PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Inria Paris

A Verification Framework for Secure Group Messaging

Soutenue par

Théophile Wallez

Le 24/06/2025

Ecole doctorale n° 386

Ecole Doctorale de Sciences Mathématiques de Paris-Centre

Spécialité

Informatique

Composition du jury :

Cas, CREMERS	
Professor, CISPA	*Rapporteur*
Stéphanie, DELAUNE	
Directrice de recherche, CNRS	*Rapportrice*
Joel, ALWEN	
Chief Cryptographer, AWS Wickr	*Examinateur*
Céline, CHEVALIER	
Maîtresse de conférences, ENS - PSL	*Examinatrice*
Véronique, CORTIER	
Directrice de recherche, CNRS	*Examinatrice*
Bruno, BLANCHET	
Directeur de recherche, Inria Paris	*Directeur de thèse*
Karthikeyan, BHARGAVAN	
Chief Research Scientist, Cryspen	*Co-Directeur de thèse*
Jonathan, PROTZENKO	
Principal Researcher, Microsoft Azure Research	*Co-Directeur de thèse*

Inria

Abstract

Messaging applications are nowadays pervasively used to communicate with each other, in particular using *group conversations* to connect people within a social circle. This is a potential threat for privacy, for example if the messaging application servers were to have access to the conversation content. To address this issue, modern messaging applications provide *end-to-end encryption*, meaning that messages are encrypted by the sender device and decrypted by the receiver device, so that their content stays hidden from the messaging application servers. Such end-to-end encryption is a feature of cryptographic protocols, whose design is notoriously error-prone. This begs the following question: *are secure messaging applications actually secure?* In this thesis, we develop a novel methodology to analyze the secure group messaging protocol Messaging Layer Security (MLS) by using formal methods on bit-precise specifications in the symbolic model, and ultimately helped to fix design flaws in MLS before its standardization.

The first axis of this thesis is developing tools to analyze cryptographic protocols, on bit-precise specifications. To handle the complexities exhibited by MLS, such as dynamic group size or recursive data structures, we present several key improvements to DY*, a symbolic analysis framework written in the F* proof assistant. To write bit-precise specifications of cryptographic protocols and handle their precise message formats, we introduce Comparse, a tool to specify and prove properties on message formats in F*. In the process, we study the broad class of *message formatting attacks*, and derive criteria cryptographic protocols should obey to avoid such attacks.

The second axis of this thesis is applying the tools we developed on MLS. We present a novel modularization of MLS to decompose it into three sub-protocols, thereby allowing us to analyze each sub-protocol independently. We then analyze and produce a machine-checked proof for two out of three sub-protocols, TreeSync and TreeKEM. Our specification for MLS is bit-precise, executable and interoperable with other MLS implementations. During our analysis, we found several design flaws and proposed fixes to the Working Group in charge of its design at the Internet Engineering Task Force (IETF), which were integrated in the MLS standard.

Résumé

Les applications de messagerie sont de nos jours utilisées de façon généralisée pour communiquer, en particulier en utilisant les *conversations de groupe* pour relier les personnes d'un cercle social. C'est une menace potentielle pour la vie privée, par exemple si les serveurs de l'application de messagerie ont accès au contenu des conversations. Pour résoudre ce problème, les applications de messagerie modernes fournissent du *chiffrement de bout en bout*, ce qui veut dire que les messages sont chiffrés par l'appareil de l'expéditeur et déchiffrés par l'appareil du destinataire, de manière à ce que les serveurs de messagerie ne puissent pas connaître le contenu des messages. Un tel chiffrement de bout en bout est de façon plus générale un protocole cryptographique, dont la conception est notoirement sujette aux erreurs. Cela soulève la question suivante: *les applications de messagerie sécurisée sont-elles réellement sécurisées ?* Dans cette thèse, nous développons une nouvelle méthodologie permettant d'analyser le protocole de messagerie de groupe Messaging Layer Security (MLS) en utilisant les méthodes formelles sur des spécifications précises à l'octet près dans le modèle symbolique et, ultimement, nous aidons à corriger des problèmes de conception dans MLS avant sa standardisation.

Le premier axe de cette thèse est le développement d'outils d'analyse de protocoles cryptographiques, sur des spécifications précises à l'octet près. Pour traiter les complexités liées à MLS, telles que la taille dynamique des groupes ou l'utilisation de structures de données récursives, nous présentons de multiples améliorations essentielles à DY*, un cadriciel d'analyse symbolique écrit dans l'assistant de preuve F*. Afin d'écrire des spécifications de protocoles cryptographiques précises à l'octet près et capturer précisément leurs formats de message, nous introduisons Comparse, un outil pour spécifier et prouver des propriétés sur des formats de message en F*. Dans le processus, nous étudions la large classe des *attaques exploitant les formats de message* et dérivons divers critères que les protocoles cryptographiques doivent suivre afin de résister à de telles attaques.

Le second axe de cette thèse est l'application des outils que nous avons développés sur MLS. Nous présentons une nouvelle modularisation de MLS pour le décomposer en trois sous-protocoles, ce qui nous permet de les analyser de manière indépendante. Ensuite, nous analysons et produisons une preuve vérifiée par ordinateur pour deux sur trois sous-protocoles, que nous avons nommés TreeSync et TreeKEM. Notre spécification de MLS est précise à l'octet près, exécutable et intéropèrable avec les autres implémentations de MLS. Durant notre analyse, nous avons trouvé plusieurs défauts de conception et nous avons proposé des corrections au groupe de travail en charge de la conception de MLS à l'Internet Engineering Task Force (IETF), qui ont été intégrées au standard de MLS.

Acknowledgements

Traditionally, this section of thesis manuscript contains multiple variants of "this thesis wouldn't have been possible without X". Unfortunately, I really enjoy noticing the "butterfly effect" on my own life, so I will first go through the random facts of life that led me to do this PhD, before jumping into the proper acknowledgements.

I couldn't have done this PhD without my parents, first for the obvious reason that without them, I wouldn't have been born in the first place, second because they were prominent in providing me cultural heritage, especially about science and mathematics (e.g. I vividly remember being explained as a kid how a mentalist managed to guess a number in the head of people in the crowd, ultimately explaining why $\frac{2x+8}{2} - x$ simplifies to 4, although we would learn algebra with unknown variables only several years later in school). My brother definitely played a role in leading me to program during my childhood, first by showcasing how cool it was to write computer programs through a series of world-class games, such as "Bob the fish" where you would shoot bubbles at jellyfishes (only writing these lines I realize, fishes in real life don't make any bubbles??) or the "Jeu du chasseur" where you would hunt rabbits (while being extra careful around the ninja rabbits that throw shurikens, of course); and maybe more importantly when before going to a dentist appointment he sat me in front of his computer and showed me a tutorial on writing HTML code (on "Le site du Zéro", RIP in peace). I really enjoyed it, it was 20 years ago when I was 9 (time flies) and haven't stopped programming since then. I don't know how my current life would look without this single event, would I eventually have started programming at some point anyway, or not? Fast-forward to the "classes préparatoires", I must warmly thank my mathematics teachers Romain Bondil and Michel Alessandri, who taught me how to do rigorous mathematical proofs for the first time in my life, as well as my physics teacher Véronique Chireux, who, although I kind of sucked at physics, was an excellent mentor during these years. Altogether, they in a way gave birth to the scientist in me. Fast-forward again during my Master's degree, I would encounter a blog post by Denis Merigoux on using SMT solvers on tax code to automatically find corner cases in the law. That looked fun, so when I would look for a research internship, I sent him an email, to which he answered something along the lines of "too late, I already have an intern, but since I see you have background on computer security and formal verification, ask my advisor Karthikeyan Bhargavan for an internship!" I didn't know this guy, neither did he know me, but still, I did my research internship with him, and eventually continued into a PhD. One of the reasons I continued into a PhD is because the lab environment was especially friendly, and for that I must warmly thank Denis, who in addition to his email-forwarding abilities made outstanding efforts to make me feel included in the team during Covid times despite the work-at-home policy, although we barely knew each other. Finally, last but not least, the fact that I did this PhD is certainly helped by the fact that I am a white cis male born in a family full of engineers, teachers and doctors, in a country that is wealthy thanks to its colonization past. Of course, these five attributes about myself are not enough for me to do a PhD, but statistics show that each of these five attributes improve the probabilities I start and succeed through a PhD, because of the privileged position they incur to me in our society. I find it is important to acknowledge this.

The paragraph above summarizes the random events in life that led me to start this PhD in this area of research, now, onto the people who played a role during the PhD itself. I am deeply grateful to my PhD advisors, Karthikeyan Bhargavan and Jonathan Protzenko, I really couldn't have dreamed of better supervision for my PhD as they both consistently provided excellent advices, both on the short term and the long term, always finding the perfect balance between allowing myself to get distracted and putting me back on tracks when it was needed. I don't know if a single human being can tick all the boxes to be a perfect advisor, but I know that together, this duo did tick all the boxes. Special mentions to Karthik, who developed striking creativity in finding places to meet, be it online, in a café, in an art gallery, in the Paris Métro (in three different lines), in his house (in three different countries), or in a {French, Indian} restaurant in {France, India} (all four possibilities), as well as a PMU (a widespread chain of working-class bars in France, also better known to feature gambling on horse races), and last but not least, in his office while other colleagues left for lunch, apparently a prominent time of the day to have deep discussions about my long-term future (thankfully, gathering his wisdom was certainly worth postponing the lunch). Special mentions to Jonathan, who I met online almost every week of my PhD (despite the 9 hours of timezone that separated us), of course to discuss science, but also, among other things, to get opinions on hiking gear, gather precious knowledge about

Chartreuse or even learn new references to “La Classe américaine” (now I think about it each time I eat chocolate mousse). A truly enjoyable moment, every week, where you additionally gather insights to progress in your thesis, what else can you ask for? But I should probably stop my praise here, as I also learned during these weekly meetings, “la flatterie ne me mènera nulle part”. So I will instead praise Laptop, Jonathan’s big hairy orange cat, who would sometimes intrude during our meetings, and whose magnificence would shatter ability to speak unless I made the conscious effort to would look somewhere else. After noticing this phenomenon, Jonathan began to send me photos of Laptop upon showing progress in my thesis, this was extremely effective, so I guess we can say that Laptop was my muse during this PhD. Note that Karthik and Jonathan were not my only PhD advisors: the careful reader would have noticed that the front page also mentions Bruno Blanchet, who had to take over the administrative works when Karthik left for a sabbatical. I have to thank Bruno for taking over this unpleasant burden, but also for carefully answering my scientific questions when I had some for him, be it on cryptography, on the subtleties of the symbolic model or on the inner workings of ProVerif. In the lab, Bruno was not the only victim of my questions, and I must also thank Adrien and Charlie for taking the time to answer numerous the questions I had about cryptography in the computational model, although it is not strictly in the scope of my PhD, I feel having knowledge in this area definitely helped me to grow in the field of cryptography. The person I definitely spent the most time with is Son, with whom I not only shared an office during most of my PhD, but also had an excellent trip to India when Karthik kindly invited us to visit him there, always patiently bearing with my silliness, such as getting extra dirty from climbing a tree hours before going to one of the fanciest restaurants of Delhi (although we certainly stood out with our hobo-looking clothes and our squeaky shoes, it is still today among the best restaurant experiences I had). I really wish we had time to work on the symbiotic project of using your tools to verify an MLS implementation in Rust against my verified specification, but as always, research projects take more time we initially expect. Finally, Son would regularly remind me that my own struggles with my PhD were not that bad, by shouting in the office (yes, of course with a very soft voice) something along the lines of “oh, mais c’est insupportable !” When Son left for his post-doc, he was replaced with Vincent, who happened to make different kind of noises by quietly playing the soft music of FIP on a speaker (best radio ever). My PhD wouldn’t have been the same without Aymeric (also known as “MC Fromherz” since his impressive delivery on rap songs in Japanese karaokes), a true genius, although sometimes misunderstood (seriously, in what universe is it a good idea to dip cookies into beer?) Even more impressive is his ability to find bad puns in every possible scenario (“pain pont”), so that it feels at least 10% of his brain is devoted to this task; thankfully this sometimes leads to actual good puns by pure accident (“c’est fâcheux”). I am lucky to have met Lucas during my PhD, first because he is an extremely nice person, second because his hacking skills are beyond my understanding, and third because he is easily nerd-snippable; these three properties were quite useful during my PhD, should a feature be missing in F*, I would only need to complain about it during a break, and a pull request would appear the next day, authored by the mighty W95Psp. Truly amazing. Conversely, I myself was nerd-sniped by Louis, who kindly brought puzzles to the lab, or as he said, “nerd traps”, and Louis also sniped me with arrows when playing Towerfall late in the afternoon (although the frequent games allowed me to progress and fight back!) Finally, I have to thank every other person that worked at Prosecco during the time of my PhD, I will not risk myself into trying to write an exhaustive list, but you all contributed to the great atmosphere that motivated me to go work on-site every day.

There are also many people outside the lab I wish to thank. First, I am deeply grateful to Cas Cremers and Stéphanie Delaune, who agreed to carefully read this manuscript, as well as Joel Alwen, Céline Chevalier and Véronique Cortier who agreed to be on the jury and patiently listen to a 45-minutes long monologue about my thesis. I need to acknowledge everyone in the MLS Working Group at the IETF, as my PhD required to regularly interact with them, and although it was certainly scary to the young PhD student I was at the beginning, they were eventually extremely nice people who took the time to thoroughly discuss my pull requests and accept my ideas when they were agreed to be good, while still challenging them when it was needed, allowing me to refine my arguments and mature my thoughts. Every year of my PhD was punctuated with traveling to the Real World Crypto Symposium, it is in my heart one of the best events of the year and I warmly thank all the people in this community who make this event so special.

As the proverb goes, “all work and no play makes Jack a dull boy”, and I need to thank all the people with whom I did non-PhD-related things, allowing myself to keep my sanity. First I need to thank my family, who in addition to make me who I am today (as written earlier) are still constantly there in my life; it really eases the mind to know that I can rely on your support whatever happens in my life. Big up to my brother who, 20 years later, is still a source of inspiration, plus, life wouldn’t be the same without the weekly chocolatine hunting followed by the well-stirred coffee brewed in a space rocket. Thank you to all the friends of ENS (&

co) who kept close contact with me during these years, such as Luc (aka “l’artiste des plans foireux”), Pablo (the gardener of formal logic), Paul-Nicolas (the only guy on earth who is actually passionate about CI??) and Thibaut (I’m glad you are still alive despite your apparent lack of self-preservation instincts); as well as friends who kept regular contact with me (less frequently, but still highly enjoyable), such as Béranger, Camille, Erkan, Étienne, Louise, Lucas, Mathieu, Michele, Samuel, Simon and Théophile. Another saying is “a healthy mind in a healthy body”, so I need to thank all the friends who I practice parkour with, such as Christophe, François, Guillaume, Marie, Mathilde, Mathilde (another one!), Nicolas, Pierre-Alain, Rubing and Thibaud. I would not have been able to practice parkour without my physiotherapist Larissa, who patiently repaired the various injuries I had (editor’s notes: mostly stemming from a bicycle accident, not from practicing parkour). Although my PhD was already filled with trees (sync, kem and dem), I am glad to have met the Parisian free solo tree climbing community, in particular Garance and Hubert, the driving forces of the group, this has been a huge amount of fun, and I cannot forget the amazing views we had on Paris from the top of oak trees. My daily life was punctuated by the practice of piano, and for that I warmly thank my piano teacher Marion who allowed me to progress faster than I could fathom. Another source of fun in my life has been to try-hard baking french pastries, and I could not have learned this skill without Simon and Théophile (yes, another one). Speaking of baking, I am grateful for the Matfer Bourgeat brand of kitchen ustensils, they are truly the greatest and an immense joy to use every day (I really couldn’t have guessed that a bowl scrapper could be this great). Finally, I need to thank the public radio FIP that brings me great (although sometimes surprising) music every day of my life, and especially the program “Transe Fip Express” as well as the program “[DEEP] Search” by Laurent Garnier. I am really glad that my taxes pay for this kind of service.

Contents

List of Figures

List of Tables

Introduction 1

This thesis lies at the intersection of two research areas: cryptography, and computer-checked mathematical proofs. In this section, we will first introduce the cryptographic and mathematics concepts relevant to this thesis (§1.1 and §1.2), then, show how these two research areas intersect (§1.3), finally, describe the goal of this thesis (§1.4).

1.1 Cryptography and secure messaging

It would be a cliché to start this dissertation with "since the dawn of humanity, mankind had the desire to ensure confidentiality of its communications". However, as we shall see, the concept of *secure messaging* traces all the way back to the Roman Empire and Ancient Greece.

1.1.1 Pre-computer cryptography

> Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.
>
> **Bruce Schneier**, *"Memo to the Amateur Cipher Designer", 1998*

Although "cryptography" nowadays rhymes with "computer", humans didn't wait for the advent of computers to use cryptographic techniques. A recurrent scenario is the following: a military general wants to send a message to another military general, but fears that the message might be intercepted by the enemy. To avoid this risk, the military generals agree beforehand on a way to *transform* messages so that the enemy cannot extract any useful information from a transformed message, but that other military generals can recover the message before its transformation. This process of "transforming" messages is known as *encryption*; in modern terms, the military general *encrypts* the message before sending it, afterward the other military general will *decrypt* the message to recover its original content.

Scytale. The Ancient Greeks and Spartans are believed to have used a "scytale" to encrypt messages, by winding a strip of paper around a cylinder, writing their message, and unwind the strip of paper, as depicted in Figure 1.1. As a result, the letters of the messages are shuffled around, so that "attack at dawn" becomes "acdtkatawatn".

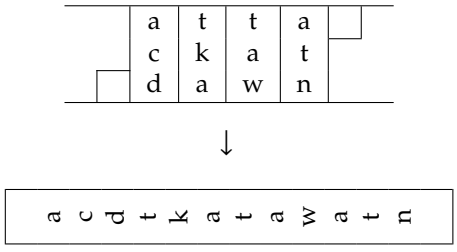


Figure 1.1: A scytale, believed to be used by the anciant Greeks and Spartans. The message "attack at dawn" becomes "acdtkatawatn".

Caesar cipher. Julius Caesar is believed to have encrypted messages by shifting each letter by three positions in the alphabet, that is, "d" becomes "a", "e" becomes "b", etc (as depicted in Figure 1.2), so that the text "attack at dawn" becomes "xqqxzhxqaxtk" (see Figure 1.3).

a	b	c	d	e	f	g	h	i	...
↓	↓	↓	↓	↓	↓	↓	↓	↓	
x	y	z	a	b	c	d	e	f	...

Figure 1.2: Caesar cipher.

a	t	t	a	c	k	a	t	d	a	w	n
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
x	q	q	x	z	h	x	q	a	x	t	k

Figure 1.3: Using Caesar cipher, the message "attack at dawn" becomes "xqqxzhxqaxtk".

Kerckhoffs's principle. In 1883, Auguste Kerckhoffs publishes an article entitled "Military Cryptography" [1], which states six design principles that military cryptographic systems should obey. One of the principles states that "it should not require secrecy, and it should not be a problem if it falls into enemy hands".[1] Notably, the scytale and Caesar cipher don't obey this design principle, whose security depends on their mechanism remaining unknown to the enemy.

[1]: Kerckhoffs (1883), *La cryptographie militaire*

1: originally, "Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi"

To obey Kerckhoffs's design principle, encryption mechanisms are parametrized by a *cryptographic key*, so that their security depend only on the secrecy of this cryptographic key, even if the encryption mechanism itself is known to the enemy. This design principle is nowadays a fundamental principle of modern cryptography, as opposed to "security by obscurity".

Cryptanalysis and security of cryptographic systems. Because cryptography is used to encrypt sensitive messages, the use of cryptographic systems systematically comes with attempts to break them, that is, to decrypt encrypted messages without knowing the corresponding cryptographic key: this is the process of *cryptanalysis*. By Kerckhoffs's design principle, we must assume that the enemy knows the cryptographic system, hence may perform cryptanalysis on it. Therefore, to ensure the security of cryptographic systems, their designers must also perform cryptanalysis on them: a cryptographic system is then assumed to be secure when the best cryptanalysis does not succeed in breaking the cryptographic system in practice.

Enigma. More recently, during World War II, Nazi Germany used the Enigma machine to encrypt their military communications. Therefore, their enemies (the Allies) worked on its cryptanalysis, as decrypting their messages would offer a military advantage. To hinder these cryptanalysis attempts, the German military worked under the assumption that it would take more than a day for the Allies to recover the cryptographic key being used, hence the German military changed their cryptographic key every day. To do so, they sent in advance sheets of paper containing one key to use each day to the Enigma operators within their military. If their enemies were to obtain these sheets of paper, they would be able to decrypt messages encrypted using these keys: this includes messages that will be sent in the future, and also messages that were sent in the past. To protect messages from the past, Enigma operators were instructed to cut off and destroy keys used for the previous days [2]. Nowadays, this technique is modernly known as "key erasure" and aims to provide "forward secrecy". To protect messages that will be sent in the future is less convenient: the only solution would be to send new sheets of paper with fresh key material.

[2]: (1944), *The US 6812 Division Bombe Report Eastcote 1944*

1.1.2 Post-computer cryptography

> I told her that we were headed into a world where people would have important, intimate, long-term relationships with people they had never met face to face. I was worried about privacy in that world, and that's why I was working on cryptography.
>
> **Whitfield Diffie**, *to his wife*

To ensure the practicality of cryptographic techniques, it is crucial that we can use them efficiently: for example, Enigma was designed as a typewriter with 26 lights, so that when a key is pressed, one light illuminates, corresponding to the letter of the encrypted or decrypted message. This notion of "efficiently" changes over time as technology evolves, notably with the invention of the computer. As a result, the advent of computers allowed for the proliferation of cryptographic techniques: although before the focus had been on *symmetric encryption*, that is, rely on a secret key shared beforehand to both encrypt and decrypt messages, we now have access to a broader set of cryptographic techniques.

Key distribution. A major inconvenience in the use of symmetric encryption is that it requires the distribution of the secret key to all the intended recipients; but how do we securely distribute the key, without revealing it to eavesdroppers, given that we cannot use symmetric encryption to protect the key distribution as this would create a chicken-and-egg problem? In a seminal paper [3], Whitfield Diffie and Martin Hellman proposed in 1976 a solution to the key distribution problem: what we nowadays call a "Diffie-Hellman key exchange" allows two parties to exchange a key so that a passive eavesdropper listening to their communications cannot recover the key.

[3]: Diffie et al. (1976), *New directions in cryptography*

Short digression: we can describe their solution to key distribution right now, because it is so elegant and only requires to understand undergraduate-level mathematics. First, the two parties agree on a cyclic group G with generator g (similarly to how they would agree on an encryption algorithm to use). Then, party A generates x (to be kept secret) and sends g^x to party B, similarly party B sends g^y to party A. At the end, both parties can compute the shared key $g^{xy} = \left(g^x\right)^y = \left(g^y\right)^x$, because we can efficiently compute g^x from g and x. However, a passive eavesdropper cannot compute g^{xy} from g^x and g^y, because computing x from g and g^x is more difficult: this is known as computing a "discrete logarithm", and the security of the Diffie-Hellman key exchange depends (in part) on the difficulty to compute a discrete logarithm. Originally, the group G was the cyclic group $(\mathbb{Z}/p\mathbb{Z}, \times)$, nowadays, we rely on elliptic curves to define G.

Although this solves the problem of key distribution in the presence of *passive* eavesdroppers, that is, eavesdroppers who listen to the conversation but don't interfere with it, the Diffie-Hellman key exchange is not secure against *active* adversaries that may tamper the messages. Indeed, when party A initiates a key exchange with party B, an active adversary could block all messages and themselves do a key exchange with party B: if afterward party B encrypts secrets using this key, the adversary will be able to decrypt them. The fundamental problem here is that party B cannot distinguish between doing a key exchange with party A or with an active adversary. One way to solve this problem is to use a cryptographic equivalent to signatures, so that party B can distinguish

between messages coming from party A (the signature is authentic) or from an adversary (the signature is wrong).

Signatures. In a seminal paper [4], Ron Rivest, Adi Shamir and Leonard Adleman proposed in 1978 a cryptographic equivalent to signatures. The system works with two keys: a *signature* key, and a *verification key*. The signature key is kept private (hence is also called a "private key"), and the verification key is made available publicly (hence is also called a "public key"). We can then use the signature key to produce a signature for this message, afterward another party may use the verification key to check the signature authenticity. Furthermore, if the message is modified after being signed, the signature verification will fail: a signature is bound to a message, unlike traditional handwritten signatures.

[4]: Rivest et al. (1978), *A method for obtaining digital signatures and public-key cryptosystems*

In the same paper, they also introduce a technique for *asymmetric encryption*, where one key (made public) is used to encrypt messages, and another key (kept private) is used to decrypt them, unlike *symmetric encryption* which relies on the same key both for encryption and decryption.

Combining cryptographic components. The creation of new cryptographic components allows for secure messaging techniques with better practicality and better security. As a recap, we started with *symmetric encryption*, which allows sending messages so that their content stay confidential even if an adversary listens to our communication channel, but comes with the drawback that we must beforehand distribute a shared, secret key; we then plugged-in the Diffie-Hellman *key exchange* procedure to do the key distribution on a communication channel that is passively listened to, but noticed that it is insecure when an adversary tampers with the communication channel; we finally used *cryptographic signatures* to ensure the authenticity of messages sent on the communication channel and detect such a tampering.

Cryptanalysis of cryptographic protocols. We assembled together many cryptographic components (e.g. symmetric encryption, key exchange, signatures) to create larger systems, which are named *cryptographic protocols*. These cryptographic protocols provide a large surface area to find flaws: there may of course be flaws in each of the cryptographic components we use (which may in turn be used to attack the cryptographic protocol), but importantly, there may also be flaws in the cryptographic protocol *itself*. Indeed, it may be that cryptanalysis found no flaws in any of the cryptographic components we use, but that cryptanalysis of the entire cryptographic protocol find flaws in the way cryptographic components are assembled together, especially in the presence of *active* adversaries that may tamper messages sent over the communication channel. Thankfully we now know methods to ensure that we don't introduce vulnerabilities in the way we assemble cryptographic components to create a cryptographic protocol; we shall discuss them in §1.1.4.

1.1.3 Secure messaging

> Privacy in an open society also requires cryptography. If I say something, I want it heard only by those for whom I intend it. If the content of my speech is available to the world, I have no privacy.
>
> **Eric Hughes**, *"A Cypherpunk's Manifesto", 1993*

As we have seen, the roots of cryptography are deeply tied with the military. However, with the advent of the Internet, cryptography gained traction outside the military. Indeed, regular people began to communicate over the Internet, using email or instant messaging services; some of them worried that their Internet or email provider could read the content of their communications, thereby hindering their privacy. It was therefore natural to start using cryptographic techniques to hide the content of their messages, through *secure messaging* techniques. In this section, we shall give a broad overview of the history and challenges of secure messaging; we refer the reader to [5] for a detailed survey.

[5]: Unger et al. (2015), *SoK: Secure Messaging*

Pretty Good Privacy (PGP). In 1991, Philip Zimmermann created the computer program Pretty Good Privacy (PGP), that aimed to encrypt and authenticate emails, using asymmetric encryption and cryptographic signatures. PGP was soon uploaded into a Usenet newsgroup [6]. Unfortunately because of the origins of cryptography being tied to the military, the United States had strict export restrictions on cryptography: soon after, Zimmermann became the target of a criminal investigation by the United States Government for "munitions export without a license" [7]. Zimmermann then invoked the First Amendment of the United States Constitution, namely freedom of speech and of the press, and published a book (therefore not subject to export restrictions) containing the source code of PGP [8]. The case was later dropped in 1996 [9].

[6]: Zimmermann (2001), *PGP Marks 10th Anniversary*

[7]: Zimmermann (1995), *Author's preface to the book: "PGP Source Code and Internals"*

[8]: Zimmermann (1995), *PGP source code and internals*

[9]: Zimmermann (1996), *Significant Moments in PGP's History: Zimmermann Case Dropped*

Bernstein v. United States. Early in the 1990s, Daniel J. Bernstein, back then a mathematics PhD student, designed an encryption algorithm and wanted to publish it online, however, to do so, the United States laws required him to register as an arms dealer, and apply for an export license. In 1995, Bernstein, with the help of the Electronic Frontier Foundation (EFF), sued the United States Government, and also invoked the First Amendment [10]. In 1999, the United States Court of Appeals declared that software source code was speech protected by the First Amendment, therefore that the export restrictions preventing its publication were unconstitutional [11].

[10]: Dame-Boyle (2015), *EFF at 25: Remembering the Case that Established Code as Speech*

[11]: (1999), *U.S. Court of Appeals for the Ninth Circuit: Bernstein v. USDOJ*

Cryptography and law, nowadays. The legal battles during the 1990s are not the end of the story: nowadays, cryptography is still regularly challenged by lawmakers, especially in the context of *secure messaging*. As a recent example: during the writing of these words, French legislators are debating a law to force the insertion of backdoors into secure messaging applications, so that the police can decrypt messages in order to help fighting against drug trafficking [12]. Such a backdoor, as we shall see, directly goes *against* the goals of secure messaging applications.

[12]: (2025), *Narcotrafic : le Sénat autorise les services de renseignement à accéder aux messageries cryptées*

Design flaws of PGP. After all these legal battles, let's go back to PGP and focus on another question, more central to this thesis: *is PGP a good solution toward secure messaging?* Unfortunately, PGP suffers from several design flaws. One of the design flaws is its *bad usability*: it is barely usable by the average computer user [13], however a secure messaging system

[13]: Whitten et al. (1999), *Why Johnny can't encrypt: a usability evaluation of PGP 5.0*

must be used in order to achieve its security goals. Another design flaw is the *lack of forward secrecy* (see Figure 1.4): if an adversary were to obtain the PGP private decryption key, they could use it to decrypt any message intended for this private key. In other words, when we use PGP to encrypt a message with a public encryption key, we must take into account the possibility that the corresponding private decryption key may be compromised by an adversary *in the future*. This design flaw is unacceptable by modern standards,[2] therefore led to the design of better secure messaging protocols, with the aim to provide forward secrecy.

2: and also by older standards, recall (§1.1.1) that Enigma was operated in a way to guarantee forward secrecy

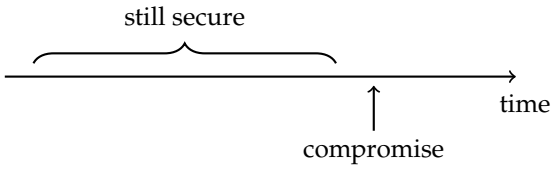


Figure 1.4: Forward secrecy: when an adversary compromises a participant and obtains cryptographic keys they store, they cannot decrypt messages sent in the past.

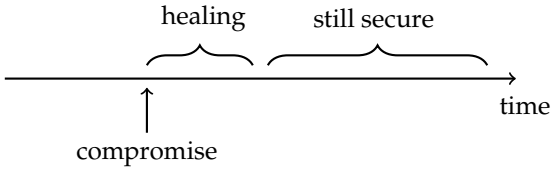


Figure 1.5: Post-compromise security: when an adversary compromises a participant and obtains cryptographic keys they store, they will not be able to decrypt messages sent in the future, after a short period of healing.

Off The Record (OTR). In response to the lack of forward secrecy in PGP, Ian Goldberg and Nikita Borisov designed in 2004 the Off The Record (OTR) messaging protocol [14], whose goal is to mimic the same security guarantees as an actual face-to-face conversation in a room.

[14]: Borisov et al. (2004), *Off-the-record communication, or, why not to use PGP*

To achieve *forward secrecy* (see Figure 1.4), OTR does not rely on asymmetric encryption for confidentiality: instead, it performs frequent Diffie-Hellman key exchanges to derive keys, which are then used with symmetric encryption to protect messages. When OTR exchanges a new symmetric key, the previous one as well as every other material involved for its computation is securely deleted. Thanks to this mechanism, OTR achieves forward secrecy: if in the future an adversary compromises one of the parties, because they will have deleted the key material of previous conversations, the adversary will not be able to decrypt them.

OTR also provides *post-compromise security* (see Figure 1.5): if an adversary compromises the cryptographic keys stored by one of the parties, the adversary will be able to decrypt messages sent using these keys, but because parties will rapidly perform new Diffie-Hellman key exchanges before encrypting next messages, the adversary will not be able to decrypt messages that will be sent in the future.

Short digression: OTR further provides *deniability*, meaning that when you send a message to someone, the recipient is cryptographically guaranteed that you sent this message, but cannot prove it to someone else, similarly to face-to-face conversations. However, nowadays, it seems that users of secure messaging protocols actually prefer non-repudiation rather than deniability, that is, the recipient of your message *can* prove to someone else that you sent the message [15]. Choosing between these two properties depends on who you want to protect: deniability protects senders of messages, non-repudiation protects receiver of messages. Therefore deniability and non-repudiation will not be in the scope of this thesis.

[15]: Yadav et al. (2023), *Cryptographic Deniability: A Multi-perspective Study of User Perceptions and Expectations*

One drawback of OTR is that it is a *synchronous* protocol: it requires the two parties of a conversation to be online at the same time, similarly to

face-to-face conversations. This means, for example, that OTR cannot be used as a replacement for Short Message Service (SMS) on mobile phones, because SMSes are asynchronous: one may send an SMS to someone who is offline, this person will receive it when they come back online.

Signal Protocol. In 2013, Trevor Perrin and Moxie Marlinspike published the TextSecure Protocol, an improvement over OTR to support asynchronous messaging [16], which was later renamed to the "Signal Protocol". They notice that OTR is synchronous only at the initialization (when a participant starts a conversation with another), however the rest of the protocol (sending messages) is asynchronous; therefore, only the initialization needs to be made asynchronous. They manage to do it through a change of paradigm: usually, long-term keys (e.g. signature keys associated to your identity) are published on servers, so that people can obtain them when you are offline, however short-term keys (e.g. Diffie-Hellman keys) are only used in a synchronous fashion; instead, the Signal protocol publishes on a server a large amount (e.g. 100) of short-term keys (called "pre-keys"), so that other people can initiate a conversation with you while you are offline. Furthermore, the server deletes pre-keys after they are issued so that they are only used once, and participants would refuse to use the same pre-key twice, therefore providing the same security guarantees as if the initialization were synchronous.

[16]: Marlinspike (2013), *Forward Secrecy for Asynchronous Messages*

Nowadays, the Signal Protocol is considered to be a state-of-the-art secure messaging protocol: it has been thoroughly analyzed [17–19], and has been deployed in Signal itself, as well as WhatsApp, Facebook Messenger, and Skype. However, it does not provide a final answer to the secure messaging problem: indeed, the Signal Protocol is a secure messaging protocol *between two persons*, whereas modern messaging applications allow for *group conversations*. Although we may implement group conversations by having group participants encrypt their messages separately to each group member [20], this does not scale well: sending a message to a group of n participants would require n times more computation and bandwidth than sending a message to only one person. The Sender Keys protocol (hinted in [20]) improves this scaling issue when sending of messages, but not when healing from a compromise. More generally, secure messaging in the context of groups comes with additional requirements, such as supporting dynamic groups (participants may join and leave the group at any time), which causes additional security challenges that must be tackled with a new protocol.

[17]: Cohn-Gordon et al. (2017), *A Formal Security Analysis of the Signal Messaging Protocol*
[18]: Alwen et al. (2019), *The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol*
[19]: Bienstock et al. (2022), *A More Complete Analysis of the Signal Double Ratchet Algorithm*

[20]: Marlinspike (2014), *Private Group Messaging*

Messaging Layer Security (MLS). In 2018, the Internet Engineering Task Force (IETF), a standards organization for the Internet, convened a working group tasked with designing a new secure group messaging protocol, dubbed Messaging Layer Security (MLS), with the ambition to scale well for large groups (e.g. thousands of participants) while providing strong security guarantees. In 2023, the MLS protocol is finalized and published in the RFC 9420 [21]. The MLS protocol is one of the objects of study of this thesis, aiming to answer the question: *is MLS really secure?* To provide an answer, we will need to define what "secure" means, furthermore, as we shall see in the next section, a positive answer to this statement always comes with fine print: indeed, we can rarely attest with 100% confidence the security of a cryptographic system. However, we have a variety of techniques to increase our confidence that a cryptographic system is indeed secure.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

1.1.4 Security of cryptographic systems

In 1949, Claude Shannon proved that symmetric encryption achieves perfect confidentiality only when the key is longer than the message it encrypts [22], or in other words, a system in which we may have 100% confidence in its security must be impractical. Therefore, because we cannot have a complete confidence in the security of practical cryptographic systems, we must rely on other methods to raise our confidence in their security. Different methods exist depending on the cryptographic system.

[22]: Shannon (1949), *Communication theory of secrecy systems*

Cryptanalysis. One way is to have many people try to break the cryptographic system: then, if no one succeeds after several years, we guess it's probably secure. This method works for any cryptographic system, but it does not provide the highest level of confidence: it may be that nobody managed to break the cryptographic system because we didn't put enough resources into trying to break it, or because everyone who tried was not clever enough. Thankfully, for many cryptographic systems, we have better ways to raise our confidence.

Proving the absence of some attacks. A complementary method is to collect general classes of attacks against the cryptographic system under study, and mathematically prove that these types of attacks do not work. For example, on symmetric encryption algorithms, it is customary to prove that they resist against differential cryptanalysis [23].

[23]: Biham et al. (1993), *Differential cryptanalysis of the data encryption standard*

Reduction proofs. On some cryptographic systems, especially on cryptographic protocols that assemble other cryptographic components (such as symmetric encryption, signatures, etc), we can mathematically prove that the security of the whole system *reduces* to the security of each cryptographic component. Informally, the mathematical proof says the following: if you show me how to efficiently break the security of my cryptographic system, I can use this knowledge to show you how to efficiently break the security of one of its cryptographic components. Because the cryptographic components are believed to be secure (e.g. because they resisted cryptanalysis so far), this implies that the cryptographic system is also believed to be secure.

Trade-offs in security proofs. When possible, reduction proofs are considered to be the gold standard. However, when it is too difficult to do such a proof, it is common to make stronger assumptions about the security of cryptographic components: for example, the Random Oracle Model [24] assumes that cryptographic hash functions are "perfect". One may say that cryptographic hash functions are not perfect, therefore these proofs have no value; however we argue that a proof with strong assumptions is better than no proof at all. In the end, this is a trade-off between the strength of the security theorem and the practical doability of its proof.

[24]: Bellare et al. (1993), *Random oracles are practical: a paradigm for designing efficient protocols*

Symbolic proofs. If we push the trade-off cursor further and make the strongest assumptions, we obtain what is called the "symbolic model". Informally, security theorems in the symbolic model prove that it is impossible for an adversary to break the whole cryptographic system when the adversary only has black-box access to cryptographic components, or in other words, if we assume that the cryptographic components being used in the system are "perfect". Again, these are strong assumptions, but a proof with strong assumptions is better than no proof at all.

Another way to understand the symbolic model is that it only considers adversaries that perform *logical attacks*, namely attacks that only have a black-box use of cryptographic components. Therefore, the symbolic model proves the absence of the whole class of logical attacks.

Computer-checked proofs. As we shall see in the next section, mathematical proofs may contain errors and security proofs are no exception, furthermore that we can rely on computers to raise our confidence that our mathematical proof is free of errors, by having the computer to mechanically verify each step of the proof.

In this thesis, we will perform computer-checked security proofs in the symbolic model.

1.2 Rigorous mathematical proofs

In §1.1, we have seen that we want to ensure that cryptographic protocols are secure. To do that, we can rely on mathematics, which can be used to define abstract objects (e.g. cryptographic protocols) and guarantee that they obey properties (e.g. they are secure). This guarantee is ensured by a "proof", which is in essence a detailed argument that aims to convince someone that the property indeed holds.

The history of mathematics has seen several crises of rigor, which led mathematicians to accept bogus proofs, or worse, false statements. In this section, we will discuss three of them.

1.2.1 Mathematical proofs, with pen and paper

> Since people have tried to prove obvious propositions, they have found that many of them are false.
>
> **Bertrand Russell**, *Mathematics and the Metaphysicians*

In the early 20th century, Bertrand Russell discovered that a standard reasoning technique in mathematics[3] led to a contradiction, now known as Russell's paradox, or in everyday terms, the barber paradox. In short, mathematics could prove the existence of a barber who "shaves all those, and those only, who do not shave themselves". However, the existence of such a barber is paradoxical when we ask: *does the barber shave itself?* Indeed, it is impossible that the barber shaves itself, because it "shaves only those who do not shave themselves", and it is also impossible that the barber does not shave itself, because it "shaves all those who do not shave themselves". Using mathematical notations, mathematicians could define the set $R = \{x | x \notin x\}$, which is paradoxical because $R \in R \iff R \notin R$.

3: known as "unrestricted comprehension principle"

Axioms. Such paradoxes are unacceptable in mathematics, which seeks to establish what is true and what is not. When we prove that some statement is true, we must rely on the knowledge that other statements are true: indeed, it is impossible to attest that something is true out of thin air. But how do we know these statements on which we rely on are indeed true? These could also be mathematically proved, but they themselves must also rely on other statements, etc. If we dig deeper and deeper into the dependencies of our proof, we must necessarily obtain statements that we assume to be true, but for which we don't have a

corresponding proof: these are called *axioms*. Such axioms are supposed to be "obviously true", for example it is obviously true that any number x must be equal to itself (i.e. $x = x$). An axiom naturally arises several kinds of questions. First, is this axiom really "obviously true"? Indeed, "obviously true" axioms led to Russell's paradox, which proved that they were actually wrong. Second, do we really need this axiom in our reasoning? Could we instead prove it using even more obviously true axioms? Third, is this axiom compatible with other axioms we use? That is, even if axioms may not create paradoxes when used alone, can they create paradoxes when used together?

The incompatibility of axioms is problematic when approaching mathematics as a community effort: it may be possible that two different mathematicians use incompatible axioms, so that we cannot combine their results without risking creating paradoxes.

Principia mathematica. In 1910, Bertrand Russell and Alfred North Whitehead published Principia Mathematica [25]. In there, they define a small set of axioms, and aim to prove as much mathematics as they can; furthermore they do not use informal prose in their proofs, instead they rely on a symbolic approach where theorem statements and proofs are written precisely using symbols. Their goal is to ensure that each theorem they prove depends on a clear set of axioms, therefore ensuring that they do not rely on theorems that use incompatible axioms. Furthermore, each proof can be verified mechanically, through manipulation of symbols. In the end, they obtain highly rigorous mathematical proofs, however it is labor-intensive: infamously, they prove, after 360 pages of definitions and intermediate lemmas, that $1 + 1 = 2$.

[25]: Russell et al. (1910), *Principia Mathematica Vol. I*

Social aspect of mathematics. In 1979, Richard DeMillo, Richard Lipton and Alan Perlis argued that successful mathematics are part of a social process [26], whereas Principia Mathematica views mathematics as a cold, formal and mechanical process. In their view, the main achievement of Principia Mathematica is to show how far we could go with a formalist approach to mathematics, which is, they argue, not very far. They explain that a highly detailed proof that can be mechanically verified does not necessarily help to understand it and gain insights into *why* the theorem statement is true, and that the latter is what makes a proof to be believed correct by the mathematical community. In short, they argue that the belief that a mathematical proof is correct by the mathematical community is a *social process*; a mathematical proof is a lively object that evolves: the insights of the proof are first explained to colleagues through informal blackboard discussions and seminars, then the proof is written and submitted for publication, it is then reviewed, published, and read by a wider audience of mathematicians that begin to internalize it, before it is finally used as a lemma in other theorems. After it stood the test of time, a proof is finally believed to be correct by the mathematical community.

[26]: De Millo et al. (1979), *Social processes and proofs of theorems and programs*

We shall see that this social approach to mathematics is fallible, hence the formalist approach made a comeback by using *computers* to mechanically check that a proof is correct, to be used *in parallel* of the social process.

1.2.2 Proof assistants

> A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail.
>
> **Vladimir Voevodsky**, *"Univalent Foundations", 2014*

In 2017, the mathematician Kevin Buzzard had a midlife crisis and losed faith in the social process that makes the mathematical community to believe a proof is correct [27]: he noticed that proofs which are accepted by the mathematical community and used to prove other theorems are sometimes incomplete or sometimes contain reasoning errors, and upon such discoveries, the mathematical community accepts when the elders of the field say "it is fixable". Furthermore, he noticed that peer-reviewed theorems are sometimes in direct contradiction with other peer-reviewed theorems, so that one of them must be wrong, but none of the theorems have been retracted.

[27]: Buzzard (2021), *Formalizing 21st century mathematics in Lean*

Proof assistants. To work against this crisis, Kevin Buzzard decided to formalize mathematics from first principles in the Lean proof assistant [28], a computer program that can mechanically verify that each step of a mathematical proof is correct. Buzzard then became involved in the development of mathlib [29], a project to formalize a significant part of modern mathematics using Lean. This project has been successful in finding and fixing mistakes in mathematical theories: in 2023, Terence Tao used Lean and the mathlib project to formalize a theorem he recently proved on paper [30], and found a mistake in his proof while formalizing it in Lean [31] (which he then fixed); in 2024, Kevin Buzzard started to formalize Fermat's Last Theorem [32] in Lean along with the mathematical theories on which it depends, soon after, he and his collaborators found that a key lemma of a theory on which Fermat's Last Theorem depends, seemed to be incorrect. Fortunately, experts of the field pointed them to an alternative presentation of the mathematical theory, which they successfully managed to formalize.

[28]: Moura et al. (2021), *The Lean 4 Theorem Prover and Programming Language*

[29]: Community (2020), *The lean mathematical library*

[30]: Tao (2023), *A Maclaurin type inequality*

[32]: Buzzard (2024), *Fermat's Last Theorem — how it's going*

Outside pure mathematics. Proof assistants have also been used with success outside pure mathematics, in domains that require mathematical reasoning. Such domains include proving properties of *programs*, or, more in the scope of this thesis, of *cryptographic protocols*. We shall discuss the use of computers to verify the security of cryptographic protocols more thoroughly in §1.3.

1.2.3 Cryptographic proofs

In 2004, Mihir Bellare and Phillip Rogaway called for a crisis of rigor in security proofs of cryptographic systems [33]: they noted that proofs in cryptography have grown so complex and tedious that they "have become essentially unverifiable". To tackle this issue, they proposed to structure security proofs as a sequence of "game-hops", also pioneered by Victor Shoup [34], arguing that it makes cryptographic proofs easier to verify, by, among other things, taming tedious probability calculations. They further suggested that the game-hopping technique could pave the way to making these proofs verified by a computer, also argued by Shai Halevi [35].

[33]: Bellare et al. (2004), *Code-Based Game-Playing Proofs and the Security of Triple Encryption*

[34]: Shoup (2004), *Sequences of games: a tool for taming complexity in security proofs*

[35]: Halevi (2005), *A plausible approach to computer-aided cryptographic proofs*

Nowadays, the game-hopping technique is pervasive in reduction proofs, and as we will see in the next section, has been successful in helping to make cryptographic proofs verifiable by computers.

1.3 Machine-checked analysis of cryptographic protocols

The use of computers to mechanically verify the security of cryptographic systems has been successful in the past decades, we now give an overview of what may be achieved using these techniques.

1.3.1 Computer-aided cryptographic proofs

In this section, we give a broad overview of the tools that aim to mechanically verify the security of cryptographic systems; we refer the reader to [36] for a detailed survey.

[36]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

In the computational model. Several tools perform traditional reduction proofs in what is called the "computational model". For example, EasyCrypt [37] allows verifying game-hopping proofs written similarly to pen & paper proofs; CryptoVerif [38] is designed with automation in mind, and aims to automatically find and verify game-hops; Squirrel [39] relies on the Bana-Comon logic [40] to provide computational guarantees.

[37]: Barthe et al. (2011), *Computer-Aided Security Proofs for the Working Cryptographer*

[38]: Blanchet (2007), *CryptoVerif: Computationally sound mechanized prover for cryptographic protocols*

[39]: Baelde et al. (2021), *An interactive prover for protocol verification in the computational model*

[40]: Bana et al. (2012), *Towards Unconditional Soundness: Computationally Complete Symbolic Attacker*

In the symbolic model. We introduced the symbolic model in §1.1.4, in which we consider an adversary that uses cryptographic functions as black-boxes. Several tools exist to prove security of cryptographic protocols in the symbolic model, which furthermore provide high automation: given a cryptographic protocol description, they either find an attack, or prove that the protocol is secure in the symbolic model. However, there is a catch: it may be that the analysis takes too much time, or uses too much memory, but when they don't, they show to be effective tools. Examples of such tools are ProVerif [41], in which we specify protocols as processes running in parallel; and Tamarin [42] in which we specify protocols as state machines.

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

In this thesis. To prove secure messaging protocols, we will use DY* [43], a recent tool which aims to provide more scalability and tackle protocols where ProVerif or Tamarin may otherwise timeout, at the expense of being less automated; in other words, DY* explores different trade-offs in the design space of symbolic analysis. We will present DY* in full details in Chapter 2, along with the improvements we made in the process of analyzing the secure messaging protocol MLS in Chapter 3.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

1.3.2 A case study: TLS 1.3

Everybody knows the padlock symbol in web browsers, or the https:// at the start of website addresses: this indicates that the connection with the website is secured via the cryptographic protocol named Transport Layer Security (TLS). As such, TLS is a wildly used cryptographic protocol, and we better hope it is indeed secure.

Attacks on TLS 1.2. TLS is a protocol with a rich history that spanned multiple versions, and aims to fit into the constraints requested by the industry that will eventually use it. As a result, TLS includes many modes of operation, backward compatibility mechanisms, etc. This leads to a broad attack surface, hence TLS went through many cycles of finding attacks and deploying mitigations [44]. Even the most thorough analysis of TLS 1.2 (e.g. [45]) did not consider every mode of operation of TLS, which led to miss for example the Triple Handshakes Attack [46] that exploits a subtle combination of session resumption and session renegotiation, two extensions of TLS.

[44]: Sheffer et al. (2015), *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*

[45]: Krawczyk et al. (2013), *On the Security of the TLS Protocol: A Systematic Analysis*

[46]: Bhargavan et al. (2014), *Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS*

With all the lessons learned by previous attacks of TLS 1.2 [44], the Internet Engineering Task Force (IETF), the standards organization in charge of TLS, started to work on a new version, now known as TLS 1.3.

TLS 1.3. Previous versions of TLS relied on a *reactive* approach toward protocol design: they were standardized and deployed, only afterward the academics studied the protocol and found attacks, that would either be mitigated via patches or fixed in the next versions. The design of TLS 1.3 relied on a *proactive* approach where the cycle of breaking and fixing by the academics happened *during* the design phase [47]. Furthermore, the tooling to mechanically verify the security of protocols was ready, and applied on TLS 1.3 during its standardization: for example, [48] analyzes draft 10 using Tamarin, [49] analyzes draft 18 using both ProVerif and CryptoVerif, and [50] analyzes draft 21 using Tamarin; each of these analysis uncovered design flaws that were subsequently fixed.

[47]: Paterson et al. (2016), *Reactive and Proactive Standardisation of TLS*

[48]: Cremers et al. (2016), *Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication*

[49]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*

[50]: Cremers et al. (2017), *A Comprehensive Symbolic Analysis of TLS 1.3*

1.4 This thesis

> Secure-messaging is the most fundamental privacy problem in cryptography: how can parties communicate in such a way that nobody knows who said what.
>
> **Phillip Rogaway**, *The Moral Character of Cryptographic Work*

In this thesis, we set out to do a *proactive* analysis of the secure group messaging protocol MLS by performing a machine-checked security proof.

1.4.1 New challenges in cryptographic protocol analysis

However, performing a machine-checked security analysis of secure group messaging protocols presents unique challenges.

