandom permutations. Hash functions are perfect one-way maps. It remains possible to build complex primitives such as authenticated encryption with associated data (AEAD) and also to model interesting use cases, such as a Diffie-Hellman exponentiation that obtains a shared secret that is outside of the algebraic group. However, in the latter case, such constructions cannot be based on on an algebra that includes the space of integers commonly considered when modeling these scenarios, since all messages are simply a combination of the core terms used to express primitives.

3.1.1 Verification Context

All generated models execute the protocol in comprehensive formal verification context: a typical run includes a process in which Alice initiates a session with Bob, a process in which Alice initiates a session with Charlie, a process in which Bob acts a responder to Alice and a process in which Bob acts as a responder to Charlie. Charlie is a compromised participant whose entire state is controlled by the attacker. Each process in the top-level process are executed in parallel. The top-level process is executed in an unbounded number of sessions. Within the processes, transport messages are again executed in an unbounded number of sessions in both directions. Fresh key material is provided for each ephemeral generated in each session within the unbounded number of sessions: no ephemeral key reuse occurs between the sessions modeled.

3.1.2 Cryptographic Primitives

Noise Handshake Patterns make use of cryptographic primitives which in this work we will treat as constructions in the symbolic model. We consider the following cryptographic primitives:

- KP(): Generates a new Diffie-Hellman key pair consisting of a private key x and a public key g^x.

- DH($x \leftarrow$ KP(), y)**:** Derives a Diffie-Hellman shared secret between the private key within the key pair x and the public key y.
- E(k, n, ad, p)**:** Encrypts and generates an authentication tag for plaintext p using key k and nonce n, optionally extending the authentication tag to cover associated data ad. The output is considered to be Authenticated Encryption with Associated Data (AEAD) [102].
- D(k, n, ad, c)**:** Decrypts and authenticates ciphertext c using key k and nonce n. Associated data ad must also be included if it was defined during the encryption step for authentication to pass on both c and ad.
- R(k)**:** Returns a new key by applying a pseudorandom function on k.
- H(d)**:** A one-way hash function on data d.
- HKDF(ck, ik)**:** A Hash-Based Key Derivation function [92] that takes keys (ck, ik) and outputs a triple of keys. In some instances, the third key output is discarded and not used. The function is similar to the original HKDF definition but with ck acting as the salt and with a zero-length "*info*" variable.

In ProVerif, Diffie-Hellman is implemented as a `letfun` that takes two `key`-type values (representing points on the Curve25519 [97] elliptic curve) along with an `equation` that essentially illustrates the Diffie-Hellman relationship $g^{ab} = g^{ba}$ in the symbolic model.[2] DH and KP (implemented as `generate_keypair`) are then implemented as `letfun`s on top of that construction:[3]

```
fun dhexp(key, key):key.
equation forall a:key, b:key;
dhexp(b, dhexp(a, g)) = dhexp(a, dhexp(b, g)).
```

Encryption is implemented as a function that produces a bitstring (representing the ciphertext) parametrized by a key, nonce, associated data and plaintext. Decryption is a reduction function that produces the correct plaintext only when the appropriate parameters are given, otherwise the process ends:

```
fun encrypt(key, nonce, bitstring, bitstring):bitstring.

fun decrypt(key, nonce, bitstring, bitstring):aead reduc
forall k:key, n:nonce, ad:bitstring, plaintext:bitstring;
decrypt(k, n, ad, encrypt(k, n, ad, plaintext)) = aeadpack(true, ad, plaintext).
```

Finally, H and HMAC are implemented as one-way functions parametrized by two bitstrings (for ease of use in modeling in the case of H, and for a keyed hash representation in the case of HMAC) while HKDF is constructed on top of them.

3.1.3 ProVerif Model Components

In the ProVerif model of a Noise Handshake Pattern, there are nine components:

1. **ProVerif parameters.** This includes whether to reconstruct a trace and whether the attacker is active or passive.

[2] Recall that, in the symbolic model, any arithmetic property such as additivity is not a given and must be modeled specifically.

[3] `keypairpack` and `keypairunpack` are a `fun` and `reduc` pair that allow compressing and decompressing a tuple of `key` values into a `keypair`-type value for easy handling throughout the model. Whenever the suffixes `pack` and `unpack` appear from now on, it is safe to assume that they function in a similar pattern.

2. **Types.** Cryptographic elements, such as keys are nonces, are given types. Noise Handshake Message state elements such as `CipherState`s, `SymmetricState`s and `HandshakeState`s (see §3.2) are given types as well as constructors and reductors.

3. **Constants.** The generator of the g Diffie-Hellman group, HKDF constants such as `zero` and the names of principals (Alice, indicating the initiator, Bob, indicating the recipient, and Charlie, indicating a compromised principal controlled by the attacker) are all declared as constants.

4. **Bitstring concatenation.** Functions are declared for bitstring concatenation, useful for constructing and destructing the message buffers involved in the Noise Protocol Framework's `WriteMessage` and `ReadMessage` functions.

5. **Cryptographic primitives.** DH, KP, E, D, H and HKDF are modeled as cryptographic primitives in the symbolic model.

6. **State transition functions.** All functions defined for `CipherState`, `SymmetricState` and `HandshakeState` are implemented in the applied-pi calculus.

7. **Channels.** Only a single channel is declared, `pub`, representing the public Internet.

8. **Events and queries.** Here, the protocol events and security queries relevant to a particular Noise Handshake Pattern are defined. This includes the four authentication queries and five confidentiality queries discussed in §3.3.

9. **Protocol processes and top-level process.** This includes the `WriteMessage` and `ReadMessage` function for each handshake and transport message, followed by the top-level process illustrating the live execution of the protocol on the network.

3.2 Representing Noise in the Applied-Pi Calculus

The Noise Protocol Framework [36] is restricted only to describing messages between two parties (initiator and responder), the public keys communicated and any Diffie-Hellman operations conducted. Messages are called Noise "Message Patterns". They make up authenticated key exchanges, which are called Noise "Handshake Patterns". Noise supports authenticated encryption with associated data (AEAD) and Diffie-Hellman key agreement. The Noise Protocol Framework does not currently support any signing operations.

The full description of a Noise-based secure channel protocol is contained within its description of a Noise Handshake Pattern, such as the one seen in Fig. 3.1. The initial messages within a Noise Handshake Pattern, which contain *tokens* representing public keys or Diffie-Hellman operations is called a *handshake message*. After handshake messages, *transport messages* may occur carrying encrypted payloads. Here is an overview of the tokens that may appear in a handshake message:

- e, s. The sender is communicating their ephemeral or static public key, respectively.

Syntax

$k ::=$	public DH keys
$\quad e$	$\quad$ ephemeral DH key
$\quad s$	$\quad$ static DH key
$t ::=$	tokens
$\quad k$	$\quad$ public DH key
$\quad k_1 k_2$	$\quad$ shared DH secret (ee, es, se, or ss)
$\quad psk$	$\quad$ pre-shared key
$p ::=$	pre-messages
$\quad \epsilon$	$\quad$ end of pre-message (empty)
$\quad k, p$	$\quad$ pre-message with public DH key
$m ::=$	messages
$\quad \epsilon$	$\quad$ end of message (empty)
$\quad t, m$	$\quad$ message with token
$h_r ::=$	handshake (responder's turn)
$\quad \epsilon$	$\quad$ end of handshake
$\quad \overset{m}{\longleftarrow} h_i$	$\quad$ responder message, then initiator's turn
$h_i ::=$	handshake (initator's turn)
$\quad \epsilon$	$\quad$ end of handshake
$\quad \overset{m}{\longrightarrow} h_r$	$\quad$ initiator message, then responder's turn
$n ::=$	noise patterns
$\quad \overset{p_1}{\longrightarrow} \overset{p_2}{\longleftarrow} h_i$	$\quad$ pre-messages, then handshake

Figure 3.3: Noise Handshake Pattern Syntax.

- ee, es, se, ss**.** The sender has locally calculated a new shared secret. The first letter of the token indicates the initiator's key share while the second indicates the responder's key share. As such, this token remains the same irrespective of who is sending the particular handshake message in which it occurs.
- psk**.** The sender is mixing a pre-shared key into their local state and the recipient is assumed to do the same.

Optionally, certain key materials can be communicated before a protocol session is initiated. A practical example of how this is useful could be secure messaging protocols, where prior knowledge of an ephemeral key pair could help a party initiate a session using a zero-round-trip protocol, which allows them to send an encrypted payload without the responder needing to be online.

These *pre-message patterns* are represented by a series of messages occuring before handshake messages. The end of the pre-message stage is indicated by a "..." sign. For example, in Fig. 3.1, we see a pre-message pattern indicating that the initiator has prior knowledge of the responder's public static key before initiating a protocol session.

3.2.1 Validating Noise Handshake Pattern Syntax

Noise Handshake Patterns come with certain validity rules:

- **Alternating message directions.** Message direction within a Noise Handshake Pattern must alternate (initiator → responder, initiator ← responder), with the first message being sent by the initiator.

Validity Rules

$d ::= \leftarrow \mid \rightarrow$ direction
$\bar{t} ::= k^d \mid k_1 k_2 \mid psk$ tokens w. DH keys
$\Gamma ::= \{\bar{t}_0, \ldots, \bar{t}_n\}$ context
$tokens^d(m) \triangleq \{k^d \mid k \in m \cap \{e, s\}\} \cup (m \setminus \{e, s\})$

Pre-Message Validity: $\Gamma \vdash^d p$

$$\text{PreEmpty}\ \frac{}{\Gamma \vdash^d \epsilon}$$

$$\text{PreKey}\ \frac{k^d \notin \Gamma \quad \Gamma \cup \{k^d\} \vdash^d p}{\Gamma \vdash^d k, p}$$

Message Validity: $\Gamma \vdash^d m$

$$\text{MsgEmpty}^{\rightarrow}\ \frac{ss \in \Gamma \Rightarrow se \in \Gamma \quad se \in \Gamma \Rightarrow ee \in \Gamma \quad psk \in \Gamma \Rightarrow e^{\rightarrow} \in \Gamma}{\Gamma \vdash^{\rightarrow} \epsilon}$$

$$\text{MsgEmpty}^{\leftarrow}\ \frac{ss \in \Gamma \Rightarrow es \in \Gamma \quad es \in \Gamma \Rightarrow ee \in \Gamma \quad psk \in \Gamma \Rightarrow e^{\leftarrow} \in \Gamma}{\Gamma \vdash^{\leftarrow} \epsilon}$$

$$\text{MsgKey}\ \frac{k^d \notin \Gamma \quad \Gamma \cup \{k^d\} \vdash^d m}{\Gamma \vdash^d k, m}$$

$$\text{MsgDH}\ \frac{k_1^{\rightarrow} \in \Gamma \quad k_2^{\leftarrow} \in \Gamma \quad k_1 k_2 \notin \Gamma \quad \Gamma \cup \{k_1 k_2\} \vdash^d m}{\Gamma \vdash^d k_1 k_2, m}$$

$$\text{MsgPSK}\ \frac{psk \notin \Gamma \quad \Gamma \cup \{psk\} \vdash^d m}{\Gamma \vdash^d psk, m}$$

Handshake Validity: $\Gamma \vdash h_i$

$$\text{HSEmpty}\ \frac{}{\Gamma \vdash \epsilon}$$

$$\text{HSMessageI}\ \frac{\Gamma \vdash^{\rightarrow} m \quad \Gamma \cup tokens^{\rightarrow}(m) \vdash h_r}{\Gamma \vdash \xrightarrow{m} h_r}$$

$$\text{HSMessageR}\ \frac{\Gamma \vdash^{\leftarrow} m \quad \Gamma \cup tokens^{\leftarrow}(m) \vdash h_i}{\Gamma \vdash \xleftarrow{m} h_i}$$

Noise Pattern Validity: $\vdash n$

$$\text{NoiseValid}\ \frac{\{\} \vdash^{\rightarrow} p_1 \quad \{\} \vdash^{\leftarrow} p_2 \quad tokens^{\rightarrow}(p_1) \cup tokens^{\leftarrow}(p_2) \vdash h_i}{\vdash \xrightarrow{p_1} \xleftarrow{p_2} h_i}$$

Figure 3.4: Noise Pattern Validity Rules

- **Performing Diffie-Hellman key agreement more than once.** Principals must not perform the same Diffie-Hellman key agreement more than once per handshake.
- **Sending keys more than once.** Principals must not send their static public key or ephemeral public key more than once per handshake.
- **Transport messages after handshake messages.** Noise Handshake Patterns can only contain transport handshake messages at the very bottom of the pattern.
- **Appropriate key share communication.** Principals cannot perform a Diffie-Hellman operation with a key share that was not communicated to them prior.
- **Unused key shares.** Noise Handshake Patterns should not contain key shares that are not subsequently used in any Diffie-Hellman operation.
- **Transport messages.** Noise Handshake Patterns cannot consist purely of transport messages.

The Noise Handshake Pattern syntax is more formally described in Fig. 3.3, while validity rules are formalized in Fig. 3.4.

3.2.2 Local State

Each principal in a Noise protocol handshake keeps three local state elements: `CipherState`, `SymmetricState` and `HandshakeState`. These states contain each other in a fashion similar to a Russian Matryoshka doll, with `HandshakeState` being the largest element, containing `SymmetricState` which in turn contains `CipherState`.

- **`CipherState`** contains k (a symmetric key) and n (a nonce), used to encrypt and decrypt ciphertexts.
- **`SymmetricState`** contains a `CipherState` tuple (k, n), an additional key ck and a hash function output h.
- **`HandshakeState`** contains a `SymmetricState` along with additional local public keys (s, e) and remote public keys (rs, re).

Each state element comes with its own set of state transformation functions. These functions are triggered by the occurence and position of tokens within a Noise Handshake Pattern. We present a description of the state transition functions as seen in the Noise Protocol Framework specification, but restricted to a representation that follows implementing Noise Handshake Patterns in the symbolic model.

CipherState

A `CipherState` comes with the following state transition functions:

- **`InitializeKey(key):`** Sets $k =$ `key`. Sets $n = 0$.
- **`HasKey():`** Returns true if k is non-empty, false otherwise.
- **`SetNonce(nonce):`** Sets $n =$ `nonce`.
- **`EncryptWithAd(ad, p):`** If k is non-empty returns $\mathsf{E}(k, n, ad, p)$ then increments n. Otherwise returns p.
- **`DecryptWithAd(ad, c):`** If k is non-empty returns $\mathsf{D}(k, n, ad, c)$ then increments n. Otherwise returns c. n is not incremented if authenticated decryption fails.
- **`Rekey():`** Sets $k = \mathsf{R}(k)$.

In ProVerif, `InitializeKey` simply returns a `cipherstate`-type value packed with the input key and a starting nonce. `hasKey` unpacks an input `cipherstate` and checks whether the key is defined. The rest of the functions are based on similarly evident constructions:

```
letfun encryptWithAd(cs:cipherstate, ad:bitstring, plaintext:bitstring) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let e = encrypt(k, n, ad, plaintext) in
      let csi = setNonce(cs, increment_nonce(n)) in
      (csi, e).

letfun decryptWithAd(cs:cipherstate, ad:bitstring, ciphertext:bitstring) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let d = decrypt(k, n, ad, ciphertext) in
      let (valid:bool, adi:bitstring, plaintext:bitstring) = aeadunpack(d) in
      let csi = setNonce(cs, increment_nonce(n)) in
      (csi, plaintext, valid).
```

```
letfun reKey(cs:cipherstate) =
      let (k:key, n:nonce) = cipherstateunpack(cs) in
      let ki = encrypt(k, maxnonce, empty, zero) in
      cipherstatepack(bit2key(ki), n).
```

SymmetricState

A `SymmetricState` comes with the following state transition functions:

- **`InitializeSymmetric(name):`** Sets $ck = h = \mathsf{H}(\texttt{name})$.
- **`MixKey(ik):`** Sets $(ck, tk) = \mathsf{HKDF}(ck, \texttt{ik})$ and calls `InitializeKey`(tk).
- **`MixHash(data):`** Sets $h = \mathsf{H}(h \parallel \texttt{data})$.[4]
- **`MixKeyAndHash(ik):`** Sets $(ck, th, tk) = \mathsf{HKDF}(ck, \texttt{ik})$, then calls `MixHash`($th$) and `InitializeKey`($tk$).
- **`GetHandshakeHash():`** Returns h.
- **`EncryptAndHash(p):`** Sets c = `EncryptWithAd`(h,`p`). Calls `MixHash`(c) and returns c.
- **`DecryptAndHash(c):`** Sets p = `DecryptWithAd`(h,`c`). Calls `MixHash`(c) and returns c and returns p.
- **`Split():`** Sets $(tk_1, tk_2) = \mathsf{HKDF}(ck, \texttt{zero})$. Creates two `CipherStates` (c_1, c_2). Calls c_1`.InitializeKey`(tk_1) and c_2`.InitializeKey`(tk_2). Returns (c_1, c_2), a pair of `CipherStates` for encrypting transport messages.[5]

In ProVerif, these functions are implemented based on `letfun` declarations that combine previously declared `fun`s and `letfun`s:

```
      letfun initializeSymmetric(protocol_name:bitstring) =
      let h = hash(protocol_name, empty) in
      let ck = bit2key(h) in
      let cs = initializeKey(bit2key(empty)) in
      symmetricstatepack(cs, ck, h).

letfun mixKey(ss:symmetricstate, input_key_material:key) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (ck:key, temp_k:key, output_3:key) = hkdf(ck, input_key_material) in
      symmetricstatepack(initializeKey(temp_k), ck, h).

letfun mixHash(ss:symmetricstate, data:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      symmetricstatepack(cs, ck, hash(h, data)).

letfun mixKeyAndHash(ss:symmetricstate, input_key_material:key) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (ck:key, temp_h:key, temp_k:key) = hkdf(ck, input_key_material) in
      let (cs:cipherstate, temp_ck:key, h:bitstring) = symmetricstateunpack(mixHash(
           symmetricstatepack(cs, ck, h), key2bit(temp_h))) in
      symmetricstatepack(initializeKey(temp_k), ck, h).

letfun getHandshakeHash(ss:symmetricstate) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      (ss, h).

letfun encryptAndHash(ss:symmetricstate, plaintext:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
```

[4] $\parallel$ denotes bitstring concatenation.

[5] `zero` is meant to denote a null bitstring.

```
      let (cs:cipherstate, ciphertext:bitstring) = encryptWithAd(cs, h, plaintext) in
      let ss = mixHash(symmetricstatepack(cs, ck, h), ciphertext) in
      (ss, ciphertext).

letfun decryptAndHash(ss:symmetricstate, ciphertext:bitstring) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (cs:cipherstate, plaintext:bitstring, valid:bool) = decryptWithAd(cs, h,
          ciphertext) in
      let ss = mixHash(symmetricstatepack(cs, ck, h), ciphertext) in
      (ss, plaintext, valid).

letfun split(ss:symmetricstate) =
      let (cs:cipherstate, ck:key, h:bitstring) = symmetricstateunpack(ss) in
      let (temp_k1:key, temp_k2:key, temp_k3:key) = hkdf(ck, bit2key(zero)) in
      let cs1 = initializeKey(temp_k1) in
      let cs2 = initializeKey(temp_k2) in
      (ss, cs1, cs2).
```

HandshakeState

A `HandshakeState` comes with the following state transition functions:

- **`Initialize(hp, i, s, e, rs, re)`:** hp denotes a valid Noise Handshake Pattern. i is a boolean which denotes whether the local state belongs to the initiator. Public keys (s, e, rs, re) may be left empty or may be pre-initialized in the event that any of them appeared in a pre-message. Calls `InitializeSymmetric(hp.name)`. Calls `MixHash()` once for each public key listed in the pre-messages within `hp`.
- **`WriteMessage(p)`:** Depending on the tokens present in the current handshake message, different operations occur:
  - *e*: Sets $e \leftarrow \mathsf{KP}()$. Appends g^e to the return buffer. Calls `MixHash`(g^e).
  - *s*: Appends `EncryptAndHash`(g^s) to the buffer.
  - *ee*: Calls `MixKey`$(\mathsf{DH}(e, re))$.
  - *es*: Calls `MixKey`$(\mathsf{DH}(e, rs))$ if initiator, `MixKey`$(\mathsf{DH}(s, re))$ if responder.
  - *se*: Calls `MixKey`$(\mathsf{DH}(s, re))$ if initiator, `MixKey`$(\mathsf{DH}(e, rs))$ if responder.
  - *ss*: Calls `MixKey`$(\mathsf{DH}(s, rs))$.

  Then, `EncryptAndHash(p)` is appended to the return buffer. If there are no more handshake messages, two new `CipherStates` are returned by calling `Split()`.
- **`ReadMessage(m)`:** Depending on the tokens present in the current handshake message, different operations occur:
  - *e*: Sets `re` to the public ephemeral key retrieved from `m`.
  - *s*: Sets `temp` to the encrypted public static key retrieved from `m`. Sets `rs` to the result of `DecryptAndHash(temp)`, failing on authenticated decryption error.
  - *ee*: Calls `MixKey`$(\mathsf{DH}(e, re))$.
  - *es*: Calls `MixKey`$(\mathsf{DH}(e, rs))$ if initiator, `MixKey`$(\mathsf{DH}(s, re))$ if responder.
  - *se*: Calls `MixKey`$(\mathsf{DH}(s, re))$ if initiator, `MixKey`$(\mathsf{DH}(e, rs))$ if responder.
  - *ss*: Calls `MixKey`$(\mathsf{DH}(s, rs))$.

  Then, `DecryptAndHash` is called on the message payload extracted from m. If there are no more handshake messages, two new `CipherStates` are returned by calling `Split()`.

```
letfun writeMessage_a(me:principal, them:principal, hs:handshakestate, payload:bitstring, sid:sessionid) =
let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in
let (ne:bitstring, ciphertext1:bitstring, ciphertext2:bitstring) = (empty, empty, empty) in
let e = generate_keypair(key_e(me, them, sid)) in
let ne = key2bit(getpublickey(e)) in
let ss = mixHash(ss, ne) in
(* No PSK, so skipping mixKey *)
let ss = mixKey(ss, dh(e, rs)) in
let s = generate_keypair(key_s(me)) in
let (ss:symmetricstate, ciphertext1:bitstring) = encryptAndHash(ss, key2bit(getpublickey(s))) in
let ss = mixKey(ss, dh(s, rs)) in
let (ss:symmetricstate, ciphertext2:bitstring) = encryptAndHash(ss, payload) in
let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
let message_buffer = concat3(ne, ciphertext1, ciphertext2) in
(hs, message_buffer).
```

```
letfun readMessage_a(me:principal, them:principal, hs:handshakestate, message:bitstring, sid:sessionid) =
let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in
let (ne:bitstring, ciphertext1:bitstring, ciphertext2:bitstring) = deconcat3(message) in
let valid1 = true in
let re = bit2key(ne) in
let ss = mixHash(ss, key2bit(re)) in
(* No PSK, so skipping mixKey *)
let ss = mixKey(ss, dh(s, re)) in
let (ss:symmetricstate, plaintext1:bitstring, valid1:bool) = decryptAndHash(ss, ciphertext1) in
let rs = bit2key(plaintext1) in
let ss = mixKey(ss, dh(s, rs)) in
let (ss:symmetricstate, plaintext2:bitstring, valid2:bool) = decryptAndHash(ss, ciphertext2) in
if ((valid1 && valid2) && (rs = getpublickey(generate_keypair(key_s(them))))) then (
let hs = handshakestatepack(ss, s, e, rs, re, psk, initiator) in
(hs, plaintext2, true)).
```

Figure 3.5: The `WriteMessage` and `ReadMessage` `letfun` constructions for the first message in `IK` (Fig. 3.1), generated according to translation rules from Noise Handshake Pattern to ProVerif. The appropriate state transition functions are invoked in accordance with the occurence and ordering of tokens in the message pattern.

3.2.3 Dynamically Generating ReadMessage and WriteMessage Functions in the Applied-Pi Calculus

In Noise Explorer (our analysis framework for Noise Handshake Patterns), cryptographic primitives and state transition functions are included from a pre-existing set of Noise Protocol Framework ProVerif headers written as a part of this work and are not automatically generated according to a set of rules. Events, queries, protocol processes and the top-level process, however, are fully generated using translation rules that make them unique for each Noise Handshake Pattern.

In our generated ProVerif models, each handshake message and transport message is given its own `WriteMessage` and `ReadMessage` construction represented as `letfun`s. These functions are constructed to invoke the appropriate state transition functions depending on the tokens included in the message pattern being translated. An example generated translation can be see in Fig. 3.5, which concerns the first message in `IK` (Fig. 3.1): $\rightarrow e, es, s, ss$.

The state transition rules described in Noise Protocol Framework specification are implicated by the tokens within the message pattern. By following these rules, Noise Explorer generates a symbolic model that implements the state transitions relevant to this particular message pattern. From the initiator's side:

- *e*: Signals that the initiator is sending a fresh ephemeral key share as part of this message. This token adds one state transformation to `writeMessage_a`: `mixHash`, which hashes the new key into the session hash.
- *es*: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds one state transformation to `writeMessage_a`: `mixKey`, which calls the HKDF using as input the existing `SymmetricState` key and DH(e, rs), the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.
- *s*: Signals that the initiator is sending a static key share as part of this message. This token adds one state transformation to `writeMessage_a`: `encryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.
- *ss*: Signals that the initiator is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds one state transformation to `writeMessage_a`: `mixKey`, which calls the HKDF function using, as input, the existing `SymmetricState` key, and DH(s, rs), the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.

Message A's payload, which is modeled as the output of the function `msg_a(initiatorIdentity, responderIdentity, sessionId)`, is encrypted as `ciphertext2`. This invokes `encryptAndHash`, which performs AEAD encryption on the payload, with the session hash as the associated data (`encryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

On the receiver end:

- *e*: Signals that the responder is receiving a fresh ephemeral key share as part of this message. This token adds one state transformation to `readMessage_a`: `mixHash`, which hashes the new key into the session hash.
- *es*: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's ephemeral key and the responder's static key as part of this message. This token adds one state transformation to `readMessage_a`: `mixKey`, which calls the HKDF function using, as input, the existing SymmetricState key, and DH(s, re), the Diffie-Hellman share calculated from the initiator's ephemeral key and the responder's static key.
- *s*: Signals that the responder is receiving a static key share as part of this message. This token adds one state transformation to `readMessage_a`: `decryptAndHash` is called on the static public key. If any prior Diffie-Hellman shared secret was established between the sender and the recipient, this allows the initiator to communicate their long-term identity with some degree of confidentiality.
- *ss*: Signals that the responder is calculating a Diffie-Hellman shared secret derived from the initiator's static key and the responder's static key as part of this message. This token adds one state transformation to `readMessage_a`: `mixKey`, which calls HKDF function using, as input, the existing `SymmetricState` key and DH(s, rs), the Diffie-Hellman share calculated from the initiator's static key and the responder's static key.

Message A's payload invokes the following operation: `decryptAndHash`, which performs AEAD decryption on the payload, with the session hash as the associated data (`decryptWithAd`) and `mixHash`, which hashes the encrypted payload into the next session hash.