Secure group messaging. In §1.1.3 we have seen several design goals of secure messaging protocols, and how the Signal Protocol provides an answer for one-to-one conversations. The setting of secure group messaging comes with further challenges that MLS aims to solve. First, MLS supports the following features:

- *asynchronous:* participants may sometimes be offline (discussed in §1.1.3)
- *dynamic groups:* people may join and leave the group at any time
- *efficient sending of messages:* the amount of computation when sending or receiving a message does not depend on the group size

- *efficient group update:* the amount of computation when updating the group state (e.g. by adding or removing people) scales less than linearly with the group size

Second, MLS aims to have the following security guarantees:

- *forward secrecy:* if an adversary compromises the cryptographic keys of a participant, they should not be able to decrypt messages sent in the past (discussed in §1.1.3)
- *post-compromise security:* if an adversary compromises the cryptographic keys of a participant, they should not be able to decrypt messages that will be sent in the future (discussed in §1.1.3)
- *add security:* when a participant joins the group, they are not able to decrypt messages that were sent before they joined the group
- *remove security:* when a participant leaves the group, they are not able to decrypt messages that will be sent after they leave the group
- *group membership agreement*: every member of the group agrees on who is in the group, in particular this precludes an adversary being in the group without your knowledge

The MLS working group was created in 2018, the same year the IETF standardized TLS 1.3. After the success of proactive protocol design with TLS 1.3 (see §1.3.2), the MLS working group decided to use the same approach.

In the following years, several works analyzed MLS using the traditional method of proving its security on pen & paper (e.g. [51] and [52]). They were successful to reveal attacks and design flaws in the intermediate designs of MLS, and helped the MLS working group to fix them. Unfortunately, the use of machine-checked security proofs was initially absent from the picture.

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*

Towards a machine-checked security proofs for MLS. The goal of security proofs is to raise the confidence in the security of a cryptographic protocol, and successfully manage to do so under three conditions:

1. the proof is free of any mistakes
2. the protocol being analyzed is actually the protocol we want to have guarantees on (e.g. MLS)
3. the security properties being proved actually correspond to the security properties we want (e.g. forward secrecy, etc)

With machine-checked security proofs, we can immediately gain confidence in the first two conditions: first, the proof is mechanically verified by a computer, which makes it unlikely to contain mistakes; second, we can make our protocol specification *executable* and test it for interoperability to ensure we analyze the correct protocol. Therefore, when reviewing such a machine-checked security proof, the focus can be made on the security properties being proved.

Challenges in analyzing MLS. Despite their advantages, no such machine-checked security proofs were performed on MLS, the reason being that it was at the time too difficult: indeed, MLS is based on Signal's double ratchet [53] but made more complex to account for groups, and at the time the best machine-checked analysis of Signal could only account for two messages [54]. Indeed, analyzing Signal (hence also MLS) comes with the unique challenge that a single session (i.e. conversation) involves encrypting an unbounded number of messages, whose keys are derived through a recursive chain of hashes. Furthermore, analyzing MLS comes with the unique challenge that there may be an

[53]: Perrin et al. (2016), *The Double Ratchet Algorithm*

[54]: Kobeissi et al. (2017), *Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach*

unbounded number of participants within a single session (i.e. group conversation) which furthermore may vary dynamically throughout the lifetime of a messaging group, as participants join and leave the group. Finally, an additional challenge posed by MLS is that to handle many participants in a group conversation, MLS internally relies on binary trees to compute its cryptography, whereas neither TLS 1.3 nor Signal use complex datastructures within their protocol specification.

1.4.2 Our solutions

This thesis aims to address the above challenges by providing the first, comprehensive, bit-precise, symbolic security analysis of the MLS protocol.

New tools for analyzing secure group messaging protocols. We saw in §1.4.1 that analyzing a protocol with the complexity of MLS was beyond the current state-of-the-art, therefore, a prerequisite for this work was to improve the foundation of tools and libraries to be able to conduct our proof over MLS.

In Chapter 3, we will first present the significant redesign and improvements we have made to the symbolic analysis framework DY* [43], which allowed us to tackle more complex invariants, thus increasing the expressive power of DY* so that MLS could be reasoned about.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

In Chapter 4, we will then present Comparse, a framework to specify and analyze message formats used in cryptographic protocols: this is an essential tool when analyzing a bit-precise specification. Furthermore, we will see that having a bit-precise specification was crucial not only to find previously-overlooked attacks, but also to find implementation flaws in existing implementations.

First machine-checked proofs for MLS. In the second part of this thesis, we use the tools we developed to *proactively* analyze MLS during its phase of standardization, and do so by analyzing a bit-precise, executable, interoperable specification.

In Chapter 5, we will show how to modularize MLS into three sub-protocols, which we name TreeSync, TreeKEM, and TreeDEM. We then analyze TreeSync, and prove that it allows MLS to provide *group membership agreement*. This is a prerequisite before proving any confidentiality property, because we cannot ensure the confidentiality of messages sent in the group if we don't agree with other participants on who is in the group. During the analysis, we initially found an attack on MLS that we reported to the MLS working group and helped to fix.

In Chapter 6, we will analyze TreeKEM, the sub-protocol in charge of continuously establishing a secret group key, and show that it provides *forward secrecy, post-compromise security, add security* and *remove security.* During the analysis, we found that forward secrecy and post-compromise security relied on application policies that were not enforced by the architecture document of MLS [55] and notified the working group.

[55]: Beurdouche et al. (2025), *The Messaging Layer Security (MLS) Architecture*

The only sub-protocol left out of our analysis is TreeDEM, which we unfortunately had to leave it as future work. Still, our analysis demonstrates that the tools we developed can be used to successfully analyze secure group messaging protocols, furthermore our analysis of MLS found several design flaws which we were able to report in time to the working group.

Developing tools and proof techniques for symbolic analysis at scale

DY*: Security proofs in the Dolev-Yao model, using F* (background)

2

The symbol is the tool which gives man his power, and it is the same tool whether the symbols are images or words, mathematical signs or mesons.

Jacob Bronowski, *"The Reach of Imagination" (1967)*

In this chapter, we give background on DY*, a tool recently developed to conduct symbolic security proofs using the F* proof assistant. As such, this chapter does not contain scientific contributions, which we reserve for the next chapter (Chapter 3).

In here, I will distill three years of personal insights of using and developing DY*. Therefore, although the purely scientific content of this section is not a contribution, how it is explained is a contribution.

Outline. In this chapter, we first give background on symbolic security analysis (§2.1), then on DY* (§2.2), and finally give a concrete example of security proof in DY* (§2.3).

2.1 Background on symbolic analysis

In this section, we explain what is symbolic security analysis, describe how the landscape looks like, and show where DY* belongs in this landscape.

2.1.1 The symbolic model

When we use a cryptographic protocol, we want to be sure it is secure. But what does "secure" means? For example, "secure" against whom? With what kind of capabilities? To answer these questions, we must come up with an *attacker model*, before proving that our cryptographic protocol is secure against *this class* of attackers.

The Dolev-Yao attacker. In their seminal work, Dolev and Yao [56] introduce, influenced by the work of Needham and Schroeder [57], an attacker model which is nowadays called "the Dolev-Yao attacker". In short, the Dolev-Yao attacker is an active attacker with a black-box usage of cryptography.

[56]: Dolev et al. (1983), *On the security of public key protocols*

[57]: Needham et al. (1978), *Using Encryption for Authentication in Large Networks of Computers*

An active attacker. We consider that the attacker is in charge of connecting protocol participants with each other. The attacker can thereby decide whether to be evil or not, for example by dropping messages, sending some messages twice, or modifying them on the fly. More precisely, when a participant sends a bytestring to the attacker (the network), this extends the knowledge of the attacker with this bytestring, and when a participant receives a bytestring from the attacker (the network), this bytestring must be known by the attacker.

With a black-box usage of cryptography. The attacker doesn't only know bytestrings sent on the network: they can also compute cryptographic

functions using the bytestrings they already know. For example, if the attacker knows a key, and an encryption of a plaintext with that key, the attacker can use the decryption function to gain the knowledge of the plaintext. However, if the attacker doesn't know the key, the ciphertext (thankfully) does not reveal the plaintext.

Symbolic model. Security proofs that prove a protocol secure against the Dolev-Yao attacker are also said to be done "in the symbolic model". This is because cryptographic functions are used in a black-box fashion, thus can be treated as uninterpreted symbols; the only thing we know about them is that the underlying cryptographic functions are *correct*, hence must obey some equations (e.g. decrypting an encrypted message with the same key yields the same message). We will develop this notion further in §2.2.3 where we will explain how we model bytestrings in DY*.

Machine-checked proofs. In the original work of Dolev and Yao [56], the symbolic model was used to conduct pen & paper security proofs. Since then, the symbolic model enjoyed a lot of success with machine-checked proofs, which led to the creation of a whole bazaar of tools. One of these tools is DY* [43], a recent contender which blends two approaches toward symbolic security proofs, that we describe in §2.1.2 and §2.1.3.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

2.1.2 Trace-based symbolic proofs

Before proving the security of a protocol in the symbolic model, we must first properly specify the cryptographic protocol in question, and precisely write the security guarantees we want to prove. One way to do that is by using *trace semantics*; this is the approach used by ProVerif [41] and Tamarin [42].

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

State machines. Participants in cryptographic protocols are specified as state machines, that receive some data from the network, modify their internal state, send back a message on the network, then wait for the next message.

The trace. To reason on the behavior of protocol participants, we will not actually have them perform impure actions (i.e. actions that are not the result of computing pure mathematical functions), such as sending messages on the actual network or generating actual fresh random bytestrings. Instead, we will store an entry corresponding to this impure action in a global, shared, append-only trace. The trace is therefore a log of every impure action that happened during a protocol execution. We can deduce from the trace all the bytestrings known by the attacker (which can be used to express confidentiality properties), or see whether Alice thinks she is talking with Bob (which can be used to express authentication properties).

Security properties. We can use the trace to define security properties, in particular *reachability* properties, which says that no matter how the attacker behaves, every trace they may *reach* by interacting with protocol participants must obey some security properties. These security properties are therefore stated as properties of the reachable traces, and will depend on the protocol being analyzed; some usual security properties are confidentiality (the attacker cannot know some secret value) or authentication (if Alice responds to Bob, then Bob initiated conversation with Alice before).

Tools such as ProVerif and Tamarin can also prove *equivalence* properties, but it is out of scope for DY* hence we will not explain these properties further.

Trace-based tools. Several tools (e.g. ProVerif [41] and Tamarin [42]) allow doing security proofs in the symbolic model in a highly automated fashion. They rely on trace semantics to automatically explore all possible behaviors of the attacker, allowing them to either find an attack, or prove that the protocol is secure (if they find none). This high automation however comes with drawbacks: another possibility is that the analysis takes too much time, or uses too much memory. It is therefore useful to consider other approaches toward symbolic analysis, such as type-based symbolic proofs.

2.1.3 Type-based symbolic proofs

Another line of work [58, 59] leverages *types* (especially *refinement types*) to guarantee security in the symbolic model.

[58]: Bengtson et al. (2008), *Refinement Types for Secure Implementations*
[59]: Bhargavan et al. (2010), *Modular verification of security protocol code by typing*

Refinement types. In mainstream programming languages, types are a tool to ensure that some behaviors never happen, e.g. that we never compute the addition of a number and a string, because in this case addition is not defined. However, in some scenarios these types may be too coarse-grained, for example, consider array indexing: how do we ensure, within the type system, that the index is within the array bounds? In mainstream programming languages, the array index is simply an integer, hence we cannot ensure that it is within the array bounds; however, in programming languages with refinement types, we can *refine* the integer with the property that it is within the array bounds (see Figure 2.1). When using this array indexing function with refined types, the typechecker will try to *prove* that the index is within bounds, and will give a type error if it doesn't succeed to prove so. We give examples of using this index function with refined types in Figure 2.1, and point where it would successfully typecheck and where it would fail.

```
val index:
  a:array α  →  i:int{0 ≤  i < len a} →
  α

let f (a:array α)  (i:int) =
  let x1 = index [10;11;12] 5 in // fail
  let x2 = index [10;11;12] 1 in // ok
  let y1 = index a i in // fail
  if 0 ≤  i < len a then
    let y2 = index a i in // ok
    ...
  else
    ...
```

Figure 2.1: The index function, with refined types.

Trace properties from refined types. We can (ab)use refinement types to prove trace properties, as we show in Figure 2.2. Imagine we receive messages, and we can validate or process them. One may want to prove the trace property: if a participant processes a message, then it must have validated the same message before. We can prove this property using refinement types: the function validate_message will return a refined type that tells the message has been validated, then the function process_message relies on refinement types to only accept messages that have been validated before. We can adapt this methodology to cryptographic functions to prove security properties on cryptographic protocols; we will show how when presenting DY* in §2.2.

```
val validate_message:
  m:msg →
  unit{message_validated m}

val process_message:
  m:msg{message_validated m} →
  ...
```

Figure 2.2: Trace property using refined types.

Modular verification. One advantage of type-based symbolic proofs is that the verification is *modular*: indeed, the typechecker verifies each function independently, unlike trace-based tools like ProVerif and Tamarin that perform whole-protocol analysis.

Executable specification. In type-based symbolic proofs, the protocol specification is a program that implements the cryptographic protocol. Therefore, another advantage of type-based symbolic proofs is that the

specification can be executed, thereby obtaining a reference implementation.

From types to trace semantics. The soundness of the type of cryptographic functions (that is, how they use refinement types) is proved on pen and paper. It means that when extending the system, this pen and paper proof must be updated: this is risky, because a mistake there might make the overall approach unsound. Thankfully, DY* proposes a solution to this problem by proving the soundess directly within F*.

2.1.4 DY*, type-based proofs with trace semantics

DY* [43] is a recent tool that combines the two aforementioned approaches: DY* users do type-based symbolic proofs (see §2.1.3) but obtain security guarantees expressed with trace semantics (see §2.1.2) without relying on an extra pen & paper proof to ensure its soundness, because the soundness proof is mechanized within the F* proof assistant. In short, DY* defines trace semantics within F*, which allows stating security properties using trace semantics, as with ProVerif or Tamarin. These security properties are then proved using type-based symbolic proof techniques, which are proven sound within F*.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Because the soundness theorem is machine-checked, this allows to gain confidence when extending the feature set of DY*, these extensions then allows to have better expressivity than previous type-based approaches.

Dynamic compromise. DY* has native support for dynamic compromise, therefore can reason on forward secrecy or post-compromise security. This is difficult to express in type-based symbolic proofs where the only way to encode trace properties is using refinement types.

Equational theories. DY* can reason on cryptographic functions that rely on equational theories, such as Diffie-Hellman which exhibits a commutative-like behavior.

Mutable state. DY* supports reasoning on protocols with mutable state. This is a requirement to prove forward secrecy or post-compromise security properties on ratcheting protocols.

All of these features will be crucial when conducting security proofs for Messaging Layer Security [21] in Chapter 5 and Chapter 6.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

2.2 Symbolic analysis with DY*

This section explains the inner workings of DY* when I started using it. Even if this section describes the work from the original DY* paper [43], we will use notations and terminology from the new version of DY* we developed and will describe in Chapter 3.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

2.2.1 Cryptographic protocols in DY*

Each tool has its own way to specify protocols and represent an attacker, in this section we describe how DY* does it.

Protocol participants in DY*. Recall (§2.1.2) that protocol participants are state machines that may perform impure actions (i.e. actions that are not the result of computing pure mathematical functions, such as sending a message, storing state, or generating fresh randomness), and that these impure actions are performed by writing them in a *trace*: a global, shared, append-only log of every impure action that happened during a protocol execution. In DY*, we specify protocol participants by implementing their state machine via a set of effectful functions that perform the transition from one state to another. The effect of these functions is handled by reading and modifying the trace: protocol participants can read messages from the network or retrieve states by obtaining them from the trace, they can send messages on the network or store states by appending corresponding entries in the trace.

Compromise. In the real world, secret state stored by a participant may leak to the attacker (e.g. a telephone is seized by the police, or a laptop is hacked using a zero-day exploit). To capture this, the DY* attacker may compromise the state of protocol participants, thereby extending their set of known bytestrings. Therefore, all security properties are expressed "modulo compromise", for example we do not say that the attacker never knows a secret value, we say that the attacker needs to perform some compromise in order to know a secret value. To express this kind of property, the compromise is represented as an entry in the trace.

The DY* attacker. The attacker is then an F* function that can call any cryptographic functions and can call the various effectful functions defined by the protocol specification. Any symbolic attack can be represented by such an F* program.

2.2.2 Proving trace properties with DY*

In this section, we give an overview of the techniques used by DY* to prove *trace properties*, which will be further developed in later sections.

Suppose we defined a protocol specification in DY*, and we know that symbolic attackers for this protocol are F* programs that are restricted to use a specific API (computing cryptographic functions, and interact with the protocol). We now want to prove that all traces reachable by such attackers satisfy some security properties.

Trace invariant. To do so, DY* requires users to strengthen the desired security properties to obtain a *trace invariant*, then prove that every reachable trace satisfies this trace invariant. We do so by requiring DY* users to prove that every effectful protocol function preserves the trace invariant, then, because every function used by the attacker preserves the trace invariant, the attacker function has no choice but to also preserve the trace invariant. This concludes that the attacker can only reach traces that satisfy the security properties. For usability, DY* doesn't actually require users to provide full-fledged trace invariant; instead DY* provides a basic skeleton for trace invariant, and allows users to complete it with protocol-specific invariants. This also allows DY* to prove once and for all that when the attacker receives messages or compromises states,

computes cryptographic functions and sends the result on the network, this preserves the trace invariant. See §2.2.9 for more details about the trace invariant.

Bytes invariant. A key ingredient of the trace invariant is the *bytes invariant*, which describes what is a "hygienic use of cryptography" by honest participants; for example they may only sign bytestrings that verify some property (which is specific to each protocol, and captures the meaning of the signature), or they may only encrypt messages to keys that are "more secret" than the message (to ensure the attacker cannot use the ciphertext and the key they know to learn something they are not allowed to). The bytes invariant is an invariant on all the bytes that appear in the protocol, whether computed by honest protocol participants or by the attacker. The bytes invariant relies on lower-level tools described below, such as labels or key usages. See §2.2.8 for more details about the bytes invariant.

Labels. Sharing ideas with Information Flow, labels capture what it means for a bytestring to be "more secret" than another. Every bytestring is associated with a label, and two labels can be compared to determine whether one is more secret than the other. The order between labels depends on the compromise events in the trace: when an attacker compromises the state of a principal, the label corresponding to this state will become equivalent to the public label. Then, the Attacker Knowledge Theorem (described below) will guarantee that any bytestring known by the attacker has a label equivalent to the public label. See §2.2.6 for more details about labels.

Key usage. Security proofs (whether symbolic or computational) often rely (sometimes implicitly) on the fact that some keys are distinct from each other. For example, if we prove a security theorem on TLS, we implicitly assume that we are not using the same long-term keys (say) in SSH. Indeed, if TLS and SSH were to share long-term keys, we would need to do a combined security proof to ensure there is no cross-protocol attack. Another example, computational security proofs rely on the fact that signature keys are not used to encrypt: doing so would prevent applying the standard EUF-CMA security assumption. In more generality, it is good hygiene to make sure a key is used with only one cryptographic primitive. In DY*, cryptographic keys are associated with a *usage*, which ensures that two keys with different purposes are distinct. See §2.2.7 for more details about key usages.

Attacker knowledge theorem. A key ingredient to prove confidentiality theorems is the *Attacker Knowledge Theorem*, which states that the attacker only knows bytestrings (1) that satisfy the bytes invariant, and (2) whose label is equivalent to the public label. This allows to prove confidentiality properties as follows: if the attacker knows some secret key (of which we know the label, e.g. only Alice is supposed to know that key), then by the Attacker Knowledge Theorem, the label of that key is equivalent to the public label, which can only happen if some compromise has happened (e.g. the attacker has compromised Alice). See §2.2.10 for more details about the Attacker Knowledge Theorem.

Outline. We just gave an overview of DY*, the rest of this section will explain DY* in details and proceed as follows:

- define symbolic bytes (§2.2.3)
- define the trace (§2.2.4)

- define attacker knowledge (§2.2.5)
- define labels (§2.2.6)
- define key usages (§2.2.7)
- define the bytes invariant (§2.2.8)
- define the trace invariant (§2.2.9)
- state the attacker knowledge theorem (§2.2.10)
- and discuss (§2.2.11)

2.2.3 Symbolic bytes

```
// Step 1: define the bytes type and equality predicate

type bytes =
  | SymEnc: key:bytes → plaintext:bytes → bytes
  | SymDec: key:bytes → ciphertext:bytes → bytes
  // ...

let bytes_equal (lhs:bytes) (rhs:bytes): prop =
  // ...

// Step 2: define encryption and decryption functions

let sym_enc (key:bytes) (plaintext:bytes): bytes =
  SymEnc key plaintext

let sym_dec (key:bytes) (ciphertext:bytes): bytes =
  SymDec key ciphertext




// Step 3: prove the reduction rule for symmetric encryption.

val decrypt_encrypt_equal:
  key:bytes → plaintext:bytes → Lemma (
    bytes_equal
      (sym_dec key (sym_enc key plaintext))
      plaintext
  )
```

(a) Straightforward definition for symbolic bytes. The reduction lemma uses a custom equality relation (bytes_equal). The symmetric encryption and decryption are only wrappers around constructors of bytes.

```
// Step 1: define the bytes type and equality predicate

type bytes =
  | SymEnc: key:bytes → plaintext:bytes → bytes
  | SymDec: key:bytes → ciphertext:bytes → bytes
  // ...

// no 'bytes_equal', we rely on F*'s standard equality (==)


// Step 2: define encryption and decryption functions

let sym_enc (key:bytes) (plaintext:bytes): bytes =
  SymEnc key plaintext

let sym_dec (key:bytes) (ciphertext:bytes): bytes =
  match ciphertext with
  | SymEnc key' plaintext →
    if key = key' then plaintext
    else SymDec key ciphertext
  | _ → SymDec key ciphertext

// Step 3: prove the reduction rule for symmetric encryption.

val decrypt_encrypt_equal:
  key:bytes → plaintext:bytes → Lemma (
    sym_dec key (sym_enc key plaintext)
    ==
    plaintext
  )
```

(b) More usable definition for symbolic bytes. The reduction lemma uses F*'s built-in equality (==). To do that, symmetric decryption checks whether the ciphertext is a symmetric encryption under the key that was used for encryption.

Figure 2.3: Definition of symbolic bytes in F*

We now describe how DY* models symbolic bytestrings. As a running example, we will show how to model symmetric encryption (without nonces nor additional data, for simplicity).

Concrete cryptography. To understand symbolic cryptography, we must first put it in contrast with concrete cryptography. In the real world, when we use an encryption function to encrypt some data with some key, the key and data are concrete sequences of bytes (that you can e.g. print out), and the output is another concrete sequence of bytes. The actual result will depend on the encryption algorithm being used, however, any encryption algorithm must ensure that when we use the decryption

function on the encrypted data with the same key, we will obtain the original data back. This property is a *functional correctness* property, and is crucial to ensure that the encryption algorithm is actually useful.

Symbolic cryptography. In the symbolic world, cryptographic functions are uninterpreted. For example, when using an encryption function, we do not actually *compute* the actual encryption algorithm, we simply remember that the output is the encryption of a given data with a given key. Because of this uninterpreted nature, symbolic bytestrings are often called "terms".[1] Despite being uninterpreted, symbolic cryptographic functions must obey some properties, namely the functional correctness properties. For symmetric encryption, this property is:

1: e.g. both in the ProVerif manual and the Tamarin manual

$$\mathsf{sym\_dec}(\mathsf{k}, \mathsf{sym\_enc}(\mathsf{k}, \mathsf{msg})) == \mathsf{msg}$$

In the symbolic world, it is the only property obeyed by symmetric encryption, hence the only way to recover the plaintext from a ciphertext is to decrypt it with the corresponding key, or in other words, it is *impossible* to decrypt an encrypted message without the key.

Straightforward F* definition. In Figure 2.3a we do a naive attempt at implementing symbolic bytestrings in F*. We implement bytestrings as an inductive type that describes how cryptographic functions were used to obtain the bytestring. With this approach, encryption and decryption do not satisfy the functional correctness property of symmetric encryption, because in F*, SymDec key (SymEnc key msg) is different from msg. To solve this problem, we introduce a custom equality relation named bytes_equal. With this new notion of equality between bytestrings, we can now ensure that encryption and decryption satisfy the functional correctness property of symmetric encryption.

Proofs and bytes_equal. This new equality on bytestrings must satisfy the standard properties of equality, namely reflexivity,[2] symmetry,[3] transitivity,[4] and congruence.[5] There are two problems with this approach. First, it means that the designers of DY* must prove these properties on bytes_equal, which can be tedious (especially for the congruence property that must be stated for every cryptographic function). Second, it means that DY* users need to rely on these properties when doing security proofs. This introduces an additional burden on the user, which we can avoid.

In the notes below, we will temporarily write $x \simeq y$ for bytes_equal x y

2: $\forall x. x \simeq x$

3: $\forall x, y.\ x \simeq y \Rightarrow y \simeq x$

4: $\forall x, y, z.\ x \simeq y \wedge y \simeq z \Rightarrow x \simeq z$

5: $\forall f, x, y.\ x \simeq y \Rightarrow f(x) \simeq f(y)$ (and same for functions with arity ≥ 2)

Better F* proof-engineering. In Figure 2.3b we show how to better implement symbolic bytestrings in F*. Now, the decryption function checks whether the ciphertext is an encryption under the good key. When this is the case, it returns the corresponding plaintext, otherwise it uses SymDec as before. Doing this allows to state the functional correctness property of symmetric encryption using F*'s built-in equality (==). The benefit is that F* reasons natively, automatically and efficiently on the reflexivity, transitivity and congruence of its built-in equality.

Normal form of bytestrings. This approach requires finding a normal form for every bytestring, so that two equal bytestrings have the same normal form; and requires every cryptographic function to keep the bytes in normal form. This requirement is easy to satisfy in practice, for example by using the encryption / decryption trick explained above. For cryptographic functions that feature commutativity (such as Diffie-Hellman), it suffices to use an arbitrary comparison function on bytestrings and sort the operands of this cryptographic function. Note that we can do this

because our bytes term is fully concrete: it doesn't contain any variable term that could be instantiated later, unlike the bytes terms in ProVerif or Tamarin.

2.2.4 Trace

We now describe how traces are represented in DY*.

List of impure actions. The trace is the list of impure actions that happened throughout the protocol execution. In Figure 2.4, we define the trace as a list of trace entries and trace entries as a sum type for each entry type. We will describe the various actions (hence each entry type) in the next paragraphs and define notation that we will use throughout the rest of the chapter.

By convention, we will use the letter τ to note traces. Several propositions will depend on an ambient trace τ, when this is the case, we will use the notation $\tau \vdash \cdots$ to say that propositions in "$\cdots$" rely on the trace τ, for example $\tau \vdash \mathcal{F}(x)$ is another way of writing that the proposition $\mathcal{F}(\tau, x)$ is true. We may sometimes omit $\tau \vdash$ and keep it implicit to avoid cluttering formulas. We will also write $\tau{+}{+}\cdots$ when we extend the trace with new entries.

```
type trace_entry =
  | SendMsg: bytes → trace_entry
  | SetState: principal → state_id → bytes → trace_entry
  | Compromise: principal → trace_entry
  | RandGen: nat → label → usage → trace_entry
  | CustomEvent: principal → string → bytes → trace_entry

type trace = list trace_entry
```

Figure 2.4: Definition of the trace in F*.

Sending and receiving message on the network. Protocol participants communicate with each other by sending messages on one end and receiving them on the other end. When a participant sends a message, the trace is extended with a SendMsg entry. However, when receiving messages, the trace is not extended: instead, participants receive a message corresponding to a SendMsg entry. We model the fact that the attacker is active using the fact that the attacker can read any message sent on the network, that they can themselves send messages on the network, and that they choose which message sent on the network protocol participant will actually receive. When a message b was sent on the network (with respect to a trace τ), that is, τ contains the entry SendMsg b, we write $\tau \vdash \boxtimes$b. We show its precise semantics in Figure 2.5.

$$\frac{}{\tau{+}{+}\mathsf{SendMsg(b)} \vdash \boxtimes \mathsf{b}}\ \textsc{Sent-SendMsg}$$

$$\frac{\tau \vdash \boxtimes \mathsf{b}}{\tau{+}{+}\mathsf{entry} \vdash \boxtimes \mathsf{b}}\ \textsc{Sent-Append}$$

Figure 2.5: Semantics of $\boxtimes$b.

Setting and retrieving state. Protocol participants can store state, for example short-term keys when waiting for the response of other participants, or long-term keys that are used in multiple sessions. When a participant stores state, the trace is extended with a SetState entry. Similarly to receiving messages, when participant retrieve some previously stored state, they receive a bytestring corresponding to a SetState entry. Participant differentiate between their multiple states using a *state identifier*, which acts as a pointer. When a principal P stored a bytestring b (with respect to a trace τ), that is, τ contains the entry SetState P sid b for some state identifier sid, we write $\tau \vdash \mathsf{P}\blacktriangleleft\mathsf{b}$ (we omit the state identifier for simplicity). We show the semantics of state storage in Figure 2.6.

$$\frac{}{\tau{+}{+}\mathsf{SetState(P, sid, b)} \vdash \mathsf{P}\blacktriangleleft\mathsf{b}}\ \textsc{Stored-SetState}$$

$$\frac{\tau \vdash \mathsf{P}\blacktriangleleft\mathsf{b}}{\tau{+}{+}\mathsf{entry} \vdash \mathsf{P}\blacktriangleleft\mathsf{b}}\ \textsc{Stored-Append}$$

Figure 2.6: Semantics of P$\blacktriangleleft$b.

Compromise. The attacker can compromise the state of any principal. When they do so, the trace is extended with a Compromise entry which contains the identity of the compromised principal. When the state of a principal P has been compromised (with respect to trace τ), we write $\tau \vdash \lightning P$ We show the semantics of compromise in Figure 2.7.

In reality, the Compromise entry also contains the state identifier that was compromised, we chose to omit it from this presentation for simplicity. In later version of DY* (Chapter 3) the Compromise entry will be even more fine-grained and point to specific SetState entry.

$$\frac{}{\tau +\!+ \text{Compromise(P)} \vdash \lightning P}\ \textsc{Compromised-Compromise}$$

$$\frac{\tau \vdash \lightning P}{\tau +\!+ \text{entry} \vdash \lightning P}\ \textsc{Compromised-Append}$$

Figure 2.7: Semantics of $\lightning P$.

Generating fresh randomness. Protocol participants and the attacker can generate fresh random bytestrings. When they do so, the trace is extended with a RandGen entry. This entry contains various properties of the bytestring: its length, its label and its usage. Looking at the bytes type (Figure 2.8), the Rand constructor also contains the properties of the bytestring, as well as a timestamp: this is the index in the trace of the corresponding RandGen entry. This ensures that two fresh random bytestrings with the same properties will be different, because they will contain different timestamps.

```
type bytes =
  | Rand:
    timestamp →
    nat → label → usage →
    bytes
  | ...
```

Figure 2.8: The "fresh randomness" constructor of bytes.

Later, in §3.3, we will revisit the Rand constructor of bytes so that it only contains its timestamp and length, but in the original DY* (hence this section) it also contains the label and usage.

Logging protocol-specific event. Finally, protocol participants can log custom, protocol-specific events, for example "I am Alice, I finished a handshake with Bob and obtained the key k". Such events are crucial to state authenticity theorems, which are usually of the form "if Alice logged the event that she finished a handshake with Bob, then before, Bob must have logged the event that he initiated a handshake with Alice". Logging an event extends the trace with a CustomEvent entry.

Implementing protocol steps in F*. Now that we defined the trace and showed how we model participants doing impure actions, we can implement protocol steps in F* as functions that work in a state monad, where the state is a trace. Furthermore, the state is monotonic, because the trace only grows. We show a slightly simplified definition of the trace monad in Figure 2.9. In practice, when implementing protocol steps, the trace is hidden from DY* users thank to helper functions that abstract the trace away: for example, when sending messages on the network, DY* users are not expected to manually append SendMsg entries, instead, they will rely on a function send_msg that will append the corresponding SendMsg entry. In the end, when implementing protocol steps, DY* user will never manipulate the trace directly, they will always do so using helper functions similar to send_msg and combine traceful functions without relying on the actual definition of traceful.

```
type traceful (a:Type) =
   trace → (a & trace)

let send_msg (b:bytes): traceful unit =
   λ tr → ((), (SendMsg b)::tr)
```

Figure 2.9: Definition of the trace monad.

Prefix. During the protocol execution, the trace is extended with every impure action performed by participants or by the attacker. Hence, if at some point in the protocol execution, the trace is τ_1, and later on, the trace is τ_2, it must be that τ_1 is a prefix of τ_2, which we write $\tau_1 \subseteq \tau_2$. When $\tau_1 \subseteq \tau_2$, we say that τ_2 is "later" than τ_1.

$$\frac{\tau_1 \vdash \cdots \qquad \tau_1 \subseteq \tau_2}{\tau_2 \vdash \cdots}\ \textsc{\cdots-Later}$$

Figure 2.10: Generic Later inference rule, that will be found in most predicates and relation in DY*.

Trace predicates monotonicity. Predicates and relations in DY* generally depend on the trace because they depend on what happened during the protocol execution. These predicates and relations will satisfy a key property being that they stay true when the trace is extended, i.e. when a property is true, it will stay true in the future. For example, if a message

was sent on the network (i.e. $\tau_1 \vdash \boxtimes \mathsf{b}$), then later (i.e. $\tau_1 \subseteq \tau_2$), the message will also have been sent on the network (i.e. $\tau_2 \vdash \boxtimes \mathsf{b}$). In full generality, these predicates and relations (such as $\boxtimes \mathsf{b}$) obey a "later" rule, of which we give a template in Figure 2.10.

2.2.5 Attacker knowledge

We now formally define the attacker knowledge. What bytestrings are known by the attacker depends on the messages sent on the network, and on the states compromised by the attacker. Therefore, the attacker knowledge depends on the trace.

Notation. We write $\tau \vdash \mathcal{A}(\mathsf{b})$ when the attacker knows b for a trace τ.

Definition. We give the semantics of attacker knowledge in Figure 2.11. The attacker knows all messages sent on the network (Att-Sent), and all compromised bytestrings (Att-Compromise). Furthermore, the attacker can compute any cryptographic functions on bytestrings they already know (Att-F). In practice, we have one rule per cryptographic function, we give an example for symmetric encryption / decryption in Figure 2.12 (Att-Encrypt and Att-Decrypt).

$$\textsc{Att-Sent}\ \frac{\boxtimes \mathsf{b}}{\mathcal{A}(\mathsf{b})} \qquad \textsc{Att-Compromise}\ \frac{\lightning \mathsf{P} \qquad \mathsf{P} \blacktriangleleft \mathsf{b}}{\mathcal{A}(\mathsf{b})}$$

$$\textsc{Att-F}\ \frac{\mathcal{A}(\mathsf{b}_1) \qquad \dots \qquad \mathcal{A}(\mathsf{b}_n)}{\mathcal{A}(\mathsf{f}(\mathsf{b}_1, \dots, \mathsf{b}_n))}$$

Figure 2.11: Semantics of attacker knowledge. The $\tau \vdash$ is left implicit when there is only one trace (i.e. everywhere). The rule Att-F is a generic rule, where f is any cryptographic function.

Example. If a ciphertext is sent on the network and the corresponding symmetric key has been compromised by the attacker, then the attacker knows the plaintext. This is knowledge is the result of the following derivation:

$$\textsc{Att-Decrypt}\ \frac{\textsc{Att-Compromise}\ \dfrac{\lightning \mathsf{P} \qquad \mathsf{P} \blacktriangleleft \mathsf{key}}{\mathcal{A}(\mathsf{key})} \qquad \textsc{Att-Sent}\ \dfrac{\boxtimes \mathsf{sym\_enc}(\mathsf{key}, \mathsf{plaintext})}{\mathcal{A}(\mathsf{sym\_enc}(\mathsf{key}, \mathsf{plaintext}))}}{\underbrace{\mathcal{A}(\mathsf{sym\_dec}(\mathsf{key}, \mathsf{sym\_enc}(\mathsf{key}, \mathsf{plaintext})))}_{\mathsf{plaintext}}}$$

$$\textsc{Att-Encrypt}\ \frac{\mathcal{A}(\mathsf{key}) \qquad \mathcal{A}(\mathsf{plaintext})}{\mathcal{A}(\mathsf{sym\_enc}(\mathsf{key}, \mathsf{plaintext}))}$$

$$\textsc{Att-Decrypt}\ \frac{\mathcal{A}(\mathsf{key}) \qquad \mathcal{A}(\mathsf{ciphertext})}{\mathcal{A}(\mathsf{sym\_dec}(\mathsf{key}, \mathsf{ciphertext}))}$$

Figure 2.12: Example instances of the Att-F rule, for symmetric encryption and decryption.

Implementing attacker knowledge in F*. The attacker knowledge predicate is the weakest predicate that obeys the rules defined in Figure 2.11. We implement it in F* as a Kleene's least fixpoint [60] in the following way: we define an increasing sequence of attacker knowledge predicates $\mathcal{A}^n(\square)$, then define the attacker knowledge as $\mathcal{A}(\mathsf{b}) := \exists n. \mathcal{A}^n(\mathsf{b})$. We define $\mathcal{A}^n(\square)$ inductively on n (we omit $\tau \vdash$ everywhere to avoid clutter):

$$\begin{aligned}
\mathcal{A}^0(\mathsf{b}) &:= \bot \\
\mathcal{A}^{n+1}(\mathsf{b}) &:= \boxtimes \mathsf{b} && \text{(Att-Sent)} \\
&\quad \vee (\exists P. \lightning \mathsf{P} \wedge \mathsf{P} \blacktriangleleft \mathsf{b}) && \text{(Att-Compromise)} \\
&\quad \vee \left(\exists \mathsf{b}_1, \dots, \mathsf{b}_n. \mathsf{b} = f(\mathsf{b}_1, \dots, \mathsf{b}_n) \wedge \forall \mathsf{i}. \mathcal{A}^n(\mathsf{b}_\mathsf{i})\right) && \text{(Att-F)} \\
&\quad \vee \dots && \text{(etc. for each } f\text{)}
\end{aligned}$$

Intuitively, $\mathcal{A}^{n+1}(\mathsf{b})$ corresponds to bytestrings that can be computed using at most n nested cryptographic function calls. As a sanity check, we prove in F* that this definition of $\mathcal{A}(\square)$ indeed obeys the rules defined in Figure 2.11.

Monotonicity. The attacker knowledge predicate is monotonic when the trace grows, the attacker knows more bytestrings, because new messages are sent on the network or new compromise happen. This property is

[60]: Kleene (1952), *Introduction to Metamathematics*

$$\textsc{Att-Later}\ \frac{\tau_1 \vdash \mathcal{A}(\mathsf{b}) \qquad \tau_1 \subseteq \tau_2}{\tau_2 \vdash \mathcal{A}(\mathsf{b})}$$

Figure 2.13: Monotonicity lemma for the attacker knowledge

stated in Figure 2.13, and is easily proved from the definition of the attacker knowledge (Figure 2.11).

Proving with attacker knowledge. In the next section, we will define an over-approximation of the attacker knowledge using the concept of *labels*. It will be useful in confidentiality proofs, where we typically want to prove that if the attacker knows some secret key, then they must have performed some compromise to obtain it. This type of property can soundly be proved using an over-approximation of the attacker knowledge.

2.2.6 Security labels

Security labels are the main workhorse of DY* to reason on the secrecy of bytestrings. For example, labels can express properties such as "this key is a secret of Alice", which means that if the attacker knows this key, then they must have compromised Alice.

Overview

We first give a high-level overview of labels before diving into their precise semantics.

Labels as an over-approximation. Every bytestring is associated with a security label which encodes an over-approximation of compromises that may lead the attacker to know this bytestring. In other words, if the attacker knows some bytestring b (with respect to trace τ), the label of b will guarantee that some compromises must have happened (in trace τ), that is, the attacker must have compromised some set of principals.

Not just set of principals. Given this description, one might think that the label of b is simply a set, namely the smallest set of principals to compromise in order to compute the bytestring b. In reality, labels are more complex than sets of principals: they are predicates on set of principals, saying whether compromising this set of principals is enough to learn the secret with this label.

As an example why it is not the case, consider a bytestring b which mixes (e.g. using a KDF) a shared secret between Alice and Bob with a shared secret between Bob and Charlie. By construction, only Bob knows the secret b. For the attacker to learn the secret b, the attacker could of course compromise Bob, but could also compromise both Alice and Charlie. This means that although the secret b is only known by Bob, it is not as secret as a key generated by Bob himself.

Comparing labels. Some secret bytestrings are less secret than others. For example, if Alice and Bob share a secret using a Diffie-Hellman computation, then the shared secret is less secret than Alice's Diffie-Hellman private key, because the latter can be used to compute the former. This notion of "less secret" is lifted to labels, using a relation dubbed *can flow*, where less secret labels flow to more (or equally) secret labels. This notion of "less secret" depend on the compromises that happened: if a key has been compromised by the attacker, from the viewpoint of the attacker this key is morally public, hence is effectively "less secret" than public bytestrings. In other words, after compromising

the key, its label flows to the public label. We will see in the rest of this section that labels have a lattice structure.

Not just compromises. In the original DY* (hence in this section), labels could only talk about the existence of compromises in the trace. In §3.2 we will generalize labels to talk about any kind of events, as well as temporal relations between these events, etc. Hence, the correct and general intuition on labels is the following: the label of a bytestring b encodes an over-approximation of the *traces* where the attacker knows b. More formally, the attacker knowledge of b is a trace predicate (that is, $\tau \mapsto (\tau \vdash \mathcal{A}(\mathsf{b}))$), and the label of b encodes a trace predicate L which is weaker than attacker knowledge, that is:

$$\forall \tau.\, (\tau \vdash \mathcal{A}(\mathsf{b})) \implies L(\tau)$$

(this is a simplified version of the Attacker Knowledge Theorem, that we precisely describe in §2.2.10). The original DY* presents the restriction that labels can only encode trace predicates L that express the existence of compromise events in the trace; we will remove this restriction in §3.2.

Labels as a restriction. We previously said that the label of a bytestring b encodes an over-approximation of the events that led the attacker to know b. For the label to effectively be an over-approximation, it means that label of b will restrict how honest participants may use b. Equipped with the notion of labels being "less secret" than others, we now hint at what are these restrictions, which we will properly formalize later with the *bytes invariant* (§2.2.8). The restrictions will ensure that if the attacker knows b (with respect to a trace), then the label of b must be less secret than (hence equivalent to) the public label (with respect to this trace), which in turn, by the semantics of labels, will imply that some events must have happened in the trace.

When a participant sends a bytestring b on the network, they will be allowed to do so only when the label of b is less secret than (hence equivalent to) the public label; this forbids sending e.g. secret keys on the network, thereby making them known to the attacker.

When a participant encrypts some data with a key, they will be allowed to do so only when the data's label is less secret than the key's label; this ensures that if the attacker knows the ciphertext (e.g. because it was sent on the network) and knows the key (e.g. via compromise), the attacker will not learn more secrets by decrypting the ciphertext. Speaking with the language of labels, when the attacker knows the key, the key's label must be equivalent to the public label, hence because the data's label is less secret than the key's label, the data's label must also be equivalent to the public label. Therefore, this restriction preserves the over-approximation property of labels when the attacker e.g. compromises the encryption key.

When a participant stores some bytestring b in their state, they will be allowed to do so only when the label of b is less secret than the label corresponding to the state they are storing, this ensures that when compromising this state, the label corresponding to this state will become equivalent to the public label, hence the label of b will also become equivalent to the public label, therefore preserving the over-approximation property of labels.

Semantics

We now formally describe the semantics of labels, which have the structure of a trace-dependent bounded lattice. We will not yet describe how labels are implemented in F*, and leave this task for §3.2.

Notation. When the label ℓ_1 flows to the label ℓ_2 with respect to the trace τ, that is, ℓ_1 is "less secret" than ℓ_2, we write

$$\tau \vdash \ell_1 \gtrsim \ell_2$$

We may leave τ implicit and write

$$\ell_1 \gtrsim \ell_2$$

The semantics of labels are formally written in Figure 2.14 and discussed below.

Public. Labels have a maximum element, the public label, noted $\top$ (see Flow-Public), which represents all the bytestrings that may be known by the attacker; this will be formally proven in the Attacker Knowledge Theorem (§2.2.10).

Secret. Labels have a minimum element, the secret label, noted $\bot$ (see Flow-Secret), which represents bytestrings known by nobody: no amount of compromise may reveal a secret bytestring to the attacker. Although unrealistic, this label is sometimes useful to do proofs cleverly (e.g. using it as a neutral element for $\sqcup$ that we introduce in a few paragraphs).

Compromised principals. When a principal is compromised (for a given trace), then the label corresponding to this principal flows to public (Flow-Compromise-Elim). The converse is also true, if we know that the label corresponding to a principal flows to public, then we deduce that this principal is compromised (Flow-Compromise-Intro).

We also have labels for principals and specific state identifiers, but as discussed in §2.2.4 we omit state identifiers for simplicity.

Meet. Labels can be intersected: this is useful for example when mixing keys with HKDF.Extract, the resulting key label will be the intersection of the input key labels. The intersection (or "meet") of two labels, noted $\ell_1 \sqcap \ell_2$, follows the standard of greatest lower bound equivalence (Flow-Meet-Eq-Elim and Flow-Meet-Eq-Intro).

Join. Labels can be united: for example when two Diffie-Hellman keyshares are combined, the Diffie-Hellman shared secret label is the union of the private key labels. The union (or "join") of two labels, noted $\ell_1 \sqcup \ell_2$, follows the standard least upper bound equivalence (Flow-Join-Eq-Elim and Flow-Join-Eq-Intro).

The DY* labels further obey a non-conventional rule in the world of lattices, Flow-Join-Public, which states that if the union of two labels flows to public, then one of the two labels must flow to public. This is useful in combination with Flow-Compromise-Intro: if we know that $A \sqcup B \gtrsim \top$ we can deduce $A \gtrsim \top$ or $B \gtrsim \top$ (by Flow-Join-Public), hence $\lightning A$ or $\lightning B$ (by Flow-Compromise-Intro).

Monotonicity. The label lattice depends on the trace, because as the trace grows, new participants will be compromised, hence more labels will

Flow-Refl
$$\frac{}{\ell \gtrsim \ell}$$

Flow-Trans
$$\frac{\ell_1 \gtrsim \ell_2 \qquad \ell_2 \gtrsim \ell_3}{\ell_1 \gtrsim \ell_3}$$

Flow-Later
$$\frac{\tau_1 \vdash \ell_1 \gtrsim \ell_2 \qquad \tau_1 \subseteq \tau_2}{\tau_2 \vdash \ell_1 \gtrsim \ell_2}$$

Flow-Public
$$\frac{}{\top \gtrsim \ell}$$

Flow-Secret
$$\frac{}{\ell \gtrsim \bot}$$

Flow-Compromise-Elim
$$\frac{\tau \vdash \lightning P}{\tau \vdash P \gtrsim \top}$$

Flow-Compromise-Intro
$$\frac{\tau \vdash P \gtrsim \top}{\tau \vdash \lightning P}$$

Flow-Meet-Eq-Elim
$$\frac{i \in \{1,2\} \qquad \ell_1 \sqcap \ell_2 \gtrsim \ell_3}{\ell_i \gtrsim \ell_3}$$

Flow-Meet-Eq-Intro
$$\frac{\ell_1 \gtrsim \ell_2 \qquad \ell_1 \gtrsim \ell_3}{\ell_1 \sqcap \ell_2 \gtrsim \ell_3}$$

Flow-Join-Eq-Elim
$$\frac{i \in \{2,3\} \qquad \ell_1 \gtrsim \ell_2 \sqcup \ell_3}{\ell_1 \gtrsim \ell_i}$$

Flow-Join-Eq-Intro
$$\frac{\ell_1 \gtrsim \ell_2 \qquad \ell_1 \gtrsim \ell_3}{\ell_1 \gtrsim \ell_2 \sqcup \ell_3}$$

Flow-Join-Public
$$\frac{\ell_1 \sqcup \ell_2 \gtrsim \top}{\ell_1 \gtrsim \top \vee \ell_2 \gtrsim \top}$$

Figure 2.14: Inference rules for labels. The $\tau \vdash$ is left implicit when there is only one trace, i.e. everywhere except in the Flow-Later rule or in the Flow-Compromise-$\cdots$ rules to emphasize that compromise depends on the trace: the label lattice is dynamic because of the Flow-Compromise-$\cdots$ rules.

flow to $\top$. This dependency on the trace is tamed by the rule Flow-Later: the relation $\gtrsim$ is preserved by extending the trace.

Flowing to the public label. In Figure 2.14, we may notice a special citizen in the semantics: $\ell \gtrsim \top$.

Indeed, if ℓ is a label corresponding to a principal, $\ell \gtrsim \top$ is equivalent to the principal being compromised (via Flow-Compromise-Intro and Flow-Compromise-Elim).

$$P \gtrsim \top \iff \lightning P$$

If ℓ is a meet $(\ell_1 \sqcap \ell_2)$ then $\ell \gtrsim \top$ is equivalent to $\ell_1 \gtrsim \top$ and $\ell_2 \gtrsim \top$ (by setting $\ell_3 = \top$ in Flow-Meet-Eq-Intro and Flow-Meet-Eq-Elim).

$$\ell_1 \sqcap \ell_2 \gtrsim \top \iff (\ell_1 \gtrsim \top \wedge \ell_2 \gtrsim \top)$$

Finally, if ℓ is a join $(\ell_1 \sqcup \ell_2)$ then $\ell \gtrsim \top$ is equivalent to $\ell_1 \gtrsim \top$ or $\ell_2 \gtrsim \top$ (the forward implication is simply Flow-Join-Public, the backward implication is proved by using Flow-Join-Eq-Elim with $\ell_1 \sqcup \ell_2 \gtrsim \ell_1 \sqcup \ell_2$ and using Flow-Refl, Flow-Trans and Flow-Public).

$$\ell_1 \sqcup \ell_2 \gtrsim \top \iff (\ell_1 \gtrsim \top \vee \ell_2 \gtrsim \top)$$

Therefore, $\tau \vdash \ell \gtrsim \top$ morally means that ℓ is "bad":[6] for this to happen, enough compromises must have happened it τ. In the Attacker Knowledge Theorem (§2.2.10), we will prove something along the lines of: when a bytestring b has label ℓ, if the attacker knows b (i.e. $\tau \vdash \mathcal{A}(\mathsf{b})$) then ℓ must flow to the public label (i.e. $\tau \vdash \ell \gtrsim \top$).

6: this notion of labels being "bad" will be discussed more thoroughly in §3.2

Bytestrings and labels

We compute the label ℓ corresponding to a bytestring b by induction on the bytestring term, and write it $\mathcal{L}(\mathsf{b})$. We discuss below the definition of $\mathcal{L}(\mathsf{b})$ for various cryptographic primitives.

Label of fresh random bytestrings. When a participant generates a fresh random bytestring, they decide what is its label. For example, when Alice generates her private long-term keys, she will choose the label Alice, but when she generates a shared symmetric key with Bob, she will choose the label Alice $\sqcup$ Bob, thereby making it safe to encrypt with Bob's private keys.

Label of public keys. When a participant transforms their private key into a public key, the public key label will be, unsurprisingly, public. This allows them to safely broadcast their public key on the network.

Label of Diffie-Hellman shared secrets. When a participant computes a Diffie-Hellman shared secret, the label of the resulting key will be the join of the two private key labels. This models the fact that if the attacker knows the shared secret, then they must know one of the two private keys.

Label of ciphertexts. When a participant encrypts a plaintext using public-key encryption or a private-key (i.e. symmetric) encryption, the label of the resulting ciphertext is public: it means it is safe to reveal to the attacker, e.g. by sending it on the network. This models the fact that the ciphertext leaks no information about the plaintext (except its length) when the key is not known by the attacker.[7]

7: When the key is known by the attacker, thanks to the restriction that one must only encrypt data less secret than the key (discussed previously and in §2.2.8), the plaintext will also be considered safe to reveal to the attacker.

Label of signatures and MACs. When a participant authenticates a bytestring via signature or Message Authentication Code (MAC), the signature or tag label has the same label as the authenticated bytestring.