3.2.4 Other Specification Features

The Noise Protocol Framework specification defines 15 "fundamental patterns", 23 "deferred patterns" and 21 "PSK patterns". `IK` (Fig. 3.1) and `IN` (Fig. 3.2) are two fundamental patterns. Deferred patterns are essentially modified fundamental patterns where the communication of public keys or the occurence of Diffie-Hellman operations is intentionally delayed. PSK pattterns are patterns in which a pre-shared key token appears. Fig. 3.6 illustrates a deferred pattern based on the fundamental pattern shown in Fig. 3.1.

The full Noise Protocol Framework specification extends somewhat beyond the description given as part of this work, including features such as "identity hiding" and "dummy keys." Some of these features are potentially valuable and slated as future work.

$I1K$:

$\leftarrow s$

$\cdots$

$\rightarrow e, es, s$

$\leftarrow e, ee$

$\rightarrow se$

Figure 3.6: An example Noise Handshake Pattern, `I1K`. This is a deferred pattern based on `IK`, shown in Fig. 3.1.

3.3 Modeling Noise Security Goals in the Symbolic Model

Since our goal is to evaluate the security guarantees achieved by arbitrary Noise Handshake Patterns, it is crucial to have a set of well-defined security goals on which to base our analysis. We want to formulate these "security grades" in ProVerif as event-based queries. This implies specifying a number of events triggered at specific points in the protocol flow as well as queries predicated on these events.

A set of the queries for the security goals described in this section is generated for each handshake and transport message within a Noise Handshake Pattern, allowing for verification to occur in the comprehensive context described in §3.1.

The Noise Protocol Framework specification defines different Noise Handshake Patterns to suit different scenarios. These patterns come with different security properties depending on which keys and shared secrets are employed and when. Two types of security grades are defined: *"authentication"* grades dealing with the authentication of a message to a particular sender (and optionally, receiver) and *"confidentiality"* grades dealing with a message's ability to resist the obtention of plaintext by an unauthorized party.

For example, the Noise Handshake Pattern illustrated in Fig. 3.1 is described in the original specification as claiming to reach strong security goals: handshake and transport message are attributed authentication grades of 1, 2, 2 and 2 respectively, and confidentiality grades of 2, 4, 5 and 5. Other Noise Handshake Patterns, such as the one described in Fig. 3.2, sacrifice security properties to deal away with the need to share public keys beforehand or to conduct additional key derivation steps (authentication: 0, 0, 2, 0 and confidentiality: 0, 3, 1 5.)

In our analysis, we leave the confidentiality grades intact. However, we introduce two new additional security grades, 3 and 4, which provide more nuance for the existing authentication grades 1 and 2. In our analysis, authentication grades 1 and 2 hold even if the authentication of the message can be forged towards the recipient if the sender carries out a separate session with a separate, compromised recipient. Authentication grades 3 and 4 do not hold in this case. This nuance does not exist in the authentication grades defined in the latest Noise Protocol Framework specification.

In all examples below, Bob is the sender and Alice is the recipient. The message in question is message D, i.e. the fourth message pattern within the Noise Handshake Pattern. In the event of a non-existent static key for either Alice or Bob, or of a non-existent PSK, the relevant `LeakS` or `LeakPsk` event is removed from the query. A principal *c* refers to any arbitrary principal on the network, which includes compromised principal Charlie.

3.3.1 Events

The following events appear in generated ProVerif models:

- **`SendMsg(principal, principal, stage, bitstring)`** takes in the identifier of the message sender, the identifier of the recipient, a "stage" value and the plaintext of the message payload. The "stage" value is the output of a function parametrized by the session ID, a unique value generated for each execution of the protocol using ProVerif's `new` keyword, and an identifier of which message this is within the Noise Handshake Pattern (first message, second message, etc.)

- **`RecvMsg(principal, principal, stage, bitstring)`** is a mirror event of the above, with the first principal referring to the recipient and the second referring to the sender.

- **`LeakS(phasen, principal)`** indicates the leakage of the long-term secret key of the principal. `phasen` refers to which "phase" the leak occured: in generated ProVerif models, phase 0 encompasses protocol executions that occur while the session is under way, while phase 1 is strictly limited to events that occur after the session has completed and has been closed.

- **`LeakPsk(phasen, principal, principal)`** indicates the leakage of the pre-shared key (PSK) of the session between an initiator (specified as the first principal) and a responder in the specified phase.

3.3.2 Authentication Grades

Grade 0 indicates no authentication: the payload may have been sent by any party, including an active attacker.

Sender authentication

In this query, we test for sender authentication and message integrity. If Alice receives a valid message from Bob, then Bob must have sent that message to someone, or Bob had their static key compromised before the session began, or Alice had their static key

compromised before the session began:

$$\begin{aligned}
&RecvMsg(alice, bob, stage(d, sid), m) \longrightarrow \\
&SendMsg(bob, c, stage(d, sid), m) \vee \\
&(LeakS(phase_0, bob) \wedge LeakPsk(phase_0, alice, bob)) \vee \\
&(LeakS(phase_0, alice) \wedge LeakPsk(phase_0, alice, bob))
\end{aligned}$$

Sender authentication and key compromise impersonation resistance

In this query, we test for sender authentication and Key Compromise Impersonation resistance. If Alice receives a valid message from Bob, then Bob must have sent that message to someone, or Bob had their static key compromised before the session began.

$$\begin{aligned}
&RecvMsg(alice, bob, stage(d, sid), m) \longrightarrow \\
&SendMsg(bob, c, stage(d, sid), m) \vee \\
&LeakS(phase_0, bob)
\end{aligned}$$

Sender and received authentication and message integrity

If Alice receives a valid message from Bob, then Bob must have sent that message to Alice specifically, or Bob had their static key compromised before the session began, or Alice had their static key compromised before the session began. This query is not present in the original Noise Protocol Framework specification and is contributed by this work.

$$\begin{aligned}
&RecvMsg(alice, bob, stage(d, sid), m) \longrightarrow \\
&SendMsg(bob, alice, stage(d, sid), m) \vee \\
&(LeakS(phase_0, bob) \wedge LeakPsk(phase_0, alice, bob)) \vee \\
&(LeakS(phase_0, alice) \wedge LeakPsk(phase_0, alice, bob))
\end{aligned}$$

Sender and receiver authentication and key compromise impersonation resistance

If Alice receives a valid message from Bob, then Bob must have sent that message to Alice specifically, or Bob had their static key compromised before the session began. This query is not present in the original Noise Protocol Framework specification and is contributed by this work.

$$\begin{aligned}
&RecvMsg(alice, bob, stage(d, sid), m) \longrightarrow \\
&SendMsg(bob, alice, stage(d, sid), m) \vee \\
&LeakS(phase_0, bob)
\end{aligned}$$

3.3.3 Confidentiality Grades

Grade 0 indicates no confidentiality: the payload is sent in cleartext.

Encryption to an ephemeral recipient

In these queries, we test for message secrecy by checking if a passive attacker or active attacker is able to retrieve the payload plaintext only by compromising Alice's static key either before or after the protocol session. Passing this query under a passive attacher achieves confidentiality grade 1, while doing so under an active attacker

achieves confidentiality grade 2 (encryption to a known recipient, forward secrecy for sender compromise only, vulnerable to replay.)

$$attacker_{p1}(msg_d(bob, alice, sid)) \longrightarrow$$
$$(LeakS(phase_0, alice) \lor LeakS(phase_1, alice)) \land$$
$$(LeakPsk(phase_0, alice, bob) \lor$$
$$LeakPsk(phase_1, alice, bob))$$

In the above, $attacker_{p1}$ indicates that the attacker obtains the message in phase 1 of the protocol execution.

Encryption to a known recipient, weak forward secrecy

In this query, we test for forward secrecy by checking if a passive attacker is able to retrieve the payload plaintext only by compromising Alice's static key before the protocol session, or after the protocol session along with Bob's static public key (at any time.) Passing this query under a passive attacker achieves confidentiality grade 3, while doing so under an active attacker achieves confidentiality grade 4 (encryption to a known recipient, weak forward secrecy only if the sender's private key has been compromised.)

$$attacker_{p1}(msg_d(bob, alice, sid)) \longrightarrow$$
$$(LeakS(phase_0, alice) \land LeakPsk(phase_0, alice, bob)) \lor$$
$$(LeakS(p_x, alice) \land LeakPsk(p_y, alice, bob) \land$$
$$LeakS(p_z, bob))$$

In the above, p_x, p_y, p_z refer to any arbitrary phases.

Encryption to a known recipient, strong forward secrecy

In this query, we test for strong forward secrecy by checking if an active attacker is able to retrieve the payload plaintext only by compromising Alice's static key before the protocol session. Passing this query achieves confidentiality grade 5.

$$attacker_{p1}(msg_d(bob, alice, sid)) \longrightarrow$$
$$(LeakS(phase_0, alice) \land LeakPsk(phase_0, alice, bob))$$

3.3.4 Limitations on Modeling Security Grades

Our analysis of authentication grades comes with an important limitation: When Noise Explorer generates the authentication queries below, it uses two different values, sid_a and sid_b, to refer to the session ID as registered in the trigger events by Alice and Bob. This differs from, and is in fact less accurate than the queries described below, which use the same session ID, sid, for both Alice and Bob. We are forced to adopt this approach due to performance limitations in our models during verification should we choose to use a single sid value for both Alice and Bob. However, we argue that since processes with differing sid values cause decryption operations that use shared secrets derived from ephemeral keys to fail, and therefore for those processes to halt, we still obtain essentially the same verification scenarios that these queries target.

Additionally, with regards to our confidentiality grades, whenever a pattern contains a PSK and `LeakPSK` events start to get involved, we ideally account for cases

Patt.	Auth.	Conf.
N	0	2
K	1	2
X	1	2
NN	0 0 0	0 1 1
NK	0 2 0	2 1 5
NX	0 2 0	0 1 5
XN	0 0 2 0	0 1 1 5
XK	0 2 4 4 4	2 1 5 5 5
XX	0 2 4 4 4	0 1 5 5 5
KN	0 0 2 0	0 3 1 5
KK	1 4 4 4	2 4 5 5
KX	0 4 4 4	0 3 5 5
IN	0 0 2 0	0 3 1 5
IK	1 4 4 4	2 4 5 5
IX	0 4 4 4	0 3 5 5
NK1	0 2 0	0 1 5
NX1	0 0 0 2 0	0 1 3 1 5

Patt.	Auth.	Conf.
X1N	0 0 0 0 2	0 1 1 3 1
X1K	0 2 0 4 4 4	2 1 5 3 5 5
XK1	0 2 4 4 4	0 1 5 5 5
X1K1	0 2 0 4 4 4	0 1 5 3 5 5
X1X	0 2 0 2 2 2	0 1 5 3 5 5
XX1	0 0 4 4 4	0 1 3 5 5
X1X1	0 0 0 4 4 4	0 1 3 3 5 5
K1N	0 0 2 0	0 1 1 5
K1K	0 4 4 4 4	2 1 5 5 5
KK1	0 4 4 4	0 3 5 5
K1K1	0 4 4 4	0 1 5 5 5
K1X	0 4 4 4 4	0 1 5 5 5
KX1	0 0 4 4 4	0 3 3 5 5
K1X1	0 0 4 4 4	0 1 3 5 5
I1N	0 0 2 0 2	0 1 1 5 1
I1K	0 4 4 4 4	2 1 5 5 5
IK1	0 4 4 4	0 3 5 5

Patt.	Auth.	Conf.
I1K1	0 4 4 4 4	0 1 5 5 5
I1X	0 4 4 4 4	0 1 5 5 5
IX1	0 0 4 4 4	0 3 3 5 5
I1X1	0 0 4 4 4	0 1 3 5 5
Npsk0	1	2
Kpsk0	1	2
Xpsk1	1	2
NNpsk0	1 1 1 1	2 3 3 3
NNpsk2	0 1 1 1	0 3 3 3
NKpsk0	1 4 1 4	2 5 3 5
NKpsk2	0 4 1 4	0 3 3 5
NXpsk2	0 4 1 4	0 3 3 5
XNpsk3	0 0 4 1 4	0 1 3 3 5
XKpsk3	0 0 4 4 4	0 1 3 5 5
KNpsk0	1 1 4 1	2 3 5 3
KNpsk2	0 1 4 1	0 3 5 3
INpsk1	1 1 4 1	2 3 5 3

Figure 3.7: Verification results for 50 Noise Handshake Patterns.

where one long-term secret is compromised but not the other. This indicates that we may need a richer notion of authenticity and confidentiality grades than the 1-5 markers that the Noise specification provides. For consistency, we are still using the old grades, but to truly understand and differentiate the security provided in many cases, we recommend that the user view the detailed queries and results as generated by Noise Explorer and available in its detailed rendering of the verification results.

3.4 Verifying Arbitrary Noise Handshake Patterns with Noise Explorer

A central motivation to this work is the obtention of a general framework for designing, reasoning about, formally verifying and comparing any arbitrary Noise Handshake Pattern. Noise Explorer is a web framework that implements all of the formalisms and ProVerif translation logic described so far in this work in order to provide these features.

Noise Explorer is ready for use by the general public today at `https://noiseexplorer.com`. Here are Noise Explorer's main functionalities:

Designing and validating Noise Handshake Patterns. This allows protocol designers to immediately obtain validity checks that verify if the protocol conforms to the latest Noise Protocol Framework specification.[6]

Generating cryptographic models for formal verification using automated verification tools. Noise Explorer can compile any Noise Handshake Pattern to a full representation in the applied-pi calculus including cryptographic primitives, state machine transitions, message passing and a top-level process illustrating live protocol execution. Using ProVerif, we can then test against sophisticated security queries starting at basic confidentiality and authentication and extending towards forward secrecy and resistance to key compromise impersonation.

Exploring the first compendium of formal verification results for Noise Handshake

[6] As of writing, Revision 34 is the latest draft of the Noise Protocol Framework. Noise Explorer is continuously updated in collaboration with the authors of the Noise Protocol Framework specification.

Patterns. Since formal verification for complex Noise Handshake Patterns can take time and require fast CPU hardware, Noise Explorer comes with a compendium detailing the full results of all Noise Handshake Patterns described in the latest revision of the original Noise Protocol Framework specification. These results are presented with a security model that is more comprehensive than the original specification, as described in §3.3.

3.4.1 Accessible High Assurance Verification for Noise-Based Protocols

Noise Explorer users are free to specify any arbitrary Noise Handshake Pattern of their own design. Once this input is validated, formal verification models are generated. The ProVerif verification output can then be fed right back into Noise Explorer, which will then generate detailed interactive pages describing the analysis results.

The initial view of the results includes a pedagogical plain-English paragraph for each message summarizing its achieved security goals. For example, the following paragraph is generated for message D (i.e. the fourth message pattern) of `IK`:

*"Message D, sent by the responder, benefits from **sender and receiver authentication** and is **resistant to Key Compromise Impersonation**. Assuming the corresponding private keys are secure, this authentication cannot be forged. Message contents benefit from **message secrecy** and **strong forward secrecy**: if the ephemeral private keys are secure and the initiator is not being actively impersonated by an active attacker, message contents cannot be decrypted by an adversary."*

Furthermore, each message comes with a detailed analysis view that allows the user to immediately access a dynamically generated representation of the state transition functions for this particular message as modeled in ProVerif and a more detailed individual writeup of which security goals are met and why. We believe that this *"pedagogy-in-depth"* that is provided by the Noise Explorer web framework will allow for useful, push-button analysis of any constructed protocol within the Noise Protocol Framework that is comprehensive.

Noise Explorer's development was done in tandem with discussions with the Noise Protocol Framework author: pre-release versions were built around revision 33 of the Noise Protocol Framework and an update to support revision 34 of the framework was released in tandem with the specification revision draft. Revision 34 also included security grade results for deferred patterns that were obtained directly via Noise Explorer's compendium of formal analysis results. We plan to continue collaborating with the Noise Protocol Framework author indefinitely to support future revisions of the Noise Protocol Framework.

3.4.2 Noise Explorer Verification Results

Noise Explorer was used to generate ProVerif models for more than 50 Noise Handshake Patterns, all of which were subsequently verified with the results shown in Fig. 3.7. We found that all of the Noise Handshake Patterns evaluated met the security goals postulated in the original Noise Protocol Framework Specification. Verification times varied between less than 30 minutes for simpler (and less secure) patterns (such as `NN`) to more than 24 hours for some of the more ambitious patterns, such as `IK`. All of the results are accessible publicly using Noise Explorer's compendium interface[7]

[7] `https://noiseexplorer.com/patterns/`

and the official Noise Protocol Framework specification has updated in order to take our results into account.

3.5 Modeling for Forgery Attacks using Noise Explorer

Using ProVerif, we were able to test for the necessity of certain Noise Handshake Patterns and to document a forgery attack within certain Noise Handshake Patterns that becomes possible when these rules are not followed appropriately. Essentially, we can compose well-known attack vectors (invalid Diffie-Hellman key shares, repeated AEAD nonces) to attack patterns that rely only on static-static key derivation (`ss`) for authentication.

Consider the pattern `KXS` below:

$$
\begin{aligned}
KXS:& \\
&\rightarrow s \\
&\ldots \\
&\rightarrow e \\
&\leftarrow e, ee, s, ss
\end{aligned}
$$

This is a variation of the Noise Handshake Pattern `KX` that uses `ss` instead of `se`, and `es`, so it is a little more efficient while satisfying the same confidentiality and authentication goals. In particular, the responder can start sending messages immediately after the second message.

However, there is an attack if the responder does not validate ephemeral public values. Suppose a malicious initiator were to send an invalid ephemeral public key `e`, say $e = 0$. Then, because of how Diffie-Hellman operations work on X25519, the responder would compute $ee = 0$ and the resulting key would depend only on the static key `ss`. Note that while the responder could detect and reject the invalid public key, the Noise specification explicitly discourages this behavior.

Since the responder will encrypt messages with a key determined only by `ss` (with a nonce set to 0), the malicious initiator can cause it to encrypt two messages with the same key and nonce, which allows for forgery attacks. A concrete man-in-the-middle attack on this pattern is as follows:[8]

In the pre-message phase, A sends a public static key share s_A to B. In the first session:

1. A malicious C initiates a session with B where he pretends to be A. C sends $e = Z$ such that Z^x would evaluate to Z for any x. This effectively allows us to model for forcing an X25519 zero-value key share in the symbolic model.
2. B receives $e = Z$ and accepts a new session with:
   - $h_{B0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{B1} = h_{B0}$
   - $h_{B1} = \mathsf{H}(h_{B0}, s_A, e = Z)$
3. B generates re_1, computes $ee = Z$ and sends back $(re_1, ee = Z, s_B, ss_{AB}, msg_a)$ where s_B is encrypted with $ck_{B2} = \mathsf{H}(ck_{B1}, ee = Z)$ as the key, 0 as the nonce and $h_{B2} = \mathsf{H}(h_{B1}, re_1, ee = Z)$ as associated data.

[8] For simplicity, here we use H to represent the more complex key derivation and mixing functions.

4. msg_a is encrypted with $ck_{B3} = \mathsf{H}(ck_{B2}, ss_{AB})$ as the key, 0 as the nonce and $h_{B3} = \mathsf{H}(h_{B2}, s_B)$ as associated data.
5. C discards this session but remembers the encrypted message.

In a second session:

1. A initiates a session with B by sending e. So, at A:
   - $h_{A0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{A1} = h_{A0}$
   - $h_{A1} = \mathsf{H}(h_{A0}, s_A, e)$
2. C intercepts this message and replaces it with the invalid public key $Z = 0$.
3. B receives $e = Z$ and accepts a new session with:
   - $h_{B0} = \mathsf{H}(\texttt{pattern\_name})$
   - $ck_{B1} = h_{B0}$
   - $h_{B1} = \mathsf{H}(h_{B0}, s_A, e = Z)$
4. B generates re_2, computes $ee = Z$ and sends back $(re_2, ee = Z, s_B, ss_{AB}, msg_b)$ where s_B is encrypted with $ck_{B2} = \mathsf{H}(ck_{B1}, ee = Z)$ as the key, 0 as the nonce and $h_{B2} = \mathsf{H}(h_{B1}, re)$ as associated data.
5. msg_b is encrypted with $ck_{B3} = \mathsf{H}(ck_{B2}, ss_{AB})$ as the key, 0 as the nonce and $h_{B3} = \mathsf{H}(h_{B2}, s_B)$ as associated data.
6. C intercepts this response.

Notably, the encryption keys (ck_{B3}) and the nonces (0) used for msg_a in session 1 and msg_b in session 2 are the same. Hence, if the underlying AEAD scheme is vulnerable to the repeated nonces attack, C can compute the AEAD authentication key for ck_{B3} and tamper with msg_a and msg_b to produce a new message msg_c that is validly encrypted under this key. Importantly, C can also tamper with the associated data h_{B3} to make it match any other hash value.

C replaces the message with $(re = Z, ee = Z, s_B, ss_{AB}, msg_c)$ and sends it to A, where s_B is re-encrypted by C using ck_{B2} which it knows and msg_c is forged by C using the AEAD authentication key for ck_{B3}. A receives the message $(re = Z, ee = Z, s_B, ss_{AB}, msg_c)$ and computes $ck_{A2} = H(ck_{A1}, ee = Z)$ and $h_{A2} = H(h_{A1}, ee = Z)$. A then decrypts s_B. A then computes $ck_{A3} = H(ck_{A2}, ss_{AB})$ and $h_{A3} = H(h_{A2}, s_{AB})$ and decrypts msg_c. This decryption succeeds since $ck_{A3} = ck_{B3}$. The attacker C therefore has successfully forged the message and the associated data.

At a high level, the above analysis can be read as indicating one of three shortcomings:

1. **Using `ss` in Noise Handshake Patterns must be done carefully.** A Noise Handshake Pattern validation rule could be introduced to disallow the usage of `ss` in a handshake unless it is accompanied by `se` or `es` in the same handshake pattern.
2. **Diffie-Hellman key shares must be validated.** Implementations must validate incoming Diffie-Hellman public values to check that they are not one of the twelve known integers [97] which can cause a scalar multiplication on the X25519 curve to produce an output of 0.

3. **Independent sessions must be checked for AEAD key reuse.** Ephemeral and static public key values are mixed into the encryption key derivation step.

As a result of this analysis, revision 34 of the Noise Protocol Framework specification included clearer wording of the validity rule which would render the `KXS` pattern invalid:

After calculating a Diffie-Hellman shared secret between a remote public key (either static or ephemeral) and the local static key, the local party must not perform any encryptions unless it has also calculated a Diffie-Hellman key share between its local ephemeral key and the remote public key. In particular, this means that:

- *After an* `se` *or* `ss` *token, the initiator must not send a payload unless there has also been an* `ee` *or* `es` *token respectively.*
- *After an* `es` *or* `ss` *token, the responder must not send a payload unless there has also been an* `ee` *or* `se` *token respectively.*

3.6 Conclusion and Future Work

In this work, we have provided the first formal treatment of the Noise Protocol Framework. We translate our formalisms into the applied-pi calculus and use this as the basis for automatically generating models for the automated formal verification of arbitrary Noise Handshake Patterns. We coalesce our results into Noise Explorer, an online framework for pedagogically designing, validating, verifying and reasoning about arbitrary Noise Handshake Patterns.

Noise Explorer has already had an impact as the first automated formal analysis targeting any and all Noise Handshake Patterns. Verification results obtained from Noise Explorer were integrated into the original specification and refinements were made to the validation rules and security goals as a result of the scrutiny inherent to our analysis.

Ultimately, it is not up to us to comment on whether the Noise Protocol Framework presents a "good" framework, per se. However, we present confident results that its approach to protocol design allows us to cross a new bridge for not only designing and implementing more robust custom secure channel protocols, but also applying existing automated verification methodologies in new and more ambitious ways.

Future work could include the automated generation of computational models to be verified using CryptoVerif and of verified implementations of Noise Handshake Patterns. The scope of our formalisms could also be extended to include elements of the Noise Protocol Framework specification, such as queries to test for identity hiding.

Chapter 4

Formal Verification for Automated Domain Name Validation

This work was published in the proceedings for the 21st International Conference on Financial Cryptography and Data Security, 2017. It was completed with co-authors Karthikeyan Bhargavan and Antoine Delignat-Lavaud.

In this chapter, we formally specify, model and verify ACME using ProVerif. Against a classic symbolic protocol adversary, ACME achieves most of its stated security goals. Notably, we show that ACME's design allows it to resist a substantially stronger threat model than the ad-hoc protocols of traditional CAs that rely on bearer tokens (passwords, cookies, authorization strings) for authentication and domain validation, thanks to its stronger cryptographic credentials and to the binding between the clients identity and the validated domain.

Nevertheless, we still discover issues and weaknesses in ACME's domain validation and account recovery features, potentially amounting to user account compromise. We attempt to address in this chapter what seem to be open questions regarding ACME: how does ACME compare to the existing security model of the actual top real-world certificate authorities? How can we most fruitfully illustrate and formally verify its security properties and what can we prove about them?

Contributions Our contributions in this chapter consist of:

- **A survey of the domain validation practices of current CAs:** in §4.1, we survey the issuance process and infrastructure of 10 of the most popular certificate authorities. We observe that traditional CAs support multiple methods for assessing domain control, that rely on different security assumptions.

- **A threat model for domain validation:** in §4.2, we specify a high-level threat model for certificate issuance based on domain validation, which applies to both traditional CAs and ACME. We relate this threat model to the various domain control validation methods surveyed in §4.1. In §4.3, we demonstrate that ACME resists a stronger threat model than other CAs.

- **Formally specifying and verifying ACME:** in §4.3, we formally specify the ACME protocol within a symbolic model in the applied π-calculus that encodes the adversarial capabilities described in §4.2. We verify the main security goals of ACME using ProVerif. Although ACME is shown to be more resistant to attacks than ad-hoc CAs, we also discover weaknesses in ACME's domain validation and account recovery and propose countermeasures.

4.1 Current State of Domain Validation

A goal of this chapter is to establish a relationship between current domain validation practices in the real world and a more formal threat model on which we base our security results. We begin by taking a closer look into the network infrastructure, user authentication and domain validation protocols currently in use by traditional CAs.

Our panel of surveyed CA is selected from the data set of Delignat-Lavaud et al. [41], which uses machine learning to classify certificates issued by domain validation. Our CA panel covers about 85% of the collected domain validated certificates from 2014, which is consistent with the January 2015 market share data from the Netcraft SSL survey[1]. For each CA, we obtain a regular one year, single-domain certificate signature for a domain name that we own.

§3.2.2.4 of the CA/Browser Forum's Baseline Requirements allow for domain validation to occur in ten different ways, including over postal mail. Of these methods, only three are in popular use: validation via email, the setting of an arbitrary DNS record, or serving some HTTP value on the target domain.

4.1.1 Domain Validation Mechanisms

With ad-hoc CAs, user C authenticates its identity C_{pk} to CA A as a simple username/-password web login, with an option for account recovery via email. C can then request that A validate some domain $C_w \subset C^\star$, where $C^\star_w$ is the set of all domain names that C controls. A's flow with the various domain validation channels proceeds thus:

- *HTTP Identifier* A sends to C a nonce A_{N^C} via an HTTPS channel that C must then advertise at some agreed-upon location under C_w. A then accesses C_w using an unauthenticated, unencrypted HTTP connection to ensure that it can retrieve A_{N^C}. This identifier depends *both* on honest DNS resolution of the validated domain's A/A6 records and an untampered HTTP connection to the domain.

  In practice, we find that CAs that allow HTTP identifiers require the nonce to be written on a text file with a long random name in the root of the validated domain. An attacker able to respond to HTTP requests for such names may get a certificate without access to the domain's DNS records.

- *DNS identifier* A sends to C a nonce A_{DNS^C} via an HTTPS channel that C must then advertise at some agreed-upon `TXT` record under the DNS records of C_w. A then queries C_w's name servers using to ensure that it can retrieve A_{DNS^C}. This identifier is dependent on honest resolution of the TXT record.

  None of the CAs we surveyed advertises DNSSEC support to ensure this DNS resolution is indeed authentic. As an experiment, we set up a DNSSEC-enabled domain and configured our nameserver to send an invalid RRSig for the TXT record of the domain validation nonce for Comodo. The validation ultimately

[1] `https://www.netcraft.com/internet-data-mining/ssl-survey/`

completed, indicating that the use of DNSSEC does not currently prevent attacks against DNS-based domain validation by current CAs.

- *Email identifier A* sends to C a URI A_{URI^C} via an email to an address E_C that A presumes to belong to C. Accessing this URI causes A to issue the certificate for C_w. This identifier is dependent on the confidential transport of the email (which may be routed through third party SMTP servers that are not guaranteed to use TLS encryption) and honest DNS resolution of the validated domain's MX records.

  In practice, we observe that CAs use dangerous heuristics to generate a list of possible E_C that C can pick from: first, they presume that any email addresses that appear in the WHOIS records of C_w is controlled by C. A large majority of registrars provide WHOIS privacy services to defend against spam. Such services can easily obtain certificates for any of their customers' domains as validation email transit through their mail servers. Second, CAs' heuristics include generic names such as `postmaster`, `webmaster`, or `admin`. If the validated domain provides an email service for which users may chose their username, an attacker may register under one of those generic names and obtain a unauthorized certificate. Such attacks have been carried successfully in the past against public email services such as Hotmail.

Once one of the above identifiers succeeds in validating C's ownership of C_w to A, A issues the certificate and the protocol ends.

4.1.2 User Authentication and Domain Validation

CA	Identifiers	Email Recovery	Public Key Auth.	Per-CSR Check
AlphaSSL	Email	Yes	No	N/A
Comodo PositiveSSL	Email	Yes	No	Yes
DigiCert	Email	Yes	No	No
GeoTrust QuickSSL	Email	Yes	No	No
GlobalSign	HTTP, DNS, Email	Yes	No	No
GoDaddy SSL	HTTP, DNS	Yes	No	No
Let's Encrypt (ACME draft-1)	HTTP	Yes	Yes	No
Network Solutions	Email	Yes	No	No
RapidSSL	Email	Yes	No	Yes
SSL.com BasicSSL	HTTP, DNS, Email	Yes	No	N/A
StartCom StartSSL	Email	Yes	Yes	No

Figure 4.1: Popular CAs, their validation methods, whether they permit account recovery via email, whether they allow login via a public-key based approach (such as client certificates) and whether domain validation is carried out once for every certificate request, even for already-validated domain names.

While CAs are required to document their certificate issuance policies in Certificate Practice Statements [134, 135, 136, 137, 138, 139, 140, 141], we find that these state-

ments are not always accurate or complete (for instance, they typically provision for validation methods that are not offered in practice; the address heuristics for email-based validation is rarely listed exhaustively). Most ad-hoc CAs in our study favor email-based validation. Unlike HTTP and DNS identifiers, email identifiers effectively rely on a *read* capability challenge instead write access proof for *C*. In §4.2, we discuss how email identifiers are the weakest available form of identification given our threat model. In §4.3, we elaborate on a weakness in ACME affecting both account recovery and domain validation. While this weakness is also generalizable to traditional certificate authorities, ACME offers an opportunity for a stronger fix.

None of CAs we surveyed offers a login mechanism that is completely independent of email. An exception almost occurs with StartSSL, which supports browser-generated X.509 client certificates for web login, but this exception is negated by the email-based account recovery in case of a lost certificate private key. Reliance on the security of the email channel can in many cases be even more serious: in many surveyed CAs, simply being able to complete a web login enables user to re-issue certificates for domains they had already validated before, without further validation.

A scan of the DNS MX and NS records of the web's top 10,000 websites (according to Alexa.com) [142] showed that roughly 45% of surveyed domain names used only six DNS providers, of which CloudFlare alone had a 18% share.[2] A similar centralization of authority exists with email, where the top six providers serve more than 55% of domain names surveyed, with Google alone holding roughly 27% market share (Figure 4.2.)

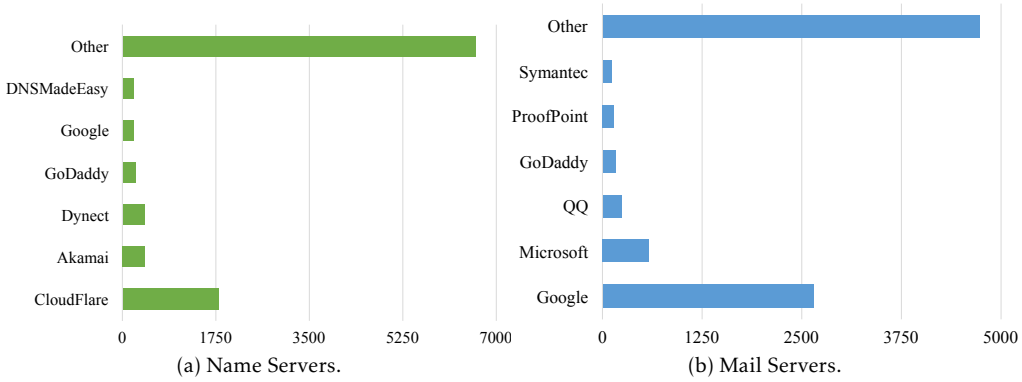

Figure 4.2: Provider repartition among the Alexa Top 10,000 global sites, as of October 2016. Notably, CloudFlare and Akamai also provide CDN services to domains under their name servers, allowing them stronger control over HTTP traffic.

These results suggest that the number of actors of which the compromise could affect traditional domain validation is significantly small. This is relevant given how top CAs allow for account recovery, certificate re-issuance and more with simple email-based validation.

[2]CloudFlare incidentally also operates Let's Encrypt's infrastructure, rendering it a centralized point of failure for ACME and ad-hoc CAs alike. While ACME is a centralization-agnostic protocol, Let's Encrypt operates with a fully centralized infrastructure.

4.2 A Security Model for Domain Validation

The protocols considered in this chapter operate between a party C claiming to serve and represent one or more domain names C_w (for which it wants certificates) and it is incumbent upon a certificate issuer A to verify that all domains in C_w are indeed controlled and managed by C. User C authenticates itself to the certificate authority A using a public key C_{pk} corresponding to the private signing key C_k. C can then link identifiers under C_k that prove that it manages and controls domains in C_w.

This and following sections are largely based on our full symbolic model[3] of ACME and ad-hoc CA protocol and network flow, which is written in the applied pi calculus and verified using ProVerif. Excerpts of this model are inlined throughout.

4.2.1 Security Goals and Threat Model

Our security goals are straightforward: for any domain $C_w \in C$, A must not sign a certificate asserting C's ownership of C_w for that domain unless C can validate C_{pk} as representing the identity that owns and manages this domain. ACME allows C to validate C_w with respect to C_{pk} by using the secret value C_k in order to demonstrate either read or write capabilities on certain pre-defined network channels, each with its own security model. A domain name C_w is considered *validated* under C_w if C_k can be used to complete a verification challenge on one of the network channels offered by the ACME protocol between C and A that in consequence asserts a relationship between C_{pk} and C_w.

The network topology, channels and actors are essentially the same for both ACME and ad-hoc CAs. However, the manner in which these actors communicate over the channels is different and leads to different attempts to establish the same security guarantees.

Channels

Intuitively, the channels we want encapsulate the following properties:

- **HTTPS Channel** Intuitively a regular web channel, we treat it as a A-authenticated duplex channel whereupon anyone can send a request to A, only A can read this request and respond, and only the sender can read this response.
- **Strong Identifier Channels** These channels must be assumed to be writable only by C. They are therefore relevant for HTTP and DNS Identifiers.
- **Weak Identifier Channel** Anyone can write to this channel, but only C can read from it. This makes it relevant for domain validation via email identifiers.

A shared consideration between ACME and ad-hoc CAs involves the critical importance of DNS resolution: if the attacker can simply produce false DNS responses for A resolving a domain request for any domain in C_w, it becomes impossible to safely carry out domain validation under any circumstances. As a sidenote, this allows us to argue that since the DNS channel must be trusted, it could also be considered as the safest channel on which to carry out domain validation using DNS Identifiers since that would allow C to avoid needlessly involving other channels.

[3] Full model available at https://github.com/Inria-Prosecco/acme-model

In formally describing our network model in ProVerif, we simulate simultaneous requests from Alice, Bob and Mallory as independent clients C. Alice and Bob both act as honest clients, while Mallory acts as a compromised participant client. All three follow the same protocol top-level process. We also simulate two independent ACME CAs, which interchangeably assume the role of A. For each C, we specify a triple of distinct channels:

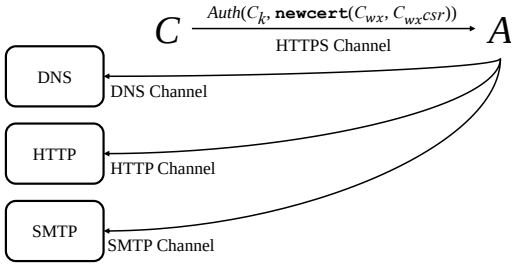

Figure 4.3: Channels overview.

$$(C_{HTTP}, C_{EMAIL}, C_{DNSTXT})$$

Each channel represents access to a different domain validation mechanism. While C is given complete access over these channels, C_{EMAIL} is only handed to A after being applied through a "write transformation" which returns a variant of the channel that is effectively write-only:

$$w(channel) \rightarrow channel$$

Similarly, a "read transformation" $r(channel) \rightarrow channel$ is applied to C_{HTTP} and C_{DNSTXT}.

A routing proxy is then specified in order to model the transportation across these channels by executing the following unbounded processes in parallel[4]:

$$in(w(C_{EMAIL}), x);\ out(r(C_{EMAIL}), x)$$
$$in(pub, x);\ out(r(C_{EMAIL}), x)$$
$$in(w(C_{HTTP}), x); out(pub, x); out(r(C_{HTTP}), x)$$
$$in(w(C_{DNSTXT}), x); out(pub, x); out(r(C_{DNSTXT}), x)$$

Threat Model

We assume that the adversary controls parts of the network and so can intercept, tamper with and inject network messages. As such, an attacker could make requests for domains they do not own, intercept and delay legitimate certificate requests and so on. Our adversary has full access to pub, $w(C_{EMAIL})$, $r(C_{HTTP})$ and $r(C_{DNSTXT})$. We also publish Mallory's channels and C_k over pub. As such, the attacker controls a set of valid participants (e.g. M) with their own valid identities (e.g. M_k, M_{pk}). The attacker may advertise any identity for its controlled principals, including false identities and may attempt to obtain a certificate for domains not legitimately under M_w.

The adversary also has at his disposal certain special functions:

- *PoisonDnsARecord*, which takes in a domain C_w and allows the attacker to poison its DNS records to redirect to a server owned by M. Calling this function triggers the $ActiveDnsAttack(C_w)$ event.
- *ManInTheMiddleHttp*, which allows the attacker to write arbitrary HTTP requests as if they were emitting from C_{HTTP} by disclosing C_{HTTP} to the attacker. Calling this function triggers the $ActiveHttpAttack(C_w)$ event.

[4]We also specify a fully public channel named `pub`.

4.2.2 ProVerif Events and Queries

Under ProVerif, queries under our symbolic model are constructed from sequences of the following events, each callable by a particular type of actor:

- **Client** The client is allowed to assert that they own some domain by triggering the event $Owner(C, C_w)$. Once C receives a certificate $C_{w_{cert}}$ for C_w from A, they also trigger
  $CertReceived(C_w, C_{w_{cert}}, C_{pk}, A_{pk})$

- **Server** The server (ACME instance or CA) triggers the event $HttpAuth(C_{pk}, C_w)$, $DnsAuth(C_{pk}, C_w)$ and $EmailAuth(C_{pk}, C_w)$ depending on the type of domain validation used. Once A issues a certificate $C_{w_{cert}}$ for C_w to C, they also trigger $CertIssued(C_w, C_{w_{cert}}, C_{pk}, A_{pk})$

- **Adversary** As noted above, the adversary may trigger the events $ActiveDnsAttack(C_w)$ and $ActiveHttpAttack(C_w)$. In addition, the adversary is allowed to masquerade as M in order assert that they own some domain by triggering the event $Owner(M_{pk}, C_w)$.

Queries

Queries encode the security properties that we expect our model to satisfy. For example, informally, we expect a *CertIssued* event may only occur following an *HttpAuth* or *DnsAuth* event for the same domain, expressing the fact that ACME should not issue a certificate for an non-validated domain under any circumstance. Running ProVerif on the query can result in three outcomes: either it diverges (in which case the model or query needs to be simplified), or it proves that the model satisfies the query, or it finds a counter-example and outputs its trace (which can be turned into an attack).

Validation with DNS Identifiers We assert that if DNS validation succeeded, then A must have been able to successfully carry out DNS validation according to spec, *or* an adversary was able to instantiate an active DNS poisoning attack (with no third possible scenario). In ProVerif, this can be expressed using injective event queries:

$$DnsAuth(C_{pk}, C_w) \implies (Owner(C_k, C_w) \vee DnsAttack(C_w))$$

Validation with HTTP Identifiers We explicitly show that HTTP authentication is weaker than DNS authentication, since an attack is possible under both cases of DNS poisoning *and* an HTTP man-in-the-middle attack:

$$HttpAuth(C_{pk}, C_w) \implies Owner(C_k, C_w) \vee (HttpAttack(C_w) \vee DnsAttack(C_w))$$

Predictable Certificate Issuance We attempt to verify that all received certificates were issued by the expected CA. This query fails to verify and leads us to the attack we discuss in §4.5:

$$CertReceived(C_w, C_{w_{cert}}, C_{pk}, A_{pk}) \implies CertIssued(C_w, C_{w_{cert}}, C_{pk}, A_{pk})$$

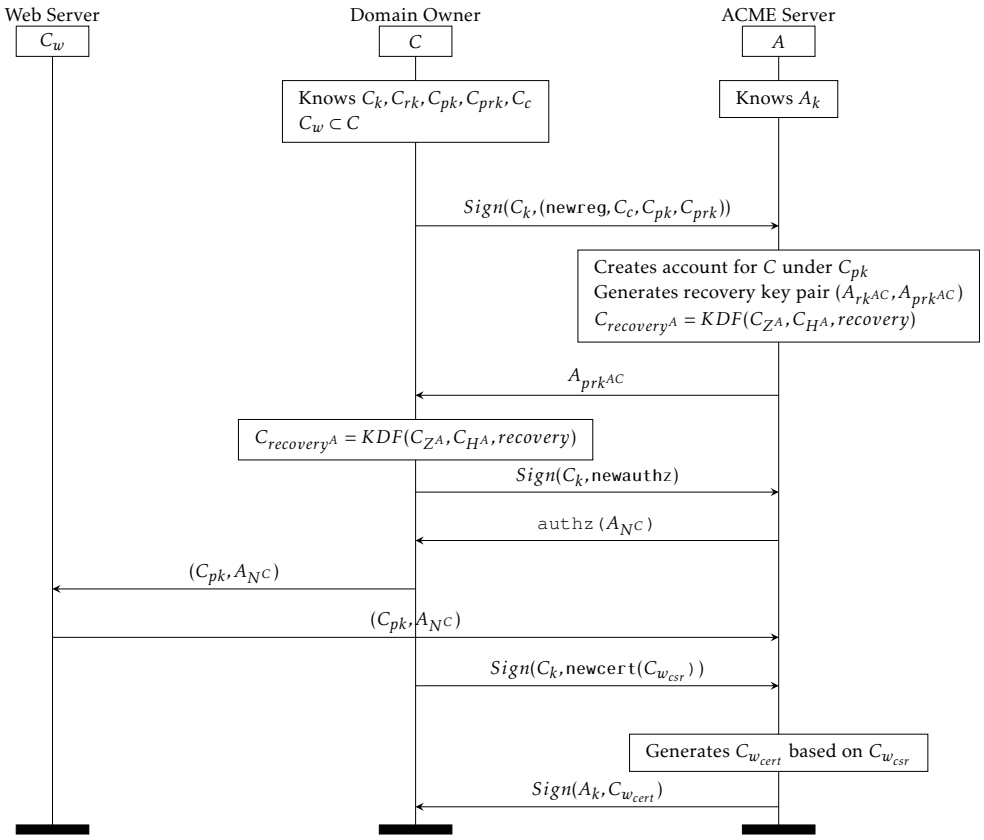

Figure 4.4: ACME draft-1 protocol functionality for C account registration, recovery key generation, and validation with certificate issuance for C_w. This chart demonstrates validation via an HTTP identifier. In draft-3 and above, the HTTP challenge (C_{pk}, A_{N^C}) is replaced with $Sign(C_k, (C_{pk}, A_{N^C}))$.

4.3 Specifying and Formally Verifying ACME

In this section we provide a formal description of the ACME protocol functionality and identify three issues that affect ACME's security. We also discuss details of how we describe the ACME protocol flow in the applied pi calculus, so that we can verify for certain queries using ProVerif.

4.3.1 ACME Network Flow

Unlike ad-hoc CAs which are limited to a web login, ACME's authentication depends on C generating a private signing key C_k with a corresponding public verification key C_{pk}, which are used to generate automated client signatures throughout the protocol.

HTTP Identifier A sends to C a nonce A_{N^C} via the HTTPS channel. C must then advertise, at an agreed-upon location under C_w, the value (C_{pk}, A_{N^C}). A then accesses C_w

using an unauthenticated, unencrypted HTTP connection to ensure that it can retrieve the intended value.

ACME also supports a very similar validation mode that operates at the level of the TLS handshake rather than at the HTTP level (using the SNI extension and a specially crafted certificate in place of the HTTP request and response). We believe this mode is intended for TLS termination software and hardware, and despite its apparent complexity, it is semantically equivalent to the HTTP identifier method. Since the details of the formatting of payloads is abstracted in our symbolic model we model both TLS-SNI and HTTP validation under the same framework in our model.

DNS Identifier A sends to C a nonce A_{nonce^C} via the HTTPS channel. C advertises this nonce in the form of a DNS record served by C_w's name servers, thereby proving ownership of C_w. A can then query its DNS server to verify that the nonce has been set. While this behavior is specified in ACME, it is not used in any implementation of Let's Encrypt: since ACME is designed to take advantage of domain validation methods that can be automated and since DNS record management depends on a series of ad-hoc protocols of its own between C and DNS service providers, it is not used by ACME.

Out-of-Band Validation The ACME standard draft supports an out-of-band validation mechanism, which can be used to implement legacy validation methods, including email-based validation. However, since this method is underspecified, we do not cover it in our models and advise against using any out-of-bound validation unless it is analyzed under a specific model.

4.3.2 ACME Protocol Functionality

In this chapter we focus on draft-1 of the IETF specification for the ACME protocol, which is as of October 2016 also the draft specification deployed in official Let's Encrypt client and server implementations. In part due to the issues we discuss in the work and have communicated with the ACME team, draft-3 (and subsequently draft-4) does away with some features, most notably Account Recovery and generally is resistant to the issues discussed here.

Preliminaries In some parts of ACME's protocol flow, C and A will need to establish a number of shared secrets, each bound to a strict protocol context, over their public keys. In ACME, this is accomplished using ANSI-X9.63-KDF:

1. C and A agree on a ECDH shared secret C_{Z^A} using their respective key pairs (C_k, C_{pk}) and (A_k, A_{pk}).
2. A hashing function C_{H^A} is chosen according to the elliptic curve used to calculate C_{Z^A}: $SHA256$ for $P256$, $SHA384$ for $P384$ and $SHA512$ for $P521$.
3. $C_{label^A} = KDF(C_{Z^A}, C_{H^A}, label)$, with $label$ indicating the chosen context for this particular key's usage.

As a protocol, ACME provides the following six certificate management functionalities (illustrated in Figure 4.4) between web server C and certificate management authority A:

- *Account Key Registration* In this step, C specifies her contact information (email address, phone number, etc.) as C_c and generates a random private signing key

C_k with (over a safe elliptic curve) a public key C_{pk}. A `POST` request is sent to A containing $Sign(C_k, (\texttt{newreg}, C_c, C_{pk}))$. The `newreg` header indicates to A that this is an account registration request. If A has no prior record of C_{pk} being used for an account and if the message's signature is valid under C_{pk}, A creates a new account for C using C_{pk} as the identifier and responds with a success message.

- *MAC-Based Account Recovery* C may choose to identify an account recovery secret with A. In order to do this, C generates an account recovery key pair (C_{rk}, C_{prk}) and simply includes C_{prk} in an optional `recoverykey` field in its initial `newreg` message to A. A generates the complementary $(A_{rk^{AC}}, A_{prk^{AC}})$ and both parties calculate $C_{recovery^A}$ using their recovery key pairs. A communicates $A_{prk^{AC}}$ in its response to C. Later, if C loses C_k, she can ask A to re-assign her account to a new identity $(C_{k'}, C_{pk'})$ by using $C_{recovery^A}$ as a key to generate a MAC of some value chosen by A.

- *Contact-Based Account Recovery* C can request that A send a verification token to one of the contact methods previously specified in C_c. For example, this could be a URI sent to an email in C_c. If C successfully opens this URI, she becomes free to replace C_{pk} with a $C_{pk'}$ for some arbitrary $C_{k'}$ at A.

- *Identifier Authorization* C can validate its ownership of a domain C_w in C_w by providing one of the identifiers discussed in §4.2 to A. C must first request authorization for C_w by sending a `newauthz` message. A then responds with the types of identifiers it is willing to accept in a `authz` message. C is then free to use any one of the permitted identifiers to validate its ownership of C_w and allow A to sign certificates for it issued to C and tied to the identity C_{pk}.

- *Certificate Issuance and Renewal* After C ties an identifier to C_w under C_{pk}, it may request that a certificate be issued for C_w simply by requesting one from A. Generally, A will send the signed certificate with no further steps required. The renewal procedure is similarly straightforward.

- *Certificate Revocation* C may ask A to revoke the certificate for C_w by sending a `POST` message containing the certificate in question, signed under either C_{pk} or the key pair for the certificate itself. C may choose which key to use for this signature. A verifies that the public key of the key pair signing the request matches the public key in the certificate and that the key pair signing the request is an account key and the corresponding account is authorized to act for all of the identifier(s) in the certificate.

Given this description of the ACME protocol and the threat model defined in §4.2, we modeled ACME using the automated verification tool ProVerif [143]. In our model, we involve three different candidates for C: Alice, Bob and Mallory and two CA candidates as A.

In our automated verification process, we consider an active attacker over the three channels specified in §4.2. As a result, we were able to find the issues discussed in §4.5. The first two are relatively minor; however, the third could lead to account compromise in the case of contact-based account recovery and potentially to the issuance of false certificate signatures if email-based domain validation were to be implemented in ACME. Furthermore, this third issue is also generalizable to affect traditional certificate authorities, as described in §4.1.

4.3.3 Model Processes

Using the modeling conventions we established in §4.2 which include channels, adversaries, actors and events, we instantiate in our ProVerif model of ACME a top-level process that executes the following processes in parallel:

- *ClientAuth* Run simultaneously by Alice, Bob and Mallory assuming the role of C (with a compromised Mallory), this process registers a new account with A and sends the queries illustrated in Figure 4.4. The events *Owner* and $CertReceived$ are triggered as part of this process.
- *ServerAuth* Run simultaneously by two independent CAs assuming the role of A, this process accepts registrations from C and follows the protocol illustrated in Figure 4.4. The events $HttpAuthenticated$ and $CertIssued$ are triggered as part of this process.

The processes $RoutingProxy$, $PoisonDnsARecord$ and $ManInTheMiddleHttp$, all described in §4.2, are also run in parallel with the above.

4.4 Weaknesses in Traditional CAs

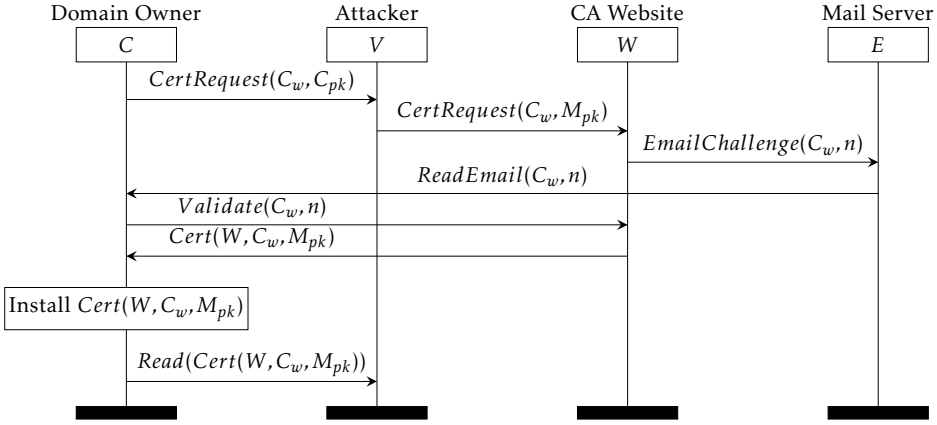

Figure 4.5: Attack on Email Validation: Concurrent Request by Active Adversary.

Traditional CA dependence on weak channels gives us a threat model where real-world attacks can have a small cost and come with severe consequences.

Email Validation In ad-hoc CAs, C is generally simply sent an email containing a URI to their email inbox, which they're supposed to click in order to validate for their chosen domain. Figure 4.5 shows an attack rendered possible by this mechanism. A could instead, upon a validation request, redirect C's browser to a secret, nonce-based URI A_{URI^C} served to C over the HTTPS channel and independently mail C the value $HMAC(A_{hk}, A_{URI^C})$ for some secret A_{hk} held by A. C would need to retrieve this second value and enter it inside the page at A_{URI^C}. This approach would largely negate the weakness discussed in §4.5, since an attacker-induced validation email would result

in an email that does not include a value matching the URI given by A to C at the beginning of the validation process.