Indeed, standard cryptographic assumptions on these primitives guarantee authenticity (via the EUF-CMA security assumption), but do not provide any guarantee of the secrecy of the authenticated bytestring.

Label of derived keys. DY* models Key Derivation Functions (KDF) following the interface of HKDF, where protocols sometimes use separately the internal functions HKDF.Extract and HKDF.Expand. First, HKDF.Extract transforms a non-uniform key (such as a Diffie-Hellman shared secret) into a uniform key, or combines two keys into one. Then, HKDF.Expand derives keys from the extracted key and an "info" bytestring. We can derive several keys by using HKDF.Expand with several distinct "info" bytestrings.

The output label of HKDF.Extract is therefore the meet of its two input key labels, modeling the fact that the attacker must know both input keys to compute the extracted key.

The output label of HKDF.Expand in practice depends on the "info" bytestring, in a protocol-specific way. Indeed, we may want to do things with the derived key that we are not allowed to do with the extracted key, such as encrypting it to new people, storing it in some intermediate state, or even publishing it on the network. Therefore, DY* expects the user to provide a function that computes the derived key label from the input key label, input key usage (see §2.2.7) and "info" bytestring.

Label of concatenation. When a user concatenates two bytestring, the label of the concatenation is the meet of the two concatenated bytestrings. Indeed, the attacker must know the two concatenated bytestrings to know their concatenation. It may not seem intuitive, because if the attacker only knows one bytestring, then they still have partial knowledge of the concatenated bytestring, hence the label should be a join? This would be unsound: if the label were a join, the concatenation of a secret key and a public bytestring would be public, hence safe to send on the network. Using a meet ensures that each of the concatenated bytestrings must be public for the concatenation to be public, hence safe to send on the network.

2.2.7 Key usages

Cryptographic protocols security often rely on the fact that keys are not used for too many purposes. We first give some examples to explain why, then show how DY* enforces this when writing security proofs.

Keys across protocols. For example, if we are proving the security of TLS 1.3, we may implicitly assume that the long-term signature key (or certificate) is not used in other protocols, for example SSH.[8] This is because if the same signature key were used in both protocols, there could be a cross-protocol attack (or at least, a first step toward such an attack) that first obtains a signature from SSH then injects it in TLS, thereby morally doing a signature forgery in TLS, something we assume is impossible when proving the security of TLS.

8: Interestingly, TLS 1.2 and TLS 1.3 may be deployed with the same long-term signature key. For this reason, TLS 1.3 signature inputs were engineered to be unambiguous with TLS 1.2 signature inputs. We will prove this in Chapter 4.

Keys across primitives. Moving the discussion from protocols to primitives, their security properties also assume that their keys are not used with other primitives. For example, signatures are assumed to be unforgeable as long as the signature key is (1) kept secret and (2) only used to compute signatures. If the signature key is used for something else,

for example to encrypt some data, then the standard unforgeability assumption (EUF-CMA) does not apply anymore, hence cannot guarantee the unforgeability of signatures.

Digression: in game-hopping computational proofs. In game-hopping computational security proofs, we may write security assumptions of primitives as follows (using a formalism similar to [61]): an attacker (with reasonable computing power) cannot distinguish the games G_1 and G_2, that is, for all attacker $\mathcal{A}$, we have $\Pr[\mathcal{A} \circ G_1 \Rightarrow 1] \approx \Pr[\mathcal{A} \circ G_2 \Rightarrow 1]$. To apply this assumption on a protocol P, we decompose $P = P' \circ G_1$, so that an attacker $\mathcal{A}$ on P induces the attacker $\mathcal{A} \circ P'$ on G_1, which cannot distinguish between G_1 and G_2 by security assumption, therefore allowing us to perform a game hop.

[61]: Rosulek (2021), *The Joy of Cryptography*

The game G_1 will typically provide an oracle that samples a fresh random key, and provide oracles to use this key in the allowed ways. When proving that P can be decomposed in $P' \circ G_1$, we implicitly prove that the key in G_1 is distinct (with overwhelming probability) from every bytestring appearing in P'.

Digression: distinct keys across primitives in the symbolic model. In the symbolic model, unlike the computational model, it is not a problem if a key is used with different cryptographic primitives. Therefore, symbolic tools such as ProVerif [41] or Tamarin [42] do not verify that keys are not used with several cryptographic primitives.[9] In DY*, we decide to perform this check. Indeed, even if it is not useful to prove symbolic security properties, DY* is more than a symbolic security framework, it is a tool to analyze cryptographic protocols. Therefore, if DY* users cannot prove that different cryptographic use distinct keys, it hints toward a design flaw in the protocol under scrutiny and should not be ignored.

9: Users of these tools could however verify this as an additional property by adding custom rules (Tamarin) or processes (ProVerif).

Proving pairwise distinctness in DY*. We want to prove many pairs of keys to be distinct: keys used with different cryptographic primitives, and keys used with different protocols. We now explain how we prove this in DY*, without adding an unreasonable burden on users. Indeed, if we were to do it naively and prove something for every such pair of keys, the proof effort would be prohibitive. The intuition behind our proof methodology is the following: we want to prove a property similar to injectivity of functions, which states that two outputs are distinct when inputs are distinct, that is, $\forall x, y. f(x) = f(y) \implies x = y$. If we were to manually check this property, we would need a quadratic number of checks, which is prohibitive. However, injectivity is equivalent to the existence of an inverse on the left, that is, $\exists g. \forall x. g(f(x)) = x$. If we exhibit such a g, then checking injectivity requires a linear number of checks.

Key usage. This is what we do in DY*: every bytestring (hence key) b is associated with a *usage*, written $\mathcal{U}$(b) or get_usage b. To prove that two bytestring are distinct, it now suffices to prove that their usages are distinct. The usage type (depicted in Figure 2.15) provides two types of information: the cryptographic primitive corresponding to this key, and a string describing the protocol where this key is used. A special usage, NoUsage corresponds to bytestrings that are not keys of cryptographic primitives.

```
type usage =
  | SignatureKey: string → usage
  | AeadKey: string → usage
  | KdfKey: string → usage
  ...
  | NoUsage: usage

val get_usage: bytes → usage
```

Figure 2.15: Type in F* for key usages.

Key usages as a restriction. The usage of a bytestring will restrict how honest participants can use them in a cryptographic protocol. For example, bytestrings with usage SignatureKey ... may be used as a key for signatures, but cannot be used as a key for encryption. Similarly, bytestrings with

usage NoUsage cannot be used as a key for any cryptographic primitive. These restrictions will be properly formalized in the *bytes invariant* (§2.2.8).

Computing the key usage. We compute the usage of a bytestring by induction on the bytestring term, we show the corresponding F* code in Figure 2.16.

When a participant generates a fresh random bytestring, they decide what is its usage. Then, get_usage simply returns this chosen usage.

The output usage of a Key Derivation Function (KDF) depends on several things. First, it depends on the protocol being analyzed, hence, we rely on the user to provide a function kdf_expand_usage that computes this usage. Second, it depends on the kind of key used for this KDF (this is the string in KdfKey in Figure 2.15). Third, it depends on the info bytestring provided to the KDF; indeed a common usage of KDFs is to derive multiple keys from a single KDF pseudo-random key (prk) by using multiple info bytestrings, which may have different usages.

```
let rec get_usage (b:bytes): usage =
  match b with
  | Rand ... usg ... → usg
  | KdfExpand prk info len →
    let prk_usage = get_usage prk in
    kdf_expand_usage prk_usage info
  // ...
  | _ → NoUsage
```

Figure 2.16: Computing key usages, in F*. The function kdf_expand_get_usage must be provided by the user.

2.2.8 The bytes invariant

In DY*, we prove security of protocols by proving that honest participants have a "hygienic" use of cryptography. This informal notion of "hygiene" is formally captured by the *bytes invariant*, an invariant on all the bytestrings that appear in a protocol execution, whether computed by honest participants or by the attacker. The bytes invariant will effectively restrict the use of cryptography by honest participants, therefore is designed as a careful trade-off between expressivity and security. Indeed, if it were too restrictive, in the extreme scenario, honest participants would not be allowed to do anything, hence all their behaviors would be safe (by quantification on the empty set); if it were too permissive, in the extreme scenario, honest participants would be allowed to do anything, including unsafe behaviors such as sending secret keys on the network.

Notation. When a bytestring b obeys the bytes invariant with respect to trace τ, we write $\tau \vdash \mathcal{B}(b)$. As with other DY* predicates and relations, the bytes invariant stays true when the trace grows (see Figure 2.17).

$$\frac{\tau_1 \vdash \mathcal{B}(b) \qquad \tau_1 \subseteq \tau_2}{\tau_2 \vdash \mathcal{B}(b)}$$

Figure 2.17: The "later" rule for bytes invariant.

Publishability. When a bytestring b obeys the bytes invariant ($\tau \vdash \mathcal{B}(b)$) and its label flows to (hence is equivalent to) the public label ($\tau \vdash \mathcal{L}(b) \gtrsim \top$), we say that b is *publishable*, and we write $\tau \vdash \mathcal{P}(b)$. Publishable bytestrings will be considered "safe to know" by the attacker. Indeed, in the Attacker Knowledge Theorem (§2.2.10) we will prove that the attacker only knows publishable bytestrings, or in other words that publishability is an over-approximation of the attacker knowledge. Furthermore, we will see in the *trace invariant* (§2.2.9) that honest participants are allowed to send publishable bytestrings on the network; this means that publishability is a "tight" over-approximation of the attacker knowledge, because publishable bytestrings may be sent on the network, thereby making them known to the attacker.

Preservation of publishability. Because publishability is a "tight" over-approximation of the attacker knowledge, and because the attacker knowledge predicate is preserved by computing cryptographic functions (Att-F in Figure 2.11), so should the publishability predicate. Therefore, a crucial design principle for the bytes invariant $\mathcal{B}(\square)$ and for the labeling

$$\frac{\forall i. \tau \vdash \mathcal{P}(b_i)}{\tau \vdash \mathcal{P}(f(b_1, \ldots, b_n))}$$

Figure 2.18: Every cryptographic function must preserve publishability.

function $\mathcal{L}(\square)$ is the following: every cryptographic function must preserve publishability. In other words, computing a cryptographic function on publishable bytestrings must output a publishable bytestring (see Figure 2.18). This design principle ensures that the bytes invariant $\mathcal{B}(\square)$ allows the attacker to compute cryptographic functions without restrictions (unlike honest participants, who must obey "hygiene" rules).

Design principles. Although each cryptographic function enjoys a tailor-made bytes invariant, the bytes invariant follows some general design principles. We just saw one of the guidelines: the bytes invariant must ensure that every cryptographic function preserves publishability, which allows the attacker to compute any cryptographic function. The other guidelines will allow honest participants to compute cryptographic functions (under some conditions).

Principle: induction. The bytes invariant has an inductive nature: to prove that the output of a cryptographic function obeys the bytes invariant, we will need to prove (among other things) that each one of its inputs obey the bytes invariant (see Figure 2.19).

$$\frac{\forall i.\tau \vdash \mathcal{B}(\mathsf{b_i}) \qquad \dots}{\tau \vdash \mathcal{B}(\mathsf{f}(\mathsf{b_1}, \dots, \mathsf{b_n}))}$$

Figure 2.19: The bytes invariant has the rough shape of an induction principle.

Principle: correct key usage. In §2.2.7, we saw that every bytestring is associated to a *usage* that describes whether it is a key, for what cryptographic function and what protocol. This entails a design principle for the bytes invariant: when we use a cryptographic function with a key, its output will obey the bytes invariant only when the key has the correct usage (see Figure 2.20). We will later show in §3.4 that we can slightly weaken this constraint, but in this section we describe the original DY*, which follows this design principle.

$$\frac{\mathcal{U}(\mathsf{key}) = \mathsf{FKey}\ \_ \qquad \dots}{\tau \vdash \mathcal{B}(\mathsf{f}(\mathsf{key}, \dots))}$$

Figure 2.20: The bytes invariant ensures that we use keys with correct usage.

Principle: confidentiality. In §2.2.6 we saw that labels encode an over-approximation of the events that must have happened for an attacker to know some bytestring, and that therefore labels must restrict how a bytestring is used by honest participants, to effectively stay an over-approximation. The bytes invariant treats restrictions related to using cryptographic functions, specifically encryption functions; other restrictions will be handled by the *trace invariant* (§2.2.9).

$$\frac{\tau \vdash \mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key}) \qquad \dots}{\tau \vdash \mathcal{B}(\mathsf{enc}(\mathsf{key}, \mathsf{ptxt}, \dots))}$$

Figure 2.21: The bytes invariant ensures that plaintexts are less secret than keys used to encrypt them.

The bytes invariant enforces that honest participants can only encrypt plaintexts (thereafter ptxt) that are less secret than keys (thereafter key) used to encrypt them, that is, $\mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key})$. To understand this restriction, recall that the decryption function must preserve publishability (see Figure 2.18). It means that the plaintext must be publishable ($\mathcal{P}(\mathsf{ptxt})$) when the key is publishable ($\mathcal{P}(\mathsf{key})$), or putting the bytes invariant aside and only dealing with labels, $\mathcal{L}(\mathsf{ptxt}) \gtrsim \top$ when $\mathcal{L}(\mathsf{key}) \gtrsim \top$. To prove this implication, it is sufficient to require $\mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key})$, which concludes by transitivity. We will see that it is also necessary that $\mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key})$, when we will understand better the relation $\gtrsim$ in §3.2.

To give an example of this design principle, imagine Alice holds a private key sk (say, a signature key) known only by her (hence $\mathcal{L}(\mathsf{sk}) = \mathsf{Alice}$), and a shared symmetric encryption key k known only by her and Bob (hence $\mathcal{L}(\mathsf{k}) = \mathsf{Alice} \sqcup \mathsf{Bob}$). Suppose she encrypts her private key sk with the shared symmetric key k and sends the ciphertext on the network. Now, there is a problem: if the attacker compromises Bob and obtains k, they can decrypt the ciphertext to obtain sk. However, this is not allowed by the label of sk ($\mathcal{L}(\mathsf{sk}) = \mathsf{Alice}$), which says that if the attacker knows sk then they must have compromised Alice; the (allegedly) over-approximation encoded by $\mathcal{L}(\mathsf{sk})$ is not an over-approximation anymore. The root cause

of this problem is that we encrypted sk with k without making sure that $\mathcal{L}(\mathsf{sk}) \gtrsim \mathcal{L}(\mathsf{k})$; indeed, in general we don't have $\tau \vdash \mathsf{Alice} \gtrsim \mathsf{Alice} \sqcup \mathsf{Bob}$ (unless Alice is compromised hence $\tau \vdash \mathsf{Alice} \gtrsim \top$, see Flow-Compromise-Elim in Figure 2.14).

Principle: authenticity. For cryptographic functions that authenticate, such as signatures, message authentication codes (MAC), or authenticated encryption (whether symmetric or asymmetric), DY* will require users to provide a predicate such that honest participants only authenticate data that satisfy this predicate. In Figure 2.22 we give a more concrete example for signatures: the user provides SigPre, then the bytes invariant only allows to sign data that satisfy SigPre. In return, when a signature verification succeeds, we are able to deduce that either (1) the signature was computed by an honest participant, hence the signed data satisfies SigPre, or (2) the signature was computed by the attacker, hence the signature key must be publishable.

$$\frac{\tau \vdash \mathsf{SigPre}(\mathcal{U}(\mathsf{sk}), \mathsf{msg}) \qquad \dots}{\tau \vdash \mathcal{B}(\mathsf{sign}(\mathsf{sk}, \mathsf{msg}, \dots))}$$

$$\frac{\mathsf{verify}(\mathsf{vk}, \mathsf{msg}, \mathsf{sig}) \qquad \dots}{\tau \vdash \mathsf{SigPre}(\mathcal{U}(\mathsf{sk}), \mathsf{msg}) \vee \mathcal{P}(\mathsf{sk})}$$

Figure 2.22: The bytes invariant ensures that participants only sign messages that satisfy some (user-provided) predicate.

These authenticity predicates may use in their logic labels ($\mathcal{L}(\square)$) and the secrecy relation between them ($\gtrsim$). Therefore, authenticity predicates can be used to transfer information from a participant to another. For example, assume Alice generates a fresh random bytestring b with some label ℓ, then sign it. Later, Bob obtains the random bytestring b and verifies the corresponding signature. We can use the signature predicate to transfer the information on $\mathcal{L}(\mathsf{b})$ from Alice to Bob, by defining $\mathsf{SigPre}(\_, b) := \mathcal{L}(\mathsf{b}) = \ell$: this predicate holds when Alice signs b, hence Bob will deduce that either $\mathcal{L}(\mathsf{b}) = \ell$, or that Alice's signature key is compromised.

Morally, the signature predicate associates a meaning to the signature: when Alice signs a message msg using her signature key sk, she means "I, Alice, thereby attests that $\mathsf{SigPre}(\mathcal{U}(\mathsf{sk}), \mathsf{msg})$".

Example: authenticated symmetric encryption. The cryptographic functions for Authenticated Encryption with Associated Data (AEAD), namely encryption (aead_enc) and decryption (aead_dec), capture all the design principles aforementioned; we show them in their full glory (only slightly simplified) in Figure 2.23.

$$\frac{\begin{array}{c}\mathcal{B}(\mathsf{key}) \qquad \mathcal{P}(\mathsf{nonce}) \qquad \mathcal{B}(\mathsf{ptxt}) \qquad \mathcal{P}(\mathsf{ad}) \qquad \mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key}) \\ \mathcal{U}(\mathsf{key}) = \mathsf{AeadKey}\ \_ \qquad \mathsf{AeadPre}(\mathcal{U}(\mathsf{key}), \mathsf{nonce}, \mathsf{ptxt}, \mathsf{ad})\end{array}}{\mathcal{P}(\mathsf{aead\_enc}(\mathsf{key}, \mathsf{nonce}, \mathsf{ptxt}, \mathsf{ad}))}$$

$$\frac{\begin{array}{c}\mathcal{B}(\mathsf{key}) \qquad \mathcal{B}(\mathsf{nonce}) \\ \mathcal{B}(\mathsf{ciphertext}) \qquad \mathcal{B}(\mathsf{ad}) \qquad \mathsf{ptxt} = \mathsf{aead\_dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{ciphertext}, \mathsf{ad})\end{array}}{\mathcal{B}(\mathsf{ptxt}) \wedge \mathcal{L}(\mathsf{ptxt}) \gtrsim \mathcal{L}(\mathsf{key}) \wedge (\mathsf{AeadPre}(\mathcal{U}(\mathsf{key}), \mathsf{nonce}, \mathsf{ptxt}, \mathsf{ad}) \vee \mathcal{P}(\mathsf{key}))}$$

Figure 2.23: Bytes invariant rules for AEAD encryption and decryption.

The requirements for encryption (aead_enc) mainly correspond to the design principles we discussed above, there is however a slight twist: we require nonce and ad to be publishable ($\mathcal{P}(\mathsf{nonce})$ and $\mathcal{P}(\mathsf{ad})$), this is stronger than what we would expect from the induction design principle (which only requires $\mathcal{B}(\mathsf{nonce})$ and $\mathcal{B}(\mathsf{ad})$). This is because the standard security assumptions on AEAD do not guarantee the confidentiality of nonce nor ad, hence we require them to be publishable. As a result, it would be safe to give to the attacker a function extract_ad such that $\mathsf{extract\_ad}(\mathsf{aead\_enc}(\mathsf{key}, \mathsf{nonce}, \mathsf{ptxt}, \mathsf{ad})) = \mathsf{ad}$, although we don't model such a function in practice.

Thanks to the requirements for encryption (aead_enc), we can deduce useful information on the plaintext (ptxt) when decrypting (with aead_dec), for example that the AEAD predicate must hold unless key is publishable (hence may be known by the attacker).

Implementing the bytes invariant in F*. The bytes invariant is defined by induction on the bytestring term. We show the excerpt of its implementation related to AEAD (only slightly simplified) in Figure 2.24. Its implementation is a straightforward combination of the encryption bytes invariant conditions (Figure 2.23) and preservation of publishability design principle (Figure 2.18).

```
let rec bytes_invariant (tr:trace) (b:bytes): prop =
 match b with
 ...
 | AeadEnc key nonce ptxt ad →
   bytes_invariant tr key ∧
   bytes_invariant tr nonce ∧ (get_label nonce) `can_flow tr` public ∧
   bytes_invariant tr ptxt ∧
   bytes_invariant tr ad ∧ (get_label ad) `can_flow tr` public ∧
   (
    (
     // Honest case
     AeadKey? (get_usage key) ∧
     aead_pred tr (get_usage key) key nonce ptxt ad ∧
     (get_label ptxt) `can_flow tr` (get_label key)
    ) ∨ (
     // Attacker case
     (get_label key) `can_flow tr` public ∧
     (get_label ptxt) `can_flow tr` public
    )
   )
 ...
```

Figure 2.24: Implementation of the bytes invariant in F*.

2.2.9 The trace invariant

We saw in §2.2.9 that the *bytes invariant* restricts how participants may use cryptographic functions. We now introduce the *trace invariant* which restricts how participants may perform impure actions, such as sending messages on the network, storing state, or logging protocol events. Recall that in DY*, protocols are specified by exposing effectful functions that implement protocol steps to the attacker. Then, by interacting with these functions, the attacker may be able to reach some set of traces. We will reason on the set of reachable traces as follows: DY* users will prove that each effectful function they expose to the attacker preserves the trace invariant, thereby proving that all reachable traces satisfy the trace invariant.

Notation. When a trace τ satisfies the trace invariant, we write $\tau\checkmark$.

Sending messages. The trace invariant restricts what messages participants are allowed to send: indeed, it would for example be unsafe to send private keys on the network. Recall that publishable bytestrings (written $\mathcal{P}(\square)$) are considered "safe to know" by the attacker, hence they are safe to send on the network. Therefore, the trace invariant restricts participants to only send publishable messages on the network, as depicted in Figure 2.25.

$$\textsc{TraceInv-SendMsg}\quad \frac{\tau\checkmark \qquad \tau \vdash \mathcal{P}(\mathsf{msg})}{(\tau +\!\!+ \mathsf{SendMsg}(\mathsf{msg}))\checkmark}$$

Figure 2.25: Trace invariant when sending a message.

Storing state. The trace invariant also restricts participants to only store states that satisfy a user-provided state invariant (StatePre). The reason is twofold. First, this provides an additional proof technique for the user: when they will retrieve some state that was stored previously, they know the state did obey the state invariant when they stored it; this may provide crucial information to conduct the proof, such as labeling information on the data stored within the state. Second, DY* puts restrictions on the state invariant, and requires that

$$\mathsf{StatePre}(\mathsf{P},\mathsf{sid},\mathsf{msg}) \implies \mathscr{B}(\mathsf{msg}) \wedge \mathscr{L}(\mathsf{msg}) \gtrsim P$$

This ensures soundness of publishability, because it implies that if the attacker obtains msg by compromise then msg is publishable, or formally

$$\lightning \mathsf{P} \wedge \mathsf{StatePre}(\mathsf{P},\mathsf{sid},\mathsf{msg}) \implies \mathscr{P}(\mathsf{msg})$$

(by definition of $\mathscr{P}(\square)$ and by Flow-Compromise-Elim in Figure 2.14).

TraceInv-SetState

$$\frac{\tau\checkmark \qquad \tau \vdash \mathsf{StatePre}(\mathsf{P},\mathsf{sid},\mathsf{msg})}{(\tau +\!\!+ \mathsf{SetState}(\mathsf{P},\mathsf{sid},\mathsf{msg}))\checkmark}$$

Figure 2.26: Trace invariant when storing state.

Logging protocol events. Similarly to storing state, the trace invariant restricts participants to only log protocol events that satisfy a user-provided event invariant (EventPre). This will be useful when writing security theorems: for example, if we know that the trace obeys the trace invariant ($\tau\checkmark$) and that some protocol event was logged (e.g. Alice successfully responded to Bob), we deduce that the event invariant (EventPre) must hold, hence further implies something useful (e.g. that Bob must have initiated conversation with Alice before). Without the event invariant, when we know some protocol event was logged, we could not deduce anything from this information.

TraceInv-CustomEvent

$$\frac{\tau\checkmark \qquad \tau \vdash \mathsf{EventPre}(\mathsf{P},\mathsf{tag},\mathsf{msg})}{(\tau +\!\!+ \mathsf{CustomEvent}(\mathsf{P},\mathsf{tag},\mathsf{msg}))\checkmark}$$

Figure 2.27: Trace invariant when logging custom protocol event.

Generating randomness and compromise. Finally, the trace invariant allows anyone (honest participants or the attacker) to generate fresh randomness, and allows the attacker to compromise anyone, without any restriction.

TraceInv-Other

$$\frac{\tau\checkmark \qquad \mathsf{ev} \in \{\mathsf{RandGen}(\_), \mathsf{Compromise}(\_)\}}{(\tau +\!\!+ \mathsf{ev})\checkmark}$$

Figure 2.28: Trace invariant when generating fresh randomness or compromising principals.

2.2.10 Attacker Knowledge Theorem

We now finally have all the ingredients to state and prove the Attacker Knowledge Theorem that was advertised throughout the past sections.

Theorem statement. The Attacker Knowledge Theorem states that for all traces that satisfy the trace invariant, if the attacker knows a bytestring (with respect to this trace), then this bytestring is publishable (with respect to this trace). In other words, publishability is a correct over-approximation of the attacker knowledge. We write the theorem statement using mathematical notations in Figure 2.29.

AttackerKnowledgeTheorem

$$\frac{\tau\checkmark \qquad \tau \vdash \mathscr{A}(\mathsf{b})}{\tau \vdash \mathscr{P}(\mathsf{b})}$$

Figure 2.29: The Attacker Knowledge Theorem.

Using the theorem. The Attacker Knowledge Theorem is crucial in confidentiality theorems, which are generally of the form: "for all reachable traces, if the attacker knows the bytestring b, then they must have compromised some participant". If the trace is reachable, it obeys the trace invariant, we then use the Attacker Knowledge Theorem to deduce that b

is publishable, hence that $\mathcal{L}(\mathsf{b}) \gtrsim \top$, meaning that if we have information on the label of b, we can deduce that some compromises must have happened.

Proving the theorem. The proof works by induction on the attacker knowledge, as defined in Figure 2.11. In the case Att-Sent, the trace invariant proves that sent messages are publishable (TraceInv-SendMsg in Figure 2.25). In the case of Att-Compromise, we already showed using the state invariant (presented in Figure 2.26) that the compromised state must be publishable. In the case of Att-F, we reason inductively using the fact that computing functions preserves publishability (Figure 2.18).

2.2.11 Discussion

We now discuss the DY* method toward symbolic security proofs. There is no particular flow within this section, the different paragraphs are mostly independent of each other.

Trusted Computing Base. We emphasize that most of what has been described in this section are only proof techniques, and not part of the security theorems we prove on cryptographic protocols. Indeed, security theorems talk about reachable traces, attacker knowledge, protocol events that were logged, and compromise. Note that the trace invariant, bytes invariant, labels, etc, do not appear in the theorem statement: they are only a proof technique, therefore they do not belong to the Trusted Computing Base.

Actually, this is not exactly true because security theorems are not of the form "for all reachable traces" but of the form "for all traces that satisfy the trace invariant". This is however safe, regardless of the precise definition of trace invariant, because each function exposed to the attacker preserves the trace invariant, hence every trace reachable by the attacker must satisfy the trace invariant. That way, we avoid having to specify precisely what is the set of reachable traces, without impacting the Trusted Computing Base.

About compromise. In DY*, the attacker can compromise any state stored by protocol participants. This is an opinionated choice: in ProVerif [41] or Tamarin [42], the protocol modeler must choose manually what states are allowed to be compromised by the attacker.

About using F*. Although DY* is entirely written in F*, it is not using F*-specific features: we could in theory implement in other proof assistants the same DY* approach toward symbolic cryptographic proofs.

Modularity of the proofs. The security proofs in DY* are modular: indeed, each protocol step is proved to preserve the trace invariant independently. It means that *a priori*, the time to verify a proof is proportional to the protocol size, meaning that DY* should scale well for large protocols, and should not suffer from exponential blow-up as the protocol size grows. We will confirm this intuition when proving the key agreement sub-protocol of Messaging Layer Security (MLS) in Chapter 6, which may be qualified as a "large protocol".

Lack of modularity of the invariants. However, note that every protocol step must be proved to satisfy the *same* trace invariant; as such, proving new protocol steps generally requires modifying the trace invariant

(thereby making it slightly more complex), and re-proving that every protocol step satisfies these new trace invariant. We therefore see that the trace invariant are defined monolithically, this results in a lack of modularity when performing security proofs. We will solve this problem in §3.1.

Protocol specification coding style. Different tools have different ways to specify protocols. ProVerif [41] relies on a process-calculus, where protocols are described as processes that may send or receive messages on the network, generate fresh randomness, compute cryptographic functions, etc. The processes then execute concurrently, in an interleaving chosen by the attacker. Tamarin [42] relies on multiset rewriting rules where protocols are described as state machines, that is, as a set of protocol steps that receive message from the network and retrieve some state, compute some cryptographic functions, then store some new state and send back messages on the network. The attacker may then execute the various protocol steps in the interleaving of its liking. DY* [43] is somewhat in-between the two styles: on the surface, the syntax reminds of ProVerif's process calculus, but deep inside, the execution model is more similar to Tamarin's state machines. Indeed, DY* allows the attacker to interleave computations around the execution of each protocol step, but not within one. For example, if a DY* protocol step were to send a message on the network, and afterward, in the same protocol step, receive a message from the network, then between the sending of the first message and the receiving of the second message, the attacker would not be able to perform computations or interleave protocol steps of other participants. Therefore, protocols in DY* must be specified in a style similar to Tamarin: first, receive a message from the network and retrieve some state, then compute cryptographic functions, finally store some state and send back messages on the network.

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Open question: link with concrete implementations. We prove a protocol specification secure in the symbolic model, this is great, but what does it tell us about its security in the real world? The original DY* paper [43] suggests we can substitute the symbolic cryptographic interface with concrete cryptographic functions to obtain an implementation we can execute, but how does the security theorems we prove with the symbolic interface translate to the corresponding implementation? We show in Figure 2.30 a degenerated protocol: we obtain from the attacker x, y and z, and leak a secret when hash x is equal to mac y z. In the symbolic model, a hash in never equal to a MAC, therefore this condition is always false and the protocol never leaks a secret. However, in the real world, it is easy to come up with such x, y and z when the MAC is HMAC, a commonly-used hash-based MAC. This degenerated protocol shows that we cannot prove a general result that lifts security theorems in the symbolic model to concrete implementations. However, could we characterize a class of "well-formed" protocols where such a theorem would hold, and such that real-world protocols are in general "well-formed"? Is it possible to do so without going all the way to computational soundness [62]?

```
let protocol () =
  let (x, y, z) = recv () in
  if hash x = mac y z then
    // leak secret
  else
    // do nothing
```

Figure 2.30: Example of a protocol secure in the symbolic model, but easily broken in the real world.

[62]: Cortier et al. (2011), *A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems*

No unification. Symbolic provers such as ProVerif [41] or Tamarin [42] crucially rely on *unification* during their search of a security proof. Note that this is not the case for DY*: nothing in this section relied on the use of unification. Even more, a prerequisite for unification is to have *variables* in the bytes term, which DY* do not have (see §2.2.3): for example we cannot write the term encrypt(key,x) and instantiate x afterward with some other term. This raises the following question: if the use unification seems

so crucial that two different tools rely on it, what is the fundamental reason that explains why DY* does not need to rely on unification? Doing so greatly simplifies reasoning on the attacker knowledge, and allows without too much difficulty to handle associative theories that are notably difficult to model in unification-based provers.

After asking this question around, we got the following answer: in a way, ProVerif and Tamarin work by automatically computing some sort of protocol invariants, furthermore unification is a natural tool to use when computing such invariants, because it can be used to find the most general bytestring term that matches some shape. In DY*, the invariants are given by the user, which is why DY* does not need unification.

Other works inspired by DY*. In [63], Arquint et al. re-implement the DY* methodology in the program verifier Gobra [64]. They use separation logic, which allows them to introduce new proof techniques that we discuss below.

[63]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

[64]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

Their approach allows to easily prove injective authentication theorems using separation logic, and claim that DY* cannot. However, it turns out we can also prove injective authentication theorems using DY*, albeit doing so is less straightforward: we can for example, in the event invariant, say that a bytestring in the event was randomly generated immediately before the event was logged, hence because random bytestrings are generated only one time (by definition of randomness), the event containing this bytestring must also be unique. Therefore, although their proof technique toward injective authentication is certainly interesting, it remains unclear whether their approach is strictly more expressive than what we can do in DY* or not: more research would be needed to properly answer this question.

Their approach better models interleaving with the attacker. As we discussed in the paragraph above "Protocol specification coding style", they also notice that the attacker cannot interleave computations within a protocol step execution, hence that DY* specifications must rely on a specific coding discipline to avoid accidental restriction of the attacker. Whereas we argued it was not a problem in practice because this coding discipline was standard is other provers (such as Tamarin), they solve this problem by modeling the attacker and protocol participants as different threads that execute concurrently, and allow for fine-grained interleaving using *concurrent separation logic*. As a result, their execution model is more similar to ProVerif's, whereas DY*'s execution model is more similar to Tamarin's.

2.3 Security proofs with DY*, an example

In §2.2 we explained the theory behind DY* that allows to prove security of cryptographic protocols in the symbolic model. We now give a concrete example on how we do such a proof in practice. We will prove security theorems on a unilaterally signed Diffie-Hellman key exchange, borrowed from the Protocol Proof Ladder [65], and depicted in Figure 2.31.

[65]: (2025), *Protocol Proof Ladder*

2.3.1 Specification

Before writing security theorems and working on its proof, we must first specify rigorously the protocol in DY*.

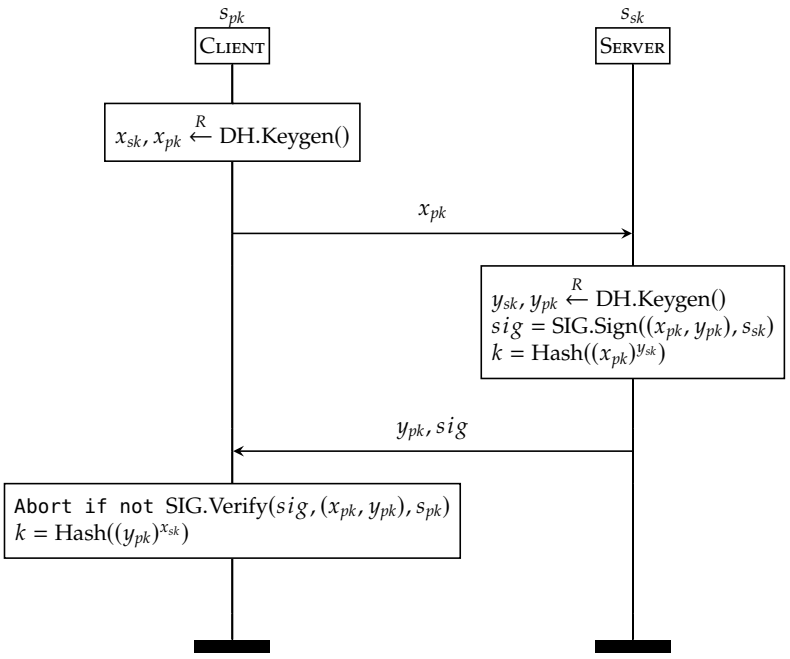


Figure 2.31: Signed Diffie-Hellman key exchange, borrowed from [65].

Types. The diagram that describes the SignedDH protocol (Figure 2.31) manipulates many objects: states, both short-term (e.g. x_{sk}) and long-term (e.g. s_{sk}), messages on the network, and inputs to the signature function. To help state security properties, in our DY* specification, we will further log protocol events for each step of the protocol execution.

We write types for each of these objects in Figure 2.32. Furthermore, using Comparse (Chapter 4), we will define proper message formats for each of these types, allowing us to serialize and parse them.

Protocol steps. We then write in DY* the specification of each protocol step, and show the code corresponding to the server protocol step in Figure 2.33. We further write (not shown in this document) two protocol steps for the client: one to initialize the protocol and another one to finish the handshake.

Long-term keys. Many cryptographic protocols rely on long-term keys, for this reason DY* provides a generic library to handle them, which we will describe in §3.5.2. For now, it suffices to say that the server stores its long-term private signature key in a state dedicated to long-term private keys, and retrieves it using the function get_private_key (see line 30). Then, a trusted process will install the corresponding public verification key on the client in a state dedicated to long-term public keys, thereby modeling a Public-Key Infrastructure (PKI).

2.3.2 Security theorems

We will prove a variety of security theorems on the protocol SignedDH.

Unilateral client-side authentication. If the client finishes the handshake and derives a key k, the server must also have finished a handshake and derived the same key k, unless the long-term signature key of the server was compromised before the client finished the handshake.

```
type client_state =
  | ClientInitiateState:
    x_sk:bytes →
    client_state
  | ClientFinishState:
    k:bytes →
    client_state
```

(a) Type for storing short-term client state.

```
type sig_input = {
  x_pk: bytes;
  y_pk: bytes;
}

type server_message = {
  y_pk: bytes;
  sig: bytes;
}
```

(b) Type for server signature input and message.

```
type signed_dh_event =
  | ClientInitiateEvent:
    // ...
  | ServerFinishEvent:
    server_sid:state_id →
    x_pk:bytes →
    y_pk:bytes →
    k:bytes →
    signed_dh_event
  | ClientFinishEvent:
    // ...
```

(c) Type for protocol events.

Figure 2.32: Various types we define in F* to represent the various objects manipulated by the SignedDH specification. We furthermore derive message formats for all of them using Comparse. We did not depict there the server state and client message, which are simpler than client state and server message.

```
1  // Shorthand label for the ephemeral Diffie-Hellman private key y_sk.
2  // It will be used line 18 when randomly sampling it.
3  let server_ephemeral_key_label (server:principal) (sid:state_id): label =
4    principal_tag_state_label server "SignedDH.ServerState" sid
5
6  val server_receive:
7    server:principal → private_keys_sid:state_id →
8    client_msg_ts:timestamp →
9    traceful (option (state_id & timestamp))
10 let server_receive server private_keys_sid client_msg_ts =
11   // Receive client message and parse it
12   let msg = parse (recv_msg client_msg_ts) in
13   let x_pk = msg.x_pk in
14
15   // Generate ephemeral dh key and compute server key
16   let server_sid = new_session_id server in
17   // - sample randomly y_sk, specify its usage, label, and length
18   let y_sk = mk_rand (DhKey "SignedDH" empty) (server_ephemeral_key_label server server_sid) 32 in
19   // - compute k
20   let k = hash (dh y_sk x_pk) in
21   // - and store it for later use
22   set_state server server_sid (ServerFinishState k);
23
24   // Log protocol event
25   let y_pk = dh_pk y_sk in
26   log_event server (ServerFinishEvent server_sid x_pk y_pk k);
27
28   // Compute signature
29   // - retrieve the long-term signature key
30   let my_sig_key = get_private_key server private_keys_sid (LongTermSigKey "SignedDH") in
31   // - generate a signature nonce
32   let sig_nonce = mk_rand SigNonce secret 32 in
33   // - serialize the signature input and sign it
34   let sig = sign my_sig_key sig_nonce (serialize { x_pk; y_pk; }) in
35
36   // Serialize and send message
37   let server_msg_ts = send_msg (serialize { y_pk; sig; }) in
38   // Return to the attacker the state_id for our short-term state, and the timestamp at which we sent the message.
39   return (Some (server_sid, server_msg_ts))
```

Figure 2.33: Specification in DY* of the server protocol step depicted in Figure 2.31. Only slightly simplified.

Forward secrecy, client-side. If the client finished the handshake and derived a key k, and if the attacker knows k, then one of the following must have been happened:

- the short-term state of the client for that session was compromised
- the short-term state of the server for that session was compromised
- the long-term signature key of the server was compromised before the client finished the handshake

Forward secrecy, server-side. Note that the client is not authenticated on the server-side, therefore we cannot say much about the secrecy of k when the server responds: indeed, the attacker could feed its own x_{pk} to the server and compute k from the server's answer. However, when x_{pk} is honest (therefore corresponds to a protocol event that a client generated it in the first protocol step), we can say the following. If the server finished the handshake using x_{pk} and derived a key k, and if a client initiated the protocol producing the same x_{pk}, and if the attacker knows k, then one of the following must have happened:

- the short-term state of the client for that session was compromised
- the short-term state of the server for that session was compromised

In DY*. We can formalize precisely these three security theorems using DY*, we show the theorem statement of client forward secrecy in Figure 2.34. The statement is mostly a straightforward translation of the security theorem written in prose. The only subtlety is when translating rigorously in DY* the statement "the short-term state of the server for that session was compromised". Indeed, we are from the viewpoint of the client, hence do not know the session identifier used by the server. We resolve this issue by existentially quantifying on the session identifier, and restrict the session identifier to appear in a protocol event logged by the server that has the same parameters as the client session (that is, x_{pk}, y_{pk}, k, etc). In DY*, this is described by the lines 18-21 of Figure 2.34.

2.3.3 Protocol invariants

In automatic tools such as ProVerif [41] or Tamarin [42], we would be done: the protocol has been specified, the security theorems have been written, the tools can now automatically check whether the security theorems hold or not.[10] This is not the case with DY*: to prove that the security theorems hold, we will strengthen them to obtain the *trace invariant* (described in §2.2.9); the trace invariant will then trivially imply the security theorems.

10: another possibility is that they take too much time or memory to verify the protocol, here, SignedDH is simple enough that this does not happen.

Event invariant. Recall (§2.2.9) that participants may only log events that satisfy a user-provided predicate, called the "event invariant". We start by describing the event invariant of SignedDH, shown in Figure 2.35. We only talk about the event ClientFinishEvent which corresponds to the client finishing the handshake (i.e. after verifying the signature and computing the shared key), the other event invariants are less interesting therefore we omit them.

The event invariant works as follows: unless the server long-term key is compromised (line 11), there exists a server session identifier (line 6) such that the server logged an event with the same parameters as the client event (line 7) and the shared key is a shared secret between the client and the server (line 9).

```
1  val client_forward_secrecy:
2   // ... for all client, server, trace, etc ...
3   Lemma
4   (requires
5    // if the trace is reachable (hence satisfies the trace invariant)
6    trace_invariant tr ∧
7    // and the client finished the handhsake with the server, using the DH keys x_pk and y_pk, resulting in the shared key k
8    event_triggered_at tr client_ev_ts client (ClientFinishEvent client_sid server x_pk y_pk k) ∧
9    // and the attacker knows the key
10   attacker_knows tr k
11  )
12  (ensures (
13   let tr_before_ev = prefix tr client_ev_ts in
14   // then either
15   // - the client ephemeral key for this session was compromised
16   is_corrupt tr (client_ephemeral_key_label client client_sid) ∨
17   // - the server ephemeral key for this session was compromised
18   (∃ server_sid.
19    event_triggered tr_before_ev server (ServerFinishEvent server_sid x_pk y_pk k) ∧
20    is_corrupt tr (server_ephemeral_key_label server server_sid)
21   ) ∨
22   // - the server long-term signature key was compromised before the client finished the handshake
23   is_corrupt tr_before_ev (long_term_key_label server)
24  ))
```

Figure 2.34: The client forward secrecy theorem for SignedDH, in DY*. is_corrupt means that the label flows to public, i.e. that there exists a corresponding compromise event in the trace (see §3.2)

Note that this event invariant is a combination of the unilateral client-side authentication and client-side forward secrecy, therefore this event invariant trivially implies these two security theorems. For example, in the client-side forward secrecy theorem (Figure 2.34), from attacker_knows tr k (or $\tau \vdash \mathcal{A}(k)$) and trace_invariant tr (or $\tau\checkmark$) we deduce, by the Attacker Knowledge Theorem that is_publishable tr k (or $\tau \vdash \mathcal{P}(k)$), hence that get_label k 'can_flow tr' public (or $\tau \vdash \mathcal{L}(k) \gtrsim \top$). The event invariant tells us that either the server long-term signature key is compromised (line 11 of Figure 2.35), or that get_label k is a join of client ephemeral key label and server ephemeral key label (line 9 of Figure 2.35). Because this label flows to public, we deduce by Flow-Join-Public from Figure 2.14 that one of these two states must have been compromised, which concludes. This reasoning is handled fully automatically by F* and its SMT solver.

We just saw how to use the event invariant when we have it as a hypothesis, to prove security theorems. However, to enjoy its guarantees, we must prove that the event invariant holds when a participant logs the corresponding event. When the client logs the event ClientFinishEvent, we know k is a hash of a Diffie-Hellman involving x_sk, hence get_label k = join (get_label x_sk) y_label, where y_label is the label of the private key corresponding to y_pk. We further know that get_label x_sk == client_ephemeral_key_label client client_sid: indeed, the client generated it at the previous protocol step and remember this fact using the state invariant (code not shown in this section, because it is straightforward). It means that with the information we know, we can prove line 8 and half of line 9 (because we don't know what is y_label). To prove line 6, 7, 11, and the other half of line 9, we will rely on the *signature predicate*.

Signature predicate. Recall (§2.2.8) that participants may only sign bytestrings that satisfy a user-provided predicate, called the "signature

```
1  let signed_dh_event_invariant =
2   λ tr me ev → (
3    match ev with
4    // ... (we omit ClientInitiateEvent and ServerFinishEvent which are straightforward)
5    | ClientFinishEvent client_sid server x_pk y_pk k → (
6      (∃ server_sid.
7         event_triggered tr server (ServerFinishEvent server_sid x_pk y_pk k) ∧
8         bytes_invariant tr k ∧
9         get_label k == join (client_ephemeral_key_label me client_sid) (server_ephemeral_key_label server server_sid)
10     ) ∨
11     is_corrupt tr (long_term_key_label server)
12   )
13  )
```

Figure 2.35: The event invariant of SignedDH. is_corrupt means that the label flows to public, i.e. that there exists a corresponding compromise event in the trace (see §3.2)

predicate". In return, when another participant verifies a signature, they will deduce that either (1) the signature was performed by an honest participant, hence the signature predicate must hold on the signed message, or (2) the signature was performed by the attacker, hence the attacker must know the signature key.

We show the signature predicate of SignedDH in Figure 2.36. Lines 4-6 deal with message formatting: although the protocol description in Figure 2.31 signs the tuple (x_{pk}, y_{pk}), the tuple must be serialized before signing, hence the signature predicate must parse the tuple back before stating properties on its components. Line 7 attests the existence of the private key y_{sk} that corresponds to the public key y_{pk} (line 8) and has a strict label (line 9) which ensures that y_{sk} stays unknown to the attacker unless they compromise the precise short-term state that contains y_{sk}. Finally, line 10 attests that the server logged an event, which says that the server finished a handshake with keys x_{pk}, y_{pk}, and the shared key k (computed as the hash of a Diffie-Hellman, as described in Figure 2.31).

The last line of code to discuss in the signature predicate is line 3. Before explaining it, note that the security of SignedDH relies on the fact that two different servers do not use the same signature keys: if that were the case, the attacker could break authentication. Indeed, when a client would initiate a handshake with server_1, the attacker could instead forward all messages to server_2; the client would not notice the difference meaning that attacker managed to break authentication. Therefore, different servers must use distinct long-term signature keys. To prove that keys are distinct, we can rely on the *key usage*, discussed in §2.2.7. Therefore, the usage of SignedDH signature keys will tell that they are (1) signature keys for (2) the SignedDH protocol and (3) some particular server identity. It means that we can retrieve the server identity from its signature key usage: this is done in line 3 of the signature predicate (Figure 2.36). There, we existentially quantify on the server identity, and say that the signature key usage must correspond to this server identity. Only one server identity may exist thanks to the injectivity of the function long_term_key_type_to_usage.

When the server computes the signature, it is easy to prove that the signature predicate holds: indeed, y_sk and y_pk were just generated (lines 18 and 25 of Figure 2.33), the event was just logged (line 26 of Figure 2.33). The only thing left to prove is a round-trip property on parse and serialize, that is, parse (serialize msg) == Some msg. Thankfully,

```
1  let signed_dh_sign_pred: sign_crypto_predicate = {
2    pred = (λ tr sk_usg msg → (
3      ∃ server. sk_usg == long_term_key_type_to_usage (LongTermSigKey "SignedDH") server ∧ (
4        match parse msg with
5        | None → ⊥
6        | Some { x_pk; y_pk; } → (
7          ∃ y_sk server_sid.
8            y_pk == dh_pk y_sk ∧
9            get_label y_sk == server_ephemeral_key_label server server_sid ∧
10           event_triggered tr server (ServerFinishEvent server_sid x_pk y_pk (hash (dh y_sk x_pk)))
11       )
12     )
13   ));
14 }
```

Figure 2.36: The signature predicate of SignedDH.

Comparse (Chapter 4) gives us this property on any message format it generates.

When the client successfully verifies the signature, they will deduce that either the signature predicate holds, or that the server long-term signature key has been compromised. This allows to finish the event invariant proof (Figure 2.35). We can prove that the server logged the same event as the client (line 7) as follows: the signature predicate ensures that the server logged an event with the key hash (dh y_sk x_pk), while the client derives the key hash (dh x_sk y_pk). Thankfully, we prove these keys are the same using the commutativity property of dh.[11] Finally, the signature predicate tells us the label of the private key associated to y_pk (in line 8 and 9 of Figure 2.36). This allows the client to know the label of k when logging its event (line 9 of Figure 2.35).

11: dh x_sk (dh_pk y_sk) == dh y_sk (dh_pk x_sk)

2.3.4 Discussion

We saw in the previous sections how we can use DY* to specify a protocol, state security properties, and prove the protocol adheres to these security properties. We give some numbers related to this proof artifact in Table 2.1. The total artifact took a few hours to produce from scratch for an expert DY* user.[12]

12: in all modesty

Component	F* LoC	Verification time
Specification	133	25s
Theorems	65	7s
Proof	160	4s

Table 2.1: Some numbers on the SignedDH security proof in DY*. The verification times are measured on an AMD 7950X. The specification includes Comparse' generation and proof of message formats, which explains its long verification time.

We saw in §2.3.3 how to write protocol invariants, which are stronger versions of the security theorems we want to prove on the protocol. Although this is an additional burden on the user compared to more automatic tools such as ProVerif [41] or Tamarin [42], we think they give insights on the profound reasons *why* the protocol is secure. For example, the signature predicate really tells what is the intent behind the signature: indeed, the signed bytestrings are *a priori* meaningless sequence of bits, zeros and ones, but the signature predicates really tells why we signed this particular bytestring, that is, what we express when we sign it.

DY*: Security proofs in the Dolev-Yao model, using F* (contributions)

3

In this chapter, we present the various usability and improvements we have made to DY*.

The content of this chapter is not yet published, therefore is new material, and is a contribution of this thesis.

Outline. We first present a new technique to define protocol invariants modularly (§3.1), then show a more expressive label framework (§3.2) and explain how we made labels erasable in the process (§3.3). Finally, we explain how to better reason with key usage (§3.4), present multiple engineering improvements (§3.5), and conclude (§3.6).

3.1 Modular protocol invariants

The DY* proof methodology relies on protocol invariants: the user proves that every protocol step preserves some invariants, and proves that the protocol invariants imply the protocol security. These two facts imply that the attacker cannot break the protocol security: indeed, the attacker can only reach traces that satisfy the invariant (first fact), and traces that satisfy these invariants obey the protocol security properties (second fact).

Although the protocols invariants are specific for each protocol security proof, DY* provides a blueprint for creating protocol invariants with holes that must be filled by the user. For example, the user provides a predicate for signatures, and the DY* protocol invariants blueprint will ensure that for any signature in the protocol, either the signature is honest and in that case the signature predicate holds on the signed message, or the signature was computed by the attacker in which case the attacker knows the private key corresponding to this signature. Similar predicates exist for Authenticated Encryption with Associated Data (AEAD), Message Authentication Code (MAC), but also state storage or protocol event logging.

```
type signature_predicate =
  trace → vk:bytes → msg:bytes →
  prop

val mk_protocol_trace_invariant:
  signature_predicate → ... →
  trace → prop
```

Figure 3.1: Type for signature predicate, and protocol trace invariant blueprint.

With this approach, the protocol verification is modular because each protocol step is verified independently. However, in the process of writing the proof for a protocol step, it may happen that the DY* user needs to tweak (say) the signature predicate. When doing so, the protocol invariants will change and every protocol steps need to be re-checked with respect to these new protocol invariants, even the protocol steps that do not deal with signatures. Although F*'s SMT-based proofs are usually stable regarding irrelevant protocol invariant changes, it may happen that the SMT gods are not in the mood that day, and will ask DY* users to sacrifice time to maintain already-done proofs.

Because of this limitation, DY* cannot compose protocol proofs; for example, to prove the security of a (big) protocol[1] by proving the security of (smaller) sub-protocols.[2] Indeed, both protocol proofs likely use different protocol invariants, however DY* requires users to provide a single protocol invariant. Hence, to combine both security proofs, the user would need to combine the two protocol invariants in a single one, and re-prove that each protocol steps preserve this combined invariant.

1: e.g. TLS 1.3

2: e.g. TLS' Handshake Protocol and Record Protocol

This is a known limitation of DY* that has been addressed in previous work: Bhargavan et al. [66] propose an approach for protocol composition in DY*, and implement a generic "communication layer" to be used by protocols that rely on an authenticated or confidential communication channel. However, their approach only works for vertical composition (when a protocol uses another protocol) but not for horizontal composition (when two protocols run in parallel).

[66]: Bhargavan et al. (2024), *Layered Symbolic Security Analysis in DY**

In this section, we present a framework to create protocol invariants that allows both horizontal and vertical composition in DY*, which was already used by and crucial to the security proofs of the TreeKEM protocol (Chapter 6).

We present this protocol invariant framework in four steps:

- first, the general shape of protocol invariants we can create,
- then, how we prove a protocol step preserves local protocol invariants,
- then, how we combine local invariants into a single global invariant,
- finally, we show three case studies: TreeKEM, and a model of HPKE

3.1.1 Shaping DY* protocol invariants as decision trees

Secure cryptographic protocols may lose their security guarantees when composed with other secure cryptographic protocols, for example when they share long-term keys, leading to cross-protocol attacks.

For this reason, it is impossible to combine arbitrary protocol invariants, as this would be unsound: this would prove the security of protocol composition, although the composition of two secure protocols is not always secure. Instead, our framework enforces a specific shape of invariants so as to allow composition. We now go through several criteria that intuitively describe under which conditions protocols can be safely composed; then, we turn these intuitive criteria into a formal shape of invariants that are composable. We later on discuss how this does not affect the expressive power of the DY* framework.

Distinct keys. Two cryptographic protocols may be safely composed when they use distinct key material (for example software developers routinely run TLS and SSH in parallel). When this is the case, we can combine the protocol invariants of the two protocols: the global DY* invariant will be of the form "if the key belongs to protocol one, then dispatch to the invariants of protocol one, and if the key belongs to protocol two, then dispatch to the invariants of protocol two". The notion of key belonging to a protocol is formalized in DY* via the *key usage* (see §2.2.7), as shown in Figure 3.2.

```
let sign_pred vk msg =
  match get_usage vk with
  | SignKey "TLS" →
   tls_sign_pred vk msg
  | SignKey "SSH" →
   ssh_sign_pred vk msg
```

Figure 3.2: Example of global invariant that dispatches to local invariants using key usages.

Domain separation. Two cryptographic protocols may use the same key, but use that key with good domain separation, for example by tagging the input of their cryptographic primitive. When this is the case, we can combine the protocol invariants of the two protocols: the global DY* invariant will be of the form "if the input tag corresponds to protocol one, then dispatch to the invariants of protocol one, and if the input tag corresponds to protocol two, then dispatch to the invariants of protocol two", as shown in Figure 3.3.

```
let tls_sign_pred vk msg =
  match get_domain_sep msg with
  | "TLS 1.2" →
   tls12_sign_pred vk msg
  | "TLS 1.3" →
   tls13_sign_pred vk msg
```

Figure 3.3: Example of global invariant that dispatches to local invariants using a domain separator. Note that TLS doesn't have such a straightforward domain separator, the code above is only for example purpose.

Shaping protocol invariants as decision trees. We can combine the two approaches we have seen: first, dispatch on the key usage, then, dispatch

on the domain separator, etc. At the end, the protocol invariant is shaped like a decision tree, where at each step we decide which branch to take depending on some tag (e.g., the key usage, or a domain separator).

Expressivity. We claim that imposing such shape on the protocol invariants does not affect DY* expressivity. Indeed, in Chapter 4 [67], we give a reasonable but sufficient set of criteria protocols should obey to avoid message-formatting attacks (a particular type of cross-protocol attack). In turn, protocols that obey these criteria can have their invariants shaped as decision trees. For example, on the criterion for signatures, [67] says that "each signature key must be used to sign messages with the same self-contained, non-ambiguous, representation-unique message format". With the lens of DY*, when protocols obey this criterion, we can first branch on the signature key usage (as in Figure 3.2), then for each usage corresponds a single message format and use the corresponding parsing function to implement get_domain_sep (as in Figure 3.3). Conversely, in Chapter 5 [68] we found a cross-protocol attack between the MLS sub-protocols TreeSync and TreeDEM: we found this attack because TreeSync and TreeDEM did not properly tag their signatures, hence we couldn't write a common signature predicate shaped as a decision tree, thereby hinting at the signature ambiguity attack.

[67]: Wallez et al. (2023), *Comparse: Provably Secure Formats for Cryptographic Protocols*

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

Our protocol invariant framework. We do not deal with full-blown decision trees directly, instead our framework only deals with one node of the decision tree: it can then be applied recursively to obtain a complete decision tree.

3.1.2 Relating local and global invariants

When we prove that a protocol step preserves the global DY* protocol invariants, our key insight is that we do not need to know the complete definition of the invariants, we only need to know how it behaves on a particular branch of the decision tree. Then, the proof works for any global protocol invariant, as long as it "contains" some local protocol invariants. For example, when proving each protocol step of TLS 1.3 preserves the protocol invariants, we only need to know that sign_pred contains the tls_sign_pred branch (Figure 3.2) and that tls_sign_pred contains the tls13_sign_pred branch (Figure 3.3). In this section, we formally relate define what it means for a global invariant to contain a local invariant.

One node at a time. We only deal with one node of the decision tree, and apply our methodology recursively to obtain a complete decision tree. Hence, the notion of "global" and "local" invariant is relative to the node of the decision we are currently considering: for example, in Figure 3.2, tls_sign_pred is a local invariant, but in Figure 3.3 it is the global invariant.

Invariants as functions. In the example of signatures, the signature invariant is a predicate on signed messages. In complete generality, an invariant is a function. This function may return propositions (hence be a predicate), but may return other things; for example KDF invariants return labels (see §2.2.6).

Setup provided by the user. Our framework rely on the user to provide types and functions that we now describe. We expect the user to provide a type for global invariant input data $\mathbb{D}_G$,[3] a type for local invariant input data $\mathbb{D}_L$,[4] a type for tags $\mathbb{T}$,[5] and a type for the invariant output $\mathbb{O}$.[6]

3: in the example of signature invariant, $\mathbb{D}_G$ would be a tuple containing verification key and signed message

4: in the simplest case it might be equal to $\mathbb{D}_G$

5: in the example of Figure 3.2, $\mathbb{T}$ is the set of key usages, in the example of Figure 3.3, $\mathbb{T}$ is the set of domain separators (here, strings)

6: for predicates, $\mathbb{O}$ is the type of propositions

Furthermore, a function $\texttt{decode\_data} : \mathbb{D}_G \to \mathbb{T} \times \mathbb{D}_L \cup \{\bot\}$ computes a tag and a local input data given a global input data.

Local invariants inside global invariant. With this setup, a local invariant is function $\mathbb{D}_L \to \mathbb{O}$ and a global invariant is function $\mathbb{D}_G \to \mathbb{O}$. Now, given a tag t, a local invariant l and global invariant g, we say that $(t, l) \subset g$ (or g "contains" l) when

$$\forall x_L x_G.\texttt{decode\_data}(x_G) = (t, x_L) \implies g(x_G) = l(x_L)$$

In other words, when $(t, l) \subset g$, we know how g behaves on inputs with tag t.

For example, in Figure 3.2, the property ("TLS", tls_sign_pred) ⊂ sign_pred would be equivalent to

```
∀vk, msg. get_usage vk = SignKey"TLS" ⟹
                sign_pred vk msg = tls_sign_pred vk msg
```

It means that when we know that vk is a key that belongs to the TLS protocol, we deduce that sign_pred vk msg behaves like tls_sign_pred vk msg, which allows us to prove a protocol step of TLS with respect to the global signature predicate sign_pred.

Reasoning on global invariants using local invariants. DY* users must deal with the various (global) protocol invariants throughout security proofs. For example, they must prove that the (global) signature predicate holds when computing a signature, or deduce that the (global) signature predicate holds when a verification succeeds. With our methodology, instead of defining top-level global protocol invariants and prove every protocol step with respect to these particular global protocol invariants, we now write proofs with respect to *any* global protocol invariants, as long as they contain local invariants specific to this protocol step. Using the relationship $(t, l) \subset g$ we can do proofs on the (abstract) global invariant by relying on our knowledge about the (concrete) local invariant, for every input tagged with t.

3.1.3 Creating a global invariant from local invariants

With our methodology, each protocol step now requires that the global protocol invariants contain a bunch of local invariants, or in mathematical notations, $(t_i, l_i) \subset g$ for $i \in \{1, \ldots, n\}$.

For example, in Figure 3.2 we need to prove that:

- ("TLS", tls_sign_pred) ⊂ sign_pred (required by the TLS proofs)
- ("SSH", ssh_sign_pred) ⊂ sign_pred (required by the SSH proofs)

Requirements from the user. We require the user to gather the set of tags and local invariants $L = \{(t_1, l_1), \ldots, (t_n, l_n)\}$ used throughout every protocol step proof. Then, we require them to prove tags are all distinct, namely $\forall i, j.t_i = t_j \implies i = j$. This is in practice easily prove by normalization, because tags are concrete values.

We give an example of such an L for the signature predicate of Figure 3.2 in Figure 3.4.

```
let L = [
   ("TLS", tls_sign_pred);
   ("SSH", ssh_sign_pred);
]

// Need to prove the following:
assert (pairwise_distinct (map fst L))
// This is trivially true because
// map fst L == ["TLS"; "SSH"]
```

Figure 3.4: Example of local invariants list for Figure 3.2.

Building the global invariant. Given the list of tags and local invariants L, our framework provides a global invariant $G(L)$. If L obeys the distinctness

```
/// HPKE's LabeledExpand function

type labeled_expand_info = {
  len: uint16;
  label: string;
  info: bytes;
}

let labeled_expand prk label info len =
  let labeled_info = serialize { len; label; info; } in
  kdf_expand prk labeled_info len

/// The HPKE single-shot API in base mode

let hpke_pk sk =
  kem_pk sk

let hpke_enc pkR entropy plaintext info ad =
  let (enc, shared_secret) = kem_encap pkR entropy in
  let aead_key = labeled_expand shared_secret "key" info 32 in
  let aead_nonce = labeled_expand shared_secret "base_nonce" info 32 in
  let ciphertext = aead_enc aead_key aead_nonce plaintext ad in
  (enc, ciphertext)

let hpke_dec skR (enc, ciphertext) info ad =
  let? shared_secret = kem_decap skR enc in
  let aead_key = labeled_expand shared_secret "key" info 32 in
  let aead_nonce = labeled_expand shared_secret "base_nonce" info 32 in
  aead_dec aead_key aead_nonce ciphertext ad
```

Figure 3.5: Specification of our simplified HPKE model. Note that the ad bytestring is authenticated via the AEAD additional data, whereas the info bytestring is authenticated within the AEAD key via the labeled_expand key derivation.

conditions, our framework proves that $\forall i.(t_i, l_i) \subset G(L)$. Therefore, this global invariant $G(L)$ satisfy the requirements of every protocol step proof.

Internals of G. The definition of G is as follows: given an input x_G, compute $\texttt{decode\_data}(x_G) = (t, x_L)$. If $t = t_i$ for some i, then $G(L)(x_G)$ is defined to be $l_i(x_L)$. If no such i exists, or if $\texttt{decode\_data}(x_G) = \bot$, we define $G(L)(x_G)$ to be equal to a default value (supplied by the user).

When tags are not distinct. We enforce the requirement that tags but be distinct, what happens when this is not the case? We argue that when tags are not distinct, this sheds light on a protocol weakness. Indeed, in the example of Figure 3.2 this would give evidence that a given key is used for too many purposes, or in the example of Figure 3.3 this would give evidence on a lack of domain separation.

3.1.4 Case study: Hybrid Public-Key Encryption

We demonstrate the expressivity of our modular protocol invariants technique with a case study on Hybrid Public Key Encryption (HPKE) [69].

[69]: Barnes et al. (2022), *RFC 9180: Hybrid public key encryption*

Modeling HPKE. We specify and prove a simplified model of HPKE [69] in Figure 3.5, more specifically its Single-Shot API in the Base mode which is used in MLS [21]. Internally, HPKE relies on three cryptographic primitives: Key Encapsulation Mechanism (KEM), Key Derivation Function (KDF), and Authenticated Encryption With Associated Data (AEAD).

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

Public-Key Encryption properties. We want DY* users to be able to use HPKE as a Public-Key Encryption (PKE) algorithm. To do so, we prove that HPKE's encryption and decryption functions obey similar security properties as DY*'s native PKE algorithm (i.e. how it preserves the bytes invariant, see §2.2.8). To recall, PKE ensures that the ciphertext is safely publishable (hence can be sent on the network) only:

- (confidentiality) when the plaintext is less secret than the private key,[7]
- (authenticity) when a user-provided global predicate (the PKE predicate) holds on the plaintext, or when the plaintext is publishable.[8]

We prove similar properties for HPKE, hence rely on a (global, user-provided) predicate for HPKE plaintexts.

7: otherwise, the attacker could use the ciphertext and the private key to learn something more secret than what they are allowed to know

8: this handles the case where the attacker performed the encryption themselves

Confidentiality. We easily prove that HPKE provides confidentiality by combining the confidentiality properties of the KEM, the KDF and the AEAD. We do not dive deeply in how we do this proof, because the interesting proof is about authenticity.

Authenticity. The authenticity guarantees of HPKE ultimately boil down to the authenticity guarantees provided by the underlying AEAD. To do so, we first combine the guarantees given by the KEM and the KDF to deduce that aead_key (see Figure 3.5) has the usage of an AEAD key for the protocol "HPKE" (more on that in the next paragraphs). Then (as shown in Figure 3.6), we compile the *global* HPKE predicate into a *local* AEAD predicate that can be composed with local AEAD predicates of other protocols as in Figure 3.2.

```
let hpke_aead_pred hpke_pred =
  λ tr key nonce plain ad →
    // We know that 'key' is an AEAD
    // key for HPKE, we retrieve the
    // underlying HPKE key usage and
    // the info field from the AEAD key
    // usage
    let AeadKey "HPKE" {
      hpke_key_usage;
      info;
    } = get_usage key in
    // either the plaintext is public
    is_publishable tr plain ∨
    // or the hpke predicate holds
    hpke_pred.pred
      tr hpke_key_usage plain info ad
  );
```

Figure 3.6: Compilation of a *global* HPKE predicate to a *local* AEAD predicate.

Modular HPKE predicate. DY* users may compose multiple protocols that internally rely on HPKE. When these protocols use distinct keys, we may write a global HPKE predicate using again our modular invariants technique and dispatch to local HPKE predicates depending on the HPKE keypair usage (similarly to Figure 3.2).

HPKE key usages. We mentioned above that aead_key (see Figure 3.5) has the usage of an AEAD key for the protocol "HPKE". This was a simplification, we now fully describe the usage of keys in HPKE. Recall (§2.2.7) that usages are a way to prove that two keys are distinct in DY*, by proving they have distinct usages. Let "Protocol X" be the protocol where we use HPKE. The usages of keys in our HPKE specification (see Figure 3.5) are as follows:

- skR / pkR must have the usage "KEM key for HPKE in Protocol X". This must be proved by the user.
- shared_secret has usage "KDF.Expand key for HPKE in Protocol X". This is proved using the properties of kem_encap and kem_decap.
- aead_key has usage "AEAD key for HPKE in Protocol X with info". This is proved using the properties of kdf_expand. The info field appears here because it is included in the key derivation.

This is how in Figure 3.6 we are able to obtain the HPKE key usage and info from the AEAD key usage.

The local HPKE decision tree. In Figure 3.7, we depict the local protocol invariants required by our HPKE proof. Our proof works with any protocol invariants, as long as the underlying KDF invariants contain local invariants for HPKE, and as long as the underlying AEAD predicate contain a local predicate for HPKE, which is itself compiled from a global

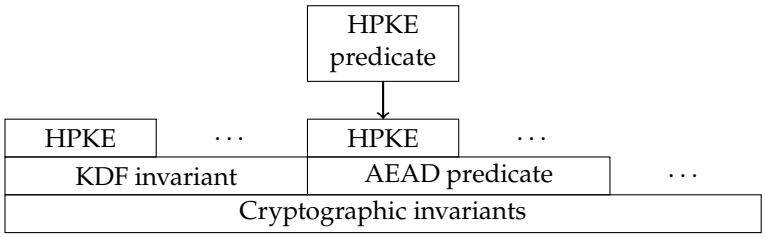


Figure 3.7: Graphical depiction of the cryptographic invariants required by HPKE. HPKE relies on a local KDF invariant, and on a local AEAD predicate which is obtained by compiling the global HPKE predicate (as shown in Figure 3.6).

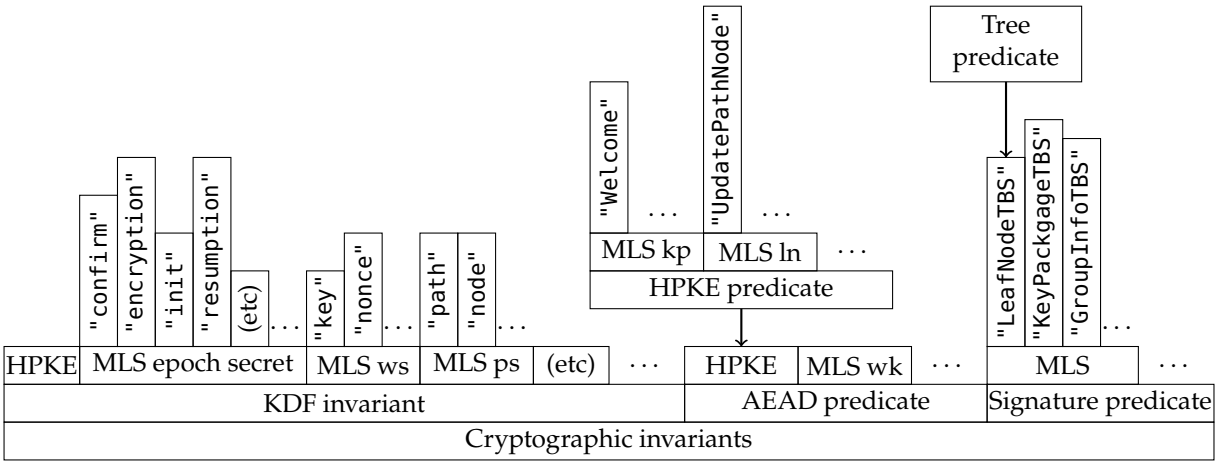


Figure 3.8: Graphical depiction of the protocol invariants required by TreeKEM. "MLS ws" means "MLS welcome secret", "MLS ps" means "MLS path secret", "MLS kp" means "MLS KeyPackage", "MLS ln" means "MLS LeafNode", "MLS wk" means "MLS welcome key". The "(etc)" blocks mean that we omitted some of the invariants for space reasons, while the "..." mean that other invariants could be added there if needed. The strings are domain-separators, which are inputs of the MLS constructions SignWithLabel, EncryptWithLabel and ExpandWithLabel. The "Tree Predicate" is a parameter of TreeSync which is instantiated by TreeKEM.

HPKE predicate, which may also be defined modularly (not depicted here).

3.1.5 Case study: TreeKEM

We demonstrate the effectiveness of our modular protocol invariants technique with a case study on TreeKEM.

TreeKEM's cryptographic invariants. In Chapter 6, we will prove the security of TreeKEM. This security proof features vertical composition because TreeKEM relies on HPKE (§3.1.4) and on TreeSync (Chapter 5). It also features horizontal composition: HPKE is used in two places in TreeKEM with a domain separator, AEAD is used both in HPKE and in TreeKEM, KDF is used in HPKE and in multiple places of TreeKEM, furthermore TreeKEM relies on multiple signatures (via TreeSync) which use the same signature key but are domain-separated.

The full cryptographic invariants decision tree is depicted in Figure 3.8. Its size demonstrates that our framework scales well and can handle complex cryptographic protocol with substantial cryptographic invariants.

Extending TreeKEM's invariants. TreeKEM is running in parallel of other protocols, for example the sub-protocol of MLS called TreeDEM. In order to perform a combined security proof of TreeKEM + TreeDEM (namely, MLS in its entirety) one would need to modify the cryptographic invariants depicted in Figure 3.8. Fortunately, this is easy: for example, TreeDEM relies on a signature with domain-separator `"FramedContentTBS"` which has its own predicate; fortunately, adding it

in the decision tree is as simple as adding an element to a list (the list `L` of Figure 3.4).

3.1.6 Discussion

This work has been motivated by the security proofs for MLS (Chapter 5 and Chapter 6). Indeed, if we were to write a single big monolithic cryptographic invariant for TreeKEM (as depicted in Figure 3.4), it would span roughly around a thousand lines of code, it would be difficult to maintain, and each time it is tweaked, existing proofs would risk to break (even the proofs not related to said tweak). Therefore, we think this modular invariant framework was a crucial technique to conduct our security proofs.

Boilerplate. Instantiating the framework requires some boilerplate, aside from defining the parameters `decode_data` etc. which cannot be compressed. We have two types of boilerplate which we tried to reduce as much as we could.

When defining the global invariant, we must prove it does contain all the local invariants of our protocol. This is done with only one line of code, which is nice because in particular it is independent on the number of local invariants to merge.

We must lift properties of our local invariants to properties of our global invariant: for example, the signature predicate must stay true when the trace grows, that is, if it is true now, it will stay true in the future. To do that, we require around 10-20 lines of boilerplate per property to lift: around 5-10 for the property statement, and around 5-10 for the proof. It seems like it could be improved, but we haven't figured out how. Still, we think this is a reasonable amount of boilerplate because it belongs to more library-ish parts of the code, and it does not affect the readability of the rest of the proofs.

History of the framework. This framework, although simple in appearance, is the result of many iterations, each time improving its expressivity or its usability.

The idea started when writing the first signature predicate for MLS, namely for `"LeafNodeTBS"` signature in TreeSync. The security proof of TreeSync was already not easy, hence we wanted this proof to be done once and for all, and did not want future extensions to the signature invariant (e.g. adding a case for `"GroupInfoTBS"`) to risk breaking our TreeSync proofs. To solve this problem, we designed a first version of the framework, hard-coded for MLS' domain-separated signatures.

When progressing through the security proof of TreeSync, we faced a similar problem for states: in DY*, protocols must use functions to store and retrieve their state, which also come with an invariant. In TreeSync, we had many different types of states, each with their own invariant: we decided to generalize the framework and parametrize it with `decode_data` etc, as described in §3.1.2.

Until this point, the framework lived within TreeSync security proofs, that is, outside DY*. We later incorporated the framework in DY* (see §3.5.2), and provided several basic invariant builders, such as defining the signature predicate modularly depending on the signature key usage (whereas in TreeSync, we only did it depending on a domain-separator). It worked well for the AEAD predicate, the signature predicate and the

PKE predicate (similar to the HPKE predicate), but it did not work for the KDF invariant, because the framework only supported predicates at that time, and the KDF invariant is a pair of functions returning labels or usages, not propositions. We revamped the framework to also parametrize the output type, and be able to modularly define the KDF invariant.

It was also at this point we realized that we could apply the framework recursively, that is, build decision trees with depth ≥ 1, so that we can write MLS' domain-separated signature invariants on top of the per-key-usage signature invariants. We didn't use this feature much on signatures, but it was a crucial ingredient to define the KDF invariant of TreeKEM.

When using the framework there was still an itch: the boilerplate to prove that the global invariant contained all the local invariants was high. Indeed, its size was linear in the number of local invariants. We managed to reduce it to only one line, which is good for the happiness of DY* users.

At this point, the framework was mostly done, but it is only when writing the security proof for HPKE that we realized how expressive it was. Now, we really think it is *the* way to write invariants in DY*, and doing it like this feels very natural. However, this is the result of several iterations and "aha!" moments.

3.2 Renovating the label construction

In symbolic protocol provers (such as DY*, but also ProVerif or Tamarin) confidentiality theorems are of the form "if the attacker knows this bytestring (e.g. a secret key), then it must have performed these type of compromises". To prove this kind of property, DY* relies on a construction called *security labels*: each bytestring is associated with a label which encodes an over-approximation of the compromises that may lead the attacker to know this bytestring. For example, the label of Alice's private signature key may ensure that if the attacker knows this key, then it must have compromised one of Alice's states. Note that this is an over-approximation because the converse is not true: for example, the attacker may have compromised the state of Alice where she stores pictures of cats, but this does not reveal Alice's private signature key to the attacker.

Security labels are built using three ingredients (depicted in Figure 3.9): a type for labels and functions to construct them (so that the type can remain abstract to users), a predicate is_corrupt that tells whether the attacker compromises fall within the over-approximation encoded by the label, and a relation can_flow that tells whether a label is less secret than another. The label of a bytestring restricts how the bytestring can be used in a cryptographic protocol: for example, a plaintext can be encrypted with a key only when the key is more secret (as per can_flow) than the plaintext. This ensures that, for example, if the attacker knows the key (hence can decrypt and obtain the plaintext), the plaintext label is corrupt (because the key label is both corrupt and more secret than the plaintext label), hence is a correct over-approximation of the compromises that may lead the attacker to know the plaintext.

```
type label

val public: label
val principal_label: principal → label
val join: label → label → label
// etc

val is_corrupt:
  trace → label → prop
val can_flow:
  trace → label → label → prop
```

Figure 3.9: The F* types of the various ingredients to build labels.

In the original DY* [43], security labels have limited expressivity: they can talk about the compromise of a principal, or of a specific state of a principal, identified by a state identifier (an integer), or of a specific version of that state, identified by a version identifier (an integer). The state

identifier is in itself just a pointer, and does not convey any information on what the state contains: does it hold an ephemeral key, or some long-term signature key? Furthermore, using these labels, one cannot express fine-grained temporal guarantees: for example that a signature key was compromised *before* we verified some corresponding signature (e.g. in a handshake protocol).

In this section, we present a new labeling framework that is more expressive, and can in particular express compromise of some particular state type (e.g. signature key) and express temporal properties (e.g. compromise of a signature key before some event). This new labeling framework was already used by and crucial to the security proofs of the TreeKEM protocol (Chapter 6).

We present the new label framework in four steps:

- first, we present the definition of labels in the original DY* [43]
- then, we simplify the definition of is_corrupt and can_flow
- then, we change the label type and is_corrupt so that labels can be extended on the user-side instead of requiring to modify DY*'s core
- finally, we show how the security proof of TreeKEM leveraged this new labeling framework

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

3.2.1 Labels in the original DY*

We begin by giving an introduction to the design of labels in the original DY* [43], then show a new, simpler design.

The label type. Labels are defined as an Abstract Syntax Tree (AST) with one node per operation on labels (e.g., meet or join) or per base label (e.g., the public label or label corresponding to a principal), shown in Figure 3.10.

```
type label =
  | Principal: principal → label
  | Join: label → label → label
  | Meet label → label → label
  | Public: label
  | Secret: label
```

Figure 3.10: Inductive type for labels.

The can_flow relation. In the original DY* [43], can_flow is defined by doing a case analysis on the two labels being compared (as shown in Figure 3.11a), therefore its definition size is quadratic in the number of node type in the label AST. Furthermore, this definition of can_flow must be a partial order, i.e., it must be reflexive and transitive. The transitivity proof is done by doing a case analysis on the three labels involved, resulting in a proof script whose size is cubic in the number of node type in the label AST. Then, a linear number of properties with linear proof script must be proved, e.g., public satisfies the greatest element property, join satisfies the least upper bound property, etc. Label corruption is then defined as flowing to the public label.

3.2.2 Simplifying labels

We design a new version of can_flow (shown in Figure 3.11b) whose definition size is linear in the number of node type in the label AST, and where every related proof (e.g., reflexivity, transitivity, least upper bound property, etc) is of constant size. As a result, this new version of can_flow will be easier to adapt when we extend the label AST with new node types.

Defining is_corrupt. In Figure 3.11b, we define is_corrupt directly, instead of defining it from can_flow (as in Figure 3.11a): the label Principal is corrupt when there exists a compromise event in the trace for the corresponding

```
let rec can_flow (tr:trace) (l1 l2:label) =
  match l1, l2 with
  | Principal p, Public → exists_compromise_event tr p
  | Join l11 l12, Join l21 l22 → ...
     (can_flow tr l1 l21 ∧ can_flow tr l1 l22) ∨
     (can_flow tr l11 l2 ∨ can_flow tr l12 l2)
  | Principal p, Meet l21 l22 → ...
  | Meet l11 l12, Join l21 l22 → ...
  ...

let is_corrupt (tr:trace) (l:label) = can_flow tr l Public
```

(a) The can_flow relation as defined in the original DY* [43]. Parts omitted for brevity.

```
let rec is_corrupt (tr:trace) (l:label) =
  match l with
  | Principal p → exists_compromise_event tr p
  | Join l1 l2 → is_corrupt tr l1 ∨ is_corrupt tr l2
  | Meet l1 l2 → is_corrupt tr l1 ∧ is_corrupt tr l2
  | Public → ⊤
  | Secret → ⊥

let can_flow (tr:trace) (l1 l2:label) =
  ∀ tr_later. tr ≤ tr_later ⟹
    (is_corrupt tr_later l2 ⟹ is_corrupt tr_later l1)
```

(b) Our new definition of can_flow. The definition as shown here is complete, no parts were omitted.

Figure 3.11: Definition of labels in DY*

principal, the Public label is always corrupt (regardless the compromises from the attacker). Finally, the Join label is interpreted as a disjunction and the Meet label is interpreted as a conjunction.

Defining can_flow from is_corrupt. We then define that a label l1 flows to a label l2 with respect to a trace tr when for any possible way to extend the trace (e.g., by adding compromise events) the label l1 is corrupt when the label l2 is corrupt. It means it is impossible for the attacker to corrupt l2 without corrupting l1, hence reflecting the intuition that l1 is "less secret" than l2.

Proving properties of can_flow. With this new definition of can_flow, most properties are now easy to prove: for example, transitivity of can_flow follows immediately from transitivity of implication, or, Public flows to any label because anything implies ⊤.

Relationship between the old and the new can_flow. We formally prove that the previous and new is_corrupt are equivalent, and that our new can_flow definition is the largest label flow relation that satisfies these three properties: (1) it is transitive, (2) it stays true when the trace grows, (3) flowing to public is equivalent to being corrupt. In particular, the previous can_flow obeys these three properties hence implies our new can_flow.

3.2.3 Generalizing labels

We now show how we modify the label type and is_corrupt to allow users to define new labels within their projects, instead of requiring modifying DY* core.

Labels as trace predicates. In §3.2.2, we can view labels as a deep-embedded domain-specific language (DSL) for trace predicates, where is_corrupt (Figure 3.11b) is an interpreter for this DSL. We use this insight to design the most general label type possible: our new labels are now full-fledged trace predicates, which then benefit from the full expressivity of F*. The labels presented in Figure 3.11b can now be implemented outside of DY*'s core (see Figure 3.12), and nothing prevents users to implement other more expressive labels as we will see in §3.2.4. Changing labels to be full-fledged trace predicates creates two technical difficulties that we explain in the rest of this section.

```
let principal_label (p:principal) = {
  is_corrupt = (λ tr →
    exists_compromise_event tr p
  );
}
let join (l1 l2:label) = {
  is_corrupt = (λ tr →
    l1.is_corrupt tr ∨ l2.is_corrupt tr
  );
}
```

Figure 3.12: Example of labels as trace predicates

Erasing labels. When labels were abstract syntax trees they could be compared using F*'s decidable equality, however this is not the case anymore because we cannot compute whether two predicates are identical or not. Indeed, before, we could write a program that checks whether two labels are equal (hence, *decide the equality* on labels). This is now impossible because labels are predicates on traces, that is, functions from traces to propositions: it is impossible to write a program that always terminates and checks whether two functions with infinite co-domain (here, traces) are equal. This causes a problem in another part of DY*, bytestrings, because the type bytes contain a label in the constructor for fresh randomness (see Figure 3.13 and Figure 2.8 in §2.2.4), however, we need decidable equality on bytes (e.g. because cryptographic protocols compare hash values). We will explain our complete solution to this problem in §3.3; in short, we succeed to remove labels from the bytes constructor, thereby making the label type *erasable* (meaning that labels are just a proof artifact and never play a role in computations).

```
type bytes =
  | Rand: ... → label → bytes
  ...

let rec get_label b =
  match b with
  | Rand ... lab → lab
  ...
```

Figure 3.13: It is convenient to store the label of a fresh random bytestring in its corresponding constructor in bytes, so that we can easily retrieve it in get_label.

Positivity of the trace type. The trace contains labels (in the event corresponding to fresh randomness generation), and labels are now predicates on the trace. Proof assistants such as F* will reject such types because they are not positive: indeed, it is unsound to accept types that contain predicates on themselves [70], otherwise (in short) this type would give a surjection from trace → prop to trace which leads to a contradiction using Cantor's diagonal argument. To work around this issue, labels are in reality not predicates on the trace, but predicates on a view of the trace where labels are removed (i.e., replaced with the unit type), as shown in Figure 3.14. This solves the issue about positivity, at the price of labels being slightly less expressive. This loss of expressivity is not a problem in practice, because reasonable labels don't depend on other labels present in the trace.

```
type trace_entry_ (label_t:Type) =
  | MkRand:
    label:label_t → ... →
    trace_entry_ label_t
  ...
type trace_ (label_t:Type) =
  list (trace_entry_ label_t)
type label = (trace_ unit → prop)
type trace = trace_ label
```

Figure 3.14: The trace type with labels as trace predicates. The trace is parametric in the label type, and labels are predicates on trace without labels. This ensures the positivity of the trace type.

New definition of can_flow. We show the new definition of is_corrupt and can_flow in Figure 3.15. The definition of can_flow is the same as in the previous section (see Figure 3.11b). The definition of is_corrupt is mostly transferred to labels themselves (e.g. compare the join label in Figure 3.12 and the Join case in Figure 3.11b); is_corrupt only takes care of removing labels of the traces via the helper function trace_forget_labels (see previous paragraph "Positivity of the trace type"). Note that the prefix relation on traces (≤) in can_flow still takes into account labels in the traces, this is fine because we don't need to compute whether a trace is a prefix of another, we only need to reason with prefixes in proofs.

```
let is_corrupt (tr:trace) (l:label) =
   l.is_corrupt (trace_forget_labels tr)

let can_flow (tr:trace) (l1 l2:label) =
   ∀ tr_later. tr ≤ tr_later ⟹ (
      is_corrupt tr_later l2 ⟹
      is_corrupt tr_later l1
   )
```

Figure 3.15: The new definition of can_flow, with labels as trace predicates.

Label extensionality. We can further prove an *extensionality* theorem on labels: if two labels flow to each other for all traces (hence their underlying is_corrupt function are pointwise equivalent), then the two labels are equal. This allows us to prove many equational theorems on labels, for example that join is commutative: before, join l1 l2 and join l2 l1 were *equivalent* from the viewpoint of can_flow, but not *equal*. This is useful because F* can reason with congruence[9] automatically with *equality* (i.e. the built-in equality of F*), but not with *equivalence* (a concept we define ourselves on labels).

9: if ℓ_1 is equivalent to ℓ_2 then $f(\ell_1)$ is equivalent to $f(\ell_2)$ for any function f.

3.2.4 Examples of more expressive labels

In the security proof of TreeKEM (Chapter 6), we rely on these new, more expressive labels, and define new kind of labels needed to conduct their

security proof without having to fork DY*, demonstrating the success of our approach. In this section, we explain how we used these new labels to conduct the TreeKEM security proof.

Signed ephemeral key. In TreeKEM, an ephemeral public encryption key is signed using a long-term signature key (in the so-called "key package"). After we verify this signature, we want the following guarantee: when we encrypt data with this ephemeral public encryption key, this data is safe from the attacker, unless (1) the attacker compromised the corresponding private decryption key, or (2) the attacker compromised this long-term signature key *before* we checked the signature. We do this by designing a label encompassing these two possibilities, and prove that this label is *less secret* than the ephemeral private decryption key label: indeed, this ensures that if the attacker knows the ephemeral private decryption key, then the label we designed is corrupt, from which we deduce that one of the two aforementioned possibilities happened.

Label for signed ephemeral key. We design two labels, ℓ_{eph} is corrupt when the attacker compromised the ephemeral key state, and ℓ_{sig} is corrupt when the attacker compromised the signature key state *before* we verified the signature. Using these two labels, we can prove that after signature verification, the following label relation holds:

$$\ell_{eph} \sqcup \ell_{sig} \gtrsim \mathcal{L}(\texttt{eph\_key})$$

Indeed, if the signature key state is compromised before we verified the signature, then $\ell_{sig} \gtrsim \top$, we conclude by transitivity via $\ell_{eph} \sqcup \ell_{sig} \gtrsim \ell_{sig}$[10] and $\top \gtrsim \mathcal{L}(\texttt{eph\_key})$.[11] If the signature key state is *not* compromised before we verified the signature, then the signature must have been computed by an honest participant, hence some corresponding signature predicate must hold on eph_key. By designing the signature predicate such that it implies $\ell_{eph} \gtrsim \mathcal{L}(\texttt{eph\_key})$, we conclude by transitivity via $\ell_{eph} \sqcup \ell_{sig} \gtrsim \ell_{eph}$.[12] We now describe how we are able to define ℓ_{eph} and ℓ_{sig} with our new labeling framework.

10: via Flow-Join-Eq-Elim in Figure 2.14

11: via Flow-Public in Figure 2.14

12: via Flow-Join-Eq-Elim in Figure 2.14

Precise state. In principal_label p (Figure 3.12), the corruption predicate says "there exist a state of principal p that is compromised by the attacker", but this does not express what this state actually contains. We can now build more precise labels, as shown in Figure 3.16: signature_key_label further says that this state contains a signature key (using the tag "SignatureKey"), and that this signature key corresponds to some given public verification key vk. This allows us to define ℓ_{eph}. To define ℓ_{sig}, we still need to deal with temporal properties.

```
let signature_key_label
  (p:principal) (vk:bytes) = {
    is_corrupt = (λ tr →
      ∃ st.
        exists_compromise_event tr st ∧
        st.who == p ∧
        st.tag == "SignatureKey" ∧
        sk_to_vk st.content.sk == vk
    );
}
```

Figure 3.16: Precise label for a signature private key of principal p corresponding to the public verification key vk.

Temporal properties. We express temporal properties with a new label combinator: guard_event l e, which is corrupt when the label l is corrupt before the event e was logged. Then, if l is the label for a signature key and e is an event which is logged after verifying the ephemeral key signature, we define ℓ_{sig} = guard_event l e, which correctly express a temporal property: it is corrupt when the attacker compromised the signature key before we verified the ephemeral key signature. The label combinator guard_event is itself built using three different combinators, that we now discuss.

The guard label combinator. The first building block of guard_event is the guard label combinator (Figure 3.17): guard l1 l2 is corrupt when there exist a time in the past where l1 was corrupt but l2 was not corrupt. This label combinator expresses a temporal property, and used with the event

```
let guard (l1 l2:label) = {
  is_corrupt = (λ tr →
    ∃ tr_before.
      tr_before ≤ tr ∧
      l1.is_corrupt tr_before ∧
      ¬(l2.is_corrupt tr_before)
  );
}
```

Figure 3.17: Implementation of guard.

label (see next paragraph) it can be used to express temporal properties with respect to an event (e.g., verification of a signature).

Event labels. The second building block of guard_event is the event label (Figure 3.18): event_label p e is corrupt when participant p has logged the event e. We can combine it with guard, to almost obtain guard_event: indeed, guard l (event_label p e) is corrupt when l was corrupt before participant p logged the event e. The last problem to solve before obtaining guard_event is the fact that participant p appears in this label. Indeed, in security proofs with DY*, it is generally useful that different participants associate the same label for the same piece of data (here, the label for the ephemeral private key). This cannot happen if the participant p who is checking the signature appears in the ephemeral private key label.

```
let event_label (p:principal) (e:event) =
{
  is_corrupt = (λ tr →
    event_was_logged tr p e
  );
}
```

Figure 3.18: Implementation of event_label.

Participant-independent event label. To solve this problem, a key insight is that to conduct an active attack by forging the signature, the attacker must compromise the signature key before *any* participant check this signature (which is in particular before participant p checks this signature). To implement this insight, we could modify event_label to be corrupt when any participant logged some event. However, a more elegant solution is to keep our general event_label and use it inside a join over all possible participants: this would correspond to the fact that any participant logged this event. In the original DY* [43], we can only do joins over two labels, and we can repeat it to do joins over a finite amount of labels. However, we cannot do it for an infinite amount of labels: with our new labels, it is now possible to do.

Unbounded join and meets. Looking back at Figure 3.12, join between two labels corresponds to the logical or. To implement an unbounded version of join, we therefore use the unbounded version of the logical or, which is the logical exists, as shown in Figure 3.19. Unbounded meet is implemented similarly with a logical forall.

```
let big_join (f:α → label) = {
  is_corrupt = (λ tr →
    ∃ (x:α) . (f x).is_corrupt tr
  );
}
```

Figure 3.19: Implementation of unbounded join.

Conclusion. We designed new labels for TreeKEM, and in the process we designed several label combinators of general purpose (guard, event_label, big_join). These label combinators are of general interest for DY* users, for this reason we now upstreamed them into DY*'s standard library.

3.2.5 Discussion

This work has been motivated by the security proofs of TreeKEM (Chapter 6), where we wanted to express what kind of state was compromised, and express temporal properties on the labels, as we have shown in §3.2.4. The work done in §3.2.2 allowed us to more easily extend labels with the features we wanted, however modifying the core of DY* each time a user has new labeling needs felt inconvenient. We solved this problem in §3.2.3, where the user can define new labels within their proofs and therefore don't need to change the core of DY* to define new kind of labels. In the end, we think this new labeling framework was a crucial technique to conduct our security proofs of TreeKEM, and will benefit future DY* users.

Limitations. During all our attempts to hack with this new labeling framework, we only found one limitation: in the new definition of can_flow (Figure 3.11b), we quantify on all future traces, and not on all *reachable* future traces, or all future traces that *satisfy the trace invariant*. As a result, some

can_flow relations are impossible to prove, although they would be provable if can_flow were to quantify on all future traces that satisfy the trace invariant. For example, if the trace invariant impose that participants log the event Event2 only after they have logged before the event Event1, we would hope that can_flow tr (event_label p Event1) (event_label p Event2) because in all future reachable traces tr_later, we can prove that if have is_corrupt tr_later (event_label p Event2), in that case we must also have is_corrupt tr_later (event_label p Event1). Unfortunately, we cannot prove this because can_flow also quantifies on unreachable traces where this implication does not hold. It is not easy to modify the definition of can_flow to account for the trace invariant, because the trace invariant uses can_flow, so there is a chicken-and-egg problem. Therefore, we did not find how to alleviate this limitation yet.

History of the framework. Now that we are used to this new labeling framework, it seems that the new definition of can_flow introduced in §3.2.2 is the natural definition of can_flow, and that the new definition of labels introduce in §3.2.3 is the natural way to define labels. However, it turns out it was not straightforward at the beginning.

My first attempt to better understand labels was to create the same construction as the original can_flow (Figure 3.11a), but step by step: start with one label per principal, then extend it by adding the secret and public label (which are minimum and maximum labels with respect to can_flow), then extend it by adding joins and meets, thereby putting a lattice structure on labels.

Then, I wanted to add a new type of labels, which could only be corrupt before some timestamp: this was a preliminary attempt at introducing labels with temporal properties as we have done in §3.2.4. To extend the label definition, I had to create new formulas for can_flow: when does the new label flows to joins and meets, when does joins and meets flow to the new label, etc. This was quite a tricky business, the first aha! moment was when I realized that the (old) can_flow formulas were all under-approximations of the new can_flow definition (which, at the time, was not a can_flow definition, and not written anywhere) where is_corrupt tr l was instead can_flow tr l public (there was no notion of "a label is corrupt" other than flowing to public). Using this insight, I filled my whiteboard with all the new formulas I should use to extend the (old) can_flow definition to support this new temporal label. The day was over, I went back home, proud of my work. Then, while doing something else during that evening, I had a second aha! moment: part of my brain working in the background told me "hang on, this thing we are under-approximating, couldn't this be *the* definition of can_flow?". The next day I tried it, and could prove all properties we proved on the old can_flow. Furthermore, adding support for this new temporal label proved to be very easy, certainly much easier than the attempt of the day before which involved filling my whiteboard with mathematical formulas.

I discussed this potential new can_flow definition with the other people working on DY*, but the feedback was mixed. The new definition seemed more elegant, but it was unclear whether there were any hidden defects with this new definition. The main concern was that the old can_flow definition is computable: hence given two concrete label, we can compute whether can_flow returns true or false. It is not the case of the new can_flow definition, since there is a ∀ involved. In the end, we agreed to use it because the actual definition of can_flow is hidden from users anyway, they are only using the theorems we proved on can_flow, therefore

because we proved the same theorems, nothing really changed for the DY* users. At this stage, using this new definition was in practice mostly about aesthetics, however it radically improved my intuition of what it means for a label to flow to another; I tried to convey this intuition when explaining labels in §2.2.6.

Later on, when I started to think seriously about starting the security proofs for TreeKEM (Chapter 6), I wanted to have more expressive labels. Indeed, the current labels could talk about the compromise of a specific participant, or of a specific session of a participant (where the session is identified by an integer provided by DY*). The labels could not express things such as "a long-term signature key of this participant", or as "an ephemeral key used in the session described by this group identifier" (where the group identifier is given by the protocol, here MLS). I tried to hack the label framework to be able to express such things, and do it in a way that is somewhat general, does not feel super specific to MLS, and could benefit other DY* users. After spending a while tweaking the labels, I realized that all I was doing was to create new trace predicates, and try to parametrize them in a way they would be as generic as possible. This was the third aha! moment: instead of hacking the label type to encode more and more general trace predicates, what if we defined labels directly as trace predicates? As we discussed in §3.2.3 there were two challenges to do that: keeping the trace type positive and making labels erasable. I quickly found the solution to make the trace type positive, although it took a while to accept I could not find a better solution. Making labels erasable was more difficult, I already thought about this in the past but did not find any satisfying solutions, they all came with drawbacks making DY* a bit more tedious to use. However, in the past, making labels erasable was mostly for aesthetics, but with the possibility to define labels as trace predicates, we now had a good reason to make labels erasable and accept these drawbacks, because they are now trade-offs. This new definition of labels offered a second round of improving the intuition what labels are, and what it means for them to be corrupt; I tried to convey that when explaining labels in §2.2.6.

3.3 Making labels erasable

We saw in §2.2.4 when participants sample fresh randomness, they choose what is its label and usage, and that in the type for symbolic bytestrings (bytes), the constructor for fresh randomness contains a variety of information related to this randomness: its time of generation, its length, and also its label and usage (see Figure 3.20). This information can then be retrieved in the function get_label. Although storing this information there is convenient, it means that when sampling fresh randomness, the random bytestring we obtain depends on the label we chose. That is, if we sample a random bytestring, go back in time to re-sample this random bytestring, this re-sampled bytestring would be identical to the first one if we chose the same label and usage, but would be different if we chose different label or usage. This raises a philosophical question: *should this be the case*? We think the answer is: *no*. Random bytestring should only depend on its time of generation and length, but should not depend on its label or usage. Indeed, labels and usages are only proof techniques, they should play no role in computations. In other words, it means they should be *erasable* from computations.

```
type trace_entry =
  | RandGen:
    nat → label → usage →
    trace_entry
  ...

type bytes =
  | Rand:
    timestamp →
    nat → label → usage →
    bytes
  ...

let get_label b =
  match b with
  | Rand _ _ lab _ → lab
  ...
```

Figure 3.20: It is convenient to store the label of a fresh random bytestring in its corresponding constructor in bytes, so that we can easily retrieve it in get_label.

Throughout the history of DY*, there were several attempts to make labels

erasable, but none of them succeeded. In this section, we show how we managed to make labels erasable. We explored a variety of solutions toward this goal, however each of them came with drawbacks. These drawbacks would be unacceptable if it were for the sole goal of resolving this philosophical issue, however in §3.2.3 we saw that it was crucial to erase labels (or at least, remove them from the bytes type) to enhance their expressivity, transforming these drawbacks into trade-offs. Therefore, we chose the solution with the most acceptable drawback. Since then, we conducted a security proof for a large protocol (see Chapter 6) which showed this drawback is perfectly acceptable, and actually rarely shows up in practice, as we will explain.

To make labels erasable, we must prevent them from appearing in the bytes type: indeed, in Figure 3.20, two Rand constructors are equal when they have the same time of generation, length, label and usage, which means a program that checks the equality of two bytes must be able to check the equality of labels, which cannot exist when labels are erasable. Hence, with erasable labels, the bytes type should look as in Figure 3.21. Furthermore, it is sufficient to prevent labels from appearing in the bytes type: it is the only place where labels play a role in computations. Therefore, our only focus in this section will be to repair DY* after changing the bytes type to the one showed in Figure 3.21.

```
type bytes =
  | Rand:
    timestamp → nat →
    bytes
  ...
```

Figure 3.21: The new bytes type, without label or usage: just time of generation, and length.

When the bytes type is changed to the one in Figure 3.21, get_label cannot be written as in Figure 3.20. We will see three ways to repair it (and the associated drawbacks): the solution we chose, if we add the trace as a parameter of get_label (§3.3.1); and two solutions we didn't choose, if we decide to keep get_label with no extra parameter (§3.3.2), if we decide to use instead a function has_label (§3.3.3). We then finally discuss (§3.3.4).

3.3.1 If get_label depends on the trace

One solution is to look for the label in the trace: indeed, it is present in the RandGen entry, which is fine even when labels are erasable because we never need to compute whether two trace entries are equal. We give the new code for get_label in Figure 3.22, however it is not clear what label to return when the bytestring is malformed, that is, when its generation time points in the future (it has not been generated yet) or when its generation time points to a trace entry that does not correspond to randomness generation. We explore a few ways to deal with this issue.

```
let get_label (tr:trace) (b:bytes) =
  match b with
  | Rand time _ →
    if time < length tr then
      match get_entry_at tr time with
      | RandGen_ lab _ → lab
      | _ → // ???
    else
      // ???
  ...
```

Figure 3.22: Modification of get_label to fetch the label in the trace. We don't know what to do in some places, though (annotated with // ???).

Returning a garbage label. One solution would be: return a garbage label (e.g. public, or anything really) when the bytestring is malformed. This would work to obtain a valid definition for get_label, but induces another problem: the label returned by get_label may change when the trace grows (in the case the time of generation points in the future). This is not desirable: recall that every property in DY* is designed to be monotonic in the trace, that is, when they are true with respect to some trace, they stay true when the trace grows (see Figure 2.10). If the label returned by get_label were to change when the trace grows, every property that involves get_label (e.g. the bytes invariant) will fail to be monotonic.