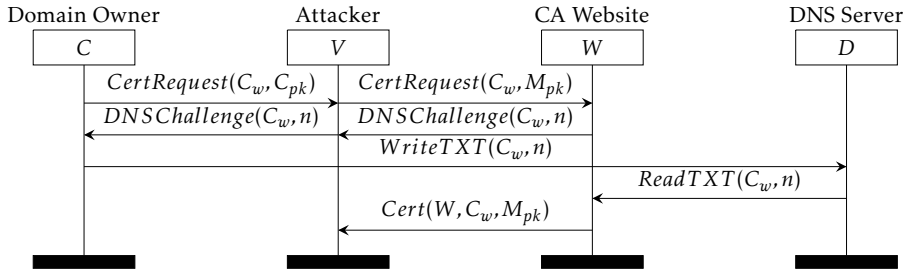

Figure 4.6: Active attack on DNS/HTTP/Email Validation when using just nonces.

Usage of Nonces Traditional CAs use random nonces with no special cryptographic properties as the values that they then verify over HTTP, email or DNS. In addition to this helping caused the attack described above, another more general attack on nonces is shown in Figure 4.6 in the case of an active attacker. For example, this attack can be used by a compromised CA website to get certificates issued for domain C_w by another (more reputable) CA, hence amplifying the compromise across CAs. None of these attacks would be effective if nonces were tied to some cryptographic properties, such as MACs or even just by deriving them from a hash of the certificate request's public key.

In order to avoid a similar attack, ACME draft-3 and draft-4 require that HTTP identifiers be validated by broadcasting $Sign(C_k, A_{N^C})$ via the web server instead of ACME draft-1's (C_{pk}, A_{N^C}).

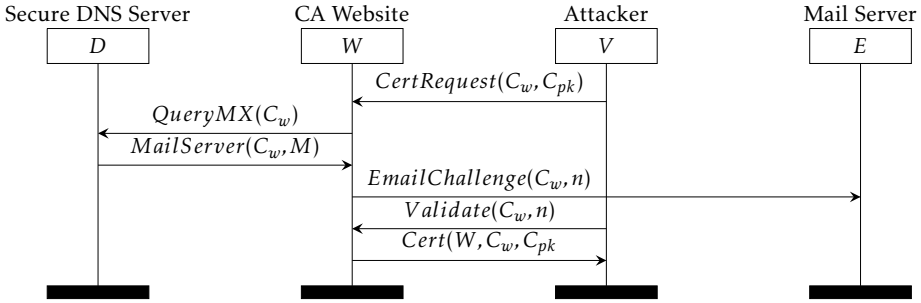

Figure 4.7: Attack on Email Validation: Passive Adversary on Email Channel

4.5 Weaknesses in ACME

Cross-CA Attacks on Certificate Issuance Suppose an ACME client C requests a certificate from A but A is malicious or the secure channel between A and C is compromised. Now, an attacker can intercept authorization and certificate requests from C to

A, and instead forward them to another ACME server A'. If A' requests domain validation with a token T, the attacker forwards the token to the client, who will dutifully place its account key K and token T on its validation channel. A' will check this token and accept the authorization and issue a certificate that the attacker can forward to C.

This means that C asked for a certificate from A, but instead received a certificate from A'. Moreover, it may have paid A for the service, but A' might have done it for free. This issue, while not critical, can be prevented if C checks the certificate it gets to make sure it was issued by the expected CA. An alternative and possibly stronger, mitigation would be for ACME to extend the Key Authorization string to include the CAs identifier.

More generally, this issue reveals that ACME does not provide channel binding, and this appears as soon as we model the ACME HTTPS Channel. We would have expected to model this as a mutually-authenticated channel since the client always signs its messages with the account key. However, although the clients signature is tunnelled inside HTTPS, the signature itself is not bound to the HTTPS channel. This means that a message from an ACME client C to A can be forwarded by A to a different A' (as long as C supports both A and A'). This kind of credential forwarding attack can be easily mitigated by channel binding. For example, ACME could rely on the Token Binding specifications to securely bind the client signature to the underlying channel. Alternatively, ACME could extend the signed request format to always include the servers name or certificate-hash, to ensure that the message cannot be forwarded to other servers.

Contact-Based Recovery Hijacking While the use of sender-authenticated channels in ACME seems to be relatively secure, more attention needs to be paid to the receiver-authenticated channels. For example, if the ACME server uses the website administrator's email address to send the domain validation token, a naïve implementation of this kind of challenge would be vulnerable to attack.

In the current specification, the contact channel (typically email) is used for account recovery when the ACME client has forgotten its account key. We show how the careless use of this channel can be vulnerable to attack and propose a countermeasure. Suppose an ACME client C issues an account recovery request for an account under C_{pk} with a new key $C_{k'}$ to the ACME server A. A network attacker M blocks this request and instead sends his own account recovery request for the account under C_{pk} (pretending to be C) with his own key $M_{k'}$. A will then send C an email asking to click on a link. C will think this is a request in response to its own account recovery request and will click on it. Similarly to the (slightly different) flow described in Figure 4.5, A will think that C has confirmed account recovery and will transfer the account under C_{pk} to the attackers key $M_{k'}$. In the above attack, the attacker did not need to compromise the contact channel (or for that matter, the ACME channel).

The key observation here is that on receiver-authenticated channels (e.g. email) the receiver does not get to bind the token provided by A with its own account key. Consequently, we need to add a further check. The email sent from A to C should contain a fresh token in addition to Cs new account key. Instead of clicking on the link (out-of-band), C should cut and paste the token into the ACME client which can first check that the account key provided by A matches the one in the ACME client and only then does it send the token back to A, or alternatively that the email recipient at C visually confirms that the account key (thumbprint) provided by A matches the one displayed in the ACME client.

The attack described here is on account recovery, but a similar attack appears if

we allow email-based domain validation. A malicious ACME server or man-in-the-middle can then get certificate issued for *C*'s domains with its own public key, without compromising the contact/validation channel. The mitigation for that attack would be very similar to the one proposed above.

4.6 Conclusion

In this work, we have provided the results of an empirical case study that allowed us to describe a real-world threat model governing both traditional certificate authorities and ACME in terms of user authentication and domain validation. We formally modeled these protocols and provided the results of security queries under our threat model, using automated verification. As a result of our disclosures to the ACME team, the latest ACME protocol version (draft-4) has been designed to avoid the pitfalls that make these attacks possible.

Given the weak threat model that traditional CAs are assuming for their domain validation process, we are not surprised by the regular occurrences of unauthorized certificate issuance (e.g. StartCom in 2008, Comodo and DigiNotar in 2011, WoSign in 2016). We advocate the CA/Browser forum to eventually mandate the use of ACME (or some other well-defined domain validation protocol that can be formally analyzed) to all certification authorities, as a long-term solution to reduce unauthorized certificate issuance. Until the issuance process for the whole PKI is unified, techniques to improve the validation of certificates such as certificate transparency [42] remain necessary to detect issuance failures and technologies such as DNSSEC [144], DANE [145], or SMTPS may help strengthen the channels involved in legacy domain validation.

Chapter 5

Formally Verified Secure Channel Implementations in Web Applications

The work in this chapter continues the research described in Chapters 1 and 2 and was published in the same proceedings as those previous chapters. It was completed with co-authors Karthikeyan Bhargavan and Bruno Blanchet, and would not have been possible without the gracious assistance of Antoine Delignat-Lavaud.

In Chapter 1, we employed automated formal verification in both the symbolic and computational models against popular secure messaging protocols. The same approach was followed in Chapter 2, where it was applied towards prototyping and formally verifying drafts of TLS 1.3 as the nascent protocol evolved. In this chapter, we present work that aims to extend our formal verification results from formal models and onto real-world implementations. We then want to produce a link between these implementations onto their basis in formal models. Our hope is that in doing so, we will be able to rule out common protocol implementation flaws such as nonce mismanagement or fundamental omissions such as missing message signatures or authentication codes.

Signal has been implemented in various programming languages, but most desktop implementations of Signal, including Cryptocat, are written in JavaScript. Although JavaScript is convenient for widespread deployability, it is not an ideal language for writing security-critical applications. Its permissive, loose typing allows for dangerous implementation bugs and provides little isolation between verified cryptographic protocol code and unverified third-party components. Rather than trying to verify general JavaScript programs, we advocate that security-critical components like SP should be written in a well-behaved subset that enables formal analysis.

We introduce ProScript (short for *"Protocol Script"*), a programming and verification framework tailored specifically for the implementation of cryptographic protocols. ProScript extends Defensive JavaScript (DJS) [77, 146], a static type system for JavaScript that was originally developed to protect security-critical code against untrusted code running in the same origin. ProScript is syntactically a subset of JavaScript, but it imposes a strong coding discipline that ensures that the resulting code is amenable to formal analysis. ProScript programs are mostly self-contained; they cannot call arbitrary third-party libraries, but are given access to carefully implemented (well-typed)

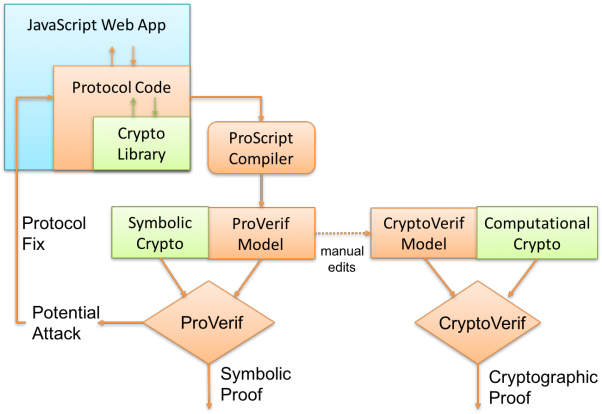

Figure 5.1: Verification Approach.

libraries such as the ProScript cryptographic library (PSCL). Protocols written in ProScript can be type-checked and then automatically translated into an applied pi calculus [147] model using the ProScript compiler. The resulting model can be analyzed directly through ProVerif and can be adapted and extended to a proof in CryptoVerif. As the code evolves, this model can be automatically refreshed to enable new analyses and proofs, if necessary.

We advocate the verification approach depicted in Figure 5.1: A ProVerif model is automatically extracted from ProScript protocol code and analyzed for its security goals against a symbolic attacker. The model is then edited by hand and extended with cryptographic assumptions to build a computational proof that is verified by CryptoVerif. The messaging application is written in JavaScript and is broken down into a cryptographic protocol core and untrusted application code that interact through a small well-typed API that hides all protocol secrets within the protocol core and only offers a simple send/receive functionality to the application. Notably, the protocol core is written in a domain-specific language and does not rely on any external libraries except for a well-vetted cryptographic library. The protocol code can be translated to an applied pi calculus model and symbolically analyzed in ProVerif to find protocol flaws and attacks. The model can also be used as the starting point for a cryptographic proof for the protocol developed using CryptoVerif.

Contributions. This chapter builds upon the contributions of Chapter 1, which centered around formal verification, to extend them into contributions targeting robust real-world protocol implementations:

- **Automated Model Extraction from JavaScript** We present the ProScript compiler, which allows for the compilation from a subset of JavaScript into a readable protocol model in the applied pi calculus. Model extraction enables formal verification to keep up with rapidly changing source code. Readable models allow the protocol analyst to experiment with different threat models and security goals and to test new features before including them in the implementation.

- **A Verified Protocol Core for Cryptocat** We integrate our verified protocol code into the latest version of Cryptocat[1] (§5.2), a popular open source messaging client with thousands of users that was originally developed by the author of

[1] https://crypto.cat

this thesis. We show how the new architecture of Cryptocat serves to protect the verified protocol code from bugs in the rest of the application.

- **A Verified Reference Implementation of TLS 1.3** Our ProVerif and CryptoVerif models capture the protocol core of TLS 1.3, but they elide many implementation details such as the protocol API and state machine. To demonstrate that our security results apply to carefully-written implementations of TLS 1.3, we present RefTLS, the first reference implementation of TLS 1.0-1.3 whose core protocol code has been formally analyzed for security. RefTLS is written in Flow, a statically typed variant of JavaScript, and is structured so that all its protocol code is isolated in a single module that can be automatically translated to ProVerif and symbolically analyzed against our rich threat model.

5.1 ProScript: Prototyping Protocols with Formal Translations

ProScript aims to be an ideal language for reliably implementing cryptographic protocols for web applications. Using ProScript, a protocol designer or implementer can implement a protocol, automatically extract a formal model from the code, verify the model using ProVerif and then run the protocol code within a JavaScript web application. The ProScript framework does not target general JavaScript code, however existing applications can be adapted to use ProScript for their security-critical protocol components.

Our goal is to allow the developer to go back and forth between their protocol implementation and the ProVerif model, in order to help understand the behavior being illustrated, the properties being verified and how detected attacks, if any, relate to their source code. For these reasons, we pay special attention to generating models that are optimized both for verifiability as well as readability. This increases their utility to a human examiner who may wish to independently expand the model to include more specific process flows or to explore variations of the protocol against a manually defined adversary.

Syntactically, ProScript is a subset of JavaScript that can be naturally translated to the applied pi calculus. This restriction produces casualties, including recursion, `for` loops and extensible objects. A closer look at the ProScript syntax shows JavaScript employed in a particular style to bring out useful features:

Isolation. ProScript is based on Defensive JavaScript (DJS) [77, 146], a typed subset of JavaScript which focuses on protecting security-critical components from malicious JavaScript code running in the same environment. DJS imposes a strict typing discipline in order to eliminate language-based attacks like prototype poisoning. In particular, it forbids the use of unknown external libraries as well as calls to tamperable object methods such as `.toString()`. It also forbids extensible objects and arrays and prevents any access to object prototypes. These restrictions result in protocol implementations that are more robust and less influenced by the runtime environment. The ProScript typechecker builds on and extends DJS and hence, inherits both its language restrictions and isolation guarantees.

Type Declarations and Inference. ProScript requires all variables and functions to be declared before they are used, hence imposing a strict scoping discipline. For example, an expression $v.x$ is well-typed if and only if v has been defined, as a local or global variable, to be an object with a property x. As an extension to the builtin types of

DJS, ProScript allows type declarations for commonly used protocol data structures. For example, an array of 32 hexadecimal integers can be declared as a key type. The ProScript compiler recognizes such type declarations and uses them to translate the code into more concise and informative ProVerif models. Moreover, the typechecker can automatically infer fine-grained sub-types. For example, ProScript differentiates between numbers declared using decimal literals (eg. `128`) and hexadecimal literals (eg. `0x80`). Numbers defined using hexadecimal are sub-typed as bytes. This feature allows us to track how numerical values are employed in the protocol and prevents type coercion bugs similar to an actual bug that we describe in §5.2, where a significant loss of entropy was caused by a byte being coerced into a decimal value.

State-Passing Functional Style. ProScript's syntax takes advantage of JavaScript's functional programming features in order to encourage and facilitate purely functional protocol descriptions, which the compiler can translate into symbolically verifiable, human-readable models in the applied pi calculus. The functional style encourages the construction of state-passing functions, leaving state modification up to the unverified application outside of the ProScript code. The majority of a ProScript implementation tends to be a series of pure function declarations. A small subset of these functions is exposed to a global namespace for access by the verified application while most remain hidden as utility functions for purposes such as key derivation, decryption and so on. This state-passing style is in contrast to DJS that allows direct modification of heap data structures. The functional style of ProScript allows protocol data structures, including objects and arrays, to be translated to simple terms in ProVerif built using constructors and destructors, hence avoiding the state-space explosion inherent in the heap-based approach that is needed to translate DJS to ProVerif [146].

5.1.1 Protocol Implementation

To implement SP in ProScript, we must first deconstruct it into various protocol components: structures for managing keys and user states, messaging functions, APIs and top-level processes. ProScript is well-equipped to handle these protocol components in a way that lends itself to model extraction and verification. We break down our ProScript SP implementation into:

Types for State and Key Management. ProScript's type declaration syntax can be used to declare types for individual elements such as encryption keys but also for collections of elements such as a conversation party's state. These declarations allow for the construction of common data structures used in the protocol and also makes their management and modification easier in the extracted ProVerif models.

Messaging Interface. The ProScript implementation exposes the generic messaging API in a single global object. All interface access provides purely state-passing functionality.

Long-Term and Session States. Protocol functions take long-term and session states (S^a, T_n^{ab}) as input and return T_{n+1}^{ab}. S^a contains long-term values such identity keys, while T includes more session-dependent values such as `ephemerals`, containing the current ephemeral and chaining keys for the session context and `status`, indicating whether the application layer should perform a state update.

Internal Functions. Utility functionality, such as key derivation, can also be described as a series of pure functions that are not included in the globally accessible interface.

Top-Level Process. A top-level process can serve as a harness for testing the proper functioning of the protocol in the application layer. Afterwards, when this top-level

process is described in the extracted ProVerif model, the implementer will be able to use it to define which events and security properties to query for.

Inferred Types in ProScript. ProScript type declarations allow for the easier maintenance of a type-checkable protocol implementation, while also allowing the ProScript compiler to translate declared types into the extracted ProVerif model. Defining a `key` as an array of 32 bytes will allow the ProScript compiler to detect all 32 byte arrays in the implementation as keys and type their usage accordingly. Figure 5.4 shows the compiled output of the key derivation function implemented in the ProScript code. The compiler infers its input and output types through its usage elsewhere in the code and assigns a proper type in case one is declared. If no type is declared, the compiler will formalize constructors and destructors to deal with the values being passed.

5.1.2 ProScript Syntax

A ProScript implementation consists of a series of *modules*, each containing a sequence of type declarations (containing constructors, assertion utilities and type converters), constant declarations and function declarations.

Type Declarations. Types are declared as object constants with the name Type_x where x is the name of the type (e.g. *key*). Type declaration objects include various properties:

1. $construct$, a function which returns the shape of the object and which is used to both define the type to the compiler and to instantiate new variables of this type throughout the code.

2. $assert$, a function allowing the ProScript compiler to infer that a function's input parameter is of type x.

3. $toBitstring, fromBitstring$, conversion functions from type x to a regular string. These are detected by the ProScript compiler and represented as ProVerif type conversion functions in the extracted model.

4. $clone$, a function for cloning a variable of type x.

Constant Declarations. In ProScript, we prefer the use of constant declarations to variable declarations due to their decreased malleability. We also prohibit the reassignment of object properties because of the pointer-like behavior of JavaScript object variables rendering this difficult to model efficiently in ProVerif. Scoping is also enforced: ProScript only allows for the declaration of variables at the top of a module or function, preceding all function declarations or calls. Further, for an object v, we say that $v.x$ is well-formed if and only if v has been declared, either as a type or scoped variable, to have a property x.

Function Declarations. The majority of a ProScript implementation tends to be a series of pure function declarations. A small subset of these functions is exposed to a global namespace for access by the verified application while most remain hidden as utility functions for purposes such as key derivation, decryption and so on. All ProScript functions are pure and state-passing, allowing them to be modeled into ProVerif functions, declared with the keyword *letfun* (See the Introduction of this thesis). All ProScript functions are typed and the ProScript compiler will enforce that they are used with the same input types and return the same output type throughout the implementation. Top-level object declarations often contain function declarations. We assume for simplicity that these object declarations are flattened into top-level function declarations. In order to enforce type safety, ProScript objects and arrays are also

non-extensible: JavaScript prototype functions such as Array.push cannot be employed and object accessors such as $v[x]$ are disallowed.

ProScript

$v ::=$	values
x	variables
n	numbers
s	strings
true, false	booleans
undefined, null	predefined constants
$e ::=$	expressions
v	values
$\{x_1 : v_1, \ldots, x_n : v_n\}$	object literals
$v.x$	field access
$[v_1, \ldots, v_n]$	array literals
$v[n]$	array access
Lib.$l(v_1, \ldots, v_n)$	library call
$f(v_1, \ldots, v_n)$	function call
$\sigma ::=$	statements
var $x; \sigma$	variable declaration
$x = e; \sigma$	variable assignment
const $x = e; \sigma$	constant declaration
if $(v_1 === v_2)$ $\{\sigma_1\}$ else $\{\sigma_2\}$	if-then-else
return e	return
$\gamma ::=$	globals
const $x = e$	constants
const f = function$(x_1, \ldots, x_n)\{\sigma\}$	functions
const Type_$x = \{\ldots\}$	user types
$\mu ::= \gamma_0; \ldots; \gamma_n$	modules

Note that we will use the defined Lib.l notation to access the ProScript Cryptography Library.

Operational Semantics. ProScript's operational semantics is a subset of JavaScript and both run on JavaScript interpreters. It is tooled based on the formal semantics of Maffeis et al. [148] and is superficially adapted for our language subset.

5.1.3 Translation

ProScript functions are translated into ProVerif pure functions. Type declarations are translated into ProVerif type declarations. Individual values, such as strings and numbers, are declared as global constants at the top-level scope of the ProVerif model with identifiers that are then employed throughout the model when appropriate. Objects and Arrays are instantiated in the model using functions, with destructors automatically generated in order to act as getters.

Translation Rules

$M_v ::= v \mid \{x_1 : v_1, \ldots, x_n : v_n\} \mid [v_1, \ldots, v_n]$	
$\mathcal{V}[\![M_v]\!] \rightarrow M$	Values to Terms
$\mathcal{V}[\![v]\!] = v$	

$\mathcal{V}[\![\{x_1 : v_1, \ldots, x_n : v_n\}]\!] = \mathsf{Obj}_t(v_1, \ldots, v_n)$
$\mathcal{V}[\![[v_1, \ldots, v_n]]\!] = \mathsf{Arr}_t(v_1, \ldots, v_n)$

$\mathcal{E}[\![e]\!] \to M$ Expressions to Terms
$\mathcal{E}[\![M_v]\!] = \mathcal{V}[\![M_v]\!]$
$\mathcal{E}[\![v.x]\!] = \mathtt{get}_{\mathtt{x}}(v)$
$\mathcal{E}[\![v[i]]\!] = \mathtt{get}_{\mathtt{i}}(v)$
$\mathcal{E}[\![\text{Lib}.l(v_1, \ldots, v_n)]\!] = \mathtt{Lib}_{\mathtt{l}}(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$
$\mathcal{E}[\![f(v_1, \ldots, v_n)]\!] = f(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$

$\mathcal{S}[\![\sigma]\!] \to E$ Statements to Enriched Terms
$\mathcal{S}[\![\text{var } x; \sigma]\!] = \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![x = e; \sigma]\!] = \mathtt{let}\ x = \mathcal{E}[\![e]\!]\ \mathtt{in}\ \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![\text{const } x = e; \sigma]\!] = \mathtt{let}\ x = \mathcal{E}[\![e]\!]\ \mathtt{in}\ \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![\text{return } v]\!] = \mathcal{V}[\![v]\!]$
$\mathcal{S}[\![\text{if } (v_1 \texttt{===} v_2)\ \{\sigma_1\}\ \text{else}\ \{\sigma_2\}]\!] =$
$\quad \mathtt{if}\ \mathcal{V}[\![v_1]\!] = \mathcal{V}[\![v_2]\!]\ \mathtt{then}\ \mathcal{S}[\![\sigma_1]\!]\ \mathtt{else}\ \mathcal{S}[\![\sigma_2]\!]$

$\mathcal{F}[\![\gamma]\!] \to \Delta$ Types and Functions to Declarations
$\mathcal{F}[\![\text{const } f = \text{function}(x_1, \ldots, x_n)\{\sigma\}]\!] =$
$\quad \mathtt{letfun}\ f(x_1, \ldots, x_n) = \mathcal{S}[\![\sigma]\!]$
$\mathcal{F}[\![\text{const Type\_}t = \{\ldots\}]\!] = \mathtt{type}\ t$

$\mathcal{C}[\![\mu]\!](P) \to P$ Constants to Top-level Process
$\mathcal{C}[\![e]\!](P) = P$
$\mathcal{C}[\![\text{const } x = e; \mu]\!](P) = \mathtt{let}\ x = \mathcal{E}[\![e]\!]\ \mathtt{in}\ \mathcal{C}[\![\mu]\!](P)$

$\mathcal{M}[\![\mu]\!](P) \to \Sigma$ Modules to Scripts
$\mathcal{M}[\![\mu]\!](P) = \mathcal{F}[\![\gamma_1]\!].\ldots.\mathcal{F}[\![\gamma_n]\!].\mathcal{C}[\![\mu_c]\!](P)$
where μ_c contains all globals const $x = e$ in μ
and $\gamma_1, \ldots, \gamma_n$ are the other globals of μ.

5.1.4 A Symbolic Model for PSCL

Besides being type-checked, the ProScript Cryptography Library is unique in that the ProScript compiler is able to detect its usage in ProScript code and model the equational relationships of cryptographic primitives automatically in the extracted model, as seen in Figure 5.4 (lines 9-18). For instance:

1. Diffie-Hellman operations have the relationship $(g^a)^b = (g^b)^a = g^{ab}$ modeled for a base point g using ProVerif equations.
2. Symmetric Encryption operations have their decryption functions related to the ciphertext with the use of destructors, enforcing the relationship $Dec(k, Enc(k, m)) = m$.
3. HMAC checking functions are similarly modeled with destructors in order to illustrate the relationship $checkHMAC(k, m, HMAC(k, m)) = true$.

More creatively, destructors can also be provided for the attacker to detect instances of nonce re-use in stream ciphers (such as AES-CTR) by modeling:
$(AESCTR(k, n, m_0), AESCTR(k, n, m_1)) \Rightarrow (m_0, m_1)$.

Translation Soundness. We currently do not formally prove translation soundness, so proofs of the resulting ProVerif model do not necessarily imply proof of the source code. Instead, we use model translation as a pragmatic tool to automatically generate readable protocol models faithful to the implementation and to find bugs in the implementation. We have experimented with multiple protocols written in ProScript, including OTR, SP, and TLS 1.3 and by carefully inspecting the source code and target models, we find that the compiler is quite reliable and that it generates models that are not so far from what one would want to write directly in ProVerif. In future work, we plan to prove the soundness of this translation to get stronger positive guarantees from the verification. To this end, we observe that our source language is a simply-typed functional programming language and hence we should be able to closely follow the methodology of [75].

State Tables Interface. We use ProVerif `tables` in order to manage persistent state parameters between parallel protocol executions. This necessitates the introduction of a key-value syntax in ProScript: tables are declared as global constants. Their access is summarized in this simple example: `return ProScript.state.insert('usernames', 'myUsername', password),`
`ProScript.state.get('usernames', 'myUsername', password)`
translate to `insert usernames(myUsername, password),`
`get tableName(=myUsername, password).`

Generating Top-Level Processes. We are also able to automatically generate top-level ProVerif processes. Aiming to implement this in a way that allows us to easily integrate ProScript code into existing codebases, we decided to describe top-level functions inside `module.exports`, the export namespace used by modules for Node.js [149], a popular client/server run-time for JavaScript applications (based on the V8 engine). This makes intuitive sense: `module.exports` is used specifically in order to define the functions of a Node.js module that should be available to the external namespace once that module is loaded and executing all this functionality in parallel can give us a reasonable model of a potential attacker process. Therefore, functions declared in this namespace will be translated into top-level processes executed in parallel. We use ProVerif `tables` in order to manage persistent state between these parallel processes: each process fetches the current state from a table, runs a top-level function in that state, and stores the updated state returned by the function in the table.

5.1.5 Trusted Libraries for ProScript

Protocol implementations in ProScript rely on a few trusted libraries, in particular, for cryptographic primitives and for encoding and decoding protocol messages.