Enforcing well-formed bytestrings. Another solution would be: design a predicate that captures "well-formedness" of bytestrings (see Figure 3.23), and add a pre-condition to get_label to ensure we only call it on well-formed bytestrings (this is what {bytes_wf tr b} means in Figure 3.23). Because we only use get_label on bytestrings that are well-formed, it means that get_label doesn't have to deal with the cases when the bytestring is malformed (annotated by *// ???* in Figure 3.22), and the label returned by get_label will not change when the trace grows. However, this creates another problem: each time DY* users call get_label, they will need to prove that the bytestring is well-formed. This is not desirable, because it introduces some additional friction each time DY* users call get_label.

```
let bytes_wf (tr:trace) (b:bytes) =
  match b with
  | Rand time _ →
    time < length tr ∧
    RandGen? (get_entry_at tr time)
  ...

let get_label
  (tr:trace) (b:bytes{bytes_wf tr b}) =
    ...
```

Figure 3.23: Well-formedness predicate on bytestrings.

Our solution. We combine the two approaches above: we return a garbage label when the bytestring is malformed, and prove that if the bytestring is well-formed, then its label don't change when the trace grows (see lemma get_label_later in Figure 3.24). With this approach, DY* users don't need to prove that the bytestring is well-formed when they use get_label, only when they want to reason on the output of get_label will they need to prove well-formedness. Thankfully, this is in general easy to prove, because every bytestring in a protocol execution satisfies the bytes invariant, which we prove to imply well-formedness (see lemma bytes_invariant_implies_bytes_wf in Figure 3.24). Furthermore, we design SMT patterns (an F* concept we describe in §3.5.4) so that these lemmas are applied automatically, making these well-formedness conditions completely transparent to DY* users.

```
val get_label_later:
  tr1:trace → tr2:trace → b:bytes →
  Lemma
  (requires bytes_wf tr1 b ∧ tr1 ≤ tr2)
  (ensures
    get_label tr1 b == get_label tr2 b
  )

val bytes_invariant_implies_bytes_wf:
  tr:trace → b:bytes →
  Lemma
  (requires bytes_invariant tr b)
  (ensures bytes_wf tr b)
```

Figure 3.24: Various lemmas to reason with well-formedness.

There is only one place where we cannot use this trick, because we don't know that bytestrings satisfy the bytes invariant: when we are defining the bytes invariant itself. For example when DY* users define what is the signature predicate, they are asked to prove that the signature predicate is monotonic (a requirement so that the bytes invariant is also monotonic); if the signature predicate involves get_label they will need to prove that the inputs of get_label are well-formed. Thankfully, this is in practice easy to prove, all the examples we have seen require only one or two straightforward lines of proof.

3.3.2 If get_label does not depend on the trace

Another design choice would be to say that we want to do minimal changes to get_label, hence do not introduce the trace as one of its parameters. We did not choose this approach because it comes with two prohibitive drawbacks.

Indirect labels. Then get_label cannot return the label of the bytestring, since it is not present in any of its parameters. The solution is to create a new type of label that adds an indirection, the Indirect label in Figure 3.25. Then, using the formalism of §3.2.2, we modify the function is_corrupt to handle this new type of label; on Indirect labels is_corrupt will simply call itself recursively on the label pointed by the timestamp. In practice, we cannot define is_corrupt exactly as in Figure 3.25 because this construct may loop indefinitely on malformed traces, but we can create a function is_corrupt with the same semantics by using fuel.

```
let get_label b =
  match b with
  | Rand time _ → Indirect time
  ...

let is_corrupt (tr:trace) (l:label) =
  match l with
  ...
  | Indirect time →
    time < length tr ∧ (
      match get_entry_at tr time with
      | RandGen _ lab _ →
        is_corrupt tr lab
      | _ → ⊥
    )
```

Figure 3.25: New type of label to represent an indirection, and corresponding modification of is_corrupt.

Equivalence instead of equality. The first drawback of this approach is that we cannot, in general, express equality on labels. Indeed, suppose we generate a fresh random bytestring b with label lab, we cannot prove

get_label b == lab because get_label b is now an Indirect label; we can however prove that get_label b and lab are *equivalent*, that is, they both flow to each other. This percolates throughout every use of labels in security proofs, meaning that we can never prove that two labels are equal (i.e. using F*'s built-in equality), we can only prove that they are equivalent.[13] This is unfortunate, because proofs using F*'s built-in equality enjoy a lot of automation, and we found that proving with a custom equivalence notion instead was difficult, especially when dealing with congruence, for example proving that if $\ell_1 \simeq \ell_1'$ and if $\ell_2 \simeq \ell_2'$ then $\ell_1 \sqcup \ell_2 \simeq \ell_1' \sqcup \ell_2'$. Although this fact is trivial when using F*'s built-in equality, proving it with a custom notion of equality showed to introduce a lot of friction, and we found this additional friction to be prohibitive.

13: we cannot rely on an extensionality property there, because the labels are equivalent on *this trace*, but not *all traces*: indeed there are (unrelated) traces where the two labels are not equivalent (e.g. the trace where we chose a different label for the random bytestring b)

Incompatibility with §3.2.3. The second drawback of this approach is that it is incompatible with our new labeling framework described in §3.2, especially the generalization of labels to make them more expressive in §3.2.3. Indeed, in §3.2.3 we want to define labels as trace predicates (that tells whether they are corrupt), but recall that to keep the trace type positive, labels are actually predicates on a view of the trace where labels are removed (i.e. replaced with the unit type). It means that we cannot implement the indirect label in this new labeling framework: indeed, when fetching the new label to call is_corrupt recursively, this label in the RandGen entry was removed.

3.3.3 If get_label becomes has_label

A final design choice would be to say that get_label is not the right notion to work with labels, and that we should instead rely on has_label (of type depicted in Figure 3.26). We worked toward this direction, but found out that this approach had the exact same drawbacks as §3.3.1, in addition to making the code related to labels more convoluted.

```
val has_label:
  trace → bytes → label →
  prop

let msg_can_flow_key_1 tr msg key =
  ∀ msg_label key_label.
     has_label tr msg msg_label ∧
     has_label tr key key_label ⟹
     can_flow tr msg_label key_label

let msg_can_flow_key_2 tr msg key =
  ∃ msg_label key_label.
     has_label tr msg msg_label ∧
     has_label tr key key_label ∧
     can_flow tr msg_label key_label
```

Figure 3.26: Type of a function has_label.

Well-formed bytestrings. When a bytestring is malformed (as defined in §3.3.1), the only sensible choice is that it has no label, that is, has_label tr b lab is always false. However, when b is well-formed, we can prove ∃ lab. has_label tr b lab. In short, being well-formed is equivalent to having a label. This fact is compatible with the monotonicity of has_label, it however causes a problem when using has_label with can_flow.

Lifting "can flow". Using the new notion of has_label, we need to translate our previous use of get_label into use of has_label, for example can_flow tr (get_label msg) (get_label key). We give two such attempts in Figure 3.26), but none of them is satisfactory. In msg_can_flow_key_1, if key or msg is malformed (with respect to tr) then has_label will be false, hence the implication will be true. When the trace grows, if key and msg become well-formed, the condition might become false, meaning that msg_can_flow_key_1 is not monotonic when the trace grows. We do another attempt in msg_can_flow_key_2, however to prove it we need to prove that both key and msg have a label, therefore that they are well-formed.

Therefore, we can see that our attempts at using can_flow with has_label bring back the same well-formedness conditions as in §3.3.1. We could hope to circumvent this issue, but taking a step back, we realized there is no hope to do so.

Set of labels. What we have done in the last paragraph is to lift our definition of can_flow on labels to label → prop, that is, sets of labels. In fact, we also need to lift operations on labels, such as joins and meets. With sets of labels being so ubiquitous, it is natural to define label_set = label → prop. Then, note that has_label has type trace → bytes → label_set, which is very similar to the get_label type we defined in §3.3.1. This explains why we hit the same well-formedness problem we had with this design choice.

In summary, moving from get_label to has_label is effectively a convoluted way to add the trace as a parameter of get_label, which is what we do directly in §3.3.1.

3.3.4 Discussion

The journey through the erasing of labels was certainly convoluted, and involved many trials and errors. At the beginning, working on this was mostly a distraction, to resolve the philosophical issue that labels *shouldn't* play a role in computations. Because each approach had drawbacks, we aborted the erasing of labels and the various attempts stayed as unmaintained git branches and notes describing the approaches and associated drawbacks.

Our new labeling framework (§3.2.3) acted as a catalyst: erasing label was crucial to improve expressivity of DY*. Initially, we were not sure how much annoying would be the drawback we chose (having to deal with bytestrings well-formedness), but we decided to accept it because the impact on small examples was reasonable. Since then, we conducted a security proof for a large protocol (TreeKEM, see Chapter 6) which involves a fair amount of labels, and confirmed that this drawback is perfectly acceptable, and actually rarely shows up in practice.

3.4 Making key usage an invariant

We described in §2.2.7 the concept of *usage of a key*, written $\mathcal{U}(\square)$, and saw in the *bytes invariant* (§2.2.8) that knowing what is the usage of a key is a precondition to safely use it with cryptographic functions (e.g. when encrypting a message with a key, we must prove that the key has the usage of an encryption key, as shown in §2.2.8, in particular Figure 2.20).

We found that this requirement on the key usage is not an invariant that can be nicely propagated throughout security proofs. In this section, we explain this issue and show how we solved it by weakening the notion of "get usage" (i.e. the function $\mathcal{U}(\square)$) to obtain a notion that propagates well throughout security proofs.

Usage of a key is uncertain. When doing the security proofs for TreeKEM (Chapter 6) we noticed that we cannot, in general, know *for sure* what is the usage of a key. Indeed, suppose we decrypt a message that contains a key k: how can we obtain information about the $\mathcal{U}(\mathsf{k})$? Maybe if k were signed, then the signature predicate may tell that $\mathcal{U}(\mathsf{k}) = \mathsf{u}$ for some u, however the signature predicate doesn't hold *for sure*: indeed, another possibility is that the attacker managed to obtain the corresponding signature key (e.g. via compromise), in which case the signature key is

publishable. In summary, the signature would ensure that we know either that $\mathcal{U}(\mathsf{k}) = \mathsf{u}$, or that some (unrelated) signature key is publishable.

Toward a notion of "has usage". We will show that we can always prove the better guarantee that we know either that $\mathcal{U}(\mathsf{k}) = \mathsf{u}$ or that k is publishable (i.e. $\mathcal{P}(\mathsf{k})$), in which case we will say that the key k *has usage* u. This notion of "has usage" will propagate nicely throughout the bytes invariant, meaning that it is an invariant of cryptographic protocols, unlike the requirement to know for sure that $\mathcal{U}(\mathsf{k}) = \mathsf{u}$ as imposed by the original DY* [43].

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

"has usage" with encryptions and decryptions. Suppose we decrypt a key k, and we want to prove that k has usage u, that is, $\mathcal{U}(\mathsf{k}) = \mathsf{u} \vee \mathcal{P}(\mathsf{k})$. We can do so by having the encryption predicate ensure that k has usage u: if the encryption predicate holds, we are good, otherwise it means the encryption was computed by the attacker in which case the message (hence k) must be publishable, which also concludes. Therefore, the fact that k has usage u is a property we can transmit from a principal to another via encryption (whether symmetric or asymmetric).

"has usage" with key derivations. Suppose we derive keys using the key k with a Key Derivation Function (KDF). If we know that k has usage u, that is, $\mathcal{U}(\mathsf{k}) = \mathsf{u} \vee \mathcal{P}(\mathsf{k})$, we can deduce that the derived keys also have some usage. Indeed, if $\mathcal{U}(\mathsf{k}) = \mathsf{u}$ then we know the usage of the derived key (as explained in §2.2.7). If instead k is publishable then the derived key is also publishable. This concludes that the derived keys also have some usage.

Weakening "get usage" into "has usage". We showed that we can, in general, prove that keys have some usage, that is, prove that either the key usage is exactly something or that the key is publishable. However, the preconditions to safely use cryptographic functions require that we prove the key usage is exactly something (as shown in §2.2.8, in particular Figure 2.20). In turns out we can safely weaken these preconditions to use our new notion of "has usage". Therefore, this notion of "has usage" is a correct invariant that can be propagated throughout all cryptographic functions.

Discussion. We found these issues when working on the proof of Welcome message processing in TreeKEM. In there, we decrypt (using public-key encryption) a key called the "joiner secret", and use it to derive a bunch of keys used for a variety of things. In the proof, we noticed that we were always performing two proofs in parallel: one proof when we know what is the usage of the joiner secret, and another proof when the joiner secret is publishable. Incidentally, the proof when the joiner secret is publishable relied on the lemmas that cryptographic functions preserve publishability (see Figure 2.18) which were initially not supposed to be used by DY* users, only be lemmas to prove the Attacker Knowledge Theorem (§2.2.10). This new notion of "has usage" merges the two proof paths together and leads to a better user experience. Initially, when we redesigned DY* around this notion of "has usage" we were not sure whether this would be *the* good notion for usage, but since then the proofs of TreeKEM exhibited no problems with "has usage", hinting that it is indeed the good notion to reason with usages. In a way, "k has usage u" still means that k is safe to use as a key with usage u. Indeed, if that is the case because k is publishable, computing cryptographic functions with key k is safe, because it is as if the attacker were doing these computations.

3.5 Quality of life and proof engineering

Along with the several scientific improvements on DY* we have done so far (§3.1, §3.2, §3.4) we also did substantial engineering improvements to DY*. These improvements don't improve scalability of DY*, nor improve expressivity of DY*. However, they reduce needless friction when writing security proofs, allowing DY* users to produce proofs more quickly. Furthermore, although we have no scientific evidence to prove so, we think reducing friction when writing security proofs leads to better mood of the human producing the proof, thereby enhancing its productivity.

3.5.1 Making specifications easier to read

In the original DY* [43], users do security proofs in the style of *intrinsic proofs* (described below). We found that this style of doing proofs had the side effect of making specifications less readable, which made the perspective of using it to prove security theorems on TreeKEM (Chapter 6) unappealing. We therefore did a big revamp of DY* to do *extrinsic proofs* (also described below), initially as a proof-of-concept, which then became the new version of DY* (after discussion with other DY* users, who agreed to migrate).

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Extrinsic proofs. The most natural way to prove properties on program is the following: we have a program, which is a mathematical object, and separately write a mathematical statement on the behavior of the program, followed by a mathematical proof. This is called an *extrinsic* proof: the proof is an artifact external to the program. The main drawback of this approach is that when writing the proof, we often must write the program again within the proof, thereby introducing some duplication.

Intrinsic proofs. We can resolve this issue of program duplication by writing the program and the proofs *at the same time*, in an interleaved fashion, as a single artifact: this is an *intrinsic proof*. Because the program code is deduplicated, the proof is reduced to only its "interesting" parts. On simple programs when the proof is mostly straightforward, the proof has few "interesting" parts, therefore there are few lines of proofs compared to lines of program code.

Specification readability. However, when the proof is subtle, it will have many "interesting" parts, hence the lines of proofs over lines of program code ratio will get worse, thereby submerging the program code with lines of proofs, making the program code less easy to read. This may be fine, for example if we were to prove that an optimized implementation adheres to a clean, high-level specification (e.g. as done in HACL* [71]). However, in DY*, this is not fine, because we are doing proofs *on the specification itself*, and the specification must be easy to read: indeed, specifications must be carefully reviewed to ensure they correctly model the protocol under scrutiny. In our security proofs of MLS components (Chapter 5 and Chapter 6) we will use another technique to increase our confidence that we correctly specify the protocol: by executing it concretely against test vectors. This can only be done only when there exists test vectors, hence is complementary to having easy-to-read specifications.

[71]: Zinzindohoué et al. (2017), *HACL*: A verified modern cryptographic library*

3.5.2 Library of reusable components

In §2.2, §3.2 and §3.4 we described what we call the "core" of DY*: it is the strict minimum to prove security properties on cryptographic protocols. However, only using this "core" to prove security properties does not offer a great user experience. To improve the user experience, we build a library of reusable components that makes a better user experience. Nowadays, it is unthinkable to use DY* without using each of the library components: even the simplest examples (e.g. SignedDH in §2.3) rely on them.

Modular invariants. We described in §3.1 a technique to define protocol invariants modularly. It turns out that this is not part of the "core" of DY*: indeed, this technique does not improve expressivity of DY*, it only makes it easier to use. Therefore, it is part of the DY* library.

Modular invariants on key usage. We further provide instantiations of the modular invariants technique to dispatch to local invariants depending on the *usage* of the key (as shown earlier in Figure 3.2). Indeed, this dispatch is generally the first one DY* users do when creating their invariants as decision trees, hence in the DY* library, we provide instantiations of the modular invariant technique on key usage for each cryptographic primitive.

Comparse glue. Cryptographic protocols pervasively rely on *message formats*: to send and receive messages on the network, to sign and verify signatures, to store and retrieve state, etc. Furthermore, security proofs may rely on properties of these message formats: for example, security proofs involving signatures generally rely on the serialization function to be injective (or that its corresponding message format is *non-ambiguous*, in Comparse lingo, see Chapter 4). We provide glue code with Comparse in the DY* library, which helps reduce a lot of boilerplate related to message formatting.

States and protocol events with high-level types. In the core of DY*, states and protocol events contain bytestrings. However, DY* users typically want to use them with high-level types, therefore need to handle parsing and serialization by hand. To help DY* users, we provide higher-level interfaces for states and protocol events that work with high-level types. Under the hood, these higher-level interfaces use Comparse to handle parsing and serialization transparently.

Reusable states. Cryptographic protocols often rely on a state to store long-term private keys, and on a model of a Public-Key Infrastructure (PKI) which associates public-keys to participant identities. To alleviate DY* users having to write this kind of state for each protocol proof, we provide generic reusable states for private keys and PKI. Every DY* example rely on these states, the example we showed in §2.3 is no exception: indeed, we mentioned there that we relied on these two states.

3.5.3 Typeclasses to reduce verbosity

In the original DY* [43] there was a lot of verbosity when using functions or predicates related to trace invariant, such as `get_label`, `bytes_invariant` or `trace_invariant`. We managed to cut this verbosity using F*'s typeclass mechanism.

Hierarchy of invariants. As we have seen in §2.2, the protocol invariants in DY* depend on user-provided invariants: the state invariant, the signature predicate, etc. As we show in Figure 3.27, we do this in F* by adding a parameter to every function and predicate related to protocol invariants. Furthermore, we define the user-provided invariants as a hierarchy, so that each function and predicate only depend on what they need: for example, get_label does not need to know the state invariant, hence the state invariant does not belong to the parameter of get_label. Conversely, the state invariant must be able to use bytes_invariant (e.g. to say that its content must satisfy the bytes invariant), and bytes_invariant must be able to use get_label (e.g. to say that the label of a message must flow to the label of a key). This justifies that DY* needs to define the type of user-provided invariants in several layers: the innermost layer to parametrize get_label, the middle layer to parametrize bytes_invariant, and the outermost layer to parametrize trace_invariant.

```
type crypto_usages = ...

val get_label:
  crypto_usages →
  bytes → label

type crypto_invariants = {
  usages: crypto_usages;
  ... // (signature predicate etc)
}

val bytes_invariant:
  crypto_invariants →
  trace → bytes → prop

type protocol_invariants = {
  crypto_invs: crypto_invariants;
  ... // (state and event invariant)
}

val trace_invariant:
  protocol_invariants →
  trace → prop
```

Figure 3.27: The hierarchy of protocols invariants, and various invariant-related functions.

Verbosity. Each time we use get_label, bytes_invariant or trace_invariant, we must feed them the user-provided invariants as a parameter, this creates a lot of friction. To illustrate the friction, let's suppose we are calling the function get_label with a bytestring b, and see how we obtain its parameter.

- If we are writing a signature predicate, we have as a parameter usages of type crypto_usages therefore we write get_label usages b.
- If we are writing a state predicate, we have as a parameter crypto_invs of type crypto_invariants, we write get_label crypto_invs.usages b.
- If we are proving that a protocol step preserves trace_invariant, we have a parameter invs of type protocol_invariants therefore we write get_label invs.crypto_invs.usages b.

Therefore, each time we use get_label, the user needs to answer two questions: first, what is the name of the user-provided invariants in the current context, second, how do we access its enclosed crypto_usages? Furthermore, there is always at most one sensible answer to these two questions: how to obtain crypto_usages is uniquely determined by the type of user-provided invariants we obtained; and in any context we have at most one user-provided invariants: either in the local context (such as bytes_invariant has in its definition access to a generic crypto_invariants in its local context), or in the global context (meaning that the user has defined invariants for its protocol, which must be unique). Therefore, there is always only one sensible way to obtain the parameter of get_label (and friends), which strongly hints that obtaining this parameter could be automatized.

Typeclasses. We (ab)use the typeclass mechanism of F* to cut this needless verbosity. We give an example of how typeclasses are typically implemented in functional programming languages in Figure 3.28, note the similarities with our protocol invariants: every function is parametrized by the typeclass instance, and functions operating higher in the typeclass hierarchy (e.g. double_g) need to recover the lower-level instances. Thankfully, F* comes with a typeclass mechanism that automatically find typeclass instances, which we use to automatically find the parameters of get_label and friends.

The typeclass resolution algorithm may find either *local* or *global* instances. An example of local instance: in double_g, we are calling double that expects an instance of has_add on the abstract type α , the typeclass resolution algorithm will find such an instance in the *local* context, that is, tc.add_tc. An example of global instance: if we call double on an int, the

```
// define a typeclass
type has_add (a:Type) = {
   add: a → a → a;
}

// instantiate a typeclass
let has_add_int: has_add int = {
   add = (λ x y → x+y);
}

// depend on a typeclass instance
let double (tc:has_add α) (x:α) =
   tc.add x x

// define a higher-level typeclass
type is_monoid (a:Type) = {
   neutral: a;
   add_tc: has_add a;
}

let double_g (tc:is_monoid α) (x:α) =
   double tc.add_tc x
```

Figure 3.28: Example code of typeclasses in a functional language that has no native support for typeclasses.

typeclass resolution algorithm will find a typeclass instance in the *global* context, that is, has_add_int.

We use the typeclass mechanism of F* on crypto_usages, crypto_invariants and protocol_invariants. Although typeclasses are generally parametrized by a type (like has_add), they also work without parameters even though it is a bit unusual. As we explained earlier, there is always at most one sensible way to instantiate the parameters of get_label and friends, meaning we can safely discharge this instantiation to the typeclass mechanism of F*.

3.5.4 Good SMT patterns

Security proofs in DY* rely on a Satisfiability Modulo Theories (SMT) solver, because DY* is a framework written in F*, and proofs in F* are performed using an SMT solver. In general, the SMT solver is not able to perform proofs fully automatically, and needs some help from the user, and F* libraries (such as DY*) may use *SMT patterns* to help provide more automation on custom theories (such as the theory of "can flow" ($\gtrsim$) shown in Figure 2.14).

Helping the SMT solver. There are (mainly) two ways to help the SMT solver. The first way is by using the assert construct: the SMT will try to prove the property being asserted, then do the rest of the proof using this asserted property. In other words, assert proves Q by proving $P \wedge (P \implies Q)$ where Q is the goal to prove and P is the asserted property. This effectively guides the proof search of the SMT solver. The second way is by instantiating a lemma that was previously proved and add it as a hypothesis, in other words we prove Q by proving $P \implies Q$ where P was proved in a previous lemma.

Lemma instantiation. In general, lemmas are universally quantified, that is, they are of the form $\forall x_1, \ldots, x_n.P(x_1, \ldots, x_n)$. In practice, this fact is a bit crude to give directly to the SMT solver, which will need to spend time to find appropriate values for the x_i, or may do a bad job at guessing the x_i, thereby flooding its context with various (useless) instantiations of $P(\ldots)$. Therefore, it is better practice to give the values for x_i when instantiating a lemma.

Boring lemmas. Unfortunately, many lemmas are not highly interesting despite being crucial for proofs. For example, consider the lemma on lists that the empty list is a neutral element for concatenation, or in mathematical notation, $\forall l.l +\!\!+ [] = l$. Users may find it inconvenient to provide all the lists on which to instantiate this lemma, as it can be seen as "trivial" and not the interesting part of the proof.

SMT patterns. To solve this problem, SMT solvers propose to instantiate lemmas automatically when some pattern is encountered. In our example on lists, we could ask the SMT solver that each time they see the pattern $l +\!\!+ []$ they automatically instantiate the lemma on the list l. This effectively enhance the automation provided by the SMT solver: users will rarely have to manually instantiate the lemma $\forall l.l +\!\!+ [] = l$. Furthermore, this does not pollute the SMT solver context with useless facts: this lemma will be instantiated only when we concatenate a list with the empty list.

Bad SMT patterns. If instead we were to instantiate our lemma each time we see a list, this would lead to infinite triggering: when the SMT

sees a list l, it would instantiate the lemma l ++ [] = l, which adds a new list in the context (l ++ []) on which we instantiate the lemma again to obtain (l ++ []) ++ [] = l ++ [], and so on. This example is degenerate and would not happen in the real world, still the point remains: SMT patterns should be carefully designed to avoid over-triggering. Furthermore, this careful design must be holistic: several SMT patterns may interact badly with each other, although they would be fine if they were alone.

Designing good SMT patterns. There is an inherent trade-off in the design of SMT patterns: too little, and boring parts of the proofs are not automated, too much, and the SMT patterns over-trigger which may lead to proof instability. Despite its importance, the design of good SMT patterns is still arcane knowledge with no proper documentation, but as a rule of thumb, the designer of SMT patterns must be careful of the new terms created by the instantiation of the lemma (and how they would trigger other SMT patterns), of how much an SMT pattern will trigger in some given context, and how much of these triggering are going to help the proof. We now give some examples.

Injectivity. Consider an injectivity lemma (e.g. injectivity of serialization), written as $\forall x, y. f(x) = f(y) \implies x = y$. Suppose we want to instantiate this lemma automatically, how should we design its SMT pattern? A naive design would be: instantiate this theorem when there is in the context $f(x)$ and $f(y)$. However, this will instantiate quadratically in the amount of $f(\square)$ in the context. Therefore, a better design is to exhibit a g such that $\forall x. g(f(x)) = x$ (this is equivalent to injectivity of f), and instantiate this theorem when there is in the context $f(x)$. When in the context there are $f(x)$ and $f(y)$, the SMT will then know that $g(f(x)) = x$ and $g(f(y)) = y$, hence if $f(x) = f(y)$, the SMT solver can deduce that $g(f(x)) = g(f(y))$ hence $x = y$.

Transitivity. Consider a transitivity lemma (e.g. transitivity of "can flow", see Flow-Trans in Figure 2.14), written $\forall x, y, z. x < y \wedge y < z \implies x < z$. Suppose we want to instantiate this lemma automatically, how should we design its SMT pattern? A naive design would be: instantiate this theorem when there is in the context $x < y$ and $y < z$ (this is an "hypothesis directed" pattern). Now suppose we have as hypothesis a chain $x_1 < x_2 < \cdots < x_n$, and we want to prove that $x_1 < x_n$. This pattern will trigger quadratically: for example $x_1 < x_2$ and $x_2 < x_3$, the pattern triggers and creates $x_1 < x_3$, which will itself trigger SMT patterns, and at the end, the context will contain every $x_i < x_j$ when $i < j$, which is a quadratic amount of facts. A better design of SMT pattern is to instantiate this transitivity lemma when there is in the context $x < y$ and $x < z$ (this is a "goal directed" pattern). When proving $x_1 < x_n$ from our chain of $<$, the context contains $x_1 < x_n$ and $x_1 < x_2$, instantiate the transitivity lemma and add in the context $x_2 < x_n$, which will be used to trigger the pattern again, until $x_1 < x_n$ is proved. At the end, the context will contain every $x_i < x_n$, which is a linear amount of facts. Note that the pattern will also spuriously instantiate the lemma as $x_1 < x_n \wedge x_n < x_2 \implies x_1 < x_2$, thankfully the newly created term ($x_n < x_2$) will not trigger because x_n is never at the left of $<$.

Example on "can flow". A common proof pattern in DY* is the following: suppose we decrypt a message that contains a secret s with a key k, and the encryption predicate tells that $\mathcal{L}(\mathsf{s}) \gtrsim \ell$ for some label ℓ. We know that $\mathcal{L}(\mathsf{s}) \gtrsim \mathcal{L}(\mathsf{k})$ (because a message is always "less secret" than the key used to encrypt it), and that $\mathcal{L}(\mathsf{s}) \gtrsim \ell \vee \mathcal{L}(\mathsf{k}) \gtrsim \top$ (because the encryption predicate

holds unless the key is public). From these facts, thanks to the SMT patterns on the lemmas on $\gtrsim$, the SMT solver can deduce that $\mathcal{L}(\mathsf{s}) \gtrsim \ell$ unconditionally: indeed, if $\mathcal{L}(\mathsf{k}) \gtrsim \top$, we have the chain $\mathcal{L}(\mathsf{s}) \gtrsim \mathcal{L}(\mathsf{k}) \gtrsim \top \gtrsim \ell$. Looking more closely, here is what happens: the context contains $\mathcal{L}(\mathsf{s}) \gtrsim \ell$ and $\mathcal{L}(\mathsf{s}) \gtrsim k$, the SMT instantiate the transitivity lemma on $\gtrsim$ and tries to prove $\mathcal{L}(\mathsf{k}) \gtrsim \ell$, which with $\mathcal{L}(\mathsf{k}) \gtrsim \top$ instantiates the transitivity lemma again and tries to prove $\top \gtrsim \ell$, which instantiates the lemma that $\top$ in a maximum element of the label lattice and concludes.

3.6 Conclusion

In this chapter, we explained the inner workings of DY*, a recent contender in the world of symbolic security proofs, and presented the different improvements we made to DY*, to enhance its expressivity, and improve its usability.

Since the beginning, DY* was designed with in mind the goal to be applicable to large protocols, by using invariant-based modular proof techniques instead of relying on whole-protocol analysis. We will show in Chapter 5 and Chapter 6 that this goal is accomplished by proving security theorems on significant parts of Messaging Layer Security [21].

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

The task of coming up with the security invariants may be seen as an additional burden, an obstacle that obstructs the path toward proving security theorems. Instead, we think the security invariants are interesting on their own: in a way, these distill the reasons *why* the protocol is secure. For example, if you wonder what some signature is achieving: that's easy to answer, simply go check the signature predicate, and you will understand what this signature really means.

Another byproduct of DY*'s security invariants is that they are also an excellent tool to think about cryptographic protocols: although the added value of DY* is to provide machine-checked proofs, working enough with DY* trains the user's brain to think about cryptographic protocols with invariants, which we find is a useful skill outside producing machine-checked proofs with DY*.

Comparse: Provably Secure Formats for Cryptographic Protocols

4

> As someone writing reduction proofs, I always thought "The encodings of tuples and these other high-level types are super-important, but I really want someone else to think about them."
>
> **Chris Brzuska**, *private conversation at Bistrot "Le Pacha"*

This chapter is adapted from the eponymous publication [67], presented at ACM CCS 2023. The text is identical, but was reformatted.

4.1 Introduction

Modern software applications rely on a variety of cryptographic protocols to protect sensitive data as it is transmitted over or stored on insecure media. They use Transport Layer Security (TLS) or Noise when they need secure channels, FileVault or Bitlocker for disk encryption, Signal or Messaging Layer Security (MLS) for secure messaging, Bitcoin or Ethereum for distributed ledgers.

Each of these *protocols* can be described as a sequence of (one or more) high-level *messages* sent between (one or more) *participants*. At each step, a sender takes a message that has a particular meaning in the context of the protocol and encodes it into a bitstring by following a protocol-specific *format*. This bitstring is then protected using some *cryptographic construction*. For example, the sender may encrypt the bitstring to guarantee the *confidentiality* for the message content (and the *privacy* of its metadata); or they may add signatures, message authentication codes (MACs), or zero-knowledge proofs to guarantee the *integrity* and *authenticity* of the message. The output bitstring is then serialized according to a *wire format* before it is sent over the network or stored on some disk. The recipient follows the protocol in reverse, parsing the received bitstring, applying its own sequence of cryptographic operations, and decoding the result to obtain the high-level protocol message that the sender (hopefully) intended to send.

Attacks on Cryptographic Protocols. As a classic example, consider the core of the Needham-Schroeder public-key protocol [57]:

[57]: Needham et al. (1978), *Using Encryption for Authentication in Large Networks of Computers*

$$\begin{aligned} A &\longrightarrow B: \quad \{N_A \| A\}_{PK(B)} \\ B &\longrightarrow A: \quad \{N_A \| N_B\}_{PK(A)} \\ A &\longrightarrow B: \quad \{N_B\}_{PK(B)} \end{aligned}$$

Each participant generates a nonce (N_A, N_B), formats it along with some identity information ($N_A \| A$), and encrypts the resulting bitstring using a public-key. Note that the formatting of the message is done *within* the encrypted payload; any wire-formatting that is done outside the encryption (for example, a header mentioning the sender and recipient) is considered to be under the control of the network adversary, and so is ignored in the analysis of the protocol.

The protocol aims to authenticate two participants (A and B) to each other, and to establish a shared secret (N_B) between them. However, it

has a famous attack, originally found by Gavin Lowe [72], which exploits the fact that the second message does not mention its recipient's name, allowing an attacker to mix messages across two sessions, breaking the security of the protocol. Adding B inside the encryption of the second message prevents the attack.

[72]: Lowe (1996), *Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR*

Interestingly, however, there is another, less-well-known attack on this protocol that relies on a message format ambiguity. Meadows [73] observed that an attacker could take the second message (from B to A) from one session and pass it off as a first message in a new session (seemingly from someone named N_B to A). In essence, the formats of the first two messages are *not disjoint*, since a priori, N_B may well be a valid identifier for a protocol participant.

[73]: Meadows (1996), *Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches*

Meadows finds two attacks that exploit this format ambiguity, and Heather et al. [74] show that Lowe's fix to the protocol does not prevent this attack. To fix the protocol against such format confusion attacks, it is necessary to change the internal formats and remove the ambiguity, say by systematically *tagging* each internal message by a distinct bitstring.

[74]: Heather et al. (2003), *How to Prevent Type Flaw Attacks on Security Protocols*

Formats in Real-World Protocols. Format specifications are ubiquitous in real-world protocols, since agreeing on formats is a necessary precondition to enabling multiple interoperable implementations. Internet standards like TLS 1.3 [75] and MLS [21] devote 20% of their text to describing message formats, relying on a custom language called the TLS presentation language. Other protocols rely on a variety of format description languages to encode cryptographic inputs, including XML [76–78], JSON [79, 80], CBOR [81], Protocol Buffers [82], and ASN.1 [83].

Each format description language aims to make it easier for developers and protocol designers to design new formats and correctly implement serializers and parsers for them. However, the sheer number of these languages should serve as a warning sign of the diversity of formatting requirements and constraints in mainstream protocols. Binary formats like CBOR and Protocol Buffers prioritize conciseness, while text formats like XML and JSON aim for Web-friendly interoperability. The TLS presentation language is specialized for a single family of protocols, while ASN.1 aims to be generic and self-describing. Furthermore, proprietary protocols often use their own custom formats according to their own needs.

Unfortunately, despite the great deal of attention given to describing formats, and although the dangers of format confusions have long been known, cryptographic protocols still get them wrong, resulting in high-profile attacks that continue to be found on a regular basis, both in published standards and in proprietary software.

Format Confusion Attacks. Mavrogiannopoulos et al. [84] describe an attack on TLS 1.2 that relies on a format confusion between the signature inputs used in two kinds of Diffie-Hellman handshakes; the attacker takes a server signature produced in one handshake and uses it to impersonate the server in another handshake. Wallez et al. [68] describe a vulnerability in MLS draft 12, where the inputs to signatures in the TreeSync and TreeDEM components of MLS can be confused for each other.

[84]: Mavrogiannopoulos et al. (2012), *A Cross-Protocol Attack on the TLS Protocol*

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

More recently, Paterson et al. [85] describe a series of attacks on Threema, including a format confusion attack between different encrypted messages in the C2S and E2E sub-protocols. Another attack was recently found on the Matrix protocol, where the inputs to MACs used in two different messages could be confused [86].

[85]: Paterson et al. (2023), *Three Lessons From Threema: Analysis of a Secure Messenger*

[86]: Albrecht et al. (2023), *Practically-exploitable Cryptographic Vulnerabilities in Matrix*

These attacks involve different kinds of protocols, and different cryptographic constructions, but in all cases, the problem can be traced to the incorrect or ambiguous use of formats within cryptographic inputs. These flaws are in the protocol itself, and so cannot be fixed by a clever implementation.

Even finding such format confusion attacks in a large standard like TLS 1.3 or MLS can be a real challenge, since the attacks often involve messages in different sub-protocols, sometimes described in different documents altogether. To systematically find and prevent format confusion attacks in real-world protocols, we need a formal framework for specifying and reasoning about secure formats.

Analyzing Crypto Protocols with Precise Formats. Lowe's attack on Needham-Schroeder and the subsequent fix have been very influential, serving as a motivating example for a whole line of protocol verification tools based on symbolic (or Dolev-Yao) analysis [36], including modern tools like ProVerif [41], Tamarin [42], and DY* [43] that have been applied to real-world cryptographic protocols like TLS, Signal, and Noise.

[36]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

However, the analysis of precise formats remains poorly supported by protocol verification tools and techniques. For example, the default model of symbolic concatenation used in ProVerif, Tamarin, and DY*, fails to find the format confusion attack on the fixed Needham-Schroeder-Lowe protocol even today. While it is possible to extend the algebra of terms to account for some format confusion attacks, one cannot be sure that all the low-level details of the format have been captured.

Pen-and-paper cryptographic proofs (and their mechanized variants in tools like CryptoVerif [38] and EasyCrypt [37]) technically reason about bitstring messages, but in terms of practical format analysis, they do even worse than symbolic tools. Proofs in the computational model of cryptography are much harder than symbolic analyses, so in order to make a security proof feasible, papers routinely disregard any formatting concerns and focus only on the cryptographic steps. As we will see in §4.4, even comprehensively studied protocols like TLS 1.3 have not been properly analyzed for format confusion attacks.

[38]: Blanchet (2007), *CryptoVerif: Computationally sound mechanized prover for cryptographic protocols*

[37]: Barthe et al. (2011), *Computer-Aided Security Proofs for the Working Cryptographer*

Our Work and Contributions. As we have seen, there is a large gap between the academic proofs for cryptographic protocols and real-world protocols with complex internal formats. To close this gap, we need a framework for protocol analysis that can specify and prove bit-level precise formats suitable for all the cryptographic inputs used in a protocol. To ensure that we got the formats correctly, we need to be able to test these formats against published test vectors and interoperable implementations. We then need to be able to automatically, or semi-automatically, search for and prove the absence of format confusion attacks across cryptographic inputs in all the message in all related sub-protocols. Finally, we need to be able to embed our formatting proofs within a protocol analysis framework that can verify the security of protocols.

In this paper, we propose a new framework that addresses these requirements. Our framework, called Comparse, uses game-based cryptographic assumptions to establish the set of requirements that formats must obey for their usage to be secure. Because our usage restrictions encompass several classes of attacks, we come up with criteria over formats that rule out not only confusion attacks, but also other well-known attacks. Comparse is implemented and embedded within the F* programming language and verification framework. We demonstrate its expressiveness by using it to

specify and analyze all the formats used in large Internet standards like TLS 1.3 and MLS, as well as classic protocols like Needham-Schroeder. We show how our framework allows us to guide the design of new variants of TLS like Compact TLS 1.3 (cTLS). Finally, we show how our framework is embedded within the DY* protocol verification framework and can be used to verify the security of cryptographic protocols while accounts for bit-level precise formats.

Ours is the first formatting framework that is embedded within a protocol security analysis tool. We provide the first formal proof of correctness for the formats of cTLS, an emerging standard for IoT, and we close an important gap in prior analyses of TLS 1.3. Although our framework does produce reference implementations of serializers and parsers for our formats, this is primarily meant for testing our specification, not as production-ready code. Producing efficient, zero-copy parsers for protocol formats is not our goal.

Outline. §4.2 provides a high-level overview of secure formats and establishes the properties they should obey via a new flavor of cryptographic games. §4.3 describes a formalization and implementation of our format analysis framework, Comparse in F*. §4.4 describes TLS 1.3, cTLS, examines the gap between the published security proofs of these protocols when deployed in parallel with each other, and shows how Comparse can address those. §4.5 shows how Comparse is integrated into DY*. §4.6 discusses our results, their impact, and their limitations. §4.7 briefly describes related work, and §4.8 concludes.

4.2 The Essence of Secure Formats

Real-world formats in protocols like TLS are described in a combination of custom language, comments, and English prose. This has made their comprehensive study difficult, and has resulted in several protocol attacks that leverage *design* flaws in these formats.

To address these shortcomings, this section introduces a formal notion of *message formats* and properties over those message formats, which form the foundation of our security analysis. In §4.3, we shall see how this notion is formalized in a proof assistant.

Throughout this section, we define formats for objects that are represented as bytes. This means that our formats are not just for messages sent over the wire, but also apply to cryptographic inputs (for signatures, MACs, transcript hashes, etc) or any protocol session state or key material that is stored in some binary format.

4.2.1 Formally Defining Message Formats

We use $\mathbb{B}$ to denote the set of byte sequences (also known as "bytestrings"), and a byte is a value in $[0, 255]$. We write ε for the empty bytestring, and $b_1 + b_2$ for the concatenation of two bytestrings. We write literal bytestrings in hexadecimal notation; for example, `c0ffee` is a bytestring of length 3.

Message Format. A message format for a type M is a relation $\rightleftarrows^M$ between M and $\mathbb{B}$. The index M over $\rightleftarrows$ disambiguates relations when there are several types M, M' involved. There may be several relations

for a given M, but we only ever manipulate one such $\rightleftarrows^M$ in any given context.

If we pick $M = \mathbb{B}^2$, and define the message format $(b_1, b_2) \rightleftarrows^M b$ to be e.g. $\exists b_0 \in \mathbb{B}.b_0 + b_1 + b_2 = b$, then the following three properties hold: $(\texttt{c0}, \texttt{ffee}) \rightleftarrows^M \texttt{c0ffee}$, but also $(\texttt{c0ff}, \texttt{ee}) \rightleftarrows^M \texttt{c0ffee}$, and $(\texttt{c0ff}, \texttt{ee}) \rightleftarrows^M \texttt{feedc0ffee}$.

Albeit simplistic, this example warrants two observations. First, we define our notion of format without any reference to a parser or a serializer. In our view, a format is defined as a relation independently of any concrete encodings. Second, this sample format enjoys almost no property of interest: among the many issues with this format, we remark that a given bytestring may correspond to multiple elements in M, and conversely, that an element in M may be represented by several bytestrings.

Serializers and Parsers. Naturally, designers and implementers do not think in terms of logical predicates relating M and $\mathbb{B}$, but rather in terms of concrete formats defined by parsers and serializers. A serializer for a message format $\rightleftarrows^M$ is a function $\text{serialize}_M : M \to \mathbb{B}$ such that (correctness) $\forall m \in M.m \rightleftarrows^M \text{serialize}_M(m)$. A parser for a message format $\rightleftarrows^M$ is a function $\text{parse}_M : \mathbb{B} \to M \cup \{\bot\}$, such that (completeness) $\forall b \in \mathbb{B}.(\exists m \in M.m \rightleftarrows^M b) \implies \text{parse}_M(b) \neq \bot$, and (correctness) $\forall b \in \mathbb{B}.\text{parse}_M \neq \bot \implies \text{parse}_M(b) \rightleftarrows^M b$.

We note that from completeness, it follows that a parser returns $\bot$ (meaning the input bytestring b is malformed) if and only if there no element of M is in relation with b (meaning b cannot be parsed).

Induced Message Format. Given two functions $\text{serialize}_M : M \to \mathbb{B}$ and $\text{parse}_M : \mathbb{B} \to M \cup \{\bot\}$, we define their induced message format on M as: $m \rightleftarrows^M b := m = \text{parse}_M(b) \vee \text{serialize}_M(m) = b$. This is the smallest message format for which parse_M is a parser and serialize_M is a serializer.

The induced message format relates the programmer-centric, concrete view (a format is defined by its parser and serializer) with the security-centric, more abstract view (a format is a relation between messages and bytestrings). Hence, it shows that our abstract formats can still be defined using a concrete parser and serializer.

An advantage of our format-centric view is that it allows us to state properties on a single concept (the format, i.e. a mathematical relation), rather than stating two separate properties (on the parser, and the serializer), and having to account for implementation details (such as the possibility of parsing failure).

Furthermore, as we will see shortly, we show that security properties on an induced message format can be turned into more familiar properties over the underlying parser and serializer, as is done in our implementation of Comparse. There is therefore no lack of expressivity, nor awkwardness, in adopting formats as our central concept for which core notions are defined.

4.2.2 Properties of Message Formats

Leveraging the definitions above, we start with two *security* properties: non-ambiguity, and representation unicity. Failure to exhibit these properties when needed typically indicates a protocol weakness. We follow with

two intermediary properties: non-extensibility, and non-emptiness. These are typically established as lemmas towards a functional correctness result, or a larger security proof.

Non-ambiguity. A message format $\rightleftarrows^M$ is non-ambiguous if it relates at most one high-level message to each bytestring. Formally: $\forall m_1, m_2 \in M, b \in \mathbb{B} . m_1 \rightleftarrows^M b \wedge m_2 \rightleftarrows^M b \implies m_1 = m_2$.

A non-ambiguous format is one for which the parser can only make a single choice for a given bytestring. It is usually easy to find and prevent ambiguity in message formats sent over the wire. When designing the message parser, the implementer will usually notice that multiple choices can be made, or they will typically find the issue during interoperability testing or by fuzzing.

The format-confusion attack on Needham-Schroeder mentioned in §4.1 relies on passing the contents of the second message as a valid format for the first message. Let us see how this can be seen as a violation of our format security properties. Since all three messages in the protocol use the same cryptographic construction (public-key encryption) with potentially the same keys, the three message payloads should conservatively be treated as three sub-cases of a single message format. However, we would then find that this shared format violates the non-ambiguity property above, since the formats of the first and second message overlap. To restore non-ambiguity for the payload format, we would need to change the protocol, by tagging each message with a distinct label, for example. Such format confusion issues are widespread in real-world protocols, as we shall later see in the context of TLS 1.2 (§4.2.3).

However, formats used in cryptographic constructions, such as signatures, do not always involve parsing messages, and so non-ambiguity is not naturally enforced or systematically tested. This has resulted in high-profile attacks, as we saw in §4.1.

As we mentioned earlier, the induced message format allows carrying properties of the *format* onto a parser and serializer.

Lemma 4.2.1 *Given a parser and serializer for M, the induced message format $\rightleftarrows^M$ is non-ambiguous if and only if:*
$\forall m \in M . \mathrm{parse}_M(\mathrm{serialize}_M(m)) = m$.

Representation unicity. A message format $\rightleftarrows^M$ is said to enjoy representation unicity when at most one bytestring can be in relation with each high-level message. Formally:
$\forall m \in M, b_1, b_2 \in \mathbb{B} . m \rightleftarrows^M b_1 \wedge m \rightleftarrows^M b_2 \implies b_1 = b_2$.

Representation unicity has sometimes been referred to as non-malleability [87]. We prefer the term "representation unicity", to avoid confusion with the standard notion of cryptographic malleability [88]. Just like non-ambiguity, representation unicity is mandatory in many security contexts, such as signed content. For instance, transaction malleability is a serious concern in Bitcoin, as described in BIP 62 [89]. Representation unicity rules out such attacks, by imposing a unique bytestring representation for each signed high-level message or transaction. We can also state representation unicity as a property of parsers and serializers.

[87]: Ramananandro et al. (2019), *Everparse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

[88]: Wikipedia contributors (2022), *Malleability (cryptography) — Wikipedia, The Free Encyclopedia*

[89]: Wuille (2014), *Dealing with malleability*

Lemma 4.2.2 *Given a parser and serializer for M, the induced message format $\rightleftarrows^M$ has representation unicity if and only if $\forall b \in \mathbb{B} . \mathrm{parse}_M(b) \neq \bot \implies \mathrm{serialize}_M(\mathrm{parse}_M(b)) = b$.*

We now look into additional properties that are not directly security-critical, but are oftentimes required in the course of proving functional correctness, or the security of a complete protocol.

Non-extensibility. A message format $\rightleftarrows^M$ is non-extensible when for every bytestring, at most one of its prefixes is in relation with a high-level message. Formally:
$\forall m_1, m_2, b_1, b_2 . m_1 \rightleftarrows^M b_1 \wedge m_2 \rightleftarrows^M (b_1 + b_2)) \implies b_2 = \varepsilon$.

We remark that our format-based approach allows us to separate this property from non-ambiguity, i.e. non-extensibility does *not* imply $m_1 = m_2$. In practice, most of the formats we care about are both non-extensible and non-ambiguous, with the notable exception of the extensible TLS transcript.

The non-extensibility property is not desirable in itself, but constitutes an important intermediary lemma in order to establish the non-ambiguity theorem for the dependent pair combinator (§4.3.2).

Non-emptiness. A message format on M is non-empty when the empty string is associated with no high-level message. Formally: $\forall m, b . m \rightleftarrows^M b \implies b \neq \varepsilon$.

Like non-extensibility, non-emptiness does not directly serve any security purpose, but is a crucial property required of a format in order to derive non-ambiguity of its list combinator (§4.3.4).

Lemma 4.2.3 *Given a parser and serializer for M, the induced message format $\rightleftarrows^M$ is non-empty if and only if $\forall b \in \mathbb{B} . \text{serialize}_M(b) \neq \varepsilon$*

Self-contained and data-dependent message formats. A message format is said to be data-dependent when it is parameterized over a piece of data not contained in the bytestring. A message format that is not data-dependent is said to be self-contained. For instance, TLS12SignatureInput (Figure 4.3), is data-dependent, over KeyExchangeAlgorithm. A concrete consequence is we cannot make sense of a bytestring for a data-dependent message format, until we know all of the data dependencies. Self-contained message formats are crucial in cryptographic protocol design (§4.2.3), and protect against protocol-confusion attacks such as [84].

[84]: Mavrogiannopoulos et al. (2012), *A Cross-Protocol Attack on the TLS Protocol*

4.2.3 A Systematic Approach to Format Security

Having defined what we mean by message format (§4.2.1), and having stated properties of interest for such message formats (§4.2.2), we now connect cryptographic primitives and secure formats.

First, a remark: almost every cryptographic primitive implicitly relies on a message format for its input. For instance, hashing an object implicitly relies on converting the object to a bytestring. The format must not introduce collisions in the process. Similarly, signatures are implicitly carried around as bytestrings; for functional correctness, the format must allow for a successful verification.

We now set out to review the standard toolkit of cryptographic primitives; we lift each primitive to a *high-level* primitive operating on high-level messages (instead of bytestrings) by relying on a message format. We then proceed by reduction: we state a high-level security assumption (for the high-level primitive operating on messages in M), and determine which properties the format should enjoy in order for this assumption

to reduce down to a standard security assumption on the low-level (bytestring-based) primitive. This allows proofs of protocol security to work off of these high-level security assumptions, and abstract message formats away.

In order to be meaningful, the security assumptions we come up with during reduction apply to all usages of a given primitive, across the entire protocol. This means that we can identify design weaknesses, such as lack of disambiguation of signature inputs, because we consider all the signatures in the entire protocol. We explain below with the example of signatures.

Additional primitives can be added to this list, taking care to equip them with suitable restrictions on formats that enforce correct cryptographic usage, following the methodology we describe here.

Our security conditions are somewhat opinionated: they are sound with respect to standard cryptographic assumptions, however there exist protocols that don't satisfy our conditions, yet remain secure; after all, the protocol where two parties can never communicate is always secure, regardless of the cryptographic operations that are performed underneath. But in reality, protocols that violate these assumptions would raise red flags in the cryptographic community, and would most likely be shunned.

Tracking all uses of all primitives across an entire protocol is non-trivial, and difficult to perform by hand; the next section (§4.3) shows how proof assistants can help scale our comprehensive format security analysis to real-world protocols.

Signature. We begin with signatures, whose security we consider in some detail; other primitives use a similar argument.

Each signature key must be used to sign messages (of high-level type M) with the same self-contained, non-ambiguous, representation-unique message format $\rightleftarrows^M$. If a signature key is used with two different message formats, or the format is ambiguous or data-dependent, this could lead to a signature confusion attack, such as the one exploited in TLS 1.2 [84] (as explained in §4.4). This condition therefore ensures the signed bytestrings correspond to the same unique message, and thus rule out signature confusion attacks.

[84]: Mavrogiannopoulos et al. (2012), *A Cross-Protocol Attack on the TLS Protocol*

This invariant on the whole protocol can then be exploited in security proofs. For example, we can lift the standard existential unforgeability under chosen message attack (EUF-CMA) assumption and specialize it with the message format: the challenger generates a pair of keys and give the public key to the attacker, then the attacker can ask the challenger for signatures of messages M, and succeeds if it manages to output $x \in M$ not queried to the challenger, along with a valid signature for it. This lifted EUF-CMA game security can then be reduced to the standard EUF-CMA security assumption, using the non-ambiguity and self-contained properties of $\rightleftarrows^M$.

We observe that this lifted EUF-CMA game doesn't say anything about a signature key used to sign two different message formats; this means that in order for the game to apply, we must have a way to ensure we only operate over signatures of messages in M. In other words, we enforce the absence of format confusion across different uses of signature within the protocol.

The input format for signatures must also enjoy representation unicity for functional correctness, so as to rule out the scenario where a message corresponds to two possible bytestrings, one used by the signer and the other by the verifier, which would lead to a verification failure for valid signatures.

Enforcing these requirements means that one must track every usage of a signature key; notably, in the case of TLS, this requires not only tracking signatures across TLS 1.2 and TLS 1.3, but also anticipating future extensions or versions of the protocol.

Symmetric MACs. The same precautions as for signatures must be used, for the exact same reasons. However, in practice, MAC keys are short-lived and used only a few times, which makes tracking all their uses easier. We recommend that all messages that may be MACed with the same key must conform to the same non-ambiguous, self-contained message format.

Authenticated Encryption with Associated Data (AEAD). As with MACs, we recommend that all encryption inputs that might use the same key must use the same non-ambiguous, self-contained message format for encrypted data and additional data. In some cases, this is stricter than necessary, and we can allow the format for encrypted data to be data-dependent on the associated data. It allows additional data to fulfill its duty: providing context in which the encryption was performed, and disambiguating identical inputs stemming from two different context. For functional correctness, the format of additional data must enjoy representation unicity (to prevent two parties from disagreeing on the serialization).

We note that the Threema messaging protocol fails these conditions, and uses ambiguous inputs to authenticated encryption, resulting in the attack [85] described earlier (§4.1).

[85]: Paterson et al. (2023), *Three Lessons From Threema: Analysis of a Secure Messenger*

Key Derivation Functions (KDFs). Given a secret key, a salt, and an info field, a KDF like HKDF generates a pseudo-random output of any desired length. In the cryptographic literature, a KDF is typically modeled as a pseudo-random function (PRF), where we assume that the attacker cannot distinguish the output of the KDF from a fresh random value. Protocols like TLS typically use a KDF to generate multiple keys for different purposes. To guarantee key independence for these keys, which is often an important precondition in security proofs, it is necessary to ensure that all the info fields used by a KDF with the same key and salt use the same non-ambiguous and self-contained message format. Furthermore, to preserve functional correctness, the format of these KDF inputs must enjoy representation unicity.

Hashing Messages and Transcripts. Many protocols use hash functions to compute digests of high-level data, including messages and protocol transcripts. A common requirement for this usage is collision resistance – two different inputs should yield two different hashes, except with negligible probability. However, even when using a collision-resistant hash function, this property may not hold for two high-level messages if they serialize to the same bitstring. Hence, we recommend that all inputs to hash functions must use non-ambiguous message formats. Furthermore, if possible, we advise that protocol authors use a single, self-contained message format for hash functions when two hashes must be collision-resistant in security proofs. Protocols might fail to obey this

restriction, but might still be secure, for instance if the data dependency is authenticated elsewhere; such a situation will call for more sophisticated proofs. For functional correctness, the hash input format must often also satisfy representation unicity.

Summary. All of the cryptographic operations considered above require non-ambiguity and representation unicity. The former rules out confusion; the latter is required for functional correctness. As we will see shortly, our format framework imposes, by construction, that every format must satisfy these two properties.

```
struct {
    ProtocolVersion legacy_version;
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16−2>;

    opaque legacy_compression_methods<1..2^8−1>;

    Extension extensions<8..2^16−1>;

} ClientHello;
```

(a) ClientHello as defined in the TLS 1.3 RFC [75]

```
type client_hello  = {
    legacy_version: protocol_version;
    random: random;
    legacy_session_id: tls_bytes {min=0; max=32};
    cipher_suites:
        tls_list cipher_suite {min=2; max=(pow2 16)−2};
    legacy_compression_methods:
        tls_bytes {min=1; max=(pow2 8)−1};
    extensions:
        tls_list extension {min=8; max=(pow2 16)−1};
}
```

(b) Equivalent client_hello type in F*

Figure 4.1: Translation of TLS 1.3 ClientHello in F*. Note how the F* type is precise: for example, the tls_bytes type represents bytes of bounded length, precisely corresponding to the opaque x<n..m> notation used in the TLS 1.3 RFC.

4.3 Verified Formats in F*

We now put our ideas in practice, and formalize secure formats within the F* proof assistant. F* is a dependently-typed programming language that supports program proof using a mixture of automatic (SMT-based) and manual (tactic-based) proofs. F* supports a wide array of programming patterns, including compile-time term synthesis, which we leverage in this section. Throughout this paper, we only ever use the pure fragment of F* and need not rely on its effect system.

Studying formats as complex as those of TLS in a monolithic fashion is unrealistic; any reasonable programmer will decompose formats into basic blocks that can each be studied in isolation, then composed together to form larger formats. To support this modular approach, this section introduces a set of format combinators that allow assembling complex message formats from simpler ones. We show how security properties of complex formats (the application of a combinator) can be deduced from the security properties of the simpler formats they are built upon (the arguments to the combinator). Authoring these formats by hand is tedious; we show how to automate the process using Meta-F*.

Although minimalistic (we only use, and describe, a mere 4 combinators), our approach is expressive enough to describe all message formats in TLS, cTLS and MLS. Our combinators guarantee that the formats they produce are secure. This proof is done once and for all, which relieves the programmer of the bulk of the proof effort. Users may also opt out of combinators and write message formats directly, but then are required to prove their correctness by hand, which is more onerous.

4.3.1 Defining Secure Message Formats in F*

Definitions, lemmas, proofs of reductions. We follow §4.2.1 and define formats as logical predicates in F*. We transcribe definitions from §4.2.2 and prove all of the corresponding lemmas.

Connection with parsers & serializers. Presenting formats as relations allowed us to capture the essence of formats, along with their security properties, free of the implementation-centric notions of parsers and serializers.

However, in our F* library, we define our formats using parsers and serializers. This design decision is a pragmatic one. First, this makes life easier for the programmer and doesn't require them to write a logical predicate $\rightleftarrows^M$ by hand. Second, this makes our formats *executable* which is crucial for authoring reference implementations that can serve for interoperability testing, but also for building further security proofs (§4.5). Third, we write and prove (in F*) the connection between induced formats and parsers and serializers (§4.2.1). Because this connection is verified, we not only do not lose any expressive power, but also provably know that our security properties over parsers and serializers are equivalent to the original security properties over the induced format.

A type definition for secure formats. Next, we devise a "user interface" for Comparse, that is, we write a concrete type definition in F* for what we mean by *format*.

As we saw earlier (§4.2.3), regardless of the context in which the formats appear, we always want them to enjoy non-ambiguity and have a unique representation. Therefore, our type of formats, below, takes not only a parser and a serializer, but also *proofs* that those two crucial properties always hold. Per the lemmas from §4.2.1, those two properties boil down to stating that the parser and serializer are inverses of each other. Because of this design choice, our library will refuse to handle ambiguous or non-unique formats: it is our position that formats that fail to exhibit these properties indicate a design weakness in the protocol.

```
type message_format_for (a:Type) = {
 parse: bytes → option a;
 serialize: a → bytes;
 // Non-ambiguity
 parse_serialize_inv: x:a → Lemma ( parse (serialize x) == Some x );
 // Representation unicity
 serialize_parse_inv: buf:bytes → Lemma (
  match parse buf with
  | Some x → serialize x == buf
  | None → ⊤ ); }
```

Non-extensible message formats. We define a separate type, called prefix_message_format_for, to represent non-extensible secure message formats, where the parser only consumes a prefix of the input bytestring, and returns the parsed element and the remaining suffix. In this type, the parse_serialize_inv property is adapted to allow for the suffix, as follows:

```
parse ((serialize x) ++ suffix) == Some (x, suffix)
```

Non-empty message formats. We say a message format is non-empty when all its serializations have non-zero lengths (Lemma 4.2.3). In F*, we offer this as a refinement over the earlier types.

4.3.2 The dependent pair combinator

We begin with our first combinator for pairs. Repeated applications of this combinator allow encoding pairs of several elements (known as tuples): we write $A \times B \times \ldots \times D$ for $A \times (B \times (\ldots \times D))$.

Tuples naturally occur in the wild, such in the ClientHello message of TLS (Figure 4.1a), which is simply the combination of all of its subfields. Because our combinators are generic, they cannot produce a specific user-defined type such as ClientHello; rather, they produce a tuple of ProtocolVersion × Random × . . .. We show in §4.3.5 how to convert a structural type (the tuple) into a nominal type suitable for the rest of the protocol definition (the user-provided client_hello, Figure 4.1b).

Sometimes, the message format for the second element of a pair depends on the contents of the first one. This is a *dependent* pair, which generalizes to dependent tuples. A dependent tuple occurs in the Handshake message of TLS (Figure 4.2), where the format of the third field depends on the value of the first one (via select), as well as the value of the second one (via the comment referring to length). Our combinator supports general dependent pairs, of which non-dependent pairs are a special case; this allows to encode, notably, the tagged union pattern of messages such as Handshake.

Message format combinator. The message format for the dependent pair $X \times Y$ is defined as a simple concatenation. Formally: $(x, y) \rightleftarrows^{X \times Y} b := \exists b_1, b_2 . b = b_1 + b_2 \wedge x \rightleftarrows^{X} b_1 \wedge y \rightleftarrows^{Y(x)} b_2$.

We write the dependency explicitly ($\rightleftarrows^{Y(x)}$), which captures the fact that the format of the second element depends on the first. We use a lightweight notation $X \times Y$ for dependent pairs to avoid cluttering the formulas, as opposed to the traditional $\sum_{x:X} Y(x)$. In practice, the dependent pair combinator allows turning a data-dependent format (the second element of the pair, §4.2.2) into a self-contained format (if the dependency is only over the first element). A common instance of this pattern is for tagged unions.

Formally proven security properties.

- $\rightleftarrows^{X \times Y}$ is non-ambiguous if $\rightleftarrows^{X}$ is non-extensible and non-ambiguous, and $\rightleftarrows^{Y(x)}$ is non-ambiguous for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ has representation unicity if $\rightleftarrows^{X}$ and $\rightleftarrows^{Y(x)}$ have representation unicity for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ is non-extensible if $\rightleftarrows^{X}$ is non-extensible and non-ambiguous, and $\rightleftarrows^{Y(x)}$ is non-extensible for every $x \in X$.
- $\rightleftarrows^{X \times Y}$ is non-empty if $\rightleftarrows^{X}$ is non-empty or $\rightleftarrows^{Y(x)}$ are non-empty for every $x \in X$.

Role of non-extensibility. In the non-ambiguity theorem, non-extensibility of the first element of the pair is crucial: for example, consider the trivial message format on $\mathbb{B}$, defined as $b_1 \rightleftarrows^{\mathbb{B}} b_2 := b_1 = b_2$. This message format is non-ambiguous, but a non-dependent pair of two such formats is, for the same reason as the message format on $\mathbb{B}^2$ studied in §4.2.1.

```
struct {
   HandshakeType msg_type;   /* handshake type */
   uint24 length;            /* remaining bytes in message */
   select (Handshake.msg_type) { /* handshake content */
      case client_hello: ClientHello;
      case server_hello: ServerHello;
      /* ... */
   };
} Handshake;
```

Figure 4.2: The Handshake message format, as defined in TLS 1.3 [75]. The msg_type determines the format to use for the handshake content (via select). Furthermore, the comment for field length encodes a semantic restriction: the total length (in bytes) of the select ... field is equal to length.

Formalization in F*. Two flavors exist for the dependent pair combinator: although the message format for the first element of the pair must always be non-extensible, there is no such restriction on the second element of the pair. Furthermore, the result is non-extensible if and only if the second element of the pair is non-extensible. We reflect this with two separate F* functions.

```
// When both mf_a and mf_b have the non-extensibility property
val prefix_message_format_for_dep_pair:
  #a:Type → #b:(a → Type) →
  mf_a:prefix_message_format_for a →
  mf_b:(x:a → prefix_message_format_for (b x)) →
  prefix_message_format_for (x:a & b x)

// When only mf_a has the non-extensibility property
val message_format_for_dep_pair:
  #a:Type → #b:(a → Type) →
  mf_a:prefix_message_format_for bytes a →
  mf_b:(x:a → message_format_for bytes (b x)) →
  message_format_for bytes (x:a & b x)
```

Encoding Handshake with dependent pairs. We illustrate the usage of the dependent pair combinator on the type of handshake (Figure 4.2). Assuming we have message formats for T (HandshakeType), U_{24} (uint24, unsigned 24-bit integers) and $M(t, l)$ (data-dependent handshake content of type t and serialized length l), Handshake can be encoded as $T \times (U_{24} \times M)$ (or more precisely: $\sum_{t:T} \sum_{l:U_{24}} M(t, l)$). We note that the resulting dependent triple is no longer data-dependent.

We cannot yet show the definition of M; is it a dependent type, along with an added restriction over its length. To express the latter, we need a new format: the refinement combinator.

4.3.3 The refinement combinator

Message formats are sometimes described as subsets of other message formats. For example, we can define a boolean as a byte restricted to the value 0 or 1.

Message format combinator. If $Y \subset X$, and we have a message format $\rightleftarrows^X$, then we can define $m \rightleftarrows^Y b := m \rightleftarrows^X b$.

Length restriction. A particularly useful usage of the refinement combinator is to enforce exact length restrictions on high-level messages. Given a set of messages M, we define its subset $\text{RestrictLen}(M, l) = \{m \in M \mid \forall b. m \rightleftarrows^M b \implies \text{length}(b) = l\}$. This refinement, when used

in conjunction with a dependent pair, allows encoding length-prefixed messages, wherein the first element of the pair is a (bounded) unsigned integer that stands for the length of the second element.

Formally proven security properties. The refinement combinator preserves non-ambiguity, non-extensibility, non-emptiness and representation unicity. When used with RestrictLen, it is unconditionally non-extensible.

Formalization in F*. The refinement combinator also comes in two flavors, depending on whether the input format is extensible or not. We show the extensible version here:

```
val refine:
  #a:Type → message_format_for a → pred:(a → bool) →
  message_format_for (x:a{pred x})
```

We provide a dedicated combinator that captures the fact that an extensible format can be turned into a non-extensible one, via a length restriction.

```
val fixed_length_format_to_non_extensible:
  #a:Type → len:nat →
  mf_a:message_format_for a{∀ x. length (mf_a.serialize x) == len} →
  prefix_message_format_for a
```

Encoding Handshake content with refinement. Now that we have refinements, we can revisit our earlier Handshake example (§4.3.2) and use RestrictLen to encode the constraint on the length of the third field. As mentioned above, this means the third field is unconditionally non-extensible, which in turn makes the whole Handshake message non-extensible. Our format for handshake is now of the form $\sum_{t:T} \sum_{l:U_{24}} \text{RestrictLen}(M'(t), l)$.

4.3.4 The list combinator

With the dependent pair combinator (§4.3.2), we can encode fixed-sized lists as n-tuples, but cannot represent lists whose length is not known at compile-time. For this, we need a new list combinator.

Message format combinator. Given a message format on M, we define a message format on M^*, the type of lists of Ms (with any number of elements), as: $[m_1, \ldots, m_n] \rightleftarrows^{M^*} b := \exists b_1, \ldots, b_n . b = b_1 + \cdots + b_n \wedge \forall i . m_i \rightleftarrows^{M} b_i$. Our format does not require that the list be prefixed by its length, although if the protocol mandates it, we can always encode it using a combination of refinement, dependent pair, and list combinator.

Formally proven security properties.

- $\rightleftarrows^{M^*}$ is non-ambiguous if $\rightleftarrows^{M}$ is non-ambiguous, non-extensible, non-empty.
- $\rightleftarrows^{M^*}$ has representation unicity if $\rightleftarrows^{M}$ has representation unicity.

Role of non-emptiness. Requiring non-emptiness rules out degenerate cases, such as the unit format $() \rightleftarrows^{unit} \varepsilon$. Lists of units all serialize to a single empty bytestring, meaning lists of unit are ambiguous.

Formalization in F*. To be secure, the list combinator takes as input a non-extensible, non-empty secure message format. It returns an extensible message format. Non-ambiguity and representation unicity are carried over automatically, since they are bolted into the two message format types.

```
val message_format_for_list:
 #a:Type → mf_a:prefix_message_format_for a{is_non_empty mf_a} →
 message_format_for (list a)
```

Encoding the TLS 1.3 transcript. The TLS transcript is a list of Handshake messages. Because Handshakes are non-ambiguous, non-extensible, non-empty, and have representation unicity, the TLS 1.3 transcript is non-ambiguous and has representation unicity, which are crucial properties that guarantee the correct behavior of transcript hashes in the security proof of TLS.

4.3.5 The isomorphism combinator

Given a message format defined in a document, such as in the TLS 1.3 RFC [75], we write a type in F* precisely capturing the expressivity of the message format. This corresponds to the type M of messages we saw earlier. We give an example for the TLS 1.3 ClientHello message in Figure 4.1.

The three combinators we have seen so far can parse ClientHello and Handshake, but return tuples that are isomorphic to, but not equal, to the type that the user would write in F* for Handshake (Figure 4.1).

We thus need one final combinator that goes from a *generic* representations (base types, lists, and dependent pairs) into the "original", user-defined message type. This is the isomorphism combinator. The isomorphism combinator is typically the final building block used to create a message format.

Message format combinator. Given a bijective function $f : T \to E$, which maps a high-level type T to an encoding E, and a message format for E, we define a message for T as $m \rightleftarrows^T b := f(m) \rightleftarrows^E b$.

Formally proven security properties. Because f is a bijection, the isomorphism combinator preserves non-ambiguity, representation unicity, non-extensibility, non-emptiness.

Formalization in F*. We rely on two functions for the bijection, which we require to be inverse of each other. We do this by adding a precondition to the isomorphism combinator: we can use it only if we prove that the two bijection functions are inverse of each other. This precondition is crucial to prove that the resulting message format is secure. The isomorphism combinator comes in two flavors, depending on whether the message format is non-extensible or not. Because they are so similar we only show the signature of the non-extensible version.

```
val prefix_message_format_for_isomorphism:
 #a:Type → #b:Type → mf_a:prefix_message_format_for a →
 a_to_b:(a → b) → b_to_a:(b → a) →
 Pure (prefix_message_format_for b)
 (requires (∀ x. a_to_b (b_to_a x) == x) ∧  (∀ x. b_to_a (a_to_b x) == x))
```

Finalizing our Handshake format. In §4.3.2, we obtained an encoding of Handshake as a dependent tuple of 3 elements. However, Handshake is not a dependent tuple, it is a nominal type in F*:

```
type handshake = {
  msg_type: handshake_type;
  length: uint24;
  msg: fixed_length_handshake_content msg_type length; }
```

We use the isomorphism combinator to link the nominal type (handshake) and its encoding (the dependent tuple of 3 elements).

4.3.6 Automating Combinator Synthesis

Writing combinator applications by hand quickly becomes repetitive. We now present a facility that allows the user to write only their top-level type, such as ClientHello (Figure 4.1). With a few strategically placed annotations, our facility inspects the type definition and automatically generates the combinators that will parse and serialize elements of that type.

Our facility relies on Meta-F* [90], a general-purpose compile-time metaprogramming framework. Using a technique known as elaborator reflection, Meta-F* essentially allows the programmer to "script" the compiler, to resolve proof obligations, or in our case, inspect terms and generate fresh definitions.

[90]: Martínez et al. (2019), *Meta-F* : Proof Automation with SMT, Tactics, and Metaprograms*

We authored a meta-program that takes an annotated type definition and produces a corresponding format, complete with proofs of non-ambiguity and representation unicity.

Inner workings. When processing a type such as client_hello (Figure 4.1b), the meta-program proceeds in two steps: first, it derives a message format using anonymous types (dependent tuples), then it uses the isomorphism combinator (§4.3.5) to produce a message format for the user-defined type (client_hello, a record with user-provided field names).

For each field, a corresponding format is looked up in the environment. If this corresponds to a user-defined type for which a format was previously generated, or to a base type for which we provide a hand-written format (such as uint24), all is well. Otherwise, the meta-program fails and the user must annotate the type by hand to indicate which format ought to be used for the given field.

Once again, this could be done entirely by hand: our automation relieves the user of a repetitive task. Importantly, it also makes the program easier to maintain: if an internet draft is updated to a new revision, the programmer just needs to change the type definition, and the formats automatically follow.

Handshake example. In the case of the handshake, the dependency of the msg field over msg_type and length is handled naturally: the format found in the environment for fixed_length_handshake_content takes two parameters, so the tactic instantiates that format with its two arguments brought in scope by the dependent tuple. Thanks to a judicious choice for our default formats, handshake serializes exactly per the TLS 1.3 RFC [75].

Other supported types. F* sum types are also handled by our tactic, which picks a tagged union scheme. By default, the tag occupies the minimum number of bytes required to encode all cases; the user can override that choice, and specify explicitly which type should hold the tag (including its size). This allows the user to obey a precise format specified e.g. in an RFC.

4.3.7 Implementation

The implementation described in this section occupies a total of about 2,500 lines of code. This includes the combinators and base types (942 lines), the derived types (776 lines) and the tactic (742 lines). The tactic is among the three largest Meta-F* programs written to date. We estimate that the total effort for formalizing Comparse in F* took a few person-months.

The overall conciseness of our development is explained by the judicious choice of a notion of *format* as our base abstraction, rather than parsers and serializers. Furthermore, the judicious choice of combinators limits the amount of work we need to perform. Finally, the fact that we do not need to rely on effects, along with careful crafting of definitions, allow us to maximize the amount of proofs performed automatically by SMT.

4.4 Verified Formats for TLS and cTLS

The Transport Layer Security (TLS) protocol, standardized by the IETF, is used to secure the vast majority of Web traffic. The most recent version, TLS 1.3, was standardized in 2018 [75] and is the the most commonly used version of TLS, followed by TLS 1.2 [91]. More recently, the IETF TLS working group has been working on standardizing Compact TLS 1.3 (cTLS) [92].

[92]: Rescorla et al. (2023), *Compact TLS 1.3*

In this section, we will discuss how we implemented all the formats used in TLS and cTLS using Comparse, and show how our work provides crucial missing properties needed for the security analyses of these protocols, both in isolation and when they are deployed in parallel with each other.

4.4.1 Format Confusion Attacks in TLS 1.0-1.2

The TLS protocol establishes a secure channel between a client and a server. It works in two phases: first, a *handshake* phase performs an authenticated key exchange to establish shared keys between the client and server, then a *transport* phase allows them to exchange application data protected with these shared keys. Typical TLS implementations support multiple versions and ciphersuites for backwards compatibility and to support maximum interoperability.

In TLS versions up to TLS 1.2, each handshake may use one of multiple key exchange modes. Depending on the mode, each TLS handshake consists of between 6 and 13 messages, which include various cryptographic constructions: 2 MACs, 3 key derivations, 2 encryptions, and up to 2 signatures. The formats for all these messages and cryptographic inputs are described in the custom TLS presentation language, which looks like C structs.

```
struct {
   select (KeyExchangeAlgorithm) {
      case dhe_rsa, dhe_dss,              // From TLS 1.2
           dhe_rsa_export, dhe_dss_export:  // From TLS 1.0
         opaque client_random[32];
         opaque server_random[32];
         ServerDHParams params;
      case ec_diffie_hellman:           // From TLS 1.2 ECC
         opaque client_random[32];
         opaque server_random[32];
         ServerECDHParams params;
   };
} TLS12SignatureInput;
```

Figure 4.3: A common format for TLS 1.0-1.2 signature inputs [91, 93].

For example, the input formats for server signatures used in implementations of TLS 1.0-1.2 is depicted in Figure 4.3. It includes the specification of Ephemeral Diffie-Hellman signatures (DHE) from TLS 1.2, the signature inputs for export ciphersuites in TLS 1.0, and the format for Elliptic Curve Diffie Hellman (ECDHE) in TLS 1.2. These formats are actually defined in three different documents, and we have brought them together for illustration.

We note that the format depends on a value (the key exchange algorithm) that is external to it, which is not authenticated by the signature itself. Indeed, in the absence of this external input, the signatures used in DHE key exchanges can be confused for those in ECDHE key exchanges, which leads to a concrete cross-protocol attack [84] on TLS 1.2. Note also that the format used in DHE is the same as the one used in Export DHE, which is one of the factors exploited by the Logjam attack [94].

[84]: Mavrogiannopoulos et al. (2012), *A Cross-Protocol Attack on the TLS Protocol*

[94]: Adrian et al. (2015), *Imperfect forward secrecy: How Diffie-Hellman fails in practice*

With our methodology (§4.2.3), we impose that every signature key be associated to a single, self-contained, non-ambiguous message format. These properties are violated here, because the format is not self-contained, owing to the key exchange algorithm which is an external input.

We formalized the signature input format for TLS 1.2 (DHE, ECDHE) in Comparse, and proved that given the data-dependency (KeyExchangeAlgorithm), the format has non-ambiguity and representation unicity. We were not able to do the same without the data-dependency, hinting at the cross-protocol attack. We once again re-emphasize that we see *all* the inputs to a *single* primitive as a *single* format, meaning that both TLS 1.2 and TLS 1.3 signature formats are seen as sub-cases of the general "signature input format". This systematic approach forces us to reason globally about *all* the signatures in the protocols.

4.4.2 Verified Formats for TLS 1.3

TLS 1.3 fixes many of the attacks in TLS 1.2, including the format confusion attacks described above. In particular, it defines a uniform format for all signature inputs, MAC inputs, and key derivations, by using the handshake transcript in all three cases. TLS 1.3 also encrypts handshake messages for privacy, hence the format of encryption inputs now includes 8 handshake messages.

The security of TLS 1.3 relies crucially on the non-ambiguity of the format of the handshake transcript at each stage of the protocol. It also relies

```
enum {
  profile(0),
  version(1),
  cipher_suite(2),
  dh_group(3),
  signature_algorithm(4),
  random(5),
  mutual_auth(6),
  handshake_framing(7),
  client_hello_extensions(8),
  server_hello_extensions(9),
  encrypted_extensions(10),
  certificate_request_extensions(11),
  known_certificates(12),
  finished_size(13),
  optional(65535)
} CTLSTemplateElementType;

struct {
  CTLSTemplateElementType type;
  opaque data<0..2^32−1>;
} CTLSTemplateElement;

struct {
  uint16 ctls_version = 0;
  CTLSTemplateElement elements<0..2^32−1>
} CTLSTemplate;
```

Figure 4.4: Compression templates for Compact TLS 1.3

on the non-ambiguity of each handshake message format. Furthermore, if TLS 1.3 is to be safely deployed alongside TLS 1.2, we need to prove non-ambiguity across both protocols.

Despite its importance, this property is not accounted for in many published proofs of TLS 1.3. The mechanized proofs of TLS 1.3 in ProVerif [49], CryptoVerif [49], and Tamarin [50] all assume that the message formats are injective and disjoint, and that the transcript can be treated unambiguously as a tuple of handshake messages. The pen-and-paper security proofs of TLS 1.3 (e.g. [95–97]) abstract away all formatting details; they typically assume distinct labels for the different cryptographic inputs in the protocol to simplify their analysis. Consequently, none of the published proofs of the TLS 1.3 handshake actually apply to the bit-level formats used in the protocol.

[49]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*

[49]: Bhargavan et al. (2017), *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*

[50]: Cremers et al. (2017), *A Comprehensive Symbolic Analysis of TLS 1.3*

[95]: Dowling et al. (2021), *A Cryptographic Analysis of the TLS 1.3 Handshake Protocol*

[96]: Kohlweiss et al. (2015), *(De-)Constructing TLS 1.3*

[97]: Li et al. (2016), *Multiple Handshakes Security of TLS 1.3 Candidates*

To close this gap in the literature, we formalized all the handshake messages of TLS 1.3, the handshake transcript, and all inputs to signatures, MACs, encryption, and key derivation, in Comparse, and proved that these formats were non-ambiguous. This means that any proof of TLS 1.3 that only considers a high-level abstraction of each message still applies to the concrete protocol that uses low-level bitstrings within the cryptographic inputs. Furthermore, we combined the TLS 1.2 and TLS 1.3 signature inputs to prove that the combined format is non-ambiguous. This result shows that it is safe to deploy TLS 1.3 and TLS 1.2 in parallel.

Some prior works do prove injectivity for TLS 1.2 transcripts [98] but they do not consider TLS 1.3, or both in parallel. There is also prior work on specifications and efficient implementations of TLS 1.3 and TLS 1.2 message formats [87], but they do not model TLS and do not prove non-ambiguity across TLS 1.3 and TLS 1.2 signatures.

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[87]: Ramananandro et al. (2019), *Everparse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

4.4.3 Verified Formats for cTLS

The handshake messages exchanged in a TLS handshake can get very large, primarily because TLS is designed to be interoperable across a large range of devices and so the messages contain information that may be needed in different scenarios.

Compact TLS 1.3 is a new proposal that aims to reduce the message size by agreeing out-of-band on a *compression template*. For example, if the client and server can agree beforehand on the ciphersuite and server certificate, several elements in the handshake can be eliminated, and others can be treated as fixed-length values (without length prefixes).

Other than the message formats being compressed, the cryptographic steps in cTLS are identical to TLS 1.3. Consequently, one might hope that all the TLS 1.3 security proofs will apply to cTLS. However, this only holds if we can prove that the cTLS messages and transcripts are unambiguous, and if we can show that the cTLS transcript is equivalent (for each template) to the TLS 1.3 transcript. Finally, since cTLS is likely to be deployed in parallel with TLS 1.3, we need to prove that the joint formats between the two protocols are unambiguous.

Compressing TLS 1.3 using Templates. The cTLS protocol defines a compression template that the client and server must agree to in advance. CTLSTemplate (Figure 4.4) depicts all the elements that a certain template can fix. Given such a template, the protocol describes how each message in TLS 1.3 can be compressed at a fine-grained level. For example, in the keyshare extension of the client hello, a length field can be omitted if the template specifies a single Diffie-Hellman group. Consequently, each message format in cTLS depends on the compression template.

We follow the methodology described in §4.3.1 by writing types that precisely capture what is representable by the message format, and depend on the compression template. On the example of the cipher suite compression in ClientHello, we define a new type for the cipher_suites field of client_hello. Although these new types are more complex than the ones used in TLS 1.3, they are fully handled by the meta-program.

```
type ctls_cipher_suites (t:ctls_template) =
  match get_template_element cipher_suite t with
  | Some _ → unit
  | None → tls_list cipher_suite ({min=2; max=(pow2 16)−2}))

type ctls_client_hello (t:ctls_template) = {
  (* ... *)
  cipher_suites: ctls_cipher_suites t;
  (* ... *)
}
```

Proofs for cTLS message formats and transcripts. We prove that given a template, each cTLS message has non-ambiguity and representation unicity. We do this by representing each cTLS message with a dependent type (on the template), and use Comparse to derive and prove a format on each cTLS message.

The cTLS transcript includes the compression template as a first (dummy) message and then continues with the list of cTLS handshake messages. We show that the first handshake message of the transcript carries enough information to deduce what modification was applied on the

transcript, whether it is hashing the first ClientHello, or it is doing a cTLS compression with a template.

We prove that all the cTLS messages are non-ambiguous with unique representations, and that cTLS transcripts and TLS 1.3 transcripts are non-ambiguous with respect to each other. This means that despite all the format changes and optimizations, cTLS is free from format confusion attacks, and that it is safe to deploy cTLS in parallel with TLS 1.3 (using the same server certificates.)

Equivalence between TLS 1.3 and cTLS transcripts. We also prove that each cTLS compressed transcript correspond to a unique TLS 1.3 transcript, up to extensions re-ordering. Each TLS 1.3 type that is modified by cTLS is associated with a cTLS compressed type, that depends on the compression template. We then write two compression and decompression functions, that convert between the TLS 1.3 type and the cTLS dependent type.

The compression function can fail, for instance if the compression template enforces the some ciphersuite, but the TLS 1.3 message tries to negociate the wrong ciphersuite. The decompression function can also fail, because compression removes some length tags, meaning that there exist valid compressed transcripts that contain elements whose length exceeds what is admissible by TLS 1.3. This is a novel insight that seems to not have been noticed by the cTLS designers before, and might be addressed in future drafts.

We prove round-trip properties on compression and decompression: if compression succeeds, then decompression on its result succeeds and returns something equal to the compression input (up to extension re-ordering); also, if decompression succeeds, then we can compress back its result, it will succeed and return something equal to the decompression input.

By proving these compression/decompression guarantees between TLS 1.3 and cTLS, we show that the messages and transcripts in the two protocols are interchangeable. Consequently, any proof of security for TLS 1.3 that relies on high-level message formats still holds when the messages are formatted (i.e. compressed) according to cTLS. This provides a foundation upon which future proofs of cTLS can build, since they now know that all of the formats are safe, and are in correspondence with those of TLS 1.3.

4.5 Embedding Comparse in DY*

Existing protocol verification tools often miss format confusion attacks since they do not account for bit-level precise formats. We now show how to close this gap, by integrating Comparse into the DY* symbolic protocol analysis framework. In so doing, we make it possible to apply symbolic protocol analysis to low-level message formats. Despite this additional precision, rather than incurring an additional proof burden, Comparse actually significantly reduces the proof effort for DY* proofs because of its support for automation.

4.5.1 Background: Symbolic verification with DY*

DY* [43] extends the F* [98] proof system with a symbolic verification framework for cryptographic protocols. In comparison to symbolic provers like Tamarin and ProVerif, DY* proofs require more manual annotations and are less automated. Conversely, DY* uses a more scalable proof technique, based on typechecking, and can exploit the full expressiveness of the F* proof assistant. Consequently, DY* is particularly well-suited to the formal analysis of large, complex cryptographic protocols with recursive protocol flows and inductive data structures. Indeed, DY* has been used to obtain state-of-the-art verification results for advanced protocols like Signal [43], ACME [99], Noise [100] and MLS [68].

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[99]: Bhargavan et al. (2021), *An In-Depth Symbolic Security Analysis of the ACME Standard*

[100]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

[56]: Dolev et al. (1983), *On the security of public key protocols*

Symbolic Message Formats. Like other symbolic provers, DY* relies on the Dolev-Yao model [56], where messages are treated as algebraic terms that can be constructed and destructed using abstract functions that obey simple symbolic equations. These constructors and destructors are used to model cryptographic primitives like encryption/decryption and signature/verification.

Most symbolic protocol analyses in tools like ProVerif and Tamarin ignore message formatting, and simply use tuples to represent message contents. DY* does a little better, by defining an abstract type and interface for bytestrings that include ASCII strings, freshly generated random values, cryptographic elements, and can furthermore be concatenated and split to implement specific message formats.

However, the underlying model is still symbolic, and hence does not precisely model concrete bytestrings or their lengths. For example, in this model, `concat(a,concat(b,c))` is different from `concat(concat(a,b),c)`. Consequently, it is possible that verified DY* code written against the *symbolic bytes* API may potentially still be vulnerable to format confusion attacks that exploit such inconsistencies. To counter this, we need to prove that the the bytestring API is also sound with respect to a concrete model of bytestrings.

Verifying Message Formats. Since DY* lacks a framework for automatic format analysis, DY* programmers are expected to write, by hand, serialization and parsing functions for all the message formats, cryptographic inputs, and session states used in the protocol and then prove non-ambiguity for all these formats, to use as lemmas within the security proof.

DY* tracks the secrecy and authenticity of each bytestring using secrecy labels and logical refinements, allowing users to reason about the security guarantees of their protocol code. Hence, the programmer also needs to prove that secrecy labels are preserved by formatting. For example, before sending a message on the network, we need to prove that the serialization of this message is "publishable", so that revealing it to the attacker will not leak secret values. To prove this, the programmer needs to reason about the format, and show that a serialized message is publishable if and only if every field of the high-level message is publishable.

These formatting proofs are currently done by hand in DY*, inducing a significant proof burden even for simple message formats. While this is feasible for small protocols, it quickly becomes tiresome or even impossible for real-world protocols like TLS. In the rest of this section, we show how Comparse can help alleviate this proof burden and close a

gap in DY* by providing a concrete model for bytestrings. Furthermore, by combining DY* and Comparse, we are able, for the first time, to execute DY* applications *concretely* to create and process wire-compatible message bytestrings.

4.5.2 Plugging DY* and Comparse together

Handling multiple bytes types. Until now, we assumed that Comparse worked on one defined type for byte sequences. However, DY* defines its own type to do symbolic protocol verification, the aforementioned *symbolic bytes*. To cover all the bytes types one might want to use, we now parameterize Comparse over a typeclass for bytes, which contains the minimal set of properties and lemmas that work for *any* instantiation of the type class (i.e. for both symbolic and concrete bytes). This means that the entire Comparse development is carefully crafted to never rely on any *extensionality* hypothesis, namely that two bytestrings are equal if they have the same length and coincide on every index. Comparse does not rely on concatenation associativity either; none of these properties hold for symbolic bytes.

To the best of our knowledge, no realistic parser framework was ever devised before to work without the use of extensionality or associativity. This is a key contribution of our work, and a core difference compared to other frameworks such as EverParse [87].

[87]: Ramananandro et al. (2019), *Everparse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

We can therefore use Comparse both for *concrete* bytes, to execute cryptographic protocols, and *symbolic* bytes, to prove security of cryptographic protocols. We now give more details.

The typeclass. To simplify the instantiation of the typeclass with various bytes types, we make it as minimal as we can. In the typeclass, we require bytes to have a length, and an empty sequence of bytes whose length is zero. We also require concat and split functions, which are well-behaved with respect to length. The split function is allowed to fail, for example if the index is out-of-bounds (in the concrete world) or if the index is not exactly at the right position (in the symbolic world). We furthermore require split and concat to be well-behaved with respect to each other. We are as liberal as we can about this, so that it is easily implementable with any symbolic bytes implementation. In particular, we know a split succeeds only when the index is at the boundary of a concatenation.

Writing message formats combinators. We ensure the combinators in §4.3 only rely on lemmas provided by the type class. This means our proofs are more difficult to conduct, since we cannot assume concat to be associative, we refer the reader to the supplementary material [101] for the full details.

[101]: (2023), *Comparse: Supplementary Material*

4.5.3 Improving message formats in DY*

Precise message formats. We are now able to precisely model message formats as they are written in the RFCs (§4.3.1) by plugging Comparse into DY*. Combined with Comparse's automation, this guarantees that users of DY* can easily and accurately model real-world formats as opposed to sketches of formats.

Non-ambiguity. Non-ambiguity is a built-in feature of the formats in Comparse (§4.3.1), meaning we immediately satisfy the DY* requirement after integrating Comparse.

Information flow. Given a predicate pred on bytes which is preserved by concatenation and splitting, we define a predicate on high-level messages is_well_formed pred, capturing the fact that every sequence of bytes in the high-level message satisfies the predicate pred. Moreover, we have the property that if is_well_formed pred msg then pred (serialize msg) and if pred buf and parse buf = Some msg then is_well_formed pred msg. This allows us to satisfy the DY* requirements regarding labeling of message, and compose them with Comparse while retaining a high degree of automation.

Impact on DY* examples. We adapted two examples of DY* to use Comparse for their formatting, as shown in Table 4.1. Before, the protocols relied on hand-written definitions of parsers and serializers, along with manual non-ambiguity proofs, resulting in 148 lines of code devoted to message formats in the NSL example and 141 in the ISO-DH example. With Comparse, this proof effort can be reduced by an order of magnitude (respectively 19 and 24 lines of code) by boiling it down to writing the type of messages, letting Comparse generates combinators and proofs automatically.

Furthermore, with the use of Comparse, we are now able to execute each DY* example both symbolically and concretely, providing two independent forms of debugging. The symbolic traces show the high-level protocol flow, while the concrete traces display the low-level bytestrings obtained by applying all the specified cryptographic and formatting functions.

Impact on MLS verification. We also used Comparse to formalize all the formats of the MLS RFC. In so doing, we prove that the signature confusion attack on MLS draft 12 [68] is now fixed, and provide strong security guarantees for all the formats used in the MLS. Indeed, our formal security proofs form part of a larger formal verification effort for MLS, and were used in a recently published work on the TreeSync sub-protocol [68]. Our proofs enjoy the automation provided by Comparse (§4.3.6) to write and verify the 82 formats defined in the RFC [21].

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

4.6 Discussion

To evaluate the effectiveness of Comparse, we applied it to several case studies, as shown in Table 4.1, including two examples from DY* (NSL, ISO-DH), TLS 1.3, cTLS and MLS.

Lessons. During the course of this work, we learned several lessons. Our approach is focused on formats used in cryptographic inputs and

Protocol	Nb. formats	RFC LoC	F* LoC	Lemmas	Verif. time
NSL	7	—	19	16	1min
ISO-DH	9	—	24	21	45s
TLS 1.3	51	311	452	105	3min15s
MLS	82	482	624	164	2min45s
cTLS	30	623	608	110	2min45s

Table 4.1: Evaluation over a set of protocol case studies. Lemmas include non-ambiguity, representation uniqueness lemmas, and disjointness. TLS 1.3 proofs include non-ambiguity with TLS 1.2; cTLS proofs include non-ambiguity with TLS 1.3, and properties of compression/decompression.

is justified by the cryptographic assumptions typically used in protocol analysis. We believe this principled approach yields more precise security conditions than other works that focus on parsers and serializers.

Second, our work makes it apparent why one needs to study all of the usages of a given primitive across an entire protocol, in order to rule out the entire class of format confusion attacks.

Third, proof assistants are crucial in making the analysis above tractable. With pen and paper, tracking every usage of a given key across all of TLS 1.3 (and TLS 1.2, because of backwards compatibility) would be impossible. With a carefully crafted library, combinators take care of the bulk of the work, and the library need not grow into a massive software artifact: four well-chosen combinators suffice.

Finally, the format analysis does not need to live in isolation as a separate development. It can be successfully integrated into a general-purpose symbolic security analysis framework (in our case, DY*), paving the way for future evolutions of other tools (e.g. Tamarin or ProVerif) that might make them, too, format-aware.

Limitations. We also identified several limitations throughout our journey in the land of formats. First, there are some real-world, non-ambiguous messages formats with unique representation that cannot be expressed using our combinators. One example is TLSInnerPlaintext, which must be parsed starting from the end. Fortunately, these are few such cases, and we can use the escape hatch we mentioned earlier: it suffices to write the formats by hand, without the distinguished combinators from §4.3. Because such hand-written formats still use the types from Comparse, they compose with the rest of the framework.

Second, there exist formats which intentionally do not enjoy representation unicity, such as protocol buffers [102]. We cannot account for such formats with Comparse, perhaps suggesting that they should not be used for secure protocols.

[102]: Varda (2008), *Protocol buffers: Google's data interchange format*

Table 4.2: Related features of other *verified* parser frameworks. We intentionally omit unverified systems.

Paper	Expressiveness					Properties			Execution		
	Structs	Internal dependency	External dependency	Support extensible formats	Textual format (e.g. XML)	Non-ambiguity	Representation unicity	Disjoint formats	Reference implementation	Efficient implementation	Symbolic execution
Comparse	●	●	●	●	○	●	●	●	●	○	●
Everparse+QD [87]	●	●	○	●	○	●	●	○	●	●	○
Everparse+3D [103]	●	●	◐	●	○	●	●	○	●	●	○
Narcissus [104]	●	◐	○	○	○	●	○	○	●	○	○
vGS17 [105]	●	●	●	○	○	●	○	○	●	○	○
MK14 [106]	●	◐	○	●	●	●	○	◐	○	○	◐
Cheerios [107]	Custom format					●	○	○	●	○	○
V2V [108]	ASN.1					●	●	○	○	●	○
Verified Protobuf [109]	Protocol Buffer v3					●	○	○	●	○	○

4.7 Related work

Table 4.2 recaps the capabilities of the various tools we describe here.

Generic verified formats. EverParse with the QuackyDucky front-end [87] generates efficient C implementation of validators and serializers, proves non-ambiguity and representation unicity. QuackyDucky has limited support for data-dependency, e.g. a union must have its tag immediately preceding it. The work introduces the idea that non-ambiguity and representation unicity are important properties for cryptographic protocols, but does not provide a theoretical justification for it, and does not exhibit a concrete set of recommendations like we do (§4.2.3).

[87]: Ramananandro et al. (2019), *Everparse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*

EverParse3D [103] generates efficient C validators from an expressive format description with good support for data-dependency. They prove validator injectivity, which is related to representation unicity, but is more implementation-centric, since it is a property of the validator, as opposed to the underlying format (the relation).

[103]: Swamy et al. (2022), *Hardening Attack Surfaces with Formally Proven Binary Format Parsers*

Narcissus [104] is a Coq library for writing non-ambiguous and non-extensible message formats. It does not consider representation unicity. Narcissus formats are defined using both a state and a relation, whereas we avoid state by relying on our powerful dependent pair combinator. Narcissus also uses combinators, but does not prove general results regarding (say) the non-ambiguity of combinator applications.

[104]: Delaware et al. (2019), *Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats*

[105] defines a deep embedding of data formats into Agda. They relate high-level data types to low-level formats by composing a series of transformations. Lacking support for any sort of frontend format or automatic combinator generation, their format descriptions are not as concise as ours. They reason about non-ambiguity and non-extensibility, but not representation unicity. They also rely on combinators such as dependent pairs. Owing to the nature of the Agda proof assistant, it is unclear to what extent this work can achieve a high degree of proof automation; furthermore, their choice of combinators seems geared towards their IPv4 example and we do not know if their work would scale to e.g. all of the formats of TLS 1.3.

[105]: Geest et al. (2017), *Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data*

Verification of specific formats. Cheerios [107] is a serialization library for Coq types, relying on combinators. They prove non-ambiguity and non-extensibility with an equation similar to ours (§4.3.1). They serialize to a Cheerios-specific custom binary data format that is not user-defined, meaning they cannot target real-world, RFC-prescribed formats.

[107]: (2016), *Cheerios*

[108] prove a C implementation of a specific ASN.1 format used in the automotive industry. They prove non-ambiguity and representation unicity, using an equation similar to ours. The work does not tackle the general question of proving those properties generically, for a certain class of ASN.1 formats.

[108]: Tullsen et al. (2018), *Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System*

[109] builds upon Narcissus [104] to prove verified parsers and serializers for Protocol Buffer 3. They prove non-ambiguity, but not representation unicity, which Protocol Buffer 3 does not enjoy.

[109]: Ye et al. (2019), *A Verified Protocol Buffer Compiler*

Message formats and symbolic security. [106] propose sufficient criteria for secure message formats in a protocol, and prove that an attack on a protocol using concrete formats implies an attack on the protocol with abstract formats. The criterion is that all message formats must be non-ambiguous, moreover they must be pairwise disjoint. In our vocabulary, it means that the protocol must rely on a single, non-ambiguous message format.

[106]: Mödersheim et al. (2014), *A Sound Abstraction of the Parsing Problem*

These criteria are significantly stricter than the ones we propose in §4.2.3, and we believe they are too strict for most real-world protocols. In

particular, composing two unrelated protocols in parallel (e.g. TLS and SSH) violates this property, even though running both at the same time does not impact the security of each protocol. Our criteria (§4.2.3) are more likely to be true on real-world protocols, are preserved by parallel composition, and can be used in DY* proofs.

Furthermore, their approach is less expressive than ours: they do not support data-dependency such as tagged union (which explains their strict disjointness conditions), and they cannot modularly analyze message formats. Finally, they neither provide a formal analysis tool nor a reference implementation for their formats.

4.8 Conclusion

Comparse is a framework to study the security of formats in cryptographic protocols, allowing the user to prove crucial properties such as non-ambiguity, representation unicity and (absence of) data dependency. We demonstrate the expressiveness and effectiveness of Comparse by using it to specify and verify all the formats in TLS 1.3, MLS, and cTLS. Our formats and their guarantees are compatible with both symbolic and computational cryptographic proofs. We integrate Comparse with DY* and show how the format proofs of Comparse can be composed with symbolic security proofs for a variety of protocols. In particular, our framework has been used as part of a security proof for a key component of MLS [68]. Comparse encodes a strong yet flexible discipline that can help protocol designers easily write formats that are provably secure. Our case study on cTLS shows that cTLS formats are secure, and paves the way to a complete security proof.

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

The Messaging Layer Security protocol, and its security analysis in the symbolic model

TreeSync: Authenticated group synchronization

5

This chapter is adapted from the eponymous publication [68], presented at USENIX Security '23. This publication is a Distinguished Paper Award Winner and Co-Winner of the 2023 Internet Defense Prize. The text is identical, but was reformatted. Although the text talks about security proofs on the draft 16 of MLS, they were later updated to the final version of MLS (RFC 9420 [21]).

5.1 Introduction

Whether WhatsApp, Signal, Facebook Messenger, or Wire, virtually all modern messaging applications prominently advertise end-to-end encryption (E2EE) as one of their core features, confirming that private, secure communications are becoming a baseline expectation for many users. Unlike short-lived HTTPS connections, however, messaging conversations can run for years, and so the security guarantees of messaging must account for the realistic possibility that one of the devices is stolen or otherwise compromised during the lifetime of the conversation. If an adversary compromises a device, it can of course read recent messages and send new messages, but we would still like to protect messages that were sent or received well in the past, i.e. *forward secrecy* (FS), and messages that will be sent or received in the future after a period of healing, i.e. *post-compromise security* (PCS).

For two-party conversations, messaging applications rely on modern protocols like Signal [110] to provide FS and PCS by regularly updating (or *ratcheting*) the message encryption keys [53]. However, many messaging conversations involve *groups* of more than two parties. Indeed, since many users have several devices, even a chat between two individuals becomes a group conversation under the hood.

[110]: Marlinspike et al. (2016), *Signal Specifications*

[53]: Perrin et al. (2016), *The Double Ratchet Algorithm*

Group Messaging. The fundamental difference between group messaging and two-party conversations is that groups are dynamic: participants may enter or leave at any time, meaning that the membership (or *roster*) evolves over time. The message encryption keys must also evolve with changes in the roster, so that, for example, a member who has been removed from the group cannot read subsequent messages. To keep track of the group membership, each member needs to continuously synchronize and authenticate the current group state, so they know who they are talking to.

The security requirements for group messaging are also more complex: confidentiality properties like FS and PCS need to be adapted for groups where any member device may be compromised, and authentication guarantees must now also include group membership authentication and sender authentication within the group. Groups can also grow quite large, so a group messaging protocol must provide all these guarantees while scaling to a roster with up to thousands of members. See [5] for a detailed discussion on the challenges of group messaging and a survey of proposed designs.

[5]: Unger et al. (2015), *SoK: Secure Messaging*

Current group messaging applications meet a subsets of these requirements. For example, WhatsApp uses the Signal Sender Keys protocol [20] which uses two-party Signal channels between each pair of members to distribute keys for group conversations. This protocol provides FS and sender authentication, but does not authenticate group membership, does not provide PCS, and its reliance on n^2 Signal channels in groups of size n does not scale to large groups. Signal recently added a private group system [111] that adds membership authentication and privacy but does not improve efficiency.

[20]: Marlinspike (2014), *Private Group Messaging*

[111]: Chase et al. (2020), *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*

To address this state of affairs, the IETF has convened a working group tasked with designing a new secure group messaging protocol, dubbed MLS (Messaging Layer Security). MLS is nearing publication, and an early implementation is already deployed in RingCentral and Cisco's Webex video conferencing platform. The security of MLS is of great interest, as it is likely to be adopted by several messaging applications.