When deployed in Node.js or within a browser, the protocol code may have access to native cryptographic APIs. However, these APIs do not typically provide all modern cryptographic primitives; for example, the W3C Web Cryptography API does not support Curve25519, which is needed in Signal. Consequently, implementations like Signal Messenger end up compiling cryptographic primitives from C to JavaScript. Even if the desired primitives were available in the underlying platform, accessing them in a hostile environment is unsafe, since an attacker may have redefined them. Consequently, we developed our own libraries for cryptography and message encoding.

The ProScript Cryptography Library (PSCL) is a trusted cryptographic library implementing a variety of modern cryptographic primitives such as X25519, AES-CCM and BLAKE2. All of its primitives are fully type-checked without this affecting speed: in the majority of our benchmarks, PSCL is as fast as or faster than popular JavaScript

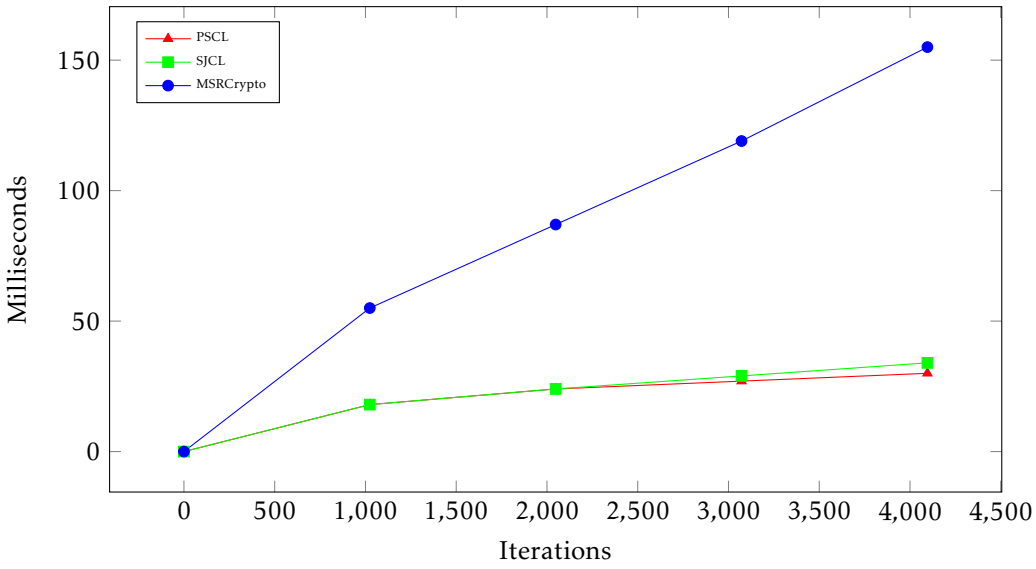

Figure 5.2: ProScript Crypto Library (PSCL) benchmarked against leading JavaScript cryptographic libraries: Stanford JavaScript Crypto Library (SJCL) and Microsoft Research Crypto Library (MSRCrypto). Each SHA256 iteration hashes 16 blocks.

cryptographic libraries like SJCL and MSR JavaScript Crypto, which do not even benefit from defensive type checking (Figure 5.2).

More crucially, PSCL functions used in ProScript code are detected by the ProScript compiler as it produces the applied pi model of the implementation, giving it the ability to convert each call to a cryptographic primitive to a call to the corresponding symbolic function in ProVerif. For example, if the ProScript compiler sees a call to PSCL's X25519 implementation, it will automatically translate it to a standard Diffie-Hellman construction in ProVerif.

5.2 A Verified Protocol Core for Cryptocat

We now describe how we can rewrite Cryptocat to incorporate our ProScript implementation of SP. We deconstruct the Cryptocat JavaScript code into the following components, as advocated in Figure 5.1.

Cryptocat is a secure messaging application that is written in JavaScript and deployed as a desktop web application. Earlier versions of Cryptocat implement a variant of the OTR (Off-The-Record) messaging protocol [6] which suffers from several shortcomings. It does not support asynchronous messaging, so both peers have to be online to be able to message each other. It does not support multiple devices or encrypted file transfer. OTR also uses legacy cryptographic constructions like DSA signatures and prime-field Diffie-Hellman, which are slower and less secure than more modern alternatives based on elliptic curves. Furthermore, Cryptocat peers did not have long-term identities and so the authentication guarantees are weak. Early version of Cryptocat suffered from many high-profile implementation bugs, including the reuse of initialization vectors for file encryption [150], bad random number generation, and a classic JavaScript type flaw that resulted in a private key of 255 bits being coerced into a string

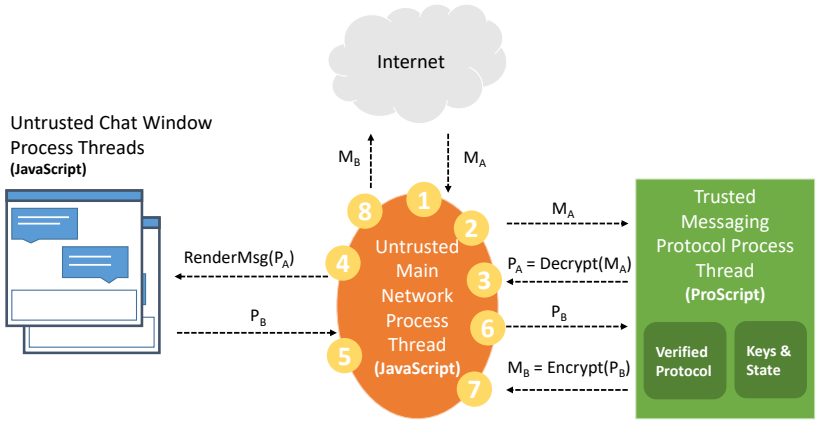

Figure 5.3: Cryptocat Architecture: isolating verified and untrusted components in Electron apps within separate processes.

that held only 55 bits. Some of these implementation flaws would have been found using a static type checker, others required deeper analysis.

Cryptocat was recently rewritten from scratch to upgrade both its messaging protocol and its implementation. The goal of this redesign was to isolate its protocol core and replace it with a verified messaging protocol written in a statically typed subset of JavaScript.

1. **Unverified JavaScript Application** This component, which comprises the majority of the code, manages the user's state, settings, notifications, graphical interface and so on. It is connected to the protocol only via the ability to call exposed protocol functions (as documented in § 1.1.1). We adopt certain assumptions regarding the unverified JavaScript application, for example that it will not modify the protocol state outside of passing it through the protocol implementation interface.

2. **Verified Protocol Implementation** This component is written in ProScript and resides in a separate namespace, functioning in a purely state-passing fashion. Namely, it does not store any internal state or make direct network calls. This implementation is type-checked, and automatically verified every time it is modified.

3. **Trusted Library** This component provides cryptographic functionality. A goal is to include modern cryptographic primitives (X25519, AES-CCM) and provide type-checking assurances without affecting speed or performance.

This layered architecture is essential for our verification methodology, but is quite different from other messaging applications. For example, the Signal Desktop application is a Chrome browser application also written in JavaScript [151]. Parts of the protocol library are compiled from C using Emscripten, presumably for performance, parts are taken from third-party libraries and other protocol-specific code is written in JavaScript. The resulting code (1.5MB, 39Kloc) is quite hard to separate into components, let alone verify for security. We hope that our layered approach can lead to verified security guarantees without sacrificing performance or maintainability.

5.2.1 Isolating Verified Code

We build Cryptocat using Electron [152], a framework for JavaScript desktop applications. Electron is built on top of the Node.js JavaScript runtime and the Chromium web

renderer. Electron has recently gained strong adoption: popular applications such as Visual Studio Code, Atom, Slack, WordPress and WhatsApp Desktop are all built on top of Electron.

By default, Electron allows applications to load any Node.js low-level module, which can in turn perform dangerous operations like accessing the file system and exfiltrate data over the network. Since all Node.js modules in a single process run within the same JavaScript environment, malicious or buggy modules can tamper with other modules via prototype poisoning or other known JavaScript attack vectors. Consequently, all Electron apps, including other desktop Signal implementations like WhatsApp and Signal messenger effectively include all of Electron and Node.js into their trusted computing base (TCB).

We propose a two-pronged approach to reduce this TCB.

Language-Based Isolation. Since ProScript is a subset of Defensive JavaScript, ProScript protocol code is isolated at the language level from other JavaScript code running within the same process, even if this code uses dangerous JavaScript features such as prototype access and modification. To ensure this isolation, ProScript code must not call any external (untyped) libraries.

Process Thread Isolation. We exploit features of Electron in order to isolate components of our application in different CPU threads as seen in Figure 5.3. When a message arrives on the network (1), the main network thread can only communicate with our Protocol TCB using a restrictive inter-process communication API. The TCB then uses its internal verified protocol functionality and state management to return a decryption of the message (3), which is then forwarded again via IPC to the chat window (4), a third separate CPU thread which handles message rendering. Furthermore, the TCB process is disallowed from loading any Node.js modules.

In particular, the network process is isolated from the chat media rendering process; neither ever obtain access to the key state or protocol functionality, which are all isolated in the ProScript protocol process. When Bob responds, a similar IPC chain of calls occurs in order to send his reply back to Alice (5, 6, 7, 8). Even if an error in the rendering code or in the XML parser escalated into a remote takeover of the entire web renderer, the calls to the protocol TCB would be restricted to those exposed by the IPC API. However, these isolation techniques only protect the ProScript code within our application when executed within a correct runtime framework. None of these techniques can guard against bugs in V8, Node.js, or Electron, or against malicious or buggy Node.js or Electron modules loaded by the application.

5.2.2 Performance and Limitations

Although we have verified the core protocol code in Cryptocat and tried to isolate this code from unverified code, the following limitations still apply: we have not formally verified the soundness of the cryptographic primitives themselves, although writing them in Defensive JavaScript does provide type safety. We have also not formally verified the Electron framework's isolation code. Similarly, we do not claim any formal verification results on the V8 JavaScript runtime or on the Node.js runtime. Therefore, we rely on a number of basic assumptions regarding the soundness of these underlying components. Cryptocat's successful deployment provides a general guideline for building, formally verifying and isolating cryptographic protocol logic from the rest of the desktop runtime. Designing better methods for implementing and isolating security-critical components within Electron apps with a minimal TCB remains an open problem.

```
free io:channel.
type number. type function. type key. [...]
const string_63:bitstring [data]. (* WhisperMessageKeys *)
[...]
equation forall a:key, b:key;
  PS_crypto_DH25519(b, PS_crypto_DH25519(a, key_83)) =
  PS_crypto_DH25519(a, PS_crypto_DH25519(b, key_83)).
fun PS_crypto_AESCTREncrypt(key, iv, bitstring):bitstring.
reduc forall k:key, i:iv, m:bitstring; PS_crypto_AESCTRDecrypt(
  k, i, PS_crypto_AESCTREncrypt(k, i, m)) = m.
[...]
letfun fun_AKEResponse(me:me, them:them, msg:msg) =
let e = PS_crypto_random32Bytes(string_80) in let ge = PS_crypto_DH25519(e, key_83) in
let shared = fun_TDH0(keypair_get_priv(me_get_identity(me)), e, them_get_identity(them),
     msg_get_prekey(msg), msg_get_ephemeral(msg)) in
let recvKeys = fun_HKDF(shared, Type_key_construct(), string_56) in
let validSig = PS_crypto_checkED25519(them_get_identity(them), Type_key_toBitstring(
     msg_get_prekey(msg)), msg_get_prekeySig(msg)) in 0 [...]
free secMsg1:bitstring [private]. free secMsg2:bitstring [private].
query attacker(secMsg1). query attacker(secMsg2).
noninterf secMsg1. noninterf secMsg2.
event Send(key, key, bitstring). event Recv(key, key, bitstring).
query a:key,b:key,m:bitstring; event(Recv(a, b, m)) ⟹event(Send(a, b, m)).
[...]
let Alice(me:me, them:them) =
  let aStartSession = fun_startSession(me, them) in
  let them = sendoutput_get_them(aStartSession) in
  out(io, sendoutput_get_output(aStartSession));
  in(io, bAcceptSession:msg);
  let aAcceptSession = fun_recv(me, them, bAcceptSession) in
  let them = recvoutput_get_them(aAcceptSession) in
  let encMsg1 = fun_send(them, secMsg1) in
  let them = sendoutput_get_them(encMsg1) in
  event Send(keypair_get_pub(me_get_identity(me)), them_get_identity(them), secMsg1);
  out(io, sendoutput_get_output(encMsg1));
  phase 1; out(io, keypair_get_priv(me_get_identity(me))); in(io, encMsg2:msg);
  let decMsg2 = fun_recv(me, them, encMsg2) in
  if (msg_get_valid(recvoutput_get_output(decMsg2)) = true) then (
    let msg2 = recvoutput_get_plaintext(decMsg2) in
    event Recv(them_get_identity(them), keypair_get_pub(me_get_identity(me)), msg2)
  ).
let Bob(me:me, them:them) = [...]
let AliceM(me:me, them:them) = [...]
let BobM(me:me, them:them) = [...]
process
let alice = fun_newIdentity() in let bob = fun_newIdentity() in let mallory =
     fun_newIdentity() in
out(io, mallory);
let bobsAlice = [...] in let alicesMallory = [...] in let bobsMallory = [...]
(Alice(alice, alicesBob) | Bob(bob, bobsAlice) | AliceM(alice, alicesMallory) | BobM(bob,
     bobsMallory))
```

Figure 5.4: Extracted model with types, equations between primitives and human-readable protocol functionality.

Despite the strong programming constraints imposed by our verification architecture, we find that Cryptocat is able to perform similarly to mainstream desktop messaging applications that do not offer end-to-end encryption, such as Skype. In order to benefit from automatic model translation of our ProScript protocol implementation (as described in §5.1.5), we use PSCL as our cryptography library, which allows us to easily handle even large file encryptions (200+MB) for Cryptocat's file sharing feature. Our application is available for Windows, Linux and Mac and currently serves over 20,000 users weekly. It is capable of handling multi-device provisioning for users with soft device revocation and device authentication. We also support video messaging, file sharing and similar usability features, which all exploit the isolation and formal verification methods described in this chapter.

```
const Type_key = {
  construct: function() {
    return [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00, 0x00. 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
  },
  assert: function(a) {
    var i = 0; for (i = 0; i < 32; i++) {
      a[i&31] & 0x00}; return a
  },
  [...]
}
[...]
const RATCHET = {
  deriveSendKeys: function(them, newEphemeralPriv) {
    const kShared = ProScript.crypto.DH25519(newEphemeralPriv, them.ephemeral)
    const sendKeys = UTIL.HKDF(kShared, them.recvKeys[0], 'WhisperRatchet')
    const kKeys = UTIL.HKDF(
      ProScript.crypto.HMACSHA256(sendKeys[1], '1'),
      Type_key.construct(), 'WhisperMessageKeys'
    )
    return {
      sendKeys: sendKeys, recvKeys: them.recvKeys,
      kENC: kKeys[0], kMAC: kKeys[1]
    }
  },
  deriveRecvKeys: function(them, newEphemeralPub) { [...] }
}
[...]
const TextSecure = {
  newIdentity: function() { [...] },
  startSession: function(me, them) {
    var me = Type_me.assert(me)
    var them = Type_them.assert(them)
    return {
      them: them,
      output: {
        status: 1,
        valid: true,
        prekey: Type_key.construct(),
        ephemeral: Type_key.construct(),
        [...]
  }}},
  acceptSession: function(me, them, msg) { [...] },
  send: function(them, plaintext) { [...] },
  recv: function(me, them, msg) {
    var me = Type_me.assert(me)
    var them = Type_them.assert(them)
    var msg = Type_msg.assert(msg)
    const themMsg = {them: them, msg: msg}
    if ((msg.status === 2) && (them.status === 0)) {
      return recvHandlers.AKEResponse(me, them, msg)
    }
    else if ((msg.status === 3) && (them.status === 2)) {
      return recvHandlers.message(recvHandlers.completeAKE(me, them, msg))
    }
    else if ((msg.status === 3) && (them.status === 3)) {
      return recvHandlers.message(themMsg)
    }
    else { return {
       them: Type_them.construct(), output: Type_msg.construct(), plaintext: ''
}}}}
```

Figure 5.5: SP functionality is written in ProScript's idiomatic style which employs JavaScript's purely functional programming language features.

5.3 RefTLS: a Reference TLS 1.3 Implementation with a Verified Protocol Core

In today's web ecosystem, TLS is used by wide variety of client and server applications to establish secure channels across the Internet. For example, Node.js servers are written in JavaScript and can accept HTTPS connections using a Node's builtin `https` module that calls OpenSSL. Popular desktop applications, such as WhatsApp messenger, are also written in JavaScript using the Electron framework (which combines Node.js with the Chromium rendering engine); they connect to servers using the same `https` module.

Our goal is to develop a high-assurance reference implementation of TLS 1.3, called RefTLS, that can be seamlessly used by Electron apps and Node.js servers. We want our implementation to be small, easy to read and analyze and effective as an early experimental version of TLS 1.3 that real-world applications can use to help them transition to TLS 1.3, before it becomes available in mainstream libraries like OpenSSL. Crucially, we want to be able to verify the security of the core protocol code in RefTLS and show that it avoids both protocol-level attacks as well as implementation bugs in its protocol state machine.

In this section, we describe RefTLS and evaluate its progress towards these goals. RefTLS has been used as a prototype implementation of TLS Draft-13 to Draft-18, interoperating with other early TLS 1.3 libraries. Its protocol core has been symbolically analyzed with ProVerif and it has been successfully integrated into Electron applications.

5.3.1 Flow and ProScript

RefTLS is written in Flow [153], a typed variant of JavaScript. Static typing in Flow guarantees the absence of a large class of classic JavaScript bugs, such as reading a missing field in an object. Consequently, our code looks very much like a program in a typed functional language like OCaml or F#. We would like to verify the security of all our Flow code, but since Flow is a fully-fledged programming language, it has loops, mutable state and many other features that are hard to automatically verify.

Earlier, we developed a typed subset of JavaScript called ProScript [73] that was designed for writing cryptographic protocol code that could be compiled automatically to ProVerif. ProScript is also a subset of Flow and so we can reuse its ProVerif compiler to extract symbolic models from the core protocol code in RefTLS, if we write it carefully.

For ease of analysis, ProScript disallows loops, recursion and only allows access mutable state through a well defined `table` interface. These are significant restrictions, but as we show, the resulting language is still expressive enough to write the core composite protocol code for TLS 1.0-1.3.

5.3.2 Implementation Structure

Figure 5.6 depicts the architecture of RefTLS and shows how it can be safely integrated into larger, unverified and untrusted applications. The library is written in Flow, a typed subset of JavaScript. The protocol core is verified by translation to ProVerif. The cryptographic library, message formatting and parsing and the runtime framework are trusted. The application and parts of the RefTLS library are untrusted (assumed to be adversarial in our model). At the top, we have Node.js and Electron applications

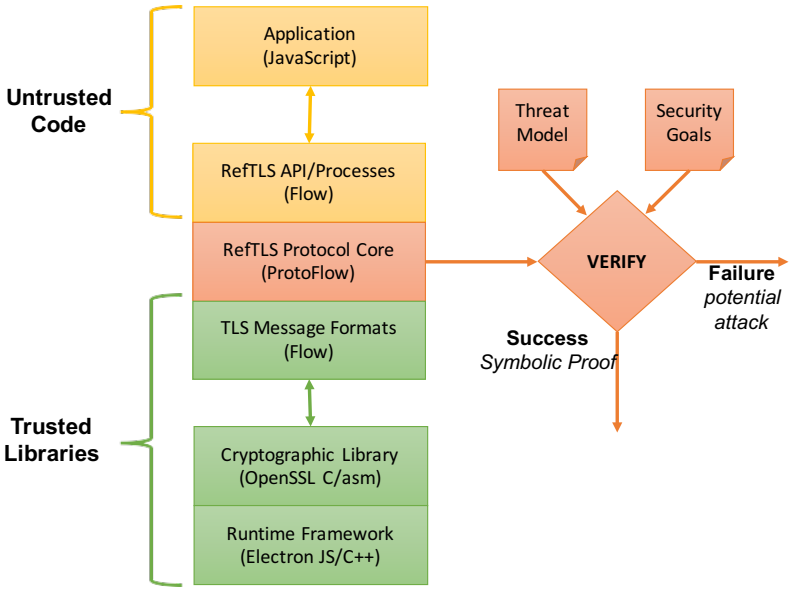

Figure 5.6: RefTLS Architecture.

written in JavaScript. RefTLS exposes an interface to these applications that exactly matches that of the default Node.js `https` module (which uses OpenSSL), allowing these applications to transparently use RefTLS instead of OpenSSL.

The RefTLS code itself is divided into untrusted Flow code that handles network connections and implements the API, a verified protocol module, written in ProScript and some trusted but unverified Flow code for parsing and serializing TLS messages. All this code is statically typechecked in Flow. The core protocol module, called RefTLS-CORE, implements all the cryptographic operations of the protocol. It exposes an interface that allows RefTLS to drive the protocol, but hides all keying material and sensitive session state within the core module. This isolation is currently implemented via the Node module system; but we can also exploit Electron's multi-threading feature in order to provide thread-based isolation to the RefTLS-CORE module, allowing it to only be accessed through a pre-defined RPC interface. Strong isolation for RefTLS-CORE allows us to verify it without relying on the correctness of the rest of the RefTLS codebase.

However, RefTLS still relies on the security and correctness of the crypto library and the underlying Electron, Node.js and JavaScript runtimes. In the future, we may be able to reduce this trusted computing base by relying on verified crypto [81], verified JavaScript interpreters [154] and least-privilege architectures, such as ESpectro [155], which can control access to dangerous libraries from JavaScript.

5.3.3 A Verified Protocol Core

In RefTLS-CORE, we develop, implement and verify (for the first time) a composite state machine for TLS 1.2 and 1.3. Each state transition is implemented by a ProScript function that processes a flight of incoming messages, changes the session state and produces a flight of outgoing messages. For TLS 1.3 clients, these functions are `get_client_hello`, `put_server_hello`, and `put_server_finished`; servers use the functions `put_client_hello`, `get_server_finished`, and `put_-`

Benchmark	RefTLS Client	Node.js Client	RefTLS Server	Node.js Server
TLS 1.2 Handshake	16ms	7ms	8ms	6ms
TLS 1.2 Handshake	36ms		59ms	
TLS 1.3 Handshake	34ms		28ms	
Fetch 100MB	1163ms	730ms		

Figure 5.7: In the first benchmark, RefTLS and Node.js's HTTPS module perform handshakes against OpenSSL. In the second and third benchmarks, we test RefTLS against itself. In the fourth benchmark, the clients attempt to fetch 100MB of data from OpenSSL over TLS 1.2.

`client_finished`.

We then use the ProScript compiler to translate this module into a ProVerif script that looks much like the protocol models described in earlier sections of this chapter. Each pure function in ProScript translates to a ProVerif function; functions that modify mutable state are translated to ProVerif processes that read and write from tables. The interface of the module is compiled to a top-level process that exposes a subset of the protocol functions to the adversary over a public channel.

The adversary can call these functions in any order and any number of times, to initiate connections in parallel, to provide incoming flights of messages, and to obtain outgoing flights of messages. The ProVerif model uses internal tables, not accessible to the attacker, to manage state updates between flights and preserve state invariants through the protocol execution.

Our approach allows us to quickly obtain verifiable ProVerif models from running RefTLS code. For example, we were able to rapidly prototype changes to the TLS 1.3 specification between Draft-13 and Draft-18, while testing for interoperability and analyzing the core protocol at the same time. In particular, we extracted a model from our Draft-18 implementation, and verified our security goals from §2.2 and §2.4 with ProVerif.

We engineered the ProScript compiler to generate readable ProVerif models that can be modified by a protocol analyst to experiment with different threat models. It is possible to extend the same automated translation approach towards CryptoVerif models. Recently, CryptoVerif syntax became similar to that of ProVerif. However, the kind of models that are easy to verify using CryptoVerif differ from the models that ProVerif can automatically verify and the assumptions on cryptographic primitives will always remain different. Therefore, even if the source syntax is the same, we may need to adapt our compiler to generate different models for ProVerif and CryptoVerif.

5.3.4 RefTLS Protocol State Machines

Client. The RefTLS client implements the composite state machine shown in Figure 5.8 for TLS 1.3 and TLS 1.2. Each state represents a point in the protocol where the client is either waiting for a flight of handshake messages from the server, or it has new session keys that it wishes to communicate to the record layer. Each arrow is annotated with the name of the function in RefTLS-CORE API that implements the corresponding state transition. Each transition may involve processing a flight of incoming messages, changing the session state and producing a flight of outgoing messages. **Server.** The RefTLS server implements a dual state machine for TLS 1.3 and TLS 1.2, as depicted

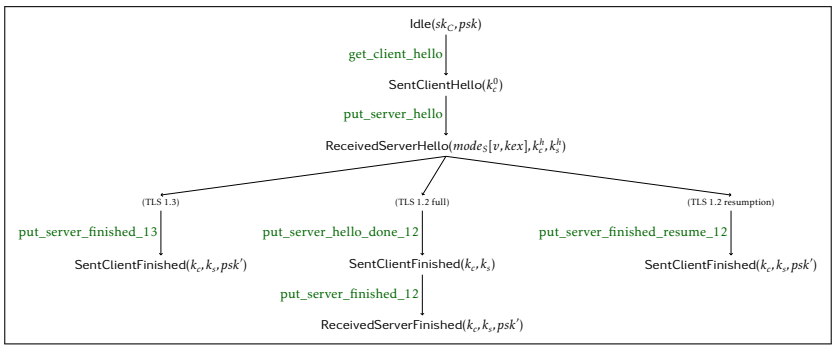

Figure 5.8: Client state machine

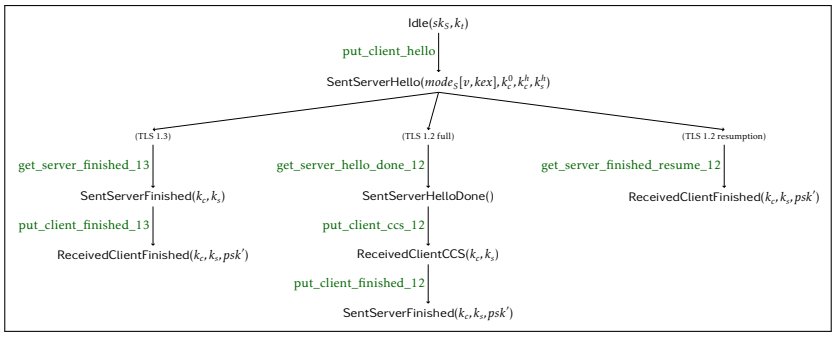

Figure 5.9: Server state machine

in Figure 5.9. The server decides which protocol version and key exchange the handshake will use, and triggers the appropriate branch in the state machine by sending a `ServerHello`. Like the client, each of its state transition functions corresponds either to a flight of messages or to a change of keys.

5.3.5 Evaluation: Verification, Interoperability, Performance

The full RefTLS codebase consists of about 6500 lines of Flow code, including 3000 lines of trusted libraries (mostly message parsing), 2500 lines of untrusted application code and 1000 lines of verified protocol core. From the core, we extracted an 800 line protocol model in ProVerif and composed it with our generic library from §2.1. Verifying this model took several hours on a powerful workstation.

RefTLS implements TLS 1.0-1.3 and interoperates with all major TLS libraries for TLS 1.0-1.2. Fewer libraries currently implement TLS 1.3, but RefTLS participated in the IETF Hackathon and achieved interoperability with other implementations of Draft-14. It now interoperates with NSS (Firefox) and BoringSSL (Chrome) for Draft-18.

By implementing Node's `https` interface, we are able to naturally integrate RefTLS within any Node or Electron application. We demonstrate the utility of this approach by integrating RefTLS into the Brave web browser, which is written in Electron. We are able to intercept all of Brave's HTTPS requests and reliably fulfill them through RefTLS.

We benchmarked RefTLS against Node.js's default OpenSSL-based HTTPS stack

when run against an OpenSSL peer over TLS 1.2. Our results are shown in Figure 5.7. In terms of computational overhead, RefTLS is two times slower than Node's native library, which is not surprising since RefTLS is written in JavaScript, whereas OpenSSL is written in C. In exchange for speed, RefTLS offers an early implementation of TLS 1.3 and a verified protocol core. Furthermore, in many application scenarios, network latency dominates over crypto, so the performance penalty of RefTLS may not be that noticeable.

5.4 Conclusion and Related Work

There have been previous attempts [76] to extract ProVerif models from typed JavaScript, such as *DJS2PV* [146]. However, DJS2PV was only tested on small code examples: attempting to translate a complete implementation such as Signal Protocol resulted in a 3,800 line model that attempts to precisely account for the heap, but could not verify due to an exploding state space. Previous efforts such as FS2PV [75] avoided this problem by choosing a purely functional source language that translated to simpler pi calculus scripts. We adopt their approach in ProScript to generate briefer, more readable models.

TypeScript [156], Flow [153], Defensive JavaScript and TS* [157] all define type systems that can improve the security of JavaScript programs. The type system in ProScript primarily serves to isolate protocol code from untrusted application and to identify a subset of JavaScript that can be translated to verifiable models. More recently, ESVerify[2] provides a subset of JavaScript that allows for implementing program verification logic within the syntax itself, which is then separately parsed with an SMT solver but avoids trigger heuristics and thereby (most) timeouts and other unpredictable results by requiring manual instantiation.

Tools like WebSpi [158] and AuthScan [159] have been used to verify the security of web security protocols such as OAuth. An expressive web security model has also been used to build manual proofs for cryptographic web protocols such as BrowserID [160]. These works are orthogonal to ProScript and their ideas can potentially be used to improve our target ProVerif models.

We plan to continue to develop and refine ProScript by evaluating how it is used by protocol designers, in the spirit of an open source project. All the code and models presented as part of this work are available online [93]. Proving the soundness of translation from ProScript to ProVerif, by relating the source JavaScript semantics to the applied pi calculus, remains future work. The process of transforming the compiled model to a verified CryptoVerif script remains a manual task, but we hope to automate this step further, based on new and upcoming developments in CryptoVerif.

Finally, a word of caution: a protocol written in ProScript and verified with ProVerif or CryptoVerif does not immediately benefit from assurance against all possible attacks. Programming in ProScript imposes a strict discipline by requiring defensive self-contained code that is statically typed and can be translated to a verifiable model and subsequent verification can be used to eliminate certain well-defined classes of attacks. We believe these checks can add confidence to the correctness of a web application, but they do not imply the absence of security bugs, since we still have a large trusted computing base. Consequently, improving the robustness and security guarantees of runtime frameworks such as Electron, Node.js and Chromium, remains an important area of future research.

[2] `https://esverify.org/`

Chapter 6

Formally Verified Secure Collaborative Document Editing

This work was completed shortly before the submission of the final draft of this thesis and is currently under peer review. It has received feedback from various sources, most notably the Security and Privacy Engineering Lab and the Decentralized and Distributed Systems Lab at the École Polytechnique Fédérale de Lausanne, who were gracious enough to invite me to present it in 2018.

Collaborative document editing software such as Google Docs and Microsoft Office365 has become indispensable in today's work environment. In spite of no confidentiality guarantees, large working groups still find themselves depending on these services even for drafting sensitive documents. Peer pressure, a rush to agree on a working solution and the sheer lack of alternatives have created an ecosystem where a majority of sensitive drafting is currently conducted with no end-to-end encryption guarantees whatsoever. Google and Microsoft servers have full access to all documents being collaborated upon and so do any parties with read access to the internal databases used by these services.

Capsule aims to solve this problem by being the first formally specified and formally verified protocol for secure collaborative document editing. Capsule comes with security goals that are validated in the symbolic model using the ProVerif [47] automated protocol verifier. It is also presented with a full client and server implementation, published as open source software.

We note that Capsule does not target the much simpler problem of encrypting cloud documents that are at rest, such as files stored on Google Drive on Microsoft OneDrive. Rather, the Capsule Protocol targets files being actively, collaboratively edited in real time.

Capsule's more involved use case scenario is responsible for providing a class of security guarantees vaguely resembling that of group secure messaging but without a need for forward secrecy. Another element that Capsule has to deal with is allowing the management of a document's incremental history with a server that cannot read the document's contents. This is accomplished by storing the document as *an authenticated hash chain of encrypted, signed diffs:* when a new participant joins a Capsule document, they pull the entire document hash chain, decrypting and parsing every diff until the entire document is reconstructed. Once within the document, each participant pushes and pulls encrypted diff blocks into this hash chain. Utilizing a hash chain also makes

it more difficult for the server to provide differing document histories to participants, without forking the hash chain entirely and with little payoff.

Capsule uses symmetric encryption to guarantee document confidentiality and integrity. Cryptographic signatures are used for authentication, although we employ only *ephemeral authentication* since identities are valid for the lifetime of the document. Symmetric primitives are also used for generating a proof value that disallows the server from adding non-existent participants to the document, in spite of these fake participants already being unable to access any document plaintext.

To the best of our knowledge, the only prior existing work regarding collaborative document encryption is CryptPad [83], an open source web client. CryptPad shares similarities with Capsule especially in that both use a hash chain of encrypted diffs in order to manage document collaboration and to reconstruct the document. However, CryptPad adopts a more relaxed threat model of an "honest but curious cloud server" and does not appear to guard against a server interfering with the document's list of participants or its history. Meanwhile, Capsule explicitly guards against a server injecting false participants by requiring a certain proof from all participants. CryptPad's software implementation is also limited within a web browser and unlike Capsule's, does not employ formally verified cryptographic primitives.

6.1 Security Goals and Threat Model

Capsule aims to guarantee the following security goals. In the following, a *valid participant* is any entity with access to the Capsule collaborative document's shared master secret. This master secret is generated upon document creation and is shared manually by the document creator in order to allow access to the document.

- **Participant List Integrity.** Only participants with access to the Capsule collaborative document's shared master secret may appear as valid entries on the participant list. Illegitimate entries injected by the server or any other parties must be detectable by valid participants.

- **Confidentiality.** Any changes made to a collaborative document may only be viewed by valid participants.

- **Integrity.** Encrypted diffs that are appended to the Capsule document's hash chain by a particular author cannot be tampered with by any other party.

- **Transcript Consistency.** A malicious server cannot selectively omit changes to the collaborative document short of entirely forking the hash chain. Furthermore, in no scenario may a malicious server maliciously *append* changes to the collaborative document.

Capsule assumes the following threat model, which is fairly standard for protocols aiming to provide end-to-end security:

- **Untrusted Network.** We assume that an attacker controls the network and so can intercept, tamper with and inject network messages.

- **Malicious Server.** We assume that a Capsule server may be potentially interested in misleading users with regards to the document's history and in the apparent list of identities participating in a collaborative document editing session.

6.2 Primitives

In designing the Capsule protocol, we wanted to focus on obtaining the smallest attack surface possible on an architectural cryptographic level. This is why our low-level cryptographic operations comprise of a restricted subset that is simple to illustrate. The practical details of which cipher suites are chosen to satisfy the security goals of these primitives is discussed in §6.5.2.

- **Hashing.** $\mathsf{HASH}(x) \longrightarrow y$. A standard one-way cryptographic hash function.
- **Hash-Based Key Derivation.** $\mathsf{HKDF}(k, \texttt{salt}) \longrightarrow (k_0, k_1, k_2, k_3)$. HKDF functions [92] are proven to be pseudorandom functions under a random oracle model, which is useful for aspects of Capsule's protocol design.
- **Encryption and Decryption.**
  - **Encryption.** $\mathsf{BENC}(ek, mk, p) \longrightarrow (\mathsf{ENC}_p, n)$. Where ek is an encryption key and mk is an authentication key. An authentication tag is generated, as is typical with authenticated symmetric ciphers.
  - **Decryption.** $\mathsf{BDEC}(ek, mk, \mathsf{ENC}_p, n) \longrightarrow \{p, \perp\}$ depending on whether decryption can be authenticated.
- **Signatures.**
  - **Key Generation.** $\mathsf{EDGEN}(sk) \longrightarrow pk$.
  - **Signing.** $\mathsf{EDSIGN}(sk, x) \longrightarrow \mathsf{SIG}_x$.
  - **Signature Verification.** $\mathsf{EDVERIF}(pk, x, sig_x) \longrightarrow \{\top, \perp\}$ depending on whether signature verification succeeds.

6.3 Protocol Description

During the lifetime of a Capsule collaborative document, there are only two subprotocols that a participant has to follow. The first is the key generation subprotocol which produces the necessary key material to conduct the session. The second is the hash chain [161] protocol, which we call *DiffChain*. By pushing and pulling encrypted diffs to the diff chain, participants can reconstruct the document upon joining the session and then begin exchanging modifications.

6.3.1 Key Material

The following key material is necessary for all participants.

Client Key Materials

A client A owns the following key material in relationship to collaborative document V:

- $V_k \xleftarrow{R} \{0,1\}^{128}$, a randomly generated master secret, selected by the document creator or otherwise obtained from the document creator. Acts as the lifetime token for legitimately joining a collaborative document.

- $(V_{ek}, V_{mk}, V_{sp}, V_{id}) \longleftarrow \mathsf{HKDF}(V_k, \texttt{CAPSULECORP})$.[1] A set of symmetric subkeys used for encryption, message authentication, proof of being a legitimate participants and to derive the document ID used by the server for bookkeeping.

- $AV_{sk} \xleftarrow{R} \{0,1\}^{256}$, a signing private key.

- $AV_{pk} \longleftarrow \mathsf{EDGEN}(AV_{sk})$.

- $AV_{pv} \longleftarrow \mathsf{PVCALC}(V_{sp}, V_{id}, A, AV_{pk}) = \mathsf{HMAC}(V_{sp},\ A \| V_{id} \| AV_{pk})$

In the above, PVCALC is simply a shorthand function for the HMAC construction following it.

Among the above values, AV_{pv} is called a *proof of participation value*: Since the server does not know V_{sp}, it cannot generate a xV_{pv} for any possible participant x. Validating this value, therefore, protects the collaborative document editing session from containing illegitimate participants who were not given access to V_k.

Server Key Materials

A server S obtains access to the following key materials in relationship to collaborative document V:

- V_{id}, used as the server-side identifier for the document.

- AV_{pk} as Alice's ephemeral identity. Optionally, the server can also use this to validate DiffChain blocks sent in by Alice for commitment.

- AV_{pv}, used by Alice to prove that her identity is being served as a legitimate participant to the collaborative document.

6.3.2 Session Setup

Suppose Alice (A) wants to create a new collaborative document. She generates the keys mentioned in §6.3.1 and then communicates the following key material to server[2] S:

$$A \xrightarrow{(V_{id}, AV_{pk}, AV_{pv})} S$$

The server creates the new document identified server-side as V_{id} and stores the triple (A, AV_{pk}, AV_{pv}) as the first participant to the document.

Session setup is now complete. In order to invite a fellow collaborator into the document, Alice simply shares V_k. Alice can choose to encode V_k as a string or QR code to make sharing easier. Once Bob obtains V_k, he too can immediately generate all necessary key material to join the document, encrypt and authenticate diff information and prove to the rest of the participants that his participation is legitimate.

When Bob (B) joins the document, the following occurs (the last message is repeated for every existing participant:)

[1] CAPSULECORP represents a salt encoded as a string.

[2] Transport layer protections for this and other communications listed below are beyond the scope of this protocol. For simplicity, we assume they are protected with a transport layer security model equivalent to a regular TLS 1.3 deployment.

$$B \xrightarrow{(V_{id}, BV_{pk}, BV_{pv})} S$$

$$B \xleftarrow{(V_{id}, AV_{pk}, AV_{pv})} S$$

At this stage, Bob may carry out several validation operations:

- Verifying that V_k maps to V_{id}.
- Depending on mutual authentication requirements, verifying that AV_{pk} is expected for some known identity for Alice.
- Verifying that $\mathsf{HMAC}(V_{sp}, A \| V_{id} \| AV_{pk}) = AV_{pv}$.

6.3.3 Managing Collaborative Document History with DiffChain

As a protocol, Capsule needs to provide a fast, efficient method to allow for the continuous update of a document by an unbounded number of participants and for the constant synchronization of this document between the participants. Aside from homomorphic encryption, which currently does not appear to be ready for use in this kind of real-world system, the other clear potential solution seemed to be an encrypted, append-only authenticated log of diff information. We construct such a data structure and call it DiffChain.

Pulling a DiffChain to reconstruct a collaborative document upon joining it is very similar to pulling the Bitcoin blockchain and parsing it in order to reconstruct the currency's current value and activity by going through all transactions since the original block. A DiffChain block contains the following data:

- **Sender's Identity and Public Keys.**
- **Encrypted Diff.** Alice generates this encrypted diff using $\mathsf{BENC}(V_{ek}, V_{mk}, \texttt{diff})$.
- **Hash of Previous Block.** This is a standard element in hash chain constructions.
- **ID of Current Block.** This is conventionally a UNIX timestamp. It is appended by the server.
- **ID of Previous Block.** This allows for faster lookups.
- **Signature.** All of the above fields are included in a single signature, with the exception of the ID of the current block since it is appended by the server.

Responsible pushing and pulling. As a rule, clients should not push any new blocks before obtaining confirmation from the server that their local diff state is current. This will help avoid merge conflicts.

Diffing algorithm. The Capsule library comes equipped with an efficient diff generation algorithm that features a JSON-compatible syntax for compatibility. Independent implementations are free to adapt their own diff payload representations, as this does not strictly affect the security or operation of the cryptographic protocol itself.

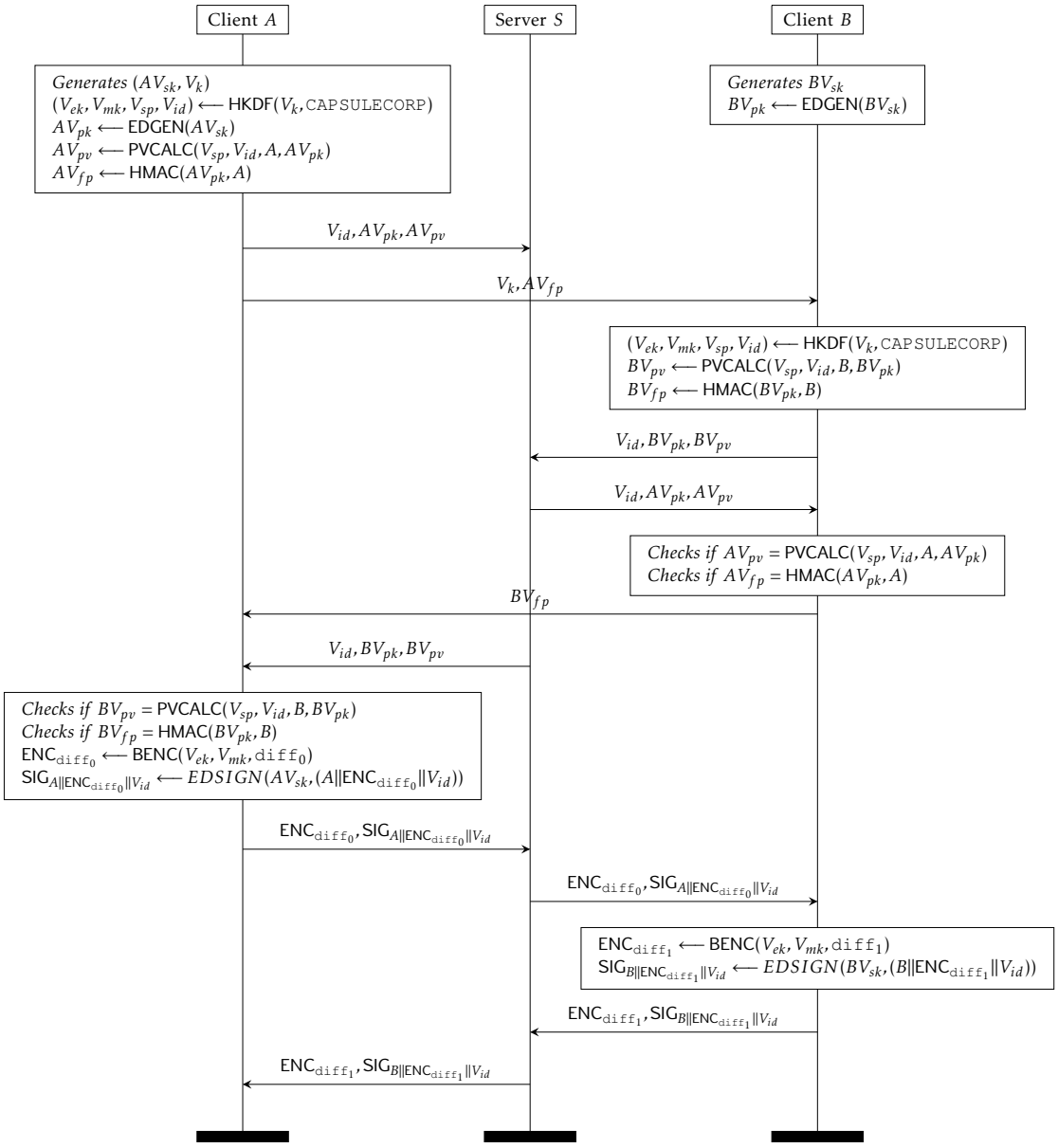

Figure 6.1: Capsule protocol session network messages. Here, A initiates a session with S and then communicates session key V_k and her own fingerprint AV_{fp} to B out of band. After B joins and all checks pass, A commits an example encrypted diff, $\texttt{diff}_0$ and B follows up with an example diff $\texttt{diff}_1$. More involved sessions can include clients C, D, etc. with a non-deterministic diff commit schedule.

6.3.4 Protocol Variant: Unauthenticated Encryption

We observe that if all participants accept ciphertexts signed only by identities with a valid XV_{pv} value for any participant X, then the authentication component of the ENC becomes redundant. This observation allows potentially replacing ENC with a unauthenticated encryption cipher, which could lead to improved performance.

Here is our argument for this: we assume that all clients are honest. We assume that HMAC is a one-way keyed hash function, that V_k is secret and that AV_{pk} is authenticated as a public signing identity belonging to A:

- If value AV_{pv} is validated, this proves that generator A must possess V_{sp}.
- Given that V_k is secret, ownership of V_{sp} in turn proves that A must possess V_k.
- At no point in a correct execution of the protocol are the values (V_{ek}, V_{mk}, V_{sp}) leaked by any participant.
- V_k maps into $(V_{ek}, V_{mk}, V_{sp}, V_{id})$. Therefore, proving ownership of V_{sp} under a secret V_k is equivalent to proving ownership of $(V_{ek}, V_{mk}, V_{sp}, V_{id})$. This satisfies the original purpose of validating the proof of participation value.
- A cannot produce AV_{pv} without also being able to produce (V_{ek}, V_{mk}). Since signature verification disallows tampering, any unauthenticated ciphertext encrypted under V_{ek} and signed by A is equivalent to a ciphertext encrypted under V_{ek}, authenticated using V_{mk} and signed by A.

6.4 Symbolic Verification with ProVerif

We describe Capsule using the applied-pi calculus and, using the ProVerif symbolic protocol verifier, verify that the Capsule protocol meets its security goals under its proposed threat model.

In the symbolic model, cryptographic primitives are logically perfect black boxes: hash functions are one-way mappings, encryption functions are permutations, random values are unique and perfectly indistinguishable. Processes can then be executed in parallel. In this setting, a Dolev-Yao [55] active attacker exists and attempts to reconstruct inputs and outputs in such a way as to reach goals defined by queries and events.

6.4.1 Capsule Processes in ProVerif

In ProVerif, we describe a single public channel, `pub`, intending to represent regular Internet network exchanges. Then, two types of client processes are described. We illustrate them here in an abbreviated format.[3]

Writer Client Processes

The writer process running under identity W joins a document V using a pre-shared V_k. It calculates WV_{pk}, emits a `ProofSent` event and sends $(V_{id}, W, WV_{pk}, WV_{pv})$ over the network. What follows next is an unbounded execution of the `writeBlock` process, with unbounded newly instantiated plaintext diff. Note that for modeling purposes, we "tag" the plaintext diff using the construction `secretdiff` which takes in a

[3] Capsule models and reference implementations are available at `https://symbolic.software/capsule/`.

value of type `plaintext` and produces a value of type `plaintext`. This constructor is paired with a reductor, `issecretdiff`, that simply reveals whether its input value was constructed using `secretdiff`.

```
let writerClient(vk:key, w:principal, sk:key) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let pv = pvcalc(vsp, vid, w, pk) in
event ProofSent(w, pv);
out(pub, (vid, w, pk, pv));
!(
  new diffx:plaintext;
  writeBlock(vk, w, sk, secretdiff(diffx))
).
```

The `writeBlock` process is self-explanatory:

```
let writeBlock(
  vk:key, w:principal, sk:key, diffx:plaintext
) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let ediff = benc(vek, vmk, diffx) in
let ediffsig = edsign(
  sk, concat3(prin2bit(w), ediff, key2bit(vid))
) in
event Pushed(vid, w, pk, ediff, ediffsig);
out(pub, (vid, w, pk, ediff, ediffsig)).
```

Reader Client Processes

The reader process running under participant identity R comes equipped with a value $UV_{fp} \longleftarrow \mathsf{HMAC}(UV_{pk}, U)$ that serves as a fingerprint for authenticating public identities and their public signing keys out of band.

Mirroring the writer processes above, the reader process begins by similarly generating the requisite values and communicating them over the network. Then, an unbounded execution of the `readBlock` process occurs.

```
let readerClient(
  vk:key, r:principal, sk:key, ufp:bitstring
) =
let pk = edgen(sk) in
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
let pv = pvcalc(vsp, vid, r, pk) in
event ProofSent(r, pv);
out(pub, (vid, r, pk, pv));
!(readBlock(vk, r, ufp)).
```

The `readBlock` process receives over the network a triple of candidate values for $(U, UV_{pk}UV_{pv})$ with a header matching the V_{id} value that the reader was able to obtain using the V_k the process was initialized with. Then, the internal UV_{fp} and UV_{pv} are checked against the candidate values received over the network. If the check succeeds, a `ProofVerified` event is emitted. Then, an encrypted diff along with its signature is received over the network with a header matching (V_{id}, U, UV_{pk}). Once signature and authenticated encryption checks pass, the *Pulled* event is emitted.

```
let readBlock(
  vk:key, r:principal, ufp:bitstring
) =
let (
  vek:key, vmk:key, vsp:key, vid:key
) = bkdf(vk, capsulecorp) in
in(pub, (
  =vid, user0:principal,
  upk0:key, upv0:bitstring
));
let ufp0 = hmac(upk0, prin2bit(user0)) in
let upv = pvcalc(vsp, vid, user0, upk0) in
if ((ufp0 = ufp) && (upv0 = upv)) then (
  event ProofVerified(user0, r, upv);
  in(pub, (
    =vid, =user0, =upk0,
    ediff0:bitstring, ediff0sig:bitstring
  ));
  let sigver0 = edverif(
    upk0, ediff0sig,
    concat3(
      prin2bit(user0), ediff0, key2bit(vid)
    )
  ) in
  if (sigver0 = true) then (
    let vs0:valid = bdec(vek, vmk, ediff0) in
    let (
      v0:bool, s0:plaintext
    ) = validunpack(vs0) in
    if (v0 = true) then (
      event Pulled(
        vid, user0, r, upk0,
        ediff0, ediff0sig
      )
    ))).
```

Attacker and Top-level Processes

Before declaring the top-level process, we must declare a process where the attacker attempts to obtain the plaintext for any diff:

```
let attackerTryingToObtainPlaintext() =
in(pub, x:plaintext);
if (issecretdiff(x) = true) then (
  event PlaintextObtainedByAttacker(x)
```

```
5| ).
```

Now, we can finally declare the top-level process. We model an unbounded number of sessions, each with a new V_k and its own unbounded instantiations of reader and writer principals with their own unique signing key pairs.

```
1  process
2  out(pub, msk) |
3  !(
4    new vkx:key;
5    !(
6      new alice:principal;
7      new bob:principal;
8      new ask:key;
9      new bsk:key;
10     out(pub, (alice, bob, sigpk(ask), sigpk(bsk)));
11     !(
12       writerClient(vkx, alice, ask) |
13       writerClient(vkx, bob, bsk) |
14       readerClient(
15         vkx, alice, ask,
16         hmac(sigpk(bsk), prin2bit(bob))
17       ) |
18       readerClient(
19         vkx, bob, bsk,
20         hmac(sigpk(ask), prin2bit(alice))
21       ) |
22       attackerTryingToObtainPlaintext()
23     )))
```

6.4.2 Security Goals in the Symbolic Model

We apply ourselves on the security goals described in §6.1 in order to model and verify participant list integrity, confidentiality, integrity and ephemeral authentication in ProVerif.[4]

Participant List Integrity

We assert that if a proof of participation value XV_{pv} is verified by participant Y, then there must exist an identity X which has issued XV_{pv}. As shown in §6.3.1, this means that whoever asserts identity X must also possess V_{sp}:

$$\text{event ProofVerified}(X, Y, XV_{pv}) \implies \text{event ProofIssued}(X, XV_{pv})$$

Confidentiality

We simply query for whether the attacker can trigger the plaintext obtention event described above for any `plaintext`-type bitstring m:

```
query event PlaintextObtainedByAttacker(m)
```

[4] Transcript consistency, while not explicitly modeled in ProVerif, is obtained through the hash chain structure of DiffChains, as described in §6.3.3.

Integrity and Ephemeral Authentication

We assert that if a DiffChain block $(\mathsf{ENC}_{\texttt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\texttt{diff}}\|V_{id}})$ was received by Y as part of document V, then this block will successfully authenticate and decrypt as originating from identity X if and only if it was pushed by X for document V.

$$\begin{aligned}\mathsf{event\ Pulled}(V_{id}, X, Y, XV_{pk}, \mathsf{ENC}_{\texttt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\texttt{diff}}\|V_{id}}) \implies \\ \mathsf{event\ Pushed}(V_{id}, X, XV_{pk}, \mathsf{ENC}_{\texttt{diff}}, \mathsf{SIG}_{X\|\mathsf{ENC}_{\texttt{diff}}\|V_{id}})\end{aligned}$$

6.4.3 Results Under a Dolev-Yao Attacker

All queries mentioned above complete successfully. By tweaking the model, we are also able to quickly test for the event where an ephemeral identity is not verified out-of-band by the other participants. Since all encryption is symmetric and no asymmetric key agreement ever occurs in the Capsule protocol, the outcome of this is minor and does not result in the compromise of confidential information.