IETF MLS. The IETF MLS working group is tasked with designing a protocol that achieves the goals described in the MLS architecture [55]. The architecture assumes the existence of a trusted authentication service (AS), which attests to relationships between member *identities* and their authentication *credentials*. It also assumes the existence of a mostly-untrusted delivery service (DS), which stores and delivers messages to endpoints and defines a globally unique order for all group modifications. A malicious DS may ignore some messages, or partition the group by selectively delivering messages, but it cannot read or write group messages.

[55]: Beurdouche et al. (2025), *The Messaging Layer Security (MLS) Architecture*

The MLS protocol is currently at version 16 [21] and is in the final stages of standardization. The first few drafts of the MLS protocol were primarily concerned with efficiently establishing group keys for large groups. The starting point was Asynchronous Ratcheting Trees (ART) [112], a protocol that uses a tree of Diffie-Hellman keys to efficiently update and distribute group keys, providing both FS and PCS. ART was replaced (in draft 2) by a more efficient alternative called TreeKEM [113] that is based on Hybrid Public Key Encryption [69]. Subsequent drafts refined and improved TreeKEM, but the fundamental key establishment mechanism remains the same. The TreeKEM protocol (and many of its variants) have been formally analyzed in the literature [51, 52, 114, 115].

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

[112]: Cohn-Gordon et al. (2018), *On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees*

[113]: Bhargavan et al. (2018), *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*

[69]: Barnes et al. (2022), *RFC 9180: Hybrid public key encryption*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*
[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*
[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*
[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*

The group key established by TreeKEM is then used to derive a tree of message encryption keys that each group member can use to send and receive application messages, via a protocol we call TreeDEM (introduced in draft 7).

Both TreeKEM and TreeDEM rely on a group state data structure that must be synchronized across all current members. Most of the remaining complexity of MLS is in defining this data structure, specifying how members can modify the group state to add and remove members, and how the group state is synchronized and authenticated between members. Indeed, many of the recent significant changes in the protocol have been motivated by strengthening the integrity and authentication guarantees of the group state against insider and outsider attacks. For example, an early attack called *double join* allowed a member to resist future removal by surreptitiously adding itself to the group. Avoiding this attack resulted in significant changes to the treatment of the member removal, at the cost of making TreeKEM less efficient. More recent authentication attacks on new members [51, 116] motivated the design of a complex *parent hash* mechanism to protect the integrity of the group state. Despite these

[51]: Alwen et al. (2022), *On The Insider Security of MLS*
[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

attacks and resulting changes, the authentication mechanisms of MLS have not been studied in their own right, and prior works have primarily seen group management from the narrow lens of its impact on key establishment in TreeKEM.

Contributions. In this paper, we focus specifically on the group state management and authentication mechanisms of MLS, which we identify as a separate sub-protocol called TreeSync. We show that MLS can be cleanly decomposed into TreeKEM, TreeDEM, and TreeSync, allowing us to state and prove the authentication and integrity guarantees of TreeSync independently of TreeKEM and TreeDEM.

We present a machine-checked formal security analysis of a byte-level precise specification of TreeSync written in the F* programming language [98]. Our specification is executable and serves as a reference implementation of MLS, which we test and evaluate against other implementations.

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

Our analysis uncovers a new attack that exploits the interaction between TreeSync and TreeDEM, and also highlights other issues in MLS. We propose fixes for these issues, which have been incorporated into MLS.

We prove a series of integrity and authentication theorems for TreeSync in MLS draft 16, using the DY* symbolic protocol analysis framework [43]. Notably, our proofs make no assumptions on the security of TreeKEM, and we only need minimal assumptions on the use of signatures in TreeDEM.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Ours is the first testable, machine-checked, formal specification for MLS. It covers all details of the protocol down to the precise message formats, and hence may be of independent interest to developers and researchers interested in MLS. Conversely, our proofs are only for TreeSync; although we formally specify both TreeKEM and TreeDEM, we leave their comprehensive security analysis for future work.

Outline. We start with a new presentation of MLS as the combination of three independent subsystems (§5.2). We then turn our attention to one of those, TreeSync, and precisely capture its behavior (§5.3). Equipped with the specification, we then set out to formally prove the security of TreeSync in the symbolic model (§5.4). Our contribution is not purely theoretical: our implementation is usable, interoperable, and has been successfully integrated in a prototype version of the Skype messaging client (§5.5). Our proof and implementation combined have influenced both the standard and other implementations: we describe changes to the MLS draft that resulted from attacks we found, as well as bugs in other implementations that were exposed through our work (§5.6). We conclude with related work (§5.7). Our verified implementation is available online as part of the anonymous supplement [117].

[117]: (2022), *TreeSync: Supplementary Material*

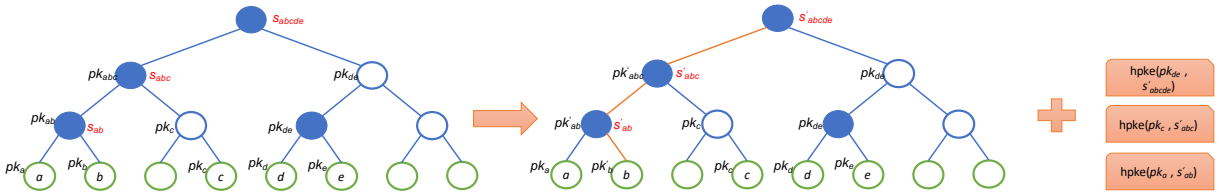


Figure 5.1: TreeKEM maintains a tree of subgroups, each associated with a secret (e.g. s_{abc}) and corresponding public key (pk_{abc}). The root secret (s_{abcde}) is the commit secret for the current epoch. Each member (e.g. b) only knows the secret keys for the subgroups it is a member of ($s_{ab}, s_{abc}, s_{abcde}$). To send a commit, a member (e.g. b) updates its subgroup secrets (to $s'_{ab}, s'_{abc}, s'_{abcde}$) and encrypts each new subgroup secret (e.g. s'_{abc}) to the corresponding sibling subgroup's public key ($hpke(pk_c, s'_{abc})$). Hence, each commit for a group of n (e.g. 8) members results in only $\log(n)$ (= 3) public key encryptions.

5.2 MLS: TreeKEM, TreeDEM, and TreeSync

The Messaging Layer Security (MLS) protocol [21] enables a set of endpoints to form a dynamic group and exchange end-to-end encrypted messages that only the current members of the group can read or write. We begin with a high-level view of this protocol before describing its cryptographic components.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

Dynamic Groups. To initiate a group conversation, an endpoint, called the *creator*, creates a new group and assigns it a fresh group secret. The creator can then add other members to this group by sending them encrypted *welcome* messages containing group information, including the current membership and the group secret. Each member is authenticated by a *credential* issued by a trusted Authentication Service, which associates the member with a signature keypair.

Any member of the group can subsequently *propose* to add or remove members, or update its own credential and/or encryption keys. A group member can *commit* a batch of pending proposals by modifying the group, updating the group secret, and conveying the new secret to the updated set of group members. Each commit is said to open a new *epoch* (group creation is at epoch 0), so the group secret at epoch n is more precisely called the *epoch secret* at n.

An epoch secret should only be known to the current members of the group in that epoch. Hence, the protocol seeks to ensure that members cannot read or write messages after they are removed, and new members cannot read old messages.

Secure Messaging. Within each epoch, the epoch secret is used to derive message encryption keys that members of the group can use to securely exchange application messages.

The protocol generates fresh encryption keys for each message to guarantee forward secrecy (FS): compromising a member's secrets should not allow the adversary to decrypt previous messages sent or received by that member. Note that the FS guarantee depends on the secure deletion of these messages on each member device [118, 119].

[118]: Albrecht et al. (2021), *Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong*
[119]: Marlinspike (2016), *Disappearing messages for Signal*

Furthermore, as long as each member regularly updates its encryption keys, the protocol provides post-compromise security: an adversary who learns a member's secrets at epoch n, but does not interfere until the member's keys are updated at epoch $m > n$, cannot decrypt messages after epoch m.

Decomposing MLS. Given this high-level view of MLS, the main cryptographic elements that need elaboration are: how a committer

computes the new epoch secret and conveys it to the current group members, how application messages are encrypted in a way that provides authentication and forward secrecy, and how the protocol guarantees that all members of the group have a consistent view of the group membership and structure. In the remainder of this section, we describe sub-protocols of MLS that implement each of these components.

5.2.1 TreeKEM: Establishing Epoch Secrets

To participate in an MLS group, each endpoint e must first upload a signed *key package* containing its credential (including a signature verification key), a public encryption key pk_e, and other protocol parameters (e.g. supported ciphersuites). So, when a group member decides to add e to a group, it can use pk_e to encrypt a *welcome* package for e containing enough information for e to initialize its state and join the group.

At epoch 0, the epoch secret is derived from a fresh random value. Thereafter, at each commit, the committer computes a *commit secret* and sends it to all the members of the new epoch. Each member then mixes the commit secret with the previous epoch secret (at $n-1$) to obtain the new epoch secret (at n). A naive approach would be for the committer to generate a fresh commit secret and encrypt it for each group member using their public keys. However, in a group of size n, this design requires n expensive public-key encryptions for each commit, which does not scale well to large groups.

TreeKEM defines a more efficient commit operation by structuring the group as a complete binary tree, as depicted in Figure 5.1. The non-empty leaves of the tree contain the key packages of the current group members (a, b, c, d, e). Each internal node (also called *parent node*) corresponds to a subgroup consisting of the members underneath that node, and is associated with a *node secret* (e.g. s_{abc}) that is known only to its members (a, b, c). Each node secret (s_{abc}) is used to derive a public encryption key for the corresponding subgroup (pk_{abc}). The node secret at the root (s_{abcde}) is known to the full group and is used as the commit secret for the epoch.

The benefit of the binary tree data structure is that when a committer (say b) wishes to convey a new commit secret to a group of size n, it only needs to compute and convey a single message containing $\log(n)$ public-key encryptions. Essentially, the committer (b) generates a fresh secret s_b, uses it to derive a sequence of node secrets for the path from its leaf to the root ($s'_{ab}, s'_{abc}, s'_{abcde}$), and conveys each new node secret to the rest of the subgroup by encrypting it under the corresponding sibling node's public key (pka, pk_c, pk_{de}). Each recipient (e.g. c) decrypts the node secret for the smallest subgroup it shares with the committer (s_{abc}), and derives the sequence of node secrets up to the root (s'_{abcde}).

In general, a parent node can be *blank*, or it may have *unmerged leaves*, which means that it is associated with not one but a set of public keys that collectively covers the members below that node. Consequently, the cost of each commit can actually vary between $\log(n)$ and n. TreeKEM also optimizes for the case when one of the children of a parent node is an empty subtree, by skipping the computation of that node's secret and treating it as blank, with the same public key as its non-empty subtree. Such nodes (e.g. the parent of de in Figure 5.1) are called *filtered nodes*.

We have given only a simplified summary of TreeKEM. The full TreeKEM protocol has many other details that we elide here, since they are unimportant for this paper. Several prior works have formally analyzed TreeKEM and its variants and shown that it implements a security definition called Continuous Group Key Agreement (CGKA) [114, 115]. Other work has analyzed the way epoch secrets are derived in TreeKEM [52].

[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*
[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*
[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*

TreeKEM Tree Invariants. For our purposes, the pertinent observation is that the security of TreeKEM crucially relies on a *tree secrecy invariant*: if an internal node is associated with a node secret, then it can only be known to the members underneath that node. Recall that each parent node secret is chosen during a commit by a member below that node, but it is then encrypted under one of its children's public keys. Consequently, the TreeKEM secrecy invariant relies on the integrity of the public keys in the tree, which can be expressed as a *tree integrity invariant*: if an internal node is associated with a public encryption key, then this public key was computed for (a subset of) the current members of the node's subgroup, by one of the members of the subgroup.

5.2.2 TreeDEM: Group Message Encryption

Given an epoch secret, the TreeDEM protocol derives message encryption keys for each member in the current group and specifies how they are used to send and receive application messages, handshake messages (containing TreeKEM proposals and commits), and Welcome messages for new members. We briefly describe how TreeDEM authenticates and encrypts application messages.

Each application message is serialized into a bitstring along with metadata indicating the group, epoch, and sender. This bitstring is then signed with the sender's signing key (to authenticate which member sent the message) and then MACed with a key derived from current epoch secret. The serialized message, its signature, and MAC are then encrypted using an authenticated encryption (AEAD) scheme using the sender's current message encryption key. The recipient performs the reverse set of operations to decrypt the message, verify the signature and MAC to ensure that the message was sent by a sender who is a member of the group.

After each message is sent or received, the sender's message encryption key is updated (or *ratcheted*) using a key derivation function to provide forward secrecy: compromising a group member after a key has been updated does not reveal prior keys or messages encrypted under those keys.

Each group member needs to keep track of the current encryption keys for all members and update these keys at every application message. In large groups, maintaining all these keys can be costly, especially if only a small minority of members send messages in a each epoch. Consequently, TreeDEM uses a tree-based message key derivation technique that lazily derives keys for each sender, to reduce the number of keys each member needs to maintain.

The security functionality provided by TreeDEM is sometimes called Forward-Secure Group Authenticated Encryption with Associated Data (FS-GAEAD) and has been analyzed in prior work [115]. For the purposes of this paper, the pertinent feature of TreeDEM is its reliance on the group

[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*

tree data structure for key derivation, and that it uses sender signatures to authenticate MLS messages.

5.2.3 TreeSync: Group State Synchronization

MLS relies on group members having a consistent view of the group state. Specifically all members must agree on (and authenticate) the membership of the group and the structure and contents of the public key tree (as depicted in Figure 5.1). Otherwise, a member may be fooled by an attacker into sending messages to groups it did not intend to communicate with.

Our key observation in this paper is that the task of synchronizing and authenticating the group state can be seen as an independent *generic* sub-protocol with minimal dependencies on TreeKEM and TreeDEM. This allows us to modularly analyze the group authentication guarantees of MLS without getting bogged down by the details of these other protocols.

We identify a protocol called TreeSync that encapsulates all operations on the MLS group state, while treating TreeKEM-related content as opaque bitstrings. We describe TreeSync in detail in §5.3, and we formalize and analyze its integrity and authentication guarantees in §5.4, under minimal assumptions on TreeKEM and TreeDEM. We also show that TreeSync provides some of the prerequisites for the security of TreeKEM and TreeDEM, such as the TreeKEM tree integrity invariant, and the Tree Hash construction.

Our treatment of TreeSync is in contrast to the MLS specification [21], which tightly interleaves its description of group synchronization with the key derivations of TreeKEM. Prior work on the authentication mechanisms in MLS [51] also follows this pattern by combining them with TreeKEM, which in our opinion results in unnecessarily complex proofs.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

Authentication Attacks on MLS. Furthermore, many prior attacks on MLS can actually be better understood as attacks on TreeSync. For example, a *double join* attack occurs when a malicious member at leaf i manages to modify the content of a parent node that is not its ancestor [120]. In the *welcome message* attack, an attacker fools a new member into accepting a tampered tree with compromised public keys [116]. In the *tree signing attack*, the attacker changes the position of leaves in the tree to fool a new member [51]. Each of these attacks resulted in major changes to the protocol, significantly raising its complexity and reducing its efficiency. By identifying and analyzing TreeSync, we provide a formal framework for finding such attacks and evaluating defenses against them. Indeed, we identified flaws in the authentication and integrity mechanisms of MLS and fixed them during this work.

[120]: Barnes (2018), *Remove without double-join (in TreeKEM)*

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

5.3 A Formal Specification of TreeSync

In this section, we describe the TreeSync protocol and its detailed formal specification in the F* language [98]. Unlike prior analyses of MLS that are based on high-level models written as pseudocode [51, 52, 114, 115], our F* specification is executable, and hence testable against the RFC test vectors and other MLS implementations. It accounts for all the low-level

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

details of MLS, and so serves as both a formal companion to the RFC and a reference implementation.

The precision of our specification also means that our analysis is less likely to miss attacks. For example, in §5.4.5, we show a new attack that results from the ambiguity of the message formats between TreeSync and TreeDEM, and would not appear in more abstract models.

The authentication mechanisms of TreeSync are complex, with performance optimizations interleaved with cryptographic constructions. Our goal is to informally explain the design and its motivations, and guide the reader to the F* specification for full details. Our full specification and all proofs are included in supplementary material [117].

[117]: (2022), *TreeSync: Supplementary Material*

5.3.1 TreeSync data structures

We present a *generic* tree data structure that can be instantiated to obtain the TreeSync and TreeKEM trees. This is in contrast with the RFC, which combines the two trees.

```
type tree (leaf_t:Type) (node_t:Type) (l:nat) (i:tree_index l) =
  | TLeaf:
    data: leaf_t{l == 0} →
    tree leaf_t node_t l i
  | TNode:
    data: node_t{l > 0} →
    left: tree leaf_t node_t (l—1) (left_index i) →
    right: tree leaf_t node_t (l—1) (right_index i) →
    tree leaf_t node_t l i
```

The type `tree left_t node_t l i` describes a complete binary tree indexed by its height l – we follow the RFC convention that a standalone leaf has height 0. The type `tree` is parametric over `leaf_t`, the payload of the leaves, and `node_t`, the payload of the non-leaf (a.k.a. "parent") nodes. The leaves of the tree (i.e. the participants) are numbered left-to-right from 0 to $2^l - 1$. Hence, each leaf has an absolute *leaf index* that represents its position in the full tree.

We leverage F*'s dependent types to encode structural invariants on the tree. Notably, the `i` argument to `tree` enforces a correct-by-construction tracking of leaf indices, rules out programmer errors, and enforces the MLS invariant that two sub-trees at different positions, even if otherwise identical, are never interchangeable.

To obtain the TreeSync tree (called `treesync`), we instantiate the tree with the content of parent and leaf nodes:

```
type parent_node = {
  opaque_content: node_content;
  parent_hash: mls_bytes;
  unmerged_leaves: mls_list uint32; }

type leaf_node = {
  opaque_content: leaf_content;
  parent_hash: leaf_node_parent_hash;
  signature_key: signature_public_key;
  ...
  signature: mls_bytes; (* signs all the fields above, and more *) }

let treesync = tree (option leaf_node) (option parent_node)
```

The TreeSync tree must include some content provided by and useful for TreeKEM, such as public keys, but our specification treats these protocols as independent modules, as evidenced by the opaque_content fields: TreeSync is oblivious to the particular payload that its nodes carry.

Note that the mls_bytes type is a convenient abbreviation that enforces that the length of bytes we manipulate does not exceed $2^{30} - 1$, a requirement coming from the compact integer encoding of the QUIC standard [121], which MLS itself adopts. We refer the reader to the supplementary material for the full definitions of all our data structures.

[121]: Iyengar et al. (2021), *QUIC: A UDP-Based Multiplexed and Secure Transport*

Parent nodes and leaf nodes. In TreeSync, participants reside at the leaves; a participant is therefore identified by its *leaf index*. Non-leaf nodes are known as *parent nodes*, and contain the cryptographic material that allows efficiently updating the tree while preserving authentication guarantees.

Blank nodes and empty leaves. In the treesync datatype above, the leaf and parent node payloads are *optional*. Empty leaf nodes happen because the RFC mandates a complete binary tree, meaning some participant (leaf) nodes might be empty, as illustrated in Figure 5.2. Empty parent (inner) nodes are called *blank nodes* in the RFC, and arise either from participant removals, or because of filtered nodes (§5.2.1).

5.3.2 TreeSync operations

TreeSync offers a series of group management operations that members can use to modify and synchronize the group state. In particular, any member can create a *proposal* message to suggest a change (e.g. add or remove a member) and send it to the rest of the group, via the Delivery Service (DS). A group member can then collect a set of proposals and send a *commit* message for these proposals along with a *path update*. None of these sending operations actually change the TreeSync tree; instead, each member waits for a commit to be accepted by the DS and sent back before executing the proposed changes. Hence, the DS resolves potential conflicts by choosing the order of commits for the whole group.

When a commit is processed, each of the proposals is executed in order to modify the local TreeSync state. In the rest of this subsection, we discuss how each of these changes is implemented. The key guiding principle for all the operations in TreeSync is that they must preserve the *tree integrity invariant*: every subtree with a non-blank node must have

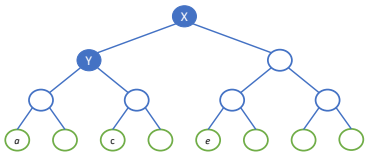


Figure 5.2: An MLS tree with three participants a, c and e at leaf indices 0, 2 and 4; other leaf nodes are empty. Due to the filtered node optimization, X and Y are the only parent nodes that are not blanked.

been authenticated by a participant at one of the leaves of the subtree. To enforce this invariant, TreeSync relies on the parent hash mechanism described in §5.3.4.

Processing Path Updates. Each commit operation ends with a *path update* that updates all the nodes on the path from the root down to the sender's leaf, updating the tree integrity mechanisms along the way. The function implementing path updates in F* has type as follows, where we omit boilerplate:

```
val apply_path: #l:nat → #li:leaf_index l 0 →
  t:treesync l 0 → p:pathsync l 0 li → treesync l 0
```

The apply_path function allows the client to update tree t of height l with a new path p, where p follows the path from root to leaf li, and carries fresh content for each node (including leaf) found along the path. The apply_path function, in addition to updating content along p, also updates the integrity protections at each node, as we will see later in Figure 5.3.

In line with our dependent type definition for trees, the leaf index li not only guarantees that the path terminates at participant (leaf) li, but also allows us to keep track of the leftmost leaf index as we move along the path. Once again, carrying such indices not only avoids errors, but greatly simplifies and automates our proofs. The 0 in the type signature is another invariant enforced "for free" by typing: this function is only intended to be called on a path starting at the root.

Processing Removal and Addition. The functions implementing add and remove have types as follows.

```
val tree_remove: #l:nat → #i:tree_index l → t:treesync l i →
  li:leaf_index l i → treesync l i
val tree_add: #l:nat → #i:tree_index l → t:treesync l i →
  li:leaf_index l i → ln:leaf_node → treesync l i
```

We note that adding a member may increase the size of the tree, and removal can shrink the tree. The full types of these functions include preconditions (omitted here, see [117]) that rule out various overflow conditions in various TreeSync structures whose length is bounded by the RFC.

[117]: (2022), *TreeSync: Supplementary Material*

If a wishes to remove c from the tree (Figure 5.2), MLS requires blanking out all of the nodes starting from c (a leaf), all the way up to, and including, the root. The net effect is that any cryptographic material that c may have authored is now gone from the tree. The RFC also mandates that each removal be enclosed in a commit that includes a path update by the committer a, which updates the contents of all nodes from a to the root, authenticates the removal, and restores integrity protections for the full tree at the root.

If a committed e wishes to add b to the tree (Figure 5.2), e fills out the first non-empty leaf (at index 1) with b's data. The path update from e then updates the nodes between e and the root (e.g. X). However, there

may be nodes between b and the root which are not updated (e.g. Y). These nodes will only be updated in a subsequent commit by one of the members under them (a or b). In the meantime, TreeSync performs supplemental book-keeping using *unmerged leaves*.

Unmerged leaves. Each node now contains a *list* of unmerged leaves (or *unmerged list*), with the invariant that participants in that list belong to the node's subtree. If b is in the unmerged list of Y, then it indicates that the addition of b to the subtree under Y postdates the modification of the node Y.

Addition now works as follows: after the first non-empty leaf has been filled with the new participant data, addition also extends the unmerged list of every node on the path from the new participant all the way up to the root. Concretely, after adding b's data at leaf index 1, e inserts b into the unmerged list of Y and X. Only then does e issue a path update.

Path updates clear the unmerged list of each node they visit, so when e issues a path update, the root's unmerged list is empty after the update, meaning that any integrity protections added to the root now cover the entire group – all is well.

Serialization and Parsing. Our specification implements the serialization and parsing of all the MLS data structures, for trees, messages, and signature contents down to the byte level – we follow the RFC to the letter. This is to be contrasted with all prior formal approaches, which study a *model* of MLS that is not guaranteed to be faithful to the RFC. As a consequence, our specification can actually be extracted and executed to establish that we are interoperable with test vectors and other implementations (§5.5); this level of precision also allowed us to find new attacks, such as signature collisions (§5.4.5).

5.3.3 Tree Hash

The MLS specification defines a tree hash operation that computes a digest for an entire tree; we rely on this operation in subsequent sections. The implementation details are of little importance for the rest of this paper: it suffices to say that the RFC implements an efficient recursive hash procedure akin to that of a Merkle Tree, and should two different MLS trees exhibit the same tree hash, then one has found a collision for the underlying hash function (§5.4.3). Proving this requires, naturally, reasoning about injectivity of serialization.

The tree hash provides an integrity mechanism for the MLS tree: if two members have the same tree hash they must have the same tree (barring hash collisions). Consequently, by including the tree hash within a signature, a sender can authenticate the full tree to a receiver. However, this integrity guarantee is not strong enough to protect new group members from tree tampering attacks by old members, such as the welcome message attack [116] and tree signing attack [51]. Consequently, MLS includes a second, stronger integrity mechanism called the Parent Hash.

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

5.3.4 Parent Hash

We have now exposed all of the TreeSync operations. Throughout our explanations, we have consistently referred to the need for a mechanism

```
let rec apply_path_loop #l #i #li (t: treesync l i) (p: pathsync l i li)
  parent_parent_hash
= match t, p with
  // End of path: apply new contents from p onto t
  | TLeaf _, PLeaf lp → TLeaf (Some lp)
  | TNode _ left right, PNode opt_opaque_content ps →
    let _, sibling = get_child_sibling t li in
    let opt_content' = // Compute the new node content
      match opt_opaque_content with
      | None → None // We skip filtered nodes
      | Some content → Some ({
        opaque_content = content;
        // Carried from previous loop iteration
        parent_hash = parent_parent_hash;
        // Notice we clear the unmerged leaves list
        unmerged_leaves = []; }) in
    // Compute the parent's parent hash for recursive call
    let parent_parent_hash' =
      match opt_opaque_content with
      | None → parent_parent_hash // We skip filtered nodes
      | Some content →
        compute_parent_hash content parent_parent_hash sibling
    in
    // Update the tree recursively
    if is_left_leaf li then ( // relative to tree of height l at position i
      let left' = apply_path_loop left ps parent_parent_hash' in
      TNode opt_content' left' right
    ) else (
      let right' = apply_path_loop right ps parent_parent_hash' in
      TNode opt_content' left right')
```

Figure 5.3: Implementation of the apply_path function, simplified. We write x' for the updated value for x.

that can protect the integrity of the whole tree, while i) correctly accounting for both unmerged leaves and blank nodes mechanisms, and ii) satisfying the constraint that a path update can only modify nodes along the path. That integrity mechanism is known as the Parent Hash, and must accommodate further requirements: first, the number of hash computations and recomputations should be minimal (for efficiency); second, the parent hash should cover the contents of all the subtrees that existed in the tree when the parent hash was last modified.

We now expose the Parent Hash mechanism as specified within the RFC. After being confronted with its ungodly amount of complexity, the reader will, we hope, be convinced that this warrants a formal analysis that can state, with full confidence, that TreeSync not only is correct, but also provides proper authentication guarantees.

Computation. Each node in the tree stores a parent hash. When a path update is applied, the parent hash of each node on the path is recomputed, starting at the root, and continuing all the way down to the leaf (participant) that issued the path update. We show the inner (recursive) loop of apply_path, in Figure 5.3.

To initialize recursion, the parent hash stored in the root node is always a special empty value. Then, at any given step along the way (with N the current node, S its sibling, and P their parent), the parent hash stored in N is updated to the hash of a serialized structure containing:

- the (new) parent hash stored in P,
- the (new) opaque payload stored in P, and

- the tree hash of S, which fully captures the contents of S, unmerged leaves included.

At the end of the path update issuance, the leaf signs its own parent hash. Doing so, the participant signs (authenticates) their own membership in the tree, as well as the content of their parent P, the entire sibling tree S, and whatever else the parent hash of P recursively covers. Transitively, this means that the leaf contains a hash value that protects the integrity of every node, sibling and parent, all the way up to the root.

Recall that when a node in the path has a blank subtree, it is called a *filtered node* and is treated as blank; in this case, apply_path_loop skips the node and moves down to its child. Figure 5.2 illustrates this optimization: if c issues a path update, its parent node is *skipped*, and only Y and X get updated.

Each path update also clears the unmerged list of every node on the path, as the nodes that were in the unmerged list are now authenticated by the update. Accounting for unmerged leaves and filtered nodes significantly complicates the implementation of all the operations in TreeSync; this is one of the many reasons that motivate a formal proof.

Verification. Perhaps harder than updating the parent hash is *verifying* its correctness to prevent against malicious actors. This happens in two circumstances: first, upon joining a group; second, upon receiving a commit from another group member. In the first scenario, the whole tree must be visited; in the second scenario, this is only an incremental process wherein a lot of values from tree hash can be cached and reused.

Several subtleties arise in the process. We give an intuition for two of those, and leave a formal discussion of correctness properties to §5.4. First, a node N might have a non-empty unmerged list. This means that in order to validate the parent hash stored at N, one must consider the subtree at the time of the last authentication of N, that is, the subtree *without* the unmerged leaves. This requires introducing a new operation revert_add(P,leaves) operation that allows us to revert back the addition of a set of unmerged leaves (leaves) from a tree rooted at a parent node (P), so that we can compute a correct, albeit outdated, hash. The second complexity arises from the filtered nodes optimization. Notably, one must ensure that a malicious actor cannot surreptitiously introduce new nodes in an otherwise filtered (skipped) subtree.

Failing to account for both of these subtleties breaks our integrity invariant and can allow attacks on the protocol. Our authentication proof for TreeSync relies on a novel criterion, the "parent-hash link", that ties together the parent hash, the blank (skipped) nodes, and unmerged leaves together (§5.4).

This concludes our overview of the main elements of TreeSync, which itself only forms a small part of the MLS standard. The protocol is large enough and complex enough that we believe that it is hard, even for experts, to understand all the details, let alone reason about its security. We provide a testable specification for all of MLS in F* [117], which readers can inspect and run to hopefully gain a better understanding of the protocol and the machine-checked authentication theorems we proof for the TreeSync component.

[117]: (2022), *TreeSync: Supplementary Material*

5.4 A security proof of TreeSync

In this section, we describe a series of invariants and lemmas we prove for our TreeSync specification leading up to the main integrity and authentication guarantees of the protocol.

5.4.1 TreeSync State Invariants

As we saw in §5.3.1, the TreeSync tree data type already incorporates several structural invariants (complete tree, correct leaf index). In addition, we state and prove a series of invariants for all TreeSync states that are reachable by a sequence of operations. We describe three of these invariants, which play important roles in our security proofs:

Unmerged Leaves. At each parent node n, the unmerged leaves list must be sorted in increasing order, each unmerged leaf must point to a leaf index within the subtree rooted at n, and the leaf at this index must be non-blank.

This invariant can be easily checked for every TreeSync tree, and is necessary to prove the parent-hash invariant described below, but surprisingly, the latter two conditions were not required by the MLS draft. On our suggestion, they are now included since draft 15.

Leaf Validation. We require and enforce an invariant that all leaf signatures in the tree have been verified, and that the credential at each leaf has been issued (out-of-band) by the Authentication Service. Hence, we can assume that the verification key in the credential belongs to the member at the leaf and has been used to sign the leaf content. These are crucial pre-conditions for the authentication guarantees of TreeSync.

Parent-hash Linking. The parent hash construction (§5.3.4) creates links between parent nodes and their descendants. Formally, if a parent node P has two children C and S, we say that there is a *direct parent link* between C and P if, once we revert all the unmerged leaves of P (revert_add(P,P.unmerged_leaves)): (1) P and C are non-blank, (2) C has no unmerged leaves, and (3) C contains a parent-hash computed from P and S (C.parent_hash is equal to ParentHash(P.content, P.parent_hash, TreeHash(S))).

More generally, we say that there is a *parent link* from a descendant node D to P (written $D \rightsquigarrow P$) if P and D satisfy the conditions above and there is a path from D up to P such that all intermediate nodes on this path are filtered, i.e. they are blank and the corresponding sibling trees are fully blank. This generalization is needed because filtered paths may introduce blank nodes between a node and its linked parent.

We show that TreeSync enforces the invariant that each non-blank node P must have a descendant D such that $D \rightsquigarrow P$. By applying this invariant recursively, we obtain a more general notion of *path linking*: a leaf L is *path linked* to an ancestor node P, if all the non-blank nodes on this path $(T_1, \ldots, T_n)$ are sequentially parent linked ($T_i \rightsquigarrow T_{i+1}$). As we shall see, this is a crucial invariant for our authentication theorem.

F* Proofs. We formalize all our invariants on the TreeSync tree as a predicate which we attach to the treesync data structure as a *refinement type*. Thereafter, we use the F* type checker to prove that all TreeSync operations that modify the tree data structure preserve this predicate. The proofs rely on some auxiliary lemmas but are mostly straightforward.

5.4.2 Verified Parsing and Serialization

Our F* specification includes parsers and serializers for all the byte formats defined in the MLS RFC, whether they represent trees, messages, or inputs to cryptographic constructions. We uniformly prove correctness properties for all these parsers and serializers, whether or not they belong in TreeSync.

In particular, for every MLS type T, we define a function serialize_T that translates T to bytes, and function parse_T that translates bytes to option T. We then prove that these functions are inverses of each other, and as a corollary, obtain that the serialization of each MLS type is *injective*.

```
∀ (x:T). parse_T (serialize_T x) = Some x
∀ (x:T) (b:bytes). parse_T b = Some x ⟹  serialize_T x = b
```

These properties are essential for functional correctness, but also for security. For example, the TreeHash construction relies on the serialization of a structure called TreeHashInput that includes the node type and hashes of the children (if any). We rely on the injectivity of this serialization to prove the integrity of TreeHash. Conversely, the failure of an injectivity lemma may point to an attack, as we will see in the case of the signature confusion attack on TreeSync authentication.

F* Proof. To prove all our parsers and serializers correct, we rely on a verified library of parser combinators in F* that largely automate the process of defining and verifying this code. This library allows us to write the RFC types as regular F* data types decorated with annotations describing how they should be serialized. Using F*'s *metaprogramming* feature, these types are automatically translated to parsers and serializers equipped with proofs of correctness.

5.4.3 Tree Hash Integrity Lemma

The TreeHash construction is used to verify the integrity of TreeSync trees: two members of a group can compare their tree hashes to verify if the trees are the same.

This integrity guarantee relies on the injectivity of TreeHash: if two subtrees t_1 and t_2 have the same tree hash (TreeHash(t_1) =TreeHash(t_2)), then either two trees are equal ($t_1 = t_2$), or else we can exhibit a pair of bitstrings b_1 and b_2 that exhibit a hash collision ($b_1 \neq b_2 \wedge H(b_1) = H(b_2)$).

In other words, a collision in TreeHash deterministically reduces to a collision in the underlying hash function. By structuring the lemma in this manner, we avoid making any symbolic or probabilistic assumption on hash functions.

The formal statement of this lemma in F* is given below:

```
val tree_hash_injectivity:
 #l1:nat → #i1:tree_index l1 → #l2:nat → #i2:tree_index l2 →
 t1:treesync l1 i1 → t2:treesync l2 i2 → Pure (bytes * bytes)
 (requires tree_hash t1 == tree_hash t2)
 (ensures λ (b1, b2) →
  // Either the trees are equal and at the same position
  (l1 == l2 ∧ i1 == i2 ∧ t1 == t2) ∨
  // Or we computed a hash collision
  (hash b1 == hash b2 ∧ ¬(b1 == b2)))
```

Importantly, note that the lemma not only guarantees that the trees have the same content and structure, but also that they are at the same position, which is needed in the Parent Hash Integrity lemma below. The integrity of TreeHash is also relevant for TreeDEM, which authenticates the current tree hash in every message, hence guaranteeing that recipients and senders of each MLS message have the same tree.

F* Proof. Our proof of this lemma in F* is by induction on the structure of the two trees and case analysis on the TreeHash definition. It relies on the injectivity of serialization for the TreeHashInput type and as it travels down the trees, it inductively constructs the bitstrings that must exhibit the hash collision if the trees are not the same.

Our proof is similar to prior proofs for Merkle Trees (e.g. see [122, Section 7]). However, we note that even well known Merkle Tree implementations sometimes have subtle bugs [123], making them good targets for formal proof.

[122]: Protzenko et al. (2020), *Evercrypt: A fast, verified, cross-platform cryptographic provider*

[123]: Voight (2012), *CVE-2012-2459 (block merkle calculation exploit)*

5.4.4 Parent Hash Integrity Lemma

Unlike the TreeHash, which is invalidated every time the tree is modified, the Parent Hash provides a more flexible integrity guarantee for subtrees that may, for example, have some unmerged leaves added after the last commit. To state the Parent Hash Integrity lemma, we first define a notion of tree equivalence that captures this flexibility, then define one step of the lemma before defining the lemma for the full tree.

Canonicalization and Equivalence. We define the canonicalization of a subtree T with respect to leaf index L, written canonicalize(T, L), by reverting the unmerged leaves at its root (revert_add(T, T.unmerged_leaves)) and by ignoring the signature value from leaf L. As we will see, if L is path-linked to T, canonicalize(T, L) captures precisely what is covered by L's signature. Because L's signature covers neither itself nor the unmerged leaves of T, we omit both in the canonicalization.

We say that two trees T and T' are equivalent with respect to a leaf index L, written $T \simeq_L T'$, if the two trees have the same canonicalization with respect to L.

Parent Link Integrity. Next we prove a lemma that shows how the parent link relation ($D \rightsquigarrow P$) protects the integrity of the tree. Consider two trees P_1 and P_2 where, P_1 has a descendant D_1 such that $D_1 \rightsquigarrow P_1$, and P_2 has a descendant D_2 such that $D_2 \rightsquigarrow P_2$. We prove that if $D_1 \simeq_L D_2$ then $P_1 \simeq_L P_2$. That is, the parent link relation ($\rightsquigarrow$) enables us to lift the equivalence relation ($\simeq_L$) up the tree.

As with TreeHash, we state this lemma in terms of a function that either proves the equivalence of P_1 and P_2 or finds a hash collision. The statement of the lemma in F* is:

```
val parent_link_integrity:
 #ld1 → #ld2 → #lp1:nat{ld1 < lp1} → #lp2:nat{ld2 < lp2} →
 #id1:tree_index ld1 → #id2:tree_index ld2 →
 #ip1:tree_index lp1 → #ip2:tree_index lp2 →
 d1:treesync ld1 id1{node_has_parent_hash d1} →
 d2:treesync ld2 id2{node_has_parent_hash d2} →
 p1:treesync lp1 ip1{node_not_blank p1} →
 p2:treesync lp2 ip2{node_not_blank p2} →
 (* leaf index of L *) li:leaf_index ld1 id1 → Pure (bytes * bytes)
 (requires equivalent d1 d2 li ∧ parent_hash_linkedP d1 p1 ∧
   parent_hash_linkedP d2 p2) // Given the hypotheses
 (ensures λ (b1, b2) →
  equivalent p1 p2 li ∨ // Either the theorem is true
  (hash b1 == hash b2 ∧ ¬(b1 == b2))) // Or we have a collision
```

Parent Hash Integrity. By recursively applying the Parent Link Integrity lemma above, we obtain the full integrity guarantee for a path from a leaf to each of its ancestor nodes. Consider two trees T_1 and T_2, where T_1 has a leaf L_1 such that L_1 is path-linked to T_1, and T_1 has a leaf L_2 such that L_2 is path-linked to T_2. We show that if L_1 and L_2 have the same content and same leaf index, and if T_1 and T_2 have the same height, then $T_1 \simeq_L T_2$.

As a corollary, we obtain a lemma that is more directly useful for TreeSync authentication: if T_1 is the root node (i.e. its parent hash field is empty) , then T_2's height cannot be greater than T_1's, and all the subtrees between L_2 to T_2 must be point-wise equivalent to the corresponding subtrees on from L_1 to T_1. In practice, after every commit, the path update corresponds to a linked path from the committing leaf (e.g. L_1) to the root (T_1). However, as other leaves subsequently commit to the tree, the linked path no longer goes to the root and may be shorter (e.g. up to T_2).

F* Proofs. The proof for the parent link integrity lemma is similar to that of the Tree Hash integrity lemma. We rely on the injectivity of serialization, and the injectivity of tree hashes, and perform a case analysis on the parent hash definition to construct a hash collision if the two trees are not equivalent. The full parent hash integrity lemma is proved by induction on the length of the trees, propagating the hash collision up the tree. Due to the subtleties and many corner cases of the parent hash computation, we found that having a proof assistant like F* to check all the cases was quite valuable.

Weakness in Previous Drafts. We note that previous drafts of MLS (before draft 13) did not satisfy the Parent Hash Integrity lemma we state and prove in this section. This is because the parent hash construction did not include the Tree Hash of the sibling and instead only included the list of public keys (called the *resolution*) in the sibling tree, i.e. C.parent_hash is equal to ParentHash(P.content, P.parent_hash, Resolution(S)). Notably, the Resolution does not include the credentials of the leaves in S. This allows an adversary to tamper with the tree, by changing the leaf credentials in S, without it being detected via the parent hash mechanism.

Incidentally, the resolution mechanism was itself introduced in response to an attack (described in [51]) on the integrity protections of the parent hash mechanism in draft 9. Our analysis shows that there still are integrity attacks on the parent hash mechanism after this fix. We proposed the

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

change to include the Tree Hash instead of the resolution and this was adopted in draft 13 of the standard. The change also has the benefit of more cleanly separating TreeSync mechanisms like parent hash from TreeKEM objects like public keys.

5.4.5 TreeSync Authentication Theorem

We can finally state the high-level TreeSync Authentication theorem. Consider the TreeSync tree T at group member b, obtained as a result of a valid sequence of TreeSync operations. Then, the theorem states that within every subtree T' of T where the root of the subtree is non-blank, there exists a leaf L in T' with a credential for some member a, such that either at some point in the past, the TreeSync tree at a contained the canonicalization of T' with respect to L (canonicalize(T', L)), or else a's signature key must have been compromised.

In other words, in every TreeSync state, every subtree with a non-blank root node is authenticated (up to the flexibility offered by equivalence) by one of the leaves in that tree. Notably, after a path update, the root of the full tree is guaranteed to neither be blank nor have unmerged leaves; the full TreeSync tree is thus always authenticated by some group member.

The authentication guarantee above is the first instance in our formal development where we are relating the state at one member (b) with the state at a different member (a). To formally state and prove this theorem, we need a runtime model that incorporates multiple parties and their interactions. To this end, we employ the DY* symbolic protocol framework.

Verifying Crypto Protocols with DY*. The DY* framework [43] defines a trace-based symbolic runtime model, where different *principals* can participate in protocols by calling cryptographic functions, generating keys and nonces, sending messages to each other, storing and modifying local state, and logging events to indicate authentication events. The attacker controls the network and can compromise principals: it can read and write any message, generate any number of keys, read the state of compromised principals, and store any amount of state for itself.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

DY* implements a symbolic (or Dolev-Yao) abstraction of cryptographic functions, modeled using constructors and functions in F*. Here, we only use the hashing and signature functions in DY*. Hash functions are modeled as opaque one-way functions with no collisions. Signature schemes are modeled as three functions: a key generation function that produces signature keypairs, a signature function that signs a bitstring using a signature key, and a verification function that takes a verification key, a bitstring, and its signature to verify. Verification succeeds if (and only if) the signature was computed with the signature function, meaning signatures are unforgeable unless the signature key is known to the attacker.

The trace-based runtime model and symbolic cryptographic assumptions of DY* are quite standard for symbolic verification and similar to models used in ProVerif [41] and Tamarin [42]. The main difference is the way proofs work in DY*. DY* is built as a library within the F* verification framework and hence has access to a rich higher-order dependently-typed programming language and a full-fledged theorem prover. Consequently, DY* is well suited to verify protocol implementations, and protocols

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

with recursive data structures like trees, which automated provers like ProVerif and Tamarin struggle with. For example, DY* has been used to verify properties like PCS for recursive protocols like Signal [43] for an unbounded number of rounds. DY* has also been used to verify detailed protocol specifications like ACME [99] and protocol implementations like Noise* [100].

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[99]: Bhargavan et al. (2021), *An In-Depth Symbolic Security Analysis of the ACME Standard*

[100]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

Applying DY* to TreeSync. The definitions we presented earlier (§5.3.1) are simplified ones. In reality, all of our TreeSync code is parametric over the type of bytes, and over operations on such bytes, which we achieve via F*'s *type class* mechanism. This allows us to write a single TreeSync, but instantiate it twice "for free": once with concrete bytes, to obtain an executable specification that can be tested over the wire, and once with symbolic DY* bytes. Similarly, our cryptographic primitives are either concrete, and call actual implementations; or symbolic, and annotated with DY* labels. To enable both concrete and symbolic crypto, we had to extend the DY* libraries with some missing features, like a proper treatment of bitstring lengths.

We then wrap the protocol code within a high-level API that offers functions for creating groups, adding and removing members, etc. This API internally stores session state for each open session, sends and receives messages, and logs events before each state change. This API is exposed to the attacker, so it can create any number of TreeSync sessions, and trigger any sequence of add, removes, and updates. However, the attacker does not get access to the internal state of uncompromised members. Our goal is to show that in all traces of honest TreeSync participants with the symbolic Dolev-Yao attacker, our confidentiality and authentication guarantees hold.

The first step is to typecheck that our protocol code obeys the DY* *labeling* discipline which ensures that secret values are kept secret; in TreeSync the only secrets are signature keys, which are used only to create signatures, so all data structures are labeled public, and the labeling proofs are straightforward.

Stating and Verifying TreeSync Authentication. Next, we need to annotate our code with *signature predicates* that describe all the possible uses of signatures in our full specification, including TreeKEM, TreeDEM, and TreeSync.

Within TreeSync, signatures are used only for leaf signatures. We require that before creating a leaf signature in a group g, the committer at leaf L must log an event of the form `Send(g,canonicalize(T,L))`, for every subtree T it modifies.

We can then state our authentication theorem as an invariant on the TreeSync state: in all reachable TreeSync session states at a member b of a group g, in every non-blank subtree T, there is a leaf L occupied by some principal a such that a previously logged an event of the form `Send(g,canonicalize(T,L))`, or else a was compromised. In DY*, this is stated as follows.

```
val treesync_authentication_theorem:
 #b:identity → #time:timestamp → #l:nat → #i:tree_index l →
 st:treesync_state → t:treesync l i →
 Lemma (requires
  is_reachable b time st ∧
  is_subtree_of t st.tree ∧
  root_node_is_not_blank t)
 (ensures ∃ li, a. has_leaf_identity t li a ∧
  // a logged the corresponding Send event
  event_happened_before a time
   (Send st.group_id (canonicalize t author_li))
  // or was corrupted by the attacker before time
  ∨ is_corrupt a time)
```

To prove this theorem, we first rely on the unforgeability of signatures to show that the leaf signature in L ensures the existence of a linked path from L to T, and of corresponding Send events in the trace. We then combine the path-link invariant and the parent hash integrity lemma to conclude that the corresponding subtrees at b and a must be equivalent, and hence have the same canonicalization, to complete the proof.

Signature Confusion Attack. In fact, our first attempt at the authentication proof for TreeSync in draft 12 failed, because we were unable to prove that the attacker could not use a TreeDEM signature to forge a TreeSync signature. This is because both protocols use the same signature keys and there is an ambiguity between their signature formats. Consequently, we could not prove that the signature predicate for TreeSync is independent of the predicate used in TreeDEM.

This proof failure actually points to a real attack, and we can generate concrete instances of the signature contents used in the two protocols that collide after serialization. We note that this attack only appears if one models bitstring-level serialization (like our specification) since the two signatures would otherwise appear to be on different MLS types.

We presented this attack to the MLS working group and it was fixed as per our recommendation in draft 13. The fix uniformly disambiguates all signatures used in MLS for different purposes using different string labels. With this fix incorporated, we completed our authentication proof.

Interpreting TreeSync Authentication. The TreeSync authentication theorem tells us that the trees at different members are consistent as long as enough honest (uncompromised) members keep creating commits. In particular, the theorem prevents all the known tree tampering attacks that plagued earlier versions of MLS [51, 116].

[51]: Alwen et al. (2022), *On The Insider Security of MLS*
[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

Interestingly, our proof makes no assumptions at all about TreeKEM, and our TreeSync specification treats all content provided by TreeKEM as opaque. We also do not make any assumptions about TreeDEM except for the signature disambiguation property described above. Consequently, TreeSync provides this authentication guarantee even if TreeKEM and TreeDEM were replaced by other (even broken) protocols.

Although we do not analyze TreeKEM and TreeDEM in this paper, TreeSync authentication is a necessary precondition for both these protocols, since they rely on tree agreement between members. We also prove that the authentication guarantee of TreeSync implies the TreeKEM tree integrity invariant, and that the tree hash used in TreeDEM provides strong integrity guarantees.

Component	F* LoC	Verification time
Library code	836	1min30s
TreeSync	1274	4min30s
TreeKEM	396	1min
TreeDEM	1384	2min45s
High level API	1024	1min30s
Library proofs	1170	1min45s
TreeSync proofs	4018	13min30s
Tests	2782	2min45s
Total specification	4914	11min15s
Total proofs	5188	15min15s

Table 5.1: Verification and coding effort for MLS on an Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz with 32GB of memory.

5.5 Implementation

MLS formal specification. Our complete F* specification totals 4914 lines of non-blank, non-comment code. We follow the modular approach described earlier (§5.2): our specification spans three namespaces, one for each subsystem. Table 5.1 gives a sense of how many lines of code (LoC) our implementation contains, grouped as run-time code, proofs, and tests.

Recall that we chose to materialize two trees for TreeSync and TreeKEM, favoring clarity and readability over conciseness; this tradeoff appears in numerous other places in our specification, where we always prefer a readable specification over a clever optimized implementation. For comparison, we evaluate mlspp and OpenMLS, two industrial implementations of MLS written in C++ and Rust respectively. The mlspp implementation, just like us, relies on an automated framework to derive parsers and serializers, and they use modern C++ with copious amounts of type inference to keep boilerplate to a minimum, totaling 4250 lines of non-blank, non-comment code. The OpenMLS implementation, in Rust, totals 15,000 lines of non-test, non-blank, non-comment code.

Based on those two points of comparison, we conclude that we successfully managed to write a compact, concise, readable modular specification that can serve as a blueprint for any future MLS implementations.

MLS reference implementation. As mentioned earlier, our specification also serves as a reference implementation: all of our code is also fully executable. To run our code, we rely on F*'s extraction feature to produce OCaml code, which we then compile and execute using the standard OCaml toolchain. Our code interoperates with mlspp and OpenMLS and we participate in the IETF MLS interoperability meetings.

Our code requires numerous cryptographic primitives: we rely on the HACL* library [71, 122, 124] for those, thereby preserving the property that the entire codebase is verified. Furthermore, HACL* is one of the few libraries that support the latest version of HPKE, which we require for interoperability.

[71]: Zinzindohoué et al. (2017), *HACL*: A verified modern cryptographic library*
[122]: Protzenko et al. (2020), *Evercrypt: A fast, verified, cross-platform cryptographic provider*
[124]: Polubelova et al. (2020), *Haclxn: Verified generic SIMD crypto (for all your favourite platforms)*

Performance evaluation. We compare our OCaml-extracted code to both mlspp (written in C++) and OpenMLS (written in Rust). We benchmark high-level integration tests that call the API functions for participant addition, participant removal, and sending of messages. The results are in Table 5.2.

We are comparing implementations written using different languages and toolchains. As such, we can only draw a broad conclusion, namely, that all

Measurement	This paper	mlspp	OpenMLS
Adds	2.7s	1.2s	0.7s
Messages	3.2s	0.6s	0.2s
Removes	5.5s	0.9s	0.7s

Table 5.2: Performance comparison between this paper and two other implementations of MLS. The time measured is the cumulative computation time for all participants in the group, measured on an Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz with 32GB of memory. *Adds:* Add 10 participants, one by one, with 20 messages from each participant after each add; *Messages:* Add 3 participants, with 400 messagers from each participant after each add; *Removes:* Add 15 participants, with 1 message from each participant after each add, then remove every participant with an odd position in the tree, then add participants until there are 15 participants again, with 1 message from each participant after each add.

implementations exhibit comparable performance, and generally execute within the same order of magnitude. We remark that our implementation, in spite of being written with no performance concerns in mind, still performs competitively. This means our code can be used off the shelf for rapid prototyping, interoperability testing, or generally, as a drop-in verified component when the highest degree of assurance is desired. Rudimentary profiling analysis indicates that a majority of the execution time is spent within the cryptographic primitives, which partially explains why our implementation has only limited overhead.