6.5 Software Implementation

Capsule is provided with a client/server reference implementation called Capsulib. It is meant to allow for quick deployment and testing of the Capsule protocol in production.

6.5.1 Capsulib: a Capsule Client/Server Implementation

Capsulib Server is a Node.js application that uses Redis as a database backend but is otherwise self-contained. It achieves low latency by communicating over WebSockets. Its main purpose is to hold a DiffChain record for every document and to facilitate the exchange of encrypted blocks.

Capsulib Client is meant to be executed within an Electron runtime. It provides a user interface, the Capsulib cryptographic library and the full Capsule client protocol implementation.

6.5.2 Cryptographic Cipher Suite

We expose the following cryptographic functions:

- **Hashing.** $\mathsf{HASH}(x) \longrightarrow y$. BLAKE2s is used as the hash function.
- **Hash-Based Key Derivation.** $\mathsf{BKDF}(k, \texttt{salt}) \longrightarrow (k_0, k_1, k_2, k_3)$. BLAKE2s is also used for key derivation, producing four 16-byte outputs.
- **Encryption and Decryption.** BLAKE2 is used for encryption and decryption. We hash the encryption key over a counter and a nonce to generate a keystream. BLAKE2 is then used as a keyed hash with a MAC key to generate an HMAC value over the ciphertext.
  - **Encryption.** $\mathsf{BENC}(ek, mk, p) \longrightarrow (\mathsf{ENC}_p, n)$.
  - **Decryption.** $\mathsf{BDEC}(ek, mk, \mathsf{ENC}_p, n) \longrightarrow p$.
- **Signatures.** Ed25519 is chosen due to its speed and minimal input validation requirements.

- **Key Generation.** $\mathsf{EDGEN}(sk) \longrightarrow pk$.
- **Signing.** $\mathsf{EDSIGN}(sk, x) \longrightarrow \mathsf{SIG}_x$.
- **Signature Verification.** $\mathsf{EDVERIF}(pk, x, sig_x) \longrightarrow \{\top, \bot\}$.

6.5.3 Formally Verified Cryptographic Primitives in WebAssembly

While Capsulib is written in JavaScript, much of the Capsulib cryptographic primitives library is automatically generated as WebAssembly (WASM) [162] code. We use a toolchain that provides strong verification guarantees, hence ruling out potentially catastrophic bugs in the cryptographic layer.

Ed25519 is specified in F* [163], an ML-like programming language with support for program verification via the use of a dependent type system, effect annotations and SMT-based automation. Our specifications are executable, which allows us to run them against the RFC test vectors to ensure their correctness.

The cryptographic algorithms themselves are written in Low*, a subset of F* that enjoys an optimized compilation scheme to low-level targets. The Low* implementation of our algorithms is shown to match the original F* specification, which ensures functional correctness and memory safety.

Based on this, we rule out both incorrect math and memory errors such as buffer overflows. In addition to function correctness and memory safety, we enforce basic side-channel resistance, by restricting secrets to a very limited set of operations. In effect, this ensures that secret-manipulating code is branch-free and cannot access memory using a secret index. This helps rule out classic timing and cache attacks.

Once verified, Low* programs can be compiled to low-level targets using the Kremlin [164] compiler, which currently supports C and WASM as backends. Compared to Emscripten, the Kremlin compiler offers a much smaller trusted computing base, smaller resulting WASM files and almost no dependencies on untrusted code, such as a `libc` implementation.

6.6 Conclusion

In this chapter, we have presented Capsule, the first formally verified protocol for secure collaborative document editing. We believe Capsule will address an important need in the space of privacy enhancing technologies. We have provided a full description of the Capsule protocol and symbolic verification results for its security goals under the specified threat model.

Finally, we also provide Capsulib, a reference implementation of Capsule based on modern web technologies. While Capsulib is built for web use, it is also the first software project to employ the HACL* formally verified cryptographic library by translating it into WASM. By adopting such technologies, Capsule protocol is able to be presented complete with a practical implementation that follows best practices and achieves sound practical security in deployment.

Conclusion

Instead of repeating the research conclusions of the preceding chapters in a rote dry and useless way that assuredly nobody will read or enjoy reading, it is likely more useful to conclude this thesis with the broad lessons that I as a student have been able to glean during the time I've spent working on this thesis.

On the bright side, the repeated conventional notion that formal methods are too isolated to have any real meaning to contribute to real world protocols being used by millions of people on smartphones is, apparently, proven wrong by the work presented in this thesis. We have been able to show, repeatedly, that modeling the most ambitious, widely deployed and important secure channel protocols can be done methodically and predictably with existing symbolic and computational verification tools.

In the case of the Noise Protocol Framework, we were able to go beyond that and even allow end-users access to an easy-to-use web interface where they can type out their own Noise Protocol Handshake designs and immediately obtain validation and formal model generation for the protocol they've just described. Then, we showed that the results of that formal model's analysis can be viewed and examined in an equally accessible and easy to understand way. This is why Noise Explorer was my favorite project in this thesis, since I was able to implement my initial vision for the sort of result I wanted in this type of research ever since I began my thesis.

Work of the sort that was done for Noise Explorer may very well be superseded by more rigorous computational models simulating game-based proofs. F⋆ implementations of TLS have already been under way for years and will very obviously be followed with similar implementations for Signal, the Noise Protocol Framework and everything else under the sun.

If frameworks such as F⋆ are to indeed be the uncontested future, however, they must overcome current ease of use, accessibility and stability hurdles that preclude their usefulness to the average cryptography engineer and protocol designer. As of the date that this thesis was submitted, F⋆ faces considerable usability barriers likely to be encountered by everyone except its co-creators and a small number of specialists. This is due to a constantly evolving toolchain as well as constantly changing fundamental types, data structures and language syntax, without an underlying namespacing convention to tie everything together. Luckily, these are all limitations that can easily be addressed with enough time and doubtlessly will be. The underlying result, at that stage, is likely to be a huge leap in how safe programs are written and how safe protocols are deployed.

In that sense, this thesis has nothing to offer in the way of leaps and bounds. It is more appropriately seen as a bridge for testing, and in most cases justifying, the reaches of formal verification into real-world cryptography. Its impact can be measured more romantically by it resulting in the first full-featured secure messenger client with a formally verified protocol core. More practically, however, the fact that we were able to automatically verify TLS 1.3 drafts *as they were being released* is a much larger impact.

This was followed by a framework where any arbitrary protocol described in a certain style (i.e. the Noise Protocol Framework) could equally be prototyped and verified. Our results on this latter front led to a more rigorous approach to the design of the Noise Protocol Framework as a whole and to the evaluation of all the secure channel protocols that are coming out of it.

This thesis ends with chapter describing a novel protocol targeting an underserved use case. We present that protocol immediately with formal models that both illustrate the design and justify its soundness under the simulation of a real-world network attacker. The sentence that this final chapter punctuates is: "Can existing major paradigms for formal protocol verification serve as guidelines for the assessment of existing protocols, the prototyping of protocols being designed, and the conception of entirely new protocols, in a way that is *meaningful* and reflective of their expected real-world properties? And can we develop novel frameworks for formal verification and more secure implementation based on these foundations?" While no multi-chaptered body of research can responsibly end with a definitive answer, my belief is that this thesis has managed to constitute perhaps the most resounding "yes" yet, and that it will serve as motivation and encouragement for a fount of future work to bring verification and implementation ever closer together.

Bibliography

[1] WhatsApp encryption overview, 2017. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[2] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[3] Trevor Perrin and Moxie Marlinspike. The X3DH key agreement protocol, 2016. https://signal.org/docs/specifications/x3dh/.

[4] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/.

[5] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage. In *USENIX Security Symposium*, pages 655--672, 2016.

[6] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 77--84. ACM, 2004.

[7] Joseph Bonneau and Andrew Morrison. Finite State Security Analysis of OTR Version 2. http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf, 2006.

[8] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. 2018.

[9] Lucian Armasu. Signal desktop affected by two similar remote code execution bugs, 2018. https://www.tomshardware.com/news/signal-desktop-remote-code-execution,37063.html.

[10] John Dunn. Serious xss vulnerability discovered in signal, 2018. https://nakedsecurity.sophos.com/2018/05/16/serious-xss-vulnerability-discovered-in-signal/.

[11] K. E.B. Hickman. The SSL protocol, 1995. IETF Internet Draft, https://tools.ietf.org/html/draft-hickman-netscape-ssl-00.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[13] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Electronic Commerce*, 1996.

[14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[15] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using SSLv2. In *USENIX Security Symposium*, pages 689--706, 2016.

[16] Nadhem AlFardan, Daniel J Bernstein, Kenneth G Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of RC4 in TLS. In *USENIX Security Symposium*, pages 305--320, 2013.

[17] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, pages 97--112, 2015.

[18] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, pages 526--540, 2013.

[19] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 5--17, 2015.

[20] Karthikeyan Bhargavan and Gaetan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[21] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. IETF RFC 5746, 2010.

[22] 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 Symposium on Security & Privacy (Oakland)*, pages 98--113, 2014.

[23] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[24] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research (SSR)*, pages 160--186, 2016.

[25] Ueli Maurer and Björn Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 505--515, 2010.

[26] Kenneth G Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, pages 372--389, 2011.

[27] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, pages 273--293, 2012.

[28] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013*, pages 429--448, 2013.

[29] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198--207, 1983.

[30] Sagar Chaki and Anupam Datta. Aspier: An automated framework for verifying security protocol implementations. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 172--185. IEEE, 2009.

[31] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Verified cryptographic implementations for TLS. *ACM TOPLAS*, 15(1):3:1--3:32, 2012.

[32] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, 2013.

[33] Gavin Lowe. An attack on the needham- schroeder public- key authentication protocol. *Information processing letters*, 56(3), 1995.

[34] Roger M Needham and Michael D Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993--999, 1978.

[35] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. Verified contributive channel bindings for compound authentication. In *Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[36] Trevor Perrin. The Noise protocol framework, 2015. Available at `http://www.noiseprotocol.org`.

[37] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. Internet X.509 Public Key Infrastructure: Certificate Policy and Certification Practices Framework. RFC 3647, Internet Engineering Task Force, November 2003.

[38] CA/Browser Forum. Baseline requirements for the issuance and management of policy-trusted certificates, v.1.1.5, May 2013. `https://www.cabforum.org/Baseline_Requirements_V1_1_5.pdf`.

[39] Gervase Markham, Ryan Sleevi, Richard Barnes, and Kathleen Wilson. Wosign and startcom. `https://docs.google.com/document/d/1c6blmbeqfn4a9zydvi2uvjbgv6szusb4smyucvrr8vq/preview`.

[40] Olivier Levillain, Arnaud Ébalard, Benjamin Morin, and Hervé Debar. One year of SSL Internet measurement. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 11--20, New York, NY, USA, 2012. ACM.

[41] Antoine Delignat-Lavaud, Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, Yinglian Xie, and Microsoft Research. Web pki: Closing the gap between guidelines and practices. In *NDSS*, 2014.

[42] Google. Certificate transparency. `https://sites.google.com/site/certificatetransparency/`.

[43] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. Arpki: Attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 382--393, New York, NY, USA, 2014. ACM.

[44] Internet Security Research Group. Let's encrypt overview, 2016. `https://letsencrypt.org`.

[45] Internet Security Research Group. Let's encrypt statistics, 2016. `https://letsencrypt.org/stats/`.

[46] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic certificate management environment (acme). `https://tools.ietf.org/html/draft-ietf-acme-acme-03`, Jul 2016.

[47] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226--246. Springer, 2013.

[48] 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.

[49] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar Formal Protocol Verification Applied*, page 117, 2007.

[50] Steve Schneider. *Using CSP for protocol analysis: the Needham-Schroeder public-key protocol*. 1996.

[51] Gavin Lowe. Lowe's fixed version of needham-schroder public key. `http://www.lsv.fr/Software/spore/nspkLowe.html`, 1995.

[52] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust (POST)*, pages 3--29, 2012.

[53] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193--207, 2008.

[54] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1--1:41, 2018.

[55] Paul Syverson, Catherine Meadows, and Iliano Cervesato. Dolev-Yao is no better than machiavelli. Technical report, Naval Research Lab, Washington DC Center for High Assurance Computing Systems, 2000.

[56] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *International conference on computer aided verification*, pages 281--285. Springer, 2005.

[57] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225--259, 2011.

[58] Bruno Blanchet, Ben Smyth, and Vincent Cheval. Proverif 1.90: Automatic cryptographic protocol verifier, user manual and tutorial, 2014.

[59] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. IACR Cryptology ePrint Archive, 2004. `http://eprint.iacr.org/2004/332`.

[60] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology (Eurocrypt)*, pages 409--426, 2006.

[61] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEEE Proceedings Information Security*, 153(1):27--39, March 2006.

[62] Bruno Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 325--339, Cambridge, MA, USA, June 2012. IEEE.

[63] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178--194, Oakland, California, May 1993.

[64] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 97--111, 2007.

[65] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1197--1210, 2015.

[66] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *IEEE European Symposium on Security & Privacy (Euro S&P)*, 2016. Cryptology ePrint Archive, Report 2015/978.

[67] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu. Multiple handshakes security of TLS 1.3 candidates. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 486--505, 2016.

[68] 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 '17, pages 1773--1788, New York, NY, USA, 2017. ACM.

[69] Jason A Donenfeld. WireGuard: next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[70] Benjamin Lipp. A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol. https://benjaminlipp.de/master-thesis.

[71] Jason Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol, 2017. https://www.wireguard.com/formal-verification/.

[72] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. Cryptology ePrint Archive, Report 2018/080, 2018. https://eprint.iacr.org/2018/080.

[73] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[74] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 483--502. IEEE, 2017.

[75] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1), 2008.

[76] M. Avalle, A. Pironti, R. Sisto, and D. Pozza. The Java SPI framework for security protocol implementation. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 746--751, Aug 2011.

[77] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium*, pages 653--670, 2013.

[78] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally secure protocol specifications. *Journal of Computer Security*, 23(3):331--402, 2015.

[79] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 445--456, 2010.

[80] 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 Symposium on Principles of Programming Languages (POPL)*, pages 256--270, 2016.

[81] 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.

[82] Ralf Küsters, Tomasz Truderung, and Juergen Graf. A framework for the cryptographic verification of Java-like programs. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 198--212, 2012.

[83] CryptPad: Zero Knowledge, Collaborative Real Time Editing, 2016. https://cryptpad.fr/what-is-cryptpad.html.

[84] K. Bhargavan, A.D. Lavaud, C. Fournet, A. Pironti, and P.Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 98--113, 2014.

[85] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How Secure is TextSecure? *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[86] Hugo Krawczyk. HMQV: A High-performance Secure Diffie-Hellman Protocol. In *International Conference on Advances in Cryptology (CRYPTO)*, pages 546--566, 2005.

[87] Nikolai Durov. Telegram mtproto protocol, 2015. `https://core.telegram.org/mtproto`.

[88] Oliver Schirokauer. The number field sieve for integers of low weight. *Mathematics of Computation*, 79(269):583--602, 2010.

[89] Daniel Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS), 2016. IETF RFC 7919.

[90] 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.

[91] Jakob Jakobsen and Claudio Orlandi. On the cca (in)security of mtproto. Cryptology ePrint Archive, Report 2015/1177, 2015. `http://eprint.iacr.org/2015/1177`.

[92] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology (CRYPTO)*, pages 631--648. 2010.

[93] Nadim Kobeissi. SP code repository. https://github.com/inria-prosecco/proscript-messaging, February 2017.

[94] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE-protocols. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 400--425. Springer, 2003.

[95] Tatsuaki Okamoto and David Pointcheval. The gap-problems: a new class of problems for the security of cryptographic schemes. In *Practice and Theory in Public Key Cryptography (PKC)*, pages 104--118, 2001.

[96] Atsushi Fujioka and Koutarou Suzuki. Designing efficient authenticated key exchange resilient to leakage of ephemeral secret keys. In *Topics in Cryptology (CT-RSA)*, pages 121--141, 2011.

[97] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography (PKC)*, pages 207--228, 2006.

[98] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281--308, April 1988.

[99] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology (CRYPTO)*, pages 430--448, 2005.

[100] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology (CRYPTO)*, pages 602--619, 2006.

[101] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, volume 3348 of *Lecture Notes on Computer Science*, pages 343--355, Chennai, India, December 2004. Springer Verlag.

[102] Phillip Rogaway. Authenticated-encryption with associated-data. In *Ninth ACM Conference on Computer and Communications Security (CCS-9)*, pages 98--107, Washington, DC, November 2002. ACM Press.

[103] Yevgeniy Dodis, Thomas Ristenpart, John Steinberger, and Stefano Tessaro. To hash or not to hash again? (In)differentiability results for H2 and HMAC. In *Advances in Cryptology (Crypto)*, pages 348--366, 2012.

[104] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On Post-Compromise Security. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 164--178. IEEE, 2016.

[105] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 451--466. IEEE, 2017.

[106] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: asynchronous group messaging with strong security guarantees. Technical report, IACR Cryptology ePrint Archive, 2017.

[107] Karthikeyan Bhargavan, Eric Rescorla and Richard Barnes. Tree-based KEM for group key management, 2018. `https://github.com/bifurcation/treekem`.

[108] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-based multiplexed and secure transport, 2016. IETF Internet Draft.

[109] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 470--485, 2016.

[110] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 452--469, 2016.

[111] Hugo Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1438--1450, 2016.

[112] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella Béguelin. Downgrade resilience in key-exchange protocols. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 506--525, 2016.

[113] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1185--1196, 2015.

[114] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*, volume 1462 of *Lecture Notes on Computer Science*, pages 1--12. Springer Verlag, 1998.

[115] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*, pages 733--748. USENIX Association, 2014.

[116] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 456--467, 2016.

[117] Martin R Albrecht and Kenneth G Paterson. Lucky microseconds: A timing attack on Amazon's S2N implementation of TLS. In *EUROCRYPT*, pages 622--643, 2016.

[118] Marsh Ray, Alfredo Pironti, Adam Langley, Karthikeyan Bhargavan, and Antoine Delignat-Lavaud. Transport Layer Security (TLS) session hash and extended master secret extension, 2015. IETF RFC 7627.

[119] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1193--1204, 2014.

[120] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 214--231, 2015.

[121] Eric Rescorla. 0-RTT and Anti-Replay. `https://www.ietf.org/mail-archive/web/tls/current/msg15594.html`, March 2015.

[122] NIST. FIPS 186-3: Digital Signature Standard (DSS). Available at `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, June 2009.

[123] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, June 2015. Available at `https://eprint.iacr.org/2015/625`.

[124] D. Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for Transport Layer Security (TLS), August 2016. `http://tools.ietf.org/rfc/rfc7919`.

[125] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. IRTF RFC 7748 https://tools.ietf.org/html/rfc7748, January 2016.

[126] Eric Rescorla. [TLS] PR#875: Additional Derive-Secret stage. https://www.ietf.org/mail-archive/web/tls/current/msg22373.html, February 2017.

[127] Ivan Bjerre Damgård. A design principle for hash functions. In *Advances in Cryptology–CRYPTO89*, pages 416--427, 1989.

[128] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT'00*, pages 531--545, 2000.

[129] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362--399, December 2000.

[130] Bruno Blanchet. Composition theorems for CryptoVerif and application to TLS 1.3. In *31st IEEE Computer Security Foundations Symposium (CSF'18)*, Oxford, UK, July 2018. IEEE Computer Society.

[131] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 78--94, 2012.

[132] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII (FOSAD)*, volume 8604 of *Lecture Notes on Computer Science*, pages 146--166. Springer Verlag, 2014.

[133] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption (FSE)*, pages 163--184, 2016.

[134] Comodo CA Ltd. Comodo Certification Practice Statement. Technical report, Comodo CA Ltd., August 2015.

[135] DigiCert. DigiCert Certification Practices Statement. Technical report, September 2016. https://www.digicert.com/docs/cps/DigiCert_CP_v410-Sept-12-2016-signed.pdf.

[136] GeoTrust Inc. GeoTrust Certification Practice Statement. Technical report, GeoTrust, September 2016. https://www.geotrust.com/resources/cps/pdfs/GeoTrustCPS-Version1.1.19.pdf.

[137] GlobalSign CA. GlobalSign CA Certification Practice Statement. Technical report, GlobalSign CA, August 2016. https://www.globalsign.com/en/repository/GlobalSign_CA_CPS_v8.3.pdf.

[138] Internet Security Research Group. Certification Practice Statement. Technical report, Internet Security Research Group, October 2016. http://cps.letsencrypt.org.

[139] Symantec Corporation. Symantec Trust Network (STN) Certification Practice Statement. Technical report, Symantec Corporation, September 2016.

[140] StartCom CA Ltd. StartCom Certificate Policy and Practice Statements. Technical report, StartCom CA Ltd., September 2016. https://www.startssl.com/policy.pdf.

[141] LLC Network Solutions. Network Solutions Certification Practice Statement. Technical report, Network Solutions, LLC, September 2016. https://assets.web.com/legal/CertificationPracticeStatement.pdf.

[142] Alexa Internet Inc. Top 1,000,000 sites (updated daily), 2013. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[143] R. K usters and T. Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 157--171, 2009.

[144] Giuseppe Ateniese and Stefan Mangard. A new approach to dns security (dnssec). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86--95. ACM, 2001.

[145] Paul Hoffman and Jakob Schlyter. The dns-based authentication of named entities (dane) transport layer security (TLS) protocol: TLSA. Technical report, 2012.

[146] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Defensive JavaScript - building and verifying secure web components. In *Foundations of Security Analysis and Design (FOSAD VII)*, pages 88--123, 2013.

[147] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104--115, London, United Kingdom, January 2001. ACM Press.

[148] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. *SIGPLAN Not.*, 47(1):31--44, January 2012.

[149] Joyent Inc. and the Linux Foundation. Node.js, 2016. https://nodejs.org/en/.

[150] Nathan Wilcox, Zooko Wilcox-O'Hearn, Daira Hopwood, and Darius Bacon. Report of Security Audit of Cryptocat, 2014. https://leastauthority.com/blog/least_authority_performs_security_audit_for_cryptocat.html.

[151] Open Whisper Systems. Signal for the browser, 2015. https://github.com/WhisperSystems/Signal-Browser.

[152] GitHub. Electron framework, 2016. https://github.com/electron/electron.

[153] Avik Chaudhuri. Flow: Abstract interpretation of javascript for type checking and beyond. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2016.

[154] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. In *ACM Symposium on the Principles of Programming Languages (POPL)*, pages 87--100, 2014.

[155] Deian Stefan. Espectro project description, 2016.

[156] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP 2014 Object-Oriented Programming*, volume 8586 of *Lecture Notes on Computer Science*, pages 257--281. Springer Verlag, 2014.

[157] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. *SIGPLAN Not.*, 48(1):371--384, January 2013.

[158] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601--657, 2014.

[159] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AUTHSCAN: automatic extraction of web authentication protocols from implementations. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[160] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browser id sso system. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 673--688. IEEE, 2014.

[161] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770--772, 1981.

[162] WebAssembly Core Specification, 2018. `https://webassembly.github.io/spec/core/bikeshed/index.html`.

[163] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017.

[164] Peng Wang, Karthikeyan Bhargavan, Jean-Karim Zinzindohoué, Abhishek Anand, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Extracting from F* to C: a progress report.

Appendix A

ProScript Definitions

A.1 ProScript Operational Semantics

Notation: Sorts and Constants.

$\mathtt{x} \in \mathcal{X}^{\mathrm{U}}$	User variable names.
$\{@proto, @this, @scope, @body\} \in \mathcal{X}^{\mathrm{I}}$	Internal variables.
$x \in \mathcal{X} \triangleq \mathcal{X}^{\mathrm{I}} * \mathcal{X}^{\mathrm{U}}$	Variable names.
$l \in \mathcal{L}_{\mathtt{null}} \triangleq \mathcal{L} \cup \{\mathtt{null}\}$	Locations.
l_{op}	`Object.prototype`.
l_g	Global object.
$L \in \mathcal{C} \triangleq \mathcal{L}^*$	Scope chains.
$\mathtt{n} \in \mathcal{N}$	Numbers.
$\mathtt{s} \in \mathcal{S}$	Strings.
$\mathtt{undefined} \in \mathcal{U}$	Undefined.
$\mathtt{e} \in \mathcal{E}$	Expressions.
$l{\cdot}\mathtt{x} \in \mathcal{R}$	References.
$\lambda \mathtt{x}{\cdot}\mathtt{e} \in \mathcal{F}$	Function code.
$\mathtt{p} \in \mathcal{P} \triangleq \mathcal{X}^{\mathrm{U}} \cup \mathcal{S}$	Property names.
$\mathtt{v} \in \mathcal{V}^{\mathrm{U}} \triangleq \mathcal{N} \cup \mathcal{S} \cup \mathcal{U} \cup \{\mathtt{null}\}$	User values.
$v \in \mathcal{V} \triangleq \mathcal{V}^{\mathrm{U}} \cup \mathcal{L}_{\mathtt{null}} \cup \mathcal{C} \cup \mathcal{F}$	Semantic values.
$H \in \mathcal{L} \times \mathcal{X} \to \mathcal{V}$	Heaps.

Notation: Functions and Judgements.

$\sigma(H, L, x)$	Find defining scope.
$\pi(H, L, x)$	Prototype resolution.

$\gamma(H,r)$	Dereferencing values.
$\mathsf{True}(E)$	E is true.
$\mathsf{False}(E)$	E is false.
$\mathsf{obj}(l,l')$	Empty object at l.
$\mathsf{fun}(l,L,\mathtt{x},\mathtt{e},l')$	New function at l.
$\mathsf{act}(l,\mathtt{x},v,l',\mathtt{e},l'')$	Activation object template.
$\mathsf{defs}(\mathtt{x},l,\mathtt{e})$	Allocate local variable.

Scope Resolution: $\sigma(H,L,x)$.

$$\sigma(H,[],x) \triangleq \mathtt{null} \qquad \frac{\pi(H,l,x) \neq \mathtt{null}}{\sigma(H,l:L,x) \triangleq l}$$

$$\frac{\pi(H,l,x) = \mathtt{null}}{\sigma(H,l:L,x) \triangleq \sigma(H,L,x)}$$

Prototype Resolution: $\pi(H,L,x)$.

$$\pi(H,\mathtt{null},x) \triangleq \mathtt{null} \qquad \frac{(l,x) \in \mathsf{dom}(H)}{\pi(H,l,x) \triangleq l}$$

$$\frac{(l,x) \notin \mathsf{dom}(H) \quad H(l,@proto) = l'}{\pi(H,l,x) \triangleq \pi(H,l',x)}$$

Dereferencing Values: $\gamma(H,r)$.

$$\frac{r \neq l{\cdot}\mathtt{x}}{\gamma(H,r) \triangleq r} \qquad \frac{\pi(H,l,\mathtt{x}) = \mathtt{null} \quad l \neq \mathtt{null}}{\gamma(H,l{\cdot}\mathtt{x}) \triangleq \mathtt{undefined}}$$

$$\frac{\pi(H,l,\mathtt{x}) = l' \quad l \neq \mathtt{null}}{\gamma(H,l{\cdot}\mathtt{x}) \triangleq H(l',\mathtt{x})}$$

Auxiliary Predicates.

$\mathsf{True}(E) \triangleq E \notin \{0,"",\mathtt{null},\mathtt{undefined}\}$

$\mathsf{False}(E) \triangleq E \in \{0,"",\mathtt{null},\mathtt{undefined}\}$

$\mathsf{obj}(l,l') \triangleq (l,@proto) \mapsto l'$

$\mathsf{fun}(F,Closure,Var,Body,Proto) \triangleq$
$(F,@scope) \mapsto Closure * (F,@body) \mapsto \lambda Var.Body *$
$(F,\mathtt{prototype}) \mapsto Proto * (F,@proto)$

$\mathsf{act}(l,\mathtt{x},v,\mathtt{e},l'') \triangleq l \mapsto$
$\{\mathtt{x}:v,@this:l'',@proto:null\} * \mathsf{defs}(\mathtt{x},l,\mathtt{e})$

Local Variable Definition.

$$\frac{\mathtt{x} \neq \mathtt{y}}{\mathsf{defs}(\mathtt{x},l,\mathtt{var\ y}) \triangleq (l,\mathtt{y}) \mapsto \mathtt{undefined}}$$

$\mathsf{defs}(\mathtt{x}, l, \mathtt{e1 = e2}) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e1})$

$\mathsf{defs}(\mathtt{x}, l, \mathtt{e1;e2}) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e1}) \cup \mathsf{defs}(\mathtt{x}, l, \mathtt{e2})$

$\mathsf{defs}(\mathtt{x}, l, \mathtt{if(e1)\{e2\}\{e3\}}) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e2}) \cup \mathsf{defs}(\mathtt{x}, l, \mathtt{e3})$

$$\frac{\textit{otherwise}}{\mathsf{defs}(\mathtt{x}, l, \mathtt{e}) \triangleq \mathtt{emp}}$$

Syntax of Terms: $\mathtt{v}$, $\mathtt{e}$.

$$\mathtt{v} \quad ::= \left(\mathtt{n} \mid \mathtt{m} \mid \mathtt{undefined} \mid \mathtt{null} \right)$$

$$\mathtt{e} \quad ::= \left(\begin{array}{c} \mathtt{e;e} \mid \mathtt{x} \mid \mathtt{v} \mid \mathtt{if(e)\ \{e\}\{e\}} \mid \mathtt{var\ x} \\ \mid \mathtt{const\ x} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e.x} \mid \mathtt{e(e)} \mid \mathtt{x = e} \\ \mid \mathtt{function(x)e} \mid \mathtt{\{x_1 : e_1 \ldots x_n : e_n\}} \mid \mathtt{e[e]} \end{array} \right)$$

Operational Semantics: $H, L, \mathtt{e} \longrightarrow H', r$.

Definition
$$\frac{}{H, L, \mathtt{(var \mid const)\ x} \longrightarrow H, \mathtt{undefined}}$$

Value
$$\frac{}{H, L, \mathtt{v} \longrightarrow H, \mathtt{v}}$$

MemberAccess
$$\frac{H, L, \mathtt{e} \longrightarrow H', l' \quad \gamma(H', l' \cdot \mathtt{x}) = v}{H, L, \mathtt{e.x} \longrightarrow H', v}$$

Variable
$$\frac{\sigma(H, L, \mathtt{x}) = l' \quad \gamma(H, l' \cdot \mathtt{x}) = v}{H, L, \mathtt{x} \longrightarrow H, v}$$

Object
$$\frac{\begin{array}{c} H_0 = H * \mathsf{obj}(l', l_{op}) \\ \forall i \in 1..n. \begin{pmatrix} H_{i-1}, L, \mathtt{e}_i \longrightarrow H_i', v_i \\ H_i = H_i'[(l', \mathtt{x}_i) \rightarrow v_i] \end{pmatrix} \end{array}}{H, L, \mathtt{\{x_1 : e_1, \ldots, x_n : e_n\}} \longrightarrow H_n, l'}$$

Assignment
$$\frac{\begin{array}{c} \sigma(H, L, \mathtt{x}) \longrightarrow l' \\ H, L, \mathtt{e} \longrightarrow H', v \\ H'' = H'[(l', \mathtt{x}) \rightarrow v] \end{array}}{H, L, \mathtt{x = e} \longrightarrow H'', v}$$

Sequence
$$\frac{H, L, \mathtt{e}_1 \longrightarrow H', v \quad H', L, \mathtt{e}_2 \longrightarrow H'', v'}{H, L, \mathtt{e}_1\ ;\ \mathtt{e}_2 \longrightarrow H'', v'}$$

BinaryOperators
$$\frac{\begin{array}{c} H, L, \mathtt{e}_1 \longrightarrow H', \mathtt{v}_1 \\ H', L, \mathtt{e}_2 \longrightarrow H'', \mathtt{v}_2 \\ \mathtt{v}_1 \oplus \mathtt{v}_2 = v \end{array}}{H, L, \mathtt{e}_1 \oplus \mathtt{e}_2 \longrightarrow H'', v}$$

$$\text{ComputedAccess}\ \frac{\begin{array}{c} H, L, \texttt{e}_1 \longrightarrow H_1, l' \\ l' \neq \texttt{null} \\ H_1, L, \texttt{e}_2 \longrightarrow H', \texttt{p} \\ \gamma(H', l'\cdot\texttt{p}) = v \end{array}}{H, L, \texttt{e}_1\texttt{[e}_2\texttt{]} \longrightarrow H', v}$$

$$\text{Function}\ \frac{H' = H * \mathsf{obj}(l, l_{op}) * \mathsf{fun}(l', L, \texttt{x}, \texttt{e}, l)}{H, L, \texttt{function(x)\{e\}} \longrightarrow H', l'}$$

$$\text{FunctionCall}\ \frac{\begin{array}{c} H, L, \texttt{e}_1 \longrightarrow H_1, l_1 \\ H_1(l_1, @body) = \lambda\ \ \texttt{x.e}_3 \quad H_1(l_1, @scope) = L' \\ H_1, L, \texttt{e}_2 \longrightarrow H_2, v \\ H_3 = H_2 * \mathsf{act}(l, \texttt{x}, v, \texttt{e}_3, \bot) \quad H_3, l : L', \texttt{e}_3 \longrightarrow H', v' \end{array}}{H, L, \texttt{e}_1\ \texttt{(e}_2\texttt{)} \longrightarrow H', v'}$$

$$\text{IfTrue}\ \frac{H, L, \texttt{e}_1 \longrightarrow H', v \quad \mathsf{True}(v) \quad H', L, \texttt{e}_2 \longrightarrow H'', v'}{H, L, \texttt{if(e}_1\texttt{)\{e}_2\texttt{\}\{e}_3\texttt{\}} \longrightarrow H'', v'}$$

$$\text{IfFalse}\ \frac{H, L, \texttt{e}_1 \longrightarrow H', v \quad \mathsf{False}(v) \quad H', L, \texttt{e}_3 \longrightarrow H'', v'}{H, L, \texttt{if(e}_1\texttt{)\{e}_2\texttt{\}\{e}_3\texttt{\}} \longrightarrow H'', v'}$$

Appendix B

Transport Layer Security

$Q_{\text{PSKH}} = C_{\text{h}}[io22();\textbf{new}\ psk : key;\overline{io23}\langle\rangle;(C_{\text{PSKClient}}[Q_{\text{PSKOnlyClient}} \mid Q_{\text{PSKDHEClient}}] \mid C_{\text{PSKServer}}[Q_{\text{PSKOnlyServer}} \mid Q_{\text{PSKDHEServer}}])]$

$Q_{\text{PSKOnlyClient}} = C_{\text{PSKOnlyClient}}[$
insert c_table$(c_cr, c_log1, c_binder, c_log1')$;
1: **event** ClientEarlyAccept1$((c_cr, c_log1, c_binder, c_log1'), cets, i_C)$;
let $c\_cets : key = cets$ **in**
$\overline{io2[i_C]}\langle$ClientHelloOut$(c_cr, c_binder)\rangle$;
$C'_{\text{PSKOnlyClient}}[$
event ClientTerm1$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m), (log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems, resumption_secret))$;
let $c\_sats : key = sats$ **in**
2: **event** ClientTerm$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, c_sats, ems))$;
3: **event** ClientAccept$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m, log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, c_sats, ems, resumption_secret), i_C)$;
let $c\_cats : key = cats$ **in**
let $c\_ems : key = ems$ **in**
let $c\_resumption\_secret : key = resumption\_secret$ **in**
$\overline{io6[i_C]}\langle$ClientFinishedOut$(cfin)\rangle]]$

$Q_{\text{PSKOnlyServer}} = C_{\text{PSKOnlyServer}}[$
$io11'[i_S]()$;
get c_table$(= s_cr, = s_log1, = s_binder, = s_log1')$ **in**
let $s\_cets2 : key =$ get_client_ets$(cets_eems)$ **in**
4: **event** ServerEarlyTerm1$((s_cr, s_log1, s_binder, s_log1'), s_cets2)$;
$\overline{io12'[i_S]}\langle\rangle$
else
let $s\_cets3 : key =$ get_client_ets$(cets_eems)$ **in**
5: **event** ServerEarlyTerm2$((s_cr, s_log1, s_binder, s_log1'), s_cets3)$;
find $us \leq N_S$ **suchthat defined**$(s\_cr[us], s\_log1[us], s\_binder[us], s\_log1'[us], s\_cets1[us]) \wedge s\_cr[us] = s\_cr \wedge s\_log1[us] = s\_log1 \wedge s\_binder[us] = s\_binder \wedge s\_log1'[us] = s\_log1'$
then
$\overline{io13'[i_S]}\langle\rangle$
else
let $s\_cets1 : key = s\_cets3$ **in**
$\overline{io14'[i_S]}\langle\rangle$,
6: **event** ServerAccept$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems), i_S)$;
let $s\_sats : key = sats$ **in**
$\overline{io18[i_S]}\langle$ServerFinishedOut$(m)\rangle$;
$C'_{\text{PSKOnlyServer}}[$
event ServerTerm1$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m), (log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems, resumption_secret))$;
let $s\_cats : key = cats$ **in**
let $s\_ems : key = ems$ **in**
let $s\_resumption\_secret : key = resumption\_secret$ **in**
7: **event** ServerTerm$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m, log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, s_cats, sats, s_ems, s_resumption_secret))$;
$\overline{io30[i_S]}\langle\rangle]]$

Figure B.1: Model of the handshakes with pre-shared key

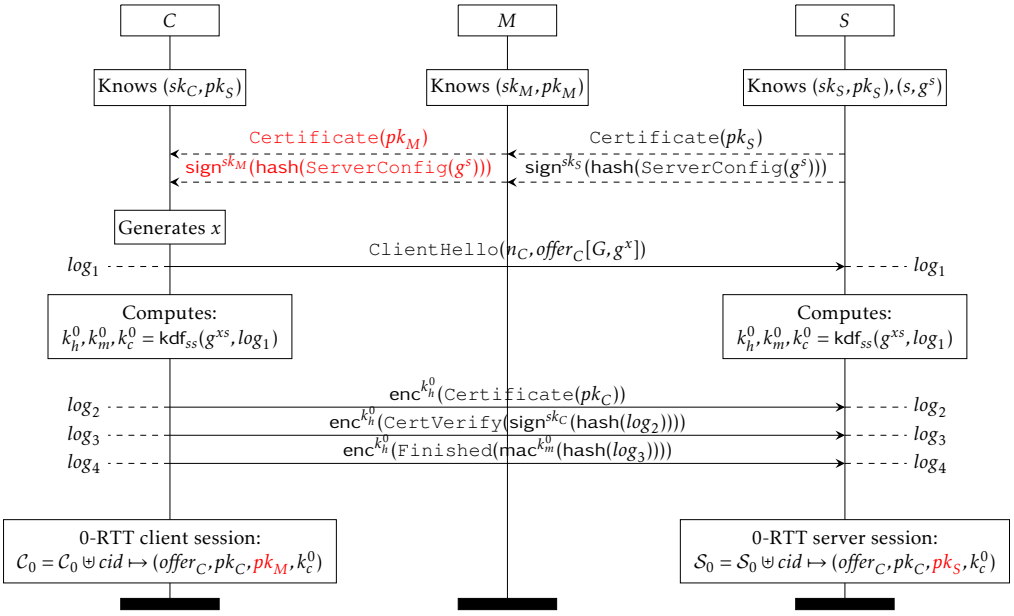

Figure B.2: Unknown Key Share (UKS) Attack in QUIC, OPTLS and on TLS 1.3 Draft-12 if the server's certificate is not added to log_1. The figure shows how a man-in-the-middle can exploit the UKS to mount a credential forwarding attack on 0-RTT client authentication. This attack led to the inclusion of `Certificate` and `ServerConfig` in the 0-RTT handshake hash in Draft-7.

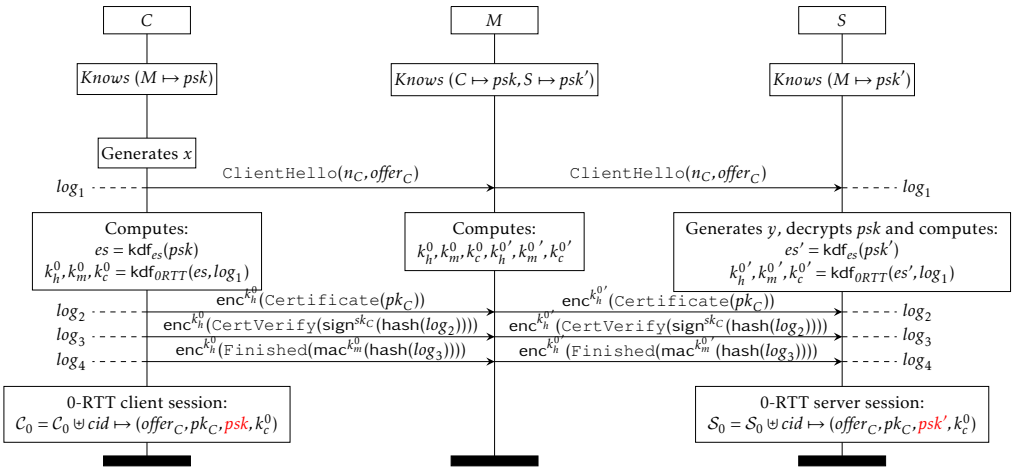

Figure B.3: Credential Forwarding Attack on PSK-based 0-RTT Client Authentication in Draft-12. This attack motivates the use of PSK binders—using *psk* to MAC the `ClientHello`)—in Draft-18.

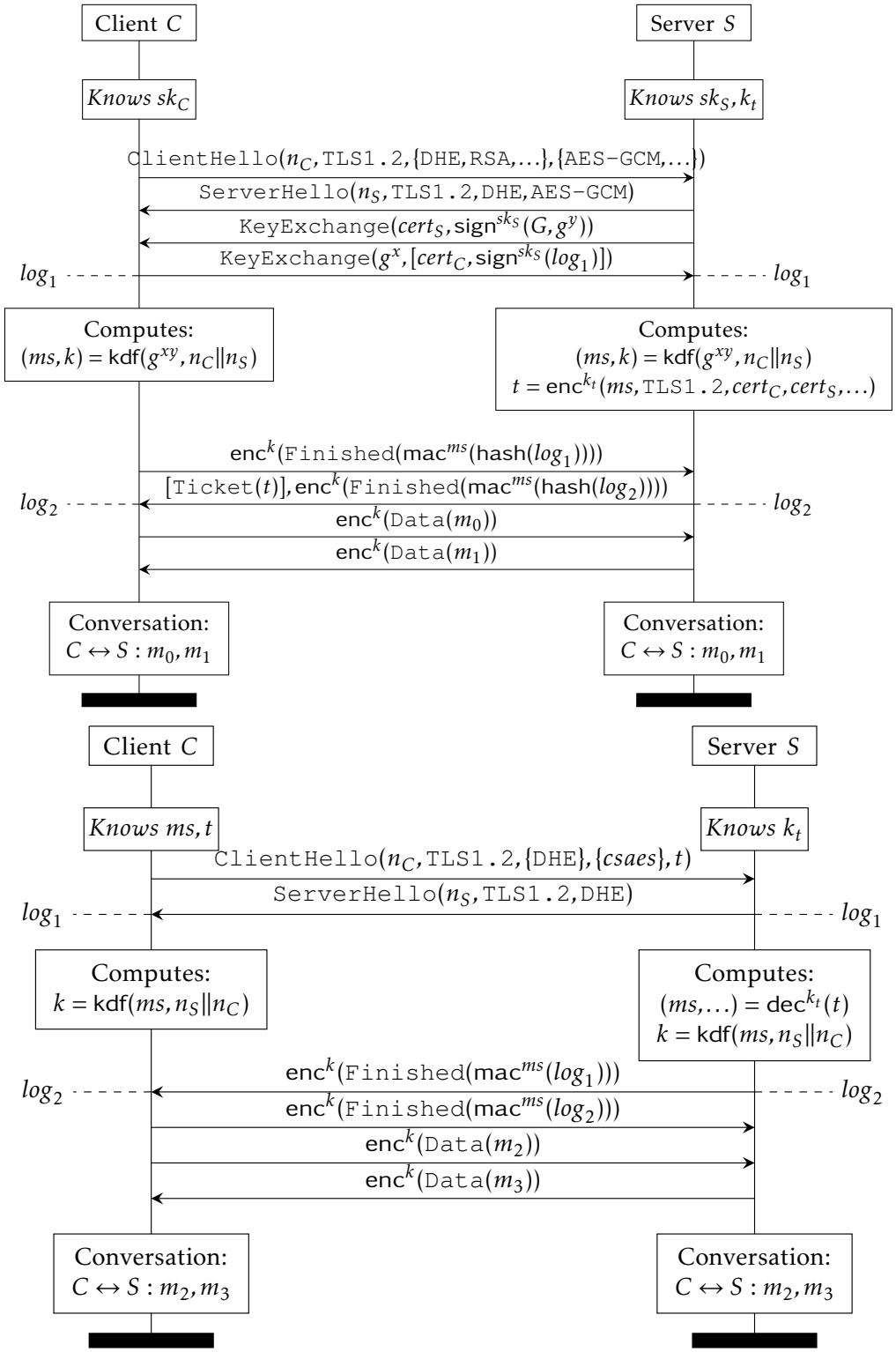

Figure B.4: TLS 1.2 full (EC)DHE handshake and a subsequent abbreviated session resumption handshake on a new connection. Parts within square brackets [...] are optional, such as client authentication and session tickets.

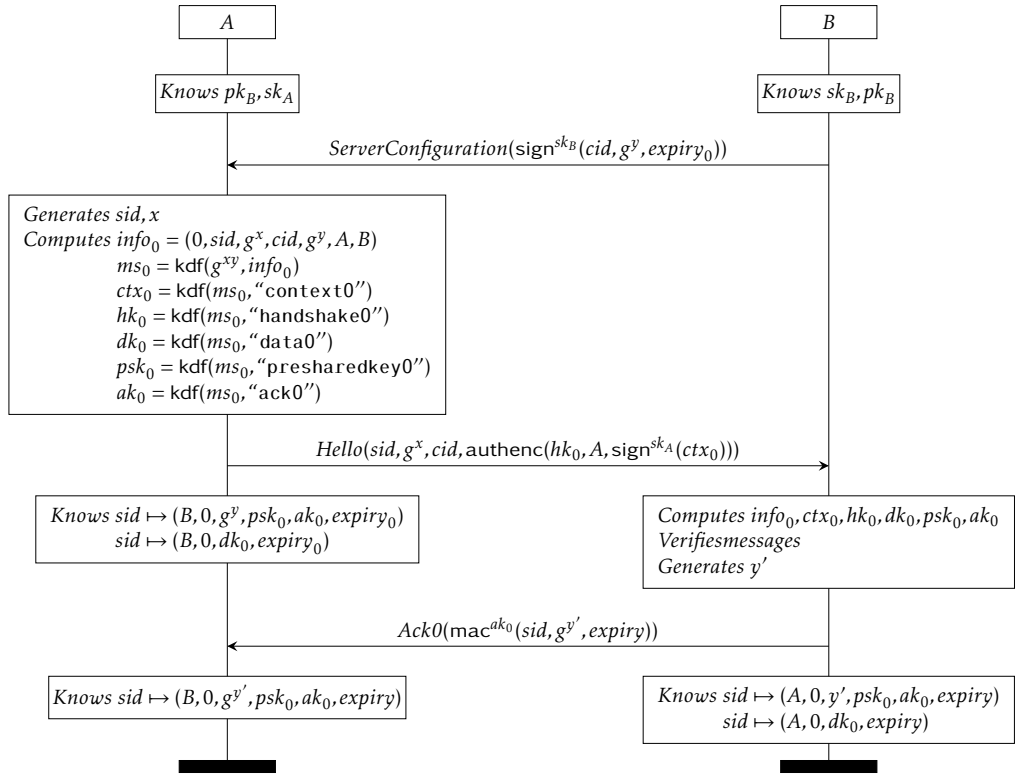

Figure B.5: The New Messaging Protocol: Start session with DH 0-RTT

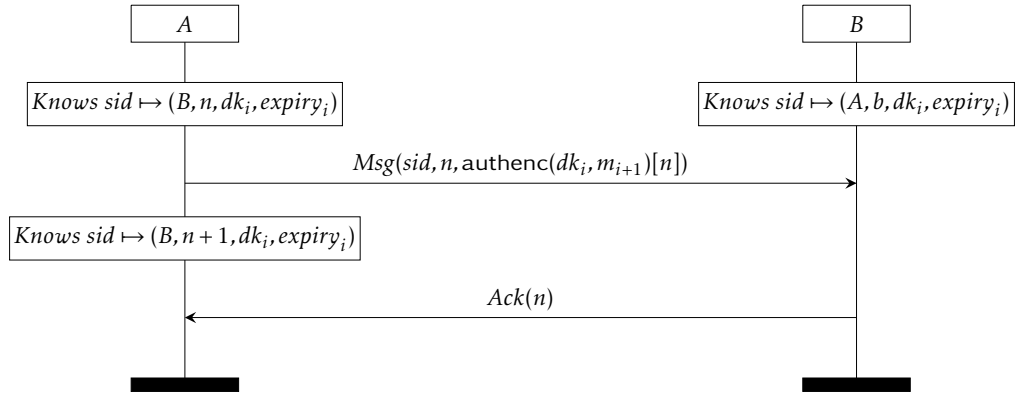

Figure B.6: The New Messaging Protocol: Sending Data

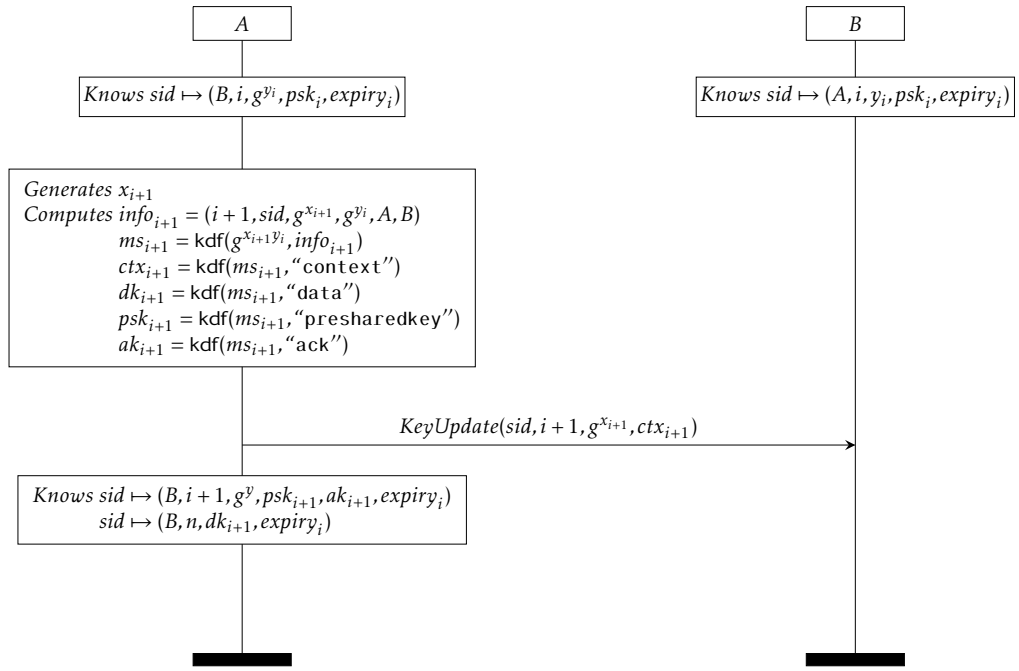

Figure B.7: The New Messaging Protocol: Key Refresh

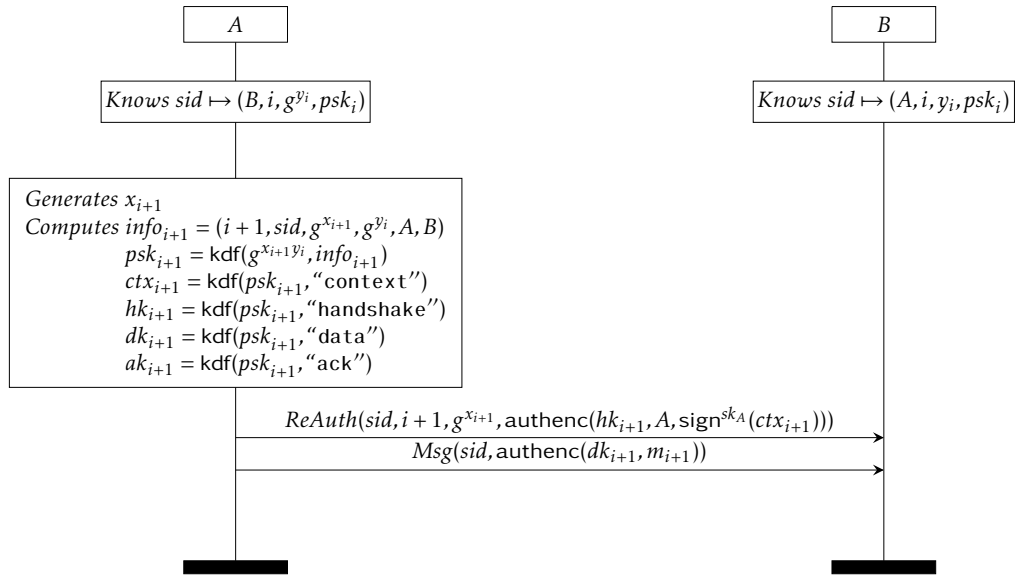

Figure B.8: The New Messaging Protocol: Re-authentication