We have several plans in the works to address the performance overhead. In the short term, we will investigate the use of better data structures (e.g. semi-persistent arrays) to make our pure, persistent byte manipulations more efficient. In the long run, we want to perform a proof of refinement that an efficient implementation, written in Rust, satisfies our high-level specification.

Skype integration. As a proof-of-concept, we integrated our reference implementation in a prototype version of the Skype messaging client. This was done as a one-time collaboration with a Microsoft team, where we added support in an experimental branch for a new feature called "secure group chats", powered by our MLS reference implementation. We tested and benchmarked the code on small groups exchanging a handful of messages. Overall, this allowed us to show that our code is deployable within a mainstream messenger.

Skype already features 1:1 private conversations using Signal; our implementation extended this functionality to actual groups. Skype is written using the Electron framework, i.e. a Web-based runtime environment. We used js_of_ocaml [125] to compile our extracted code to JavaScript, and linked it against HACL-WASM [126], a version of HACL* compiled directly to WebAssembly [127] while preserving security properties. The Skype team generously enabled the backend changes to implement the so-called Directory Service and Authentication Service that MLS relies upon.

We were able to successfully converse across endpoints, and there were no noticeable slowdowns in the user interface once we linked our code against efficient WASM-based cryptographic primitives. We conclude that the efficiency of MLS is bounded by the underlying cryptographic primitives, and that our reference implementation is a valid choice for security-conscious consumers.

[125]: Vouillon et al. (2014), *From bytecode to JavaScript: the Js_of_ocaml compiler*

[126]: Protzenko et al. (2019), *Formally verified cryptographic web applications in webassembly*

[127]: Haas et al. (2017), *Bringing the web up to speed with WebAssembly*

5.6 Impact

Improving the standard. Our work identified several issues and attacks in the MLS drafts, and led to our proposing numerous changes that were ultimately adopted by the IETF.

The first issue we found was the signature confusion attack described in §5.4.5. We fixed this defect by uniformly adding labels to all signatures in MLS, to disambiguate their intent. This change was adopted in draft 13 and is required for our authentication theorem.

A second issue we found was that the integrity guarantee provided by the parent hash mechanism was too weak (§5.4.4), since it authenticated only TreeKEM related content in the tree. We proposed replacing this mechanism with one that uses the tree hash to authenticate the full content of the tree, including leaf credentials. This change, which enables our strong parent hash integrity lemma, was adopted in draft 13.

A third series of issues we found relates to the parent hash computation. In the process of performing the proof, we ended up with the four conditions for the well-formedness of the parent-hash link in the presence of unmerged leaves and filtered paths. We also identified several key criteria that must be met for the parent hash to recursively authenticate the whole tree, and for the corresponding inductive reasoning to succeed. The RFC was failing to enforce some of these, and we showed protocol traces that would break the TreeSync property.

Finally, we identified further well-formedness conditions for unmerged leaves that were not enforced upon joining a group (an unmerged leaf *must* point to a non-blank leaf). The protocol was missing this check, which we showed could break the parent-hash invariant. This is also fixed in draft 15.

Fixing Implementation Bugs. In addition to bugs in the standard itself, we also found implementation issues throughout the course of our interoperability testing. The first faulty implementation we identified was ours: we had some serialization errors, e.g. serializing a field as a `uint8` instead of `uint16`. We also found issues in both mlspp and OpenMLS, the two major industrial implementations of MLS at the time of writing. Both bugs were reported, and fixed.

A benefit of *executable* specifications is that they can be extensively tested for interoperability, like we did. This is the only way to gain confidence that the security theorem refers to the *actual* protocol, not a variant of it with an alternate serialization scheme. It is our opinion that any serious security analysis of a real-world protocol must include an executable specification; otherwise, one might prove properties over a different protocol, without realizing.

Lessons Learned. During our engagement with the IETF MLS standardization process, we found that the benefits of formal verification are now appreciated and understood when it comes to designing a new secure protocol. Notably, the MLS working group was highly reactive and appreciative of any bugs found by various teams; gladly accepts well-argued revisions and improvements; and, we posit, enjoys the added confidence that a formal analysis brings. We suspect that the many successes from the earlier TLS 1.3 have created a fruitful ground for this sort of collaboration.

Our approach of building an executable specification of the standard proved very useful for interactions with the working group. This not only makes security proofs much easier (as opposed to, say, having to perform them on a production codebase), but also allows rapid prototype and testing of proposed changes: for example, we were able to modify the specification and adapt the proofs to understand the security implications of a last-minute protocol modification. We encourage other standardization efforts to promote reference implementations written in high-level languages.

Conversely, MLS has grown to become a large protocol standard, and even understanding, let alone analyzing, the full protocol is a challenge even for cryptographic developers and protocol experts. One of the contributions of this paper is the modular decomposition of MLS from a monolithic protocol into three independent components with a clean separation of concerns. In retrospect, this kind of modular design should have been built into the protocol standard itself, and perhaps should be a goal for the next version of MLS.

5.7 Related Work

Although group key establishment has been well studied in the literature (see e.g. [128, 129]), group messaging differs from traditional group protocols in that it supports asynchronous messaging in dynamic groups. Unger et al. [5] provide a survey of messaging protocols, including some that support groups, conclude that "conversations between larger groups still lack a good solution". Since that survey, most academic work on group messaging has either been in the context of Signal or MLS. The extension of Signal with private authenticated groups was formally described and analyzed by Chase et al. [111], but Signal's sender-driven group messaging protocol does not scale to large groups. In the rest of this section, we compare our work with work on MLS and on other efforts to formally analyze cryptographic protocols.

[128]: Manulis (2006), *Security-Focused Survey on Group Key Exchange Protocols*
[129]: Poettering et al. (2021), *SoK: Game-Based Security Models for Group Key Exchange*

[5]: Unger et al. (2015), *SoK: Secure Messaging*

[111]: Chase et al. (2020), *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*

Prior Analyses of MLS. The initial draft of MLS relied on Asynchronous Ratcheting Trees (ART) whose authors provide a cryptographic proof of their tree-based protocol design [112]. The original design of the TreeKEM protocol was presented in [113] and was adopted in MLS draft 2, and has since been extended with many features including blank nodes, unmerged leaves, and the proposal-commit pattern. Various versions of TreeKEM have been analyzed in a variety of security models. [114] presents a cryptographic analysis of TreeKEM in draft 7 against a passive, non-adaptative attacker, and defines continuous group key agreement (CGKA). They also analyze using Updatable Public Key Encryption to improve forward secrecy guarantees of TreeKEM. [115] modularly analyzes MLS in draft 11 against an active attacker. Their proof reason on high-level messages and miss the signature ambiguity attack, which we found by doing proofs on byte-level precise executable specifications. [52] analyzes the key derivation component of MLS in draft 11. [130] studies the multi-group security of MLS. All of these focus on the key exchange (TreeKEM) and data encapsulation (TreeDEM) components of MLS and do not consider tree integrity and authentication (TreeSync), our main focus.

[112]: Cohn-Gordon et al. (2018), *On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees*

[113]: Bhargavan et al. (2018), *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*

[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*

[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*

[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*

[130]: Cremers et al. (2021), *The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter*

Alwen et al. [51] study the security of TreeKEM against insider attacks and find a flaw on tree authentication. They propose different fixes by modifying the parent hash scheme, one of which is used in draft 16 and

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

we study in this work (the "tree parent hash"). However, unlike this work, they study TreeKEM and TreeSync together as a monolithic protocol.

All the works mentioned above rely on manual pen-and-paper proofs. As the MLS standard grows, so do these manual proofs, making them hard to check and maintain. In this work, we use a formal verification tool to build a byte-level precise machine-checked specification for MLS that can be independently tested, modified, and verified.

A symbolic analysis of TreeKEM for forward security in Tamarin appears in [131] but it does not consider PCS or authentication. [116] uses F* to symbolically analyze TreeKEM in draft 7, finding an attack on tree authentication. However they do not identify TreeSync as an independent protocol and do not analyze the current parent hash design.

[131]: Cremers et al. (2023), *Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis*

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

Mechanized Proofs of Crypto Protocols. Our approach follows a long line of work on the mechanized formal verification of cryptographic protocols (see [36] for a survey). Some protocol verification tools, like ProVerif [41], Tamarin [42], and DY* [43], rely on the *symbolic model* which treats cryptography abstractly and focuses on logical protocol flaws. Other tools, like CryptoVerif [38], EasyCrypt [37], and Squirrel [39], rely on the *computational model* which includes a more precise model of cryptography but provides less automation. Both kinds of tools have been applied to the analysis of real-world protocols like Signal [43, 54] and TLS 1.3 [49, 50].

[36]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[54]: Kobeissi et al. (2017), *Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach*

Except for DY*, most existing tools struggle to analyze protocols with unbounded state (like trees) and with recursive structure (like ratcheting). Indeed, very little prior work applies to the mechanized analysis of group protocols [131, 132] and even these works do not consider authenticated data structures like TreeSync trees.

[131]: Cremers et al. (2023), *Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis*

[132]: Schmidt et al. (2014), *Automated Verification of Group Key Agreement Protocols*

Finally, many prior works verify reference implementations of protocols like Signal [133], Noise [100], and TLS [134]. Like us, these handle the full complexity of the protocol, including detailed message formats, yielding precise theorems that apply to running protocol code, not just abstract models.

5.8 Conclusion

We present a precise formal specification of the current version of the MLS protocol, along with a machine-checked proof of its TreeSync component. This work is part of a long-term engagement between the authors and the MLS working group, where we analyzed multiple intermediate versions of the protocol, found and fixed issues, and contributed design improvements to the protocol. Our specification consolidates our understanding of MLS and we hope it can serve as a formal guide to readers interested in this protocol.

Our proofs are only for TreeSync and do not cover TreeDEM and TreeKEM, although we formally specify and account for the interaction between TreeSync and these. We leave the comprehensive composite security analysis of all three components of MLS for future work.

Acknowledgments

We are indebted to Franziskus Kiefer, Raphael Robert and Richard Barnes for proofreading a draft of this paper and providing precious feedback. We are grateful to Jaroslav Franek for setting up a hackathon that allowed us to try out our MLS implementation in the Skype client, along with team members Jakub Kermaschek, Jurav Blazek, Lukas Liska and Katerina Cizkova.

TreeKEM: Efficient continuous group key establishment

6

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> **C. A. R. Hoare**, *in his 1980 Turing Award Lecture*

This chapter is adapted from the eponymous publication [135], to appear at IEEE S&P 2025. The text is identical, but was reformatted.

6.1 Introduction

The Messaging Layer Security (MLS) standard [21], published in 2023, is the first and till-date only Internet standard for secure end-to-end encrypted messaging. It is currently implemented by multiple messaging software vendors, including Cisco, AWS, Wire, and XMTP, and several vendors have announced their intention to support it in the future. With the advent of new regulations that require messaging interoperability, like the EU Digital Markets Act, an open standard like MLS is seen by many as the basis for the next generation of secure messaging applications.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

Compared to popular and widely-deployed messaging protocols like Signal [110] and its many variants, the design of MLS distinguishes itself in two important ways.

[110]: Marlinspike et al. (2016), *Signal Specifications*

First, MLS puts group messaging front and center and seeks to scale up to groups with thousands of members. To achieve this, MLS is built around a new tree-based protocol that scales logarithmically with the group size (in the ideal case) and linearly in the worst case. In contrast, protocols that build group messaging using two-party channels, such as Signal Sender Keys [136], scale linearly with group size in the best case and quadratically in the worst. Furthermore, these protocols do not provide important properties like membership agreement and post-compromise security for group conversations. With the growth in popularity of group messaging, and with the increase in message sizes entailed by post-quantum cryptography, the improved security and scalability of MLS is increasingly desirable.

[136]: Balbás et al. (2023), *WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs*

Second, inspired by the experience of the Transport Layer Security (TLS) working group in the standardization of TLS 1.3, the design of MLS was structured as a collaboration between protocol designers and cryptographic experts with the goal of developing security proofs of the protocol alongside standardization. This process resulted in a number of formal security analyses of MLS (and its variants) using a variety of security models and techniques [51, 52, 68, 114–116, 131]. The current work is also a result of this long-term collaboration, and it contributes a new machine-checked security proof for TreeKEM, the core key agreement component of MLS.

MLS: TreeSync, TreeKEM, and TreeDEM. In previous work, Wallez et al. [68] identified a modular decomposition of MLS into three sub-

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

protocols, as depicted in Figure 6.1:

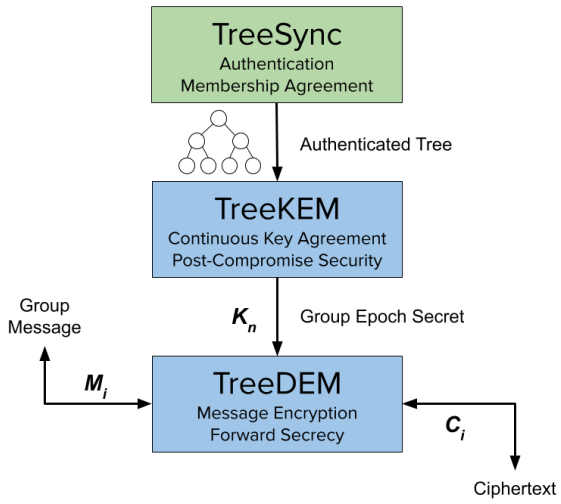


Figure 6.1: A Modular Treatment of Messaging Layer Security: TreeSync, TreeKEM, and TreeDEM

- **TreeSync**: a protocol that synchronizes the shared group state across group members. The shared state includes the current group membership and is structured as a tree, with each occupied leaf corresponding to a member, and each internal node representing a subgroup. TreeSync uses signatures and Merkle-tree style hash computations to authenticate the initial group state provided to a member and all subsequent changes to the state. It also ensures that the tree data structure maintains an internal integrity invariant. This authenticated, synchronized state is then passed to TreeKEM.
- **TreeKEM**: a protocol that allows each member to use its private keys and the sequence of authenticated states provided by TreeSync to derive a sequence of group keys, called *epoch secrets* (K_n). TreeKEM uses the tree structure to efficiently update the epoch secret; in the best case, this requires only a logarithmic number of public key encryptions and a single decryption at each recipient. Furthermore, TreeKEM provides post-compromise security, and in particular, security against members that have been removed.
- **TreeDEM**: a protocol that takes the epoch secret K_n computed by TreeKEM and uses it to derive message encryption keys for each group member. These keys are then used to encrypt and decrypt group messages so that only the current members can send or receive them. After each message, the message encryption keys are ratcheted forward to provide forward secrecy.

When compared with a two-party secure channel protocol like TLS, TreeKEM corresponds to the handshake protocol, and TreeDEM corresponds to the record layer. Of course, the complexity of MLS is in handling the dynamic group setting where the list of participants can grow and change. While all three of these protocols are novel and deserve close scrutiny via formal security analyses, in this paper, we will focus on modeling and analyzing TreeKEM.

Security Analyses for Abstract Models of MLS. A key challenge when analyzing a protocol standard is in finding the right level of abstraction. The MLS standard is 132 pages long; it defines the high-level cryptographic constructions and algorithms of TreeSync, TreeKEM, and TreeDEM, but also defines the concrete tree data structure and operations on it, the precise low-level formats of all messages and cryptographic inputs, and handles the negotiation of versions and ciphersuites. Most

prior works on analyzing MLS ignore most of these low-level details and instead model MLS as an abstract group key agreement protocol so that its specification can fit in a few pages and a formal proof of its security can be feasible.

Some works have analyzed the core key agreement of MLS with pen-and-paper proofs: [137] defines a new security definition called *continuous group key agreement* (CGKA) for protocols like TreeKEM; [114] presents a proof that TreeKEM as specified in MLS draft 7 is a CGKA; [115] presents a modular proof of MLS draft 11, by decomposing it into a CGKA protocol (essentially TreeSync+TreeKEM) and a stateful group AEAD protocol (i.e. TreeDEM); [51] analyzes the security of MLS draft 12 against malicious group members, by focusing on the integrity mechanisms within TreeSync. Other pen-and-paper proofs focus on aspects of MLS outside the core TreeKEM component: [52] analyzes the MLS key schedule, [130] studies post-compromise security for group messaging protocols like MLS.

[137]: Alwen et al. (2020), *Continuous group key agreement with active security*

[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*

[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*

[130]: Cremers et al. (2021), *The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter*

There have also been some attempts at using (semi-)automated tools to obtain machine-checked symbolic security proofs for abstract models of TreeKEM: [116] analyzes the original version of TreeKEM [113] using a symbolic model in F* [98] and compares its security with alternate designs; [131] shows how a simplified version of TreeKEM can be analyzed in the Tamarin prover for forward secrecy (but not post-compromise or post-remove security).

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

[113]: Bhargavan et al. (2018), *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[131]: Cremers et al. (2023), *Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis*

Several of these works suggest improvements to MLS, some of which were incorporated into the MLS standard before publication. Still other works present and analyze new group messaging protocols inspired by MLS, but we do not consider these works here. However, none of the proofs in these works applies to the published MLS standard, since they analyze abstract models that leave out many of the details and options that make MLS complex.

Verifying an Executable Specification of TreeKEM. In contrast to the above works, our work is directly inspired by the work of [68], which presents a proof for a bit-level precise, executable, testable specification of the TreeSync component of MLS. The advantage of working on such a specification is that one can run it against protocol test vectors, or test it for interoperability with other implementations, to gain confidence that the model we are proving security for is not missing any important protocol details.

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

Handling low-level details can be crucial for security. For example, none of the papers on MLS cited above precisely model the signatures used in MLS. Even the pen-and-paper security proof for MLS in [51] abstracts away from the formats of the signature inputs, which would have been tedious to handle in a manual proof, and assumes that these signatures cannot be confused for each other. As a consequence, this proof misses an important signature ambiguity attack on MLS, which was subsequently found in the machine-checked proof of TreeSync [68] which did model all the low-level signature formats.

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

In this paper, we present an executable, testable, interoperable model of TreeKEM and a security proof for this model using a symbolic proof framework called DY* (the same methodology as [68]). Consequently, our proof accounts for all the low-level details of the MLS standard, and our confidentiality theorem for TreeKEM composes with the authentication theorem for TreeSync in [68].

Contributions. We present the first machine-checked proof for the TreeKEM component of the published MLS standard. Ours is also the first proof for a bit-level precise, executable, interoperable specification of TreeKEM, which can be seen as a reference implementation. Our proof shows how to modularly compose the guarantees of TreeKEM and TreeSync, and provides some important insights on key management and erasure for MLS implementations and deployments. Finally, ours is likely the first machine-checked symbolic security proof for group key exchange in dynamic groups (supporting add, remove, and update), and the first to establishing properties like post-compromise security in the group setting.

Outline. We start with an informal, accessible description of TreeKEM (§6.2); next, we show how to capture TreeKEM in formal language, encoding its specification using the F* proof assistant (§6.3). Then, we state the security properties we proved (§6.4), with a precise and extensive discussion of potential paths to compromise, followed by insights about our proof techniques (§6.5). Finally, we discuss our results and conclude (§6.6).

6.2 The MLS TreeKEM Protocol

We now describe TreeKEM as specified by the MLS standard [21]. We start by describing the overall goals of TreeKEM (§6.2.1), then define the two main mechanisms of TreeKEM: the use of a tree to produce a fresh *commit secret* shared by the group, guaranteeing post-compromise security and remove-security (§6.2.2), and key schedule that provides forward secrecy and add-security (§6.2.3).

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

6.2.1 Goals of TreeKEM

The goal of TreeKEM is, at each epoch, to establish an *epoch secret* that is known to exactly the participants currently in the group. This epoch secret is then used in TreeDEM to derive the same message encryption keys at each participant. The functionality provided by TreeKEM is sometimes called *continuous group key agreement* [137].

[137]: Alwen et al. (2020), *Continuous group key agreement with active security*

Design constraints. The initial design of group key establishment in MLS was based around Asynchronous Ratcheting Trees [112] which used a tree of Diffie-Hellman operations to enable efficient asymmetric ratcheting for groups. TreeKEM [113] was proposed as a KEM-based more efficient alternative to ART. The current version of TreeKEM in the MLS standard is the culmination of multiple revisions and extensions since these early designs. It aims to satisfy several constraints; in particular, the protocol must i) handle dynamic groups (i.e. participants can join and leave the group over time), ii) be asynchronous, (i.e. participants are not required to always be online), iii) be efficient (i.e. scale better than linearly on the number of participants) and iv) provide security properties like key confidentiality, forward secrecy and post-compromise security. Goal iv) is the main topic of study in this paper.

[112]: Cohn-Gordon et al. (2018), *On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees*

[113]: Bhargavan et al. (2018), *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*

Additionally, TreeKEM makes certain assumptions about the design of the overall system, which are captured in the MLS architecture document [55]. Notably, TreeKEM relies on an untrusted *Delivery Service*, which is tasked with receiving messages from individual group participants, and

[55]: Beurdouche et al. (2025), *The Messaging Layer Security (MLS) Architecture*

broadcasting them back to all other group participants. In other words, MLS is *not* a peer-to-peer service where messages are sent directly from one participant to another.

TreeKEM Terminology. MLS is an asynchronous, distributed protocol. The TreeKEM specification therefore distinguishes the construction (locally, by a participant) of operations over the group (e.g. addition and removal of participants), known as *proposals*; the bundling of possibly many such operations into a *commit*; the application of this commit to the group to reach the next epoch.

The matter of how competing concurrent commits (by two different participants) are dealt with falls outside the scope of the MLS protocol; this is a matter handled by the untrusted delivery service, which we do not cover in the present paper. MLS assumes that each participant receives a sequence of commits from the delivery service and attempts to process them in order.

Upon processing a commit, the group enters a new *epoch* and TreeKEM outputs an *epoch secret* for the group to use. In other words, each commit does a round of key derivation which produces a fresh epoch secret: intuitively, this means that should anything change with the group (the membership, a member's public key, etc.), then a new epoch secret will be derived for the group.

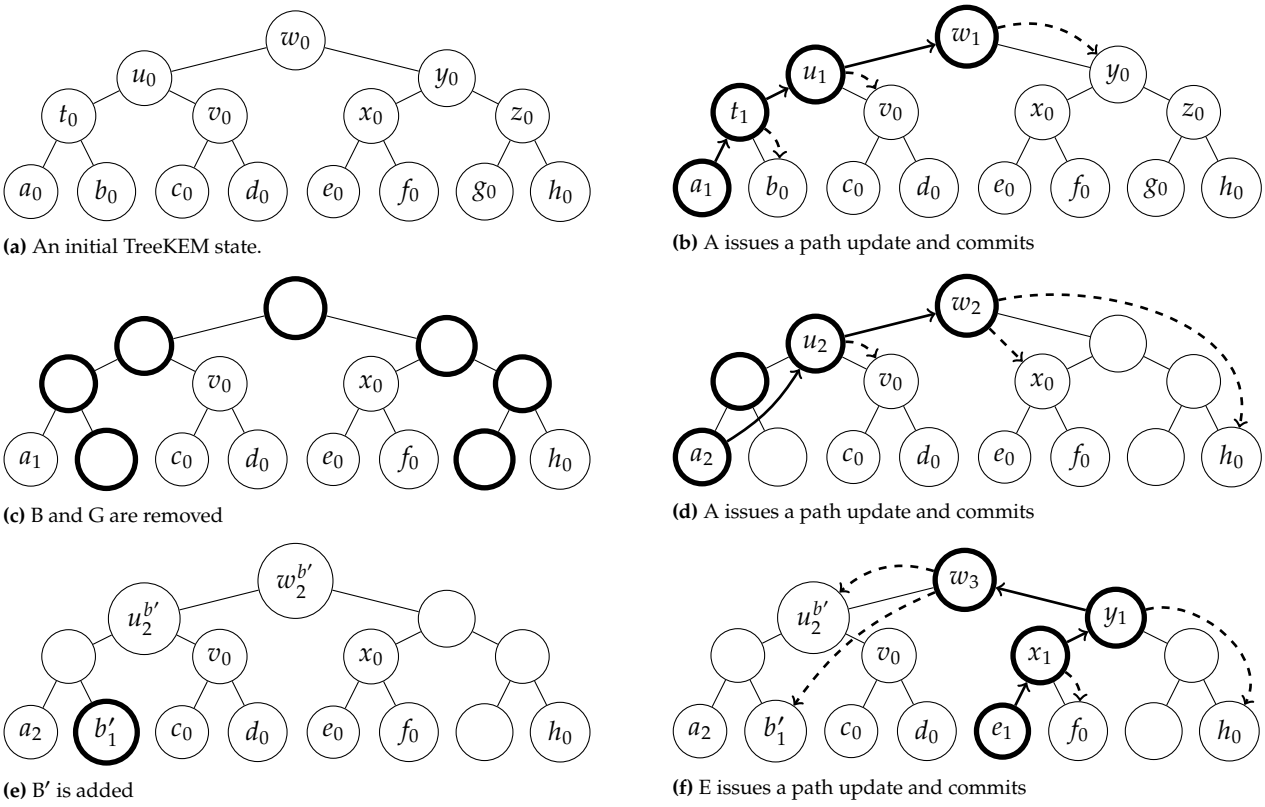
Crucially, a commit may apply a *path update* operation on the internal state of TreeKEM (explained in §6.2.2) and output a *commit secret*, which is used in the computation of the next epoch secret (explained in §6.2.3). Such path updates are mandatory except in *add-only commits*, a special flavor of commit that does not contain a path update operation (we leave the description of add-only commits to §6.2.3). We explain path updates and commit secrets in detail in the remainder of this section.

An additional element that flows into the construction of the epoch secret is the *group context*, which summarizes information about the current state of the group – as we will see later, this is important for the security proofs.

Security properties, informally. TreeKEM aims to offer several security guarantees on the epoch secret:

- add security (i.e. new participants must not know epoch secrets that predate their joining the group),
- remove security (i.e. removed participants must not know epoch secrets after they have left the group),
- forward secrecy (i.e. the compromise of a participant by an attacker must not reveal past epoch secrets) and
- post-compromise security (i.e. the epoch secret can eventually heal from a past compromise).

These properties will be more formally studied in §6.4.4. Suffices to say, for now, that forward secrecy and add security are achieved by judicious *key erasure* and that post-compromise security and remove security are achieved with by sharing *fresh randomness* (through the path update operation). In the rest of this section, we describe the state and mechanisms of the TreeKEM protocol and informally explain how it achieves these desired security guarantees.



(a) An initial TreeKEM state.
(b) A issues a path update and commits
(c) B and G are removed
(d) A issues a path update and commits
(e) B′ is added
(f) E issues a path update and commits

Figure 6.2: Evolution of a group's tree in TreeKEM. Nodes in **bold** are the nodes updated by the current operation, plain arrow (⟶) indicates hashing the path secret and dashed arrow (⇢) indicates encrypting the path secret (the cryptographic operations are detailed in Figure 6.3).

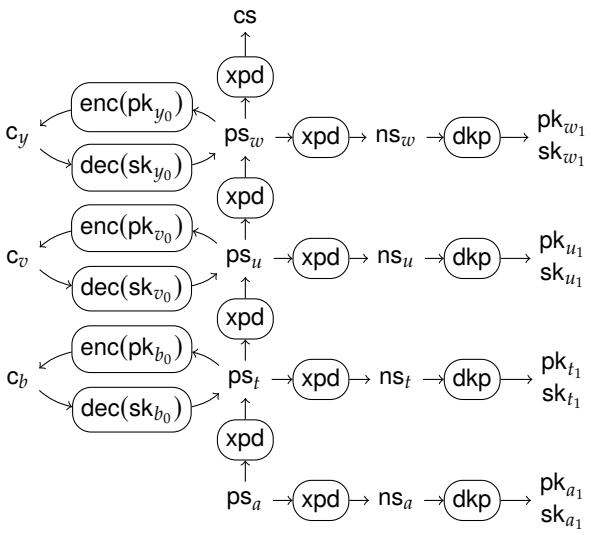


Figure 6.3: Cryptographic operations performed during A's path update in Figure 6.2b. xpd is HKDF.Expand, dkp is HPKE.DeriveKeyPair, enc is HPKE.Seal, dec is HPKE.Open, ps is "path secret", ns is "node secret", cs is "commit secret", c is "ciphertext".

6.2.2 A Tree for Group Key Agreement

Throughout this section, we will use uppercase letters to denote nodes of TreeKEM's tree (A to H for leaves and T to Z for internal nodes) and lowercase letters to denote the content stored in these nodes during the lifetime of the group (e.g. a_0, a_1, etc).

In TreeKEM, group participants are arranged in the leaves of a complete binary tree, as depicted in Figure 6.2a (nodes A to H). Each node contains a public-key encryption keypair, whose secret key is known by (and only by) the participants in the subtree rooted at that node (e.g. the secret key of v_0 is known to c_0 and d_0, and the secret key of the root w_0 is known to every participant in the group). This property is called the *tree invariant* in the MLS standard [21].

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

By relying on the tree invariant, we can efficiently encrypt data to specific sub-groups in the tree. For example, we can perform one encryption to y_0 instead of separate encryptions to e_0, f_0, g_0 and h_0. This optimization is the essence of TreeKEM; we will see how it permits the efficient creation of path updates, i.e. refreshing secrets from a leaf node to the root without modifying every node in the tree.

For encryption, TreeKEM relies on the Hybrid Public Key Encryption (HPKE) construction [69], which uses a key encapsulation mechanism (KEM), a key derivation function (HKDF), and an authenticated encryption (AEAD) algorithm to build a public-key encryption scheme that provides integrity for the plaintext and for additional data.

[69]: Barnes et al. (2022), *RFC 9180: Hybrid public key encryption*

We now describe how the group evolves through a series of updates depicted in Figure 6.2, and explain how each tree modification preserves the tree invariant.

Path update. In Figure 6.2b (and more precisely in Figure 6.3), A wants to recover from a potential compromise and benefit from post-compromise security properties of TreeKEM. Hence, it updates any secret (potentially compromised) it knows in the tree. To do so, A updates the HPKE keypairs of the nodes between its leaf and the root (i.e. of the nodes A, T, U, W, shown in **bold** in Figure 6.2b), while ensuring the new secret keys are known by participants in the corresponding subtree (e.g. the secret key of u_1 is transmitted to b_0, c_0 and d_0), and issues a new commit secret to the group.

Here is how the new secrets are generated. A generates an initial *path secret* ps_a (at the bottom of Figure 6.3), from which the new commit secret and all new keypairs will be derived. Here is how it happens: each updated node is associated with a path secret ($\mathsf{ps}_\square$ in Figure 6.3), from which two secrets are derived: the *node secret* for the same node ($\mathsf{ns}_\square$ in Figure 6.3), and the path secret for the node directly above (depicted as $\longrightarrow$ in Figure 6.2b). The node secret is used to derive a new HPKE keypair ($\mathsf{pk}_\square$, $\mathsf{sk}_\square$ in Figure 6.3). The commit secret (cs in Figure 6.3) is the path secret corresponding to the node that would be above the root.

Here is how the new secrets are transmitted to the participants that should know them (e.g. b_0 must learn the new secret key of t_1). The path secret of a node can be used to compute the secret keys of this node and all the nodes between them and the root, and hence can be used to compute the commit secret. Therefore, it is sufficient for every participant to obtain the path secret of the least common ancestor between them and A (the updater). We could use this criterion to encrypt one path secret to each group participant: this requires a linear number of encryptions in the size

of the group. We can instead do a logarithmic number of encryptions, by relying on the fact that group participants that obtain a given path secret are arranged in a subtree, and benefit from the tree invariant to do only one encryption to this subtree root. Hence it is sufficient to encrypt ps_t to b_0, ps_u to v_0 and ps_w to y_0 (depicted as $\dashrightarrow$ in Figure 6.2b). The ciphertexts obtained ($\mathsf{c}_\square$ in Figure 6.3) and the new public keys are then sent to every group participant through the Delivery Service.

Removing participants. In Figure 6.2c, B and G are removed from the group. This action is performed using the concept of *blank node*. To remove B and G, the contents of their leaf nodes are erased: their leaf nodes are now blank. Without further action, this breaks the tree invariant: for example, B knows t_1 although it is not in its subtree anymore. As a drastic solution to restore the tree invariant, any node whose secret value is known by B (such as t_1) is also blanked, and the same is done for G (the nodes that were blanked are shown in **bold** in Figure 6.2c).

In Figure 6.2d, A issues a path update to create a commit secret known neither by B nor G, thereby obtaining security after their removal. Blank nodes make this operation more complex than in Figure 6.2b, we now describe how a path update is performed in this situation and introduce two new concepts: *filtered nodes* and *resolution*.

Path update & filtered nodes. In Figure 6.2d, A issues a path update to obtain security back after the removal of B and G. It happens similarly as in Figure 6.2b and Figure 6.3, with one difference about the path secret of T. In Figure 6.2b, the path secret of T is encrypted to B, but in Figure 6.2d, B is not here because it was removed in Figure 6.2c. There is no c_b as in Figure 6.3, hence computing a path secret for T does not achieve any purpose. The node T is therefore *filtered*: it stays blank, and what should have been its path secret is now the path secret of U, or graphically, the path-secret arrow ($\longrightarrow$) goes directly from A to U.

This optimization ensures that from the viewpoint of the tree invariant, there are no redundant non-blank nodes in the tree. Indeed, if during a path update, a bigger subtree (e.g. rooted at T) covers the same set of participants as a smaller subtree (e.g. rooted at A), because of the tree invariant their secret keys will be known by the same set of participants (e.g. {A}) hence the bigger subtree (e.g. rooted T) is redundant and its root (e.g. T) is filtered.

This filtering happens each time a bigger subtree covers the same set of participants as a smaller subtree. For example, if C and D were blanked, then the node U would also be filtered, and what should have been the path secret of T would become the path secret of W.

Path update & resolution. In Figure 6.2d (again), A issues a path update to obtain security back after the removal of B and G. It happens similarly as in Figure 6.2b and Figure 6.3, with one difference about the encryption of the path secret of W. In Figure 6.2b, the path secret of W is encrypted to Y, but in Figure 6.2d, Y is blank after the removal of G in Figure 6.2c. Instead, we encrypt ps_w with the smallest set of public keys such that all participants in the subtree rooted at Y can decrypt, here, x_0 and h_0. This set of public keys is called the *resolution* of the node Y. In the simplest case (as it happened in Figure 6.2b), the resolution of a node is the public key at that node (when it is not blank). In the other cases, the resolution of a node is computed by descending in the tree until encountering a non-blank node and collecting the public keys of all these non-blank

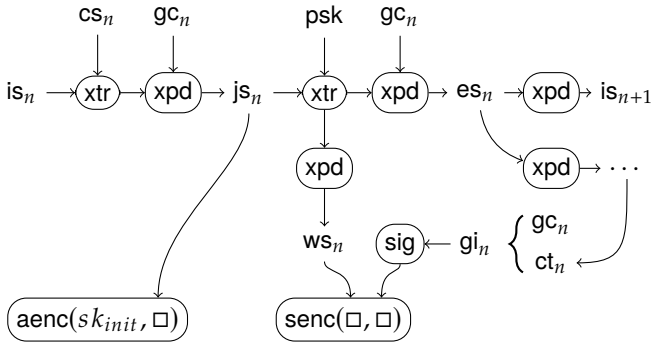


Figure 6.4: Cryptographic operations performed in the key schedule of TreeKEM (§6.2.3) and Welcome (§6.2.4). xtr is HKDF.Extract, xpd is HKDF.Expand, aenc is HPKE.Seal, senc is AEAD.Seal, sig is signature, is is init secret, cs is commit secret (see §6.2.2), gc is group context (appears 3 times), js is joiner secret, psk is pre-shared key, es is epoch secret (the output of TreeKEM), ws is welcome secret. gi is group info (see §6.2.4), ct is confirmation tag. In "..." are the various keys derived from TreeKEM. Decryption functions of the Welcome process (§6.2.4) are left implicit to simplify the diagram.

nodes. This is how we find in Figure 6.2d that the resolution of Y is the set of public keys $\{x_0, h_0\}$.

Adding participants. In Figure 6.2e, B′ is added to the group. To perform this operation, we place the keypair of B′ in the left-most blank leaf. If there were no such blank leaf, we would extend the tree to the right, adding a new root whose left child is the current tree and right child is an all-blank tree, thereby doubling the number of leaf nodes and creating new blank leaves.

This operation breaks the tree invariant, which specifies that the private key of u_2 is known by (and only by) a_2, b'_1, c_0 and d_0. This is not true, because b'_1 doesn't know the private key of u_2 which was generated by the path update in Figure 6.2d when b'_1 was not in the tree at that time.

To account for this fact, we keep track that b'_1 doesn't know the private key of u_2: we say that b'_1 is *unmerged* for u_2. We do this bookkeeping for all nodes above b'_1, and note that b'_1 is also unmerged for w_2. With this new concept, we now reveal the complete formulation of the tree invariant: the secret key of each *non-blank* node is known by (and only by) the *merged* participants in the subtree rooted at that node.

Path update & unmerged leaves. In Figure 6.2f, E issues a path update. It happens similarly to Figure 6.2b and Figure 6.3, with one difference about the encryption of the path secret of W: it is encrypted to the resolution of u_2 which is the set $\{u_2, b'_1\}$. Indeed, recall that the resolution of u_2 is the smallest set of keys to cover all participants in the subtree of U, and b'_1 is unmerged for u_2, meaning that it does not know the private key at u_2. Hence, the resolution of u_2 must include the public key of all its unmerged leaves.

In Figure 6.2e, b'_1 was unmerged for w_2. Now, the path secret of w_3 has been encrypted to b'_1, hence b'_1 is *not* unmerged for w_3: a path update clears unmerged leaves on the updated nodes.

6.2.3 The MLS Key Schedule

Intuitively, the tree component of TreeKEM provides post-compromise security (because secrets are refreshed upon a path update), and remove security (because a new commit secret is derived after removing a participant). We leave a precise characterization of these security guarantees to §6.4, and continue our tour of TreeKEM, now describing the second component of TreeKEM: its key schedule.

Recall that the path secret above the root of the tree is the commit secret (§6.2.2). This secret has add-security, remove-security and post-compromise security. We can deduce this from the tree invariant: indeed, the commit secret is as secret as the root node secret key, hence is known by (and only by) participants in the tree, because after a path update the root has no unmerged leaves. However, this ensures only a weak form of forward secrecy: for example, compromising h_0 in Figure 6.2f would allow the attacker to decrypt the path secret of w_2 in Figure 6.2d (because it is encrypted with a key now known by the attacker and we suppose the attacker knows the ciphertexts), hence compute the commit secret of this previous epoch.

Strong forward secrecy. Therefore, the commit secret cannot be used directly for our purposes. We now explain how to derive the epoch secret (§6.2.1) from the commit secret; the epoch secret has the guarantees we desire, such as strong forward secrecy: a compromise of a participant should not reveal past epoch secrets. To obtain strong forward secrecy, the commit secret is injected into a *key schedule* from which the epoch secret is computed. The key schedule inherits all security properties of the commit secret, and further provides add-security and strong forward secrecy (independently of the commit secret security) because previous secrets can be erased upon key derivation. The key schedule is depicted in Figure 6.4 and is explained below. It is structured as a loop, we present the keys in the order they are derived, starting with the epoch secret and ending with the epoch secret of the next epoch.

Key schedule. The **epoch secret** (es_n in Figure 6.4) is the main key established by TreeKEM, which is used to derive the keys used by TreeDEM (§6.2.1). It is also used to derive the next *init secret*, which serves to initialize the next epoch. The **init secret** (is_n and is_{n+1} in Figure 6.4) is combined with the commit secret and the *group context* to obtain the *joiner secret*. The group context is a summary of the current group state, in particular, it contains (in hashed form) the *initialization keys* (see §6.2.4) with which the joiner secret is encrypted (see below and §6.2.4), this will be crucial in the security proof in §6.5.3. The **joiner secret** (js_n in Figure 6.4) is encrypted with the initialization key of new participants (or "joiners") in the group. (We explain initialization keys in §6.2.4.) It is then combined with the pre-shared keys and the group context to obtain the epoch secret (thereby closing the keyschedule loop), and is also used to derive the *welcome secret*. The **welcome secret** (ws_n in Figure 6.4) produces a symmetric key that is used to encrypt the group context to new participants (as we will see in §6.2.4). Only a minimal amount of information is encrypted with the initialization key; the information that is the same for every joiner (such as the group context) is encrypted symmetrically via the welcome secret.

Add-only commits. Because the key schedule provides forward secrecy and add-security, when the set of group proposals since the last epoch only contains participant additions (hence contains no participant removal), it is not necessary to issue a path update to obtain a commit secret: instead, we can move to the next epoch using an empty commit secret. Doing such "add-only commits" still provides forward secrecy and add-security (because we do a round of key schedule) and remove-security (because we only do that when there were no removals). However, this doesn't provide post-compromise security (because no new randomness was injected in the key schedule), hence shouldn't be used when we want to recover from a potential compromise.

6.2.4 Welcoming New Group Members

We briefly mentioned how new participants join a group in §6.2.2 and §6.2.3, we now describe in depth how it happens. These explanations support the description of our security proofs in §6.5.

Key packages. Because participants are added asynchronously, they publish *key packages* on the Delivery Service, which can be used by any group member to add them to an MLS group. A key package contains a leaf node that is added to the tree (§6.2.2), and an *initialization key* (sk_{init} in Figure 6.4) that is used to encrypt the joiner secret (js_n in Figure 6.4), which bootstraps the key schedule. Notice that as we have described things, two asymmetric encryptions are required to add a new participant to the group: the joiner secret is encrypted with the initialization key (in §6.2.3), and the path secret is encrypted with the leaf node key (described in §6.2.2, but omitted from Figure 6.4). In reality, TreeKEM features an optimization and performs only one asymmetric encryption: both the joiner secret and the path secret are encrypted with the initialization key, in a package called *encrypted group secret* – this is the aenc node in Figure 6.4. Our TreeKEM API (§6.3.2) is designed to support this behavior, and the fact that the path secret is encrypted with the initialization key and not the leaf node key will also need to be taken into account in the security proof in §6.5.4.

Group info. To join a messaging group, it is not sufficient to know the group secrets. For example, the TreeKEM protocol (§6.2.2) requires each participant to know the tree of public keys, and the key schedule requires each participant to know the group context which summarizes the group state (gc_n in Figure 6.4). The tree may come from an untrusted source (e.g. the Delivery Service), and the group context is packaged in the *group info* (gi_n in Figure 6.4), which also contains a value derived from the current epoch secret, called *confirmation tag* (ct_n in Figure 6.4). The group info is further signed by a group participant, this will be crucial in the security proofs for the Welcome process (§6.5.2). Although the group info contains in principle only public data, it is opportunistically encrypted with the welcome secret (ws_n in Figure 6.4, see §6.2.3).

6.3 An executable specification of TreeKEM

In §6.2 we have explained at a high-level the inner workings of the TreeKEM protocol. We now describe how we specify TreeKEM in F* [98], a dependently-typed functional programming language. The specification is byte-level precise, passes the published test-vectors [138], and is used in a broader MLS specification that interoperates with other MLS implementations. Although the explanations in §6.2 are from a global viewpoint, we here specify the local computations performed by one TreeKEM participant.

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[138]: (n.d.), *MLS test vectors*

6.3.1 TreeKEM's Tree in F*

Earlier work [68] modularizes MLS into three sub-protocols (TreeSync, TreeKEM and TreeDEM) and proves that the TreeSync sub-protocol authenticates all of the TreeKEM state. Our specification is based on their work, and in particular, we reuse their definition of trees to define TreeKEM trees as follows:

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

```
1  type treekem_leaf = {
2    public_key: bytes; }
3
4  type treekem_node = {
5    public_key: bytes;
6    unmerged_leaves: list nat; }
7
8  type treekem_public =
9    tree (option treekem_leaf) (option treekem_node)
10
11 type treekem_private = path (bytes) (option bytes)
```

To each node is associated an HPKE keypair (§6.2.2); since we are implementing TreeKEM from the (local) point of view of a participant, we know the public HPKE keys of all participants; but we only know the private keys on the path from the leaf (us) to the root. Therefore, tree nodes and leaves contain public keys only (lines 2 and 5), and internal nodes additionally contain the list of unmerged leaves (line 6). An additional data structure, named treekem_private (line 11) contains the private keys known to us. Because nodes can be blank, we use the option type whose empty value represent blank nodes, except for the private HPKE key for leaf nodes (second argument of path line 11), because it points to our leaf that is non-blank.

We give an example of code that decrypts the path secret in Figure 6.5. This function searches for the least common ancestor between the updater and us (e.g. node U for participant C in Figure 6.2b), finds which ciphertext we must decrypt depending on our position in the resolution (e.g. second ciphertext for participant H in Figure 6.2d) and find for which private key it was encrypted (e.g. private key of u_2 for participant A in Figure 6.2f, but private key of b'_1 for B′ because it is unmerged for U). We remark that this function exhibits many different behaviors depending on the participant executing it. This level of complexity, combined with the asymmetry between the sets of operations performed by different participants, is exactly why a mechanized proof of security is, in our opinion, necessary to trust that MLS provides the expected security guarantees.

6.3.2 TreeKEM API

Users of TreeKEM are not expected to use low-level functions as shown in Figure 6.5. Instead, they use a high-level API that handles modifications to both the public state (the tree of public keys) and the private state (our path of private keys from our leaf to the root). We structure our F* code to implement a high-level API that only exposes functions to process proposals and commits, and to generate commits. Note that the group management functions (add, remove, update) are part of the TreeSync API and are orthogonal to TreeKEM. We focus on the functions for TreeKEM commits as they are the most interesting.

Processing a commit. Each participant needs to process two kinds of commits: add-only commits (without path update), and full commits (§6.2.3). For this reason, we process commits in two steps: first, we update our state and compute the commit secret, second, we perform a round of key schedule. Furthermore, the first step comes in two flavors, one for each commit type.

```
val decrypt_path_secret:
  my_li:leaf_index → upd_li:leaf_index {my_li ≠ upd_li} →
  treekem_public → treekem_private → update_path →
  bytes
let rec decrypt_path_secret my_li upd_li t p_priv p_upd =
  if leaf_index_same_side t my_li upd_li then (
    // The update path and the path to our leaf are on the same
    // side of the tree. Recurse in that subtree.
    let (child, _) = get_child_sibling t upd_li in
    decrypt_path_secret child (next p_priv) (next p_upd)
  ) else (
    // We are at the least common ancestor between us and the
    // updater. Obtain the path secret by decryption.
    let ciphertext_list = get_data p_upd in
    let (_, sibling) = get_child_sibling t upd_li in
    // Find our ciphertext by descending in the tree until we find
    // a non-blank node, and recover the index in the resolution.
    let my_index = find_resolution_index sibling my_li in
    let my_ciphertext = ciphertext_list[my_index] in
    // Find the corresponding decryption key. This involves
    // checking whether we were encrypted to as an unmerged leaf.
    let private_key = find_private_key sibling (next p_priv) in
    // With all this data gathered, we can now decrypt.
    decrypt private_key my_ciphertext
  )
```

Figure 6.5: Implementation of the decrypt_path_secret function, simplified.

```
val prepare_process_full_commit:
  treekem_state → path_update → group_context →
  result pending_process_commit

val prepare_process_add_only_commit:
  treekem_state →
  result pending_process_commit

val finalize_process_commit:
  pending_process_commit →
  pre_shared_keys → group_context →
  result (treekem_state & bytes)
```

These functions might fail, as indicated by the fact that return values are wrapped in a result. Reasons for failure include failed decryptions, or malformed path updates – the error case of result describes the nature of the error so that the client can act accordingly.

Creating a commit. Just like processing commits, creating new commits happens in two steps. In the case of a full commit, the first step, which handles the refresh of the tree and the commit secret, must itself be decomposed into two sub-steps, below.

```
val prepare_create_commit:
  treekem_state ⟶ entropy ⟶
  result (pending_create_commit & pre_path)

val continue_create_commit:
  pending_create_commit ⟶
  added_leaves:list nat ⟶ group_context ⟶
  entropy ⟶
  result (pending_create_commit_2 & path & list bytes)
```

The first function generates fresh path secrets and outputs the new public keys (in pre_path), for nodes along the affected path. The user can feed these new public keys into TreeSync to compute the new signature of our leaf node (that authenticates these new public keys) and compute a hash of the new tree – as mentioned earlier, we treat TreeSync as a signature primitive for the tree itself. This new tree hash is used within the group context, which is itself used when encrypting path secrets: this is what the second function does. It returns a pending commit creation object, an updated path, and the path secret that will be sent over to the joiners along with their welcome package to invite them into the group.

6.3.3 Execution model

One detail we omitted from our presentation (for conciseness and readability) is that all of those specification-level functions are actually parametric over the type of *bytes*, and over operations that operate on such bytes. We do so efficiently using the type class mechanism of F*.

This allows us to instantiate the specification either with concrete bytes (i.e. bitstrings) or with abstract symbolic bytes that are used in DY* [43] proofs (see §6.4.1). The former allows us to show that we are byte-for-byte conformant with the MLS standard, by running our specification (via F*'s extraction mechanism to OCaml) against test vectors and other implementations for interoperability testing. The latter, naturally, allows us to conduct our proof of security, in the next section.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Furthermore, we point out that our specification is free of any side effects: there is no memory (we never use a pointer or reference type), meaning the functions take, and return, a state, rather than modifying a global memory. Should some IO action (or, effect) need to happen, it suffices for the function to return, e.g., a list of messages to be effectively sent on the network.

To execute our specification and test it for interoperability, we wrote some glue code to allow our pure specification to interact with the effectful libraries such as networking. For proofs, we embed our specification in the trace-based semantics of DY*, as explained next.

6.4 A security theorem for TreeKEM

6.4.1 Background on DY*

DY* [43] is an F* [98] framework to state and prove security properties of cryptographic protocols. DY* uses a symbolic trace-based runtime model, where various participants can participate in a cryptographic protocol by calling cryptographic functions, generating random bytestrings, storing

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

[98]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

local state, logging events that indicate progress in the protocol execution, and sending messages on the network.

Threat model. DY* considers an active attacker that controls the network (hence can intercept, replay, or modify messages) and can dynamically compromise participants to learn the content of their private state.

Cryptographic assumptions. DY* abstracts cryptographic functions using the Dolev-Yao (or symbolic) model [56]. The symbolic model treats cryptographic functions as being perfect: for example, when sending a ciphertext on the network, the attacker learns nothing about the associated plaintext unless the attacker knows the corresponding decryption key (e.g. by compromising a participant), in which case they also learn the content of the plaintext.

[56]: Dolev et al. (1983), *On the security of public key protocols*

Security theorems. DY* users can express security properties as *reachability properties*, meaning that all traces reachable through protocol execution satisfy some security property (specific to each protocol). An example of trace property that encodes confidentiality would be: if a participant finishes the key exchange protocol and the attacker knows the exchanged key, then the attacker must have compromised one of the participants involved in the key exchange.

Security proofs. DY* relies on its user to provide a *trace invariant*, then prove that each protocol step preserves the invariant (hence any reachable trace satisfies the trace invariant) and prove that the trace invariant implies the desired security properties. Note that the trace invariants are not trusted, they are only a proof technique to prove properties on all reachable traces. To define the trace invariant, DY* provides two tools. The first tool, related to confidentiality, are security *labels*, which encode an over-approximation of the compromises for an attacker to know some given bytestring. Hence, if the attacker knows some bytestring (e.g. the private signature key of participant p) then this bytestring's label ensures that the attacker must have compromised some particular state (e.g. the state where participant p stores their private signature keys). Some labels are more secure than others, in which case we say the less secure label *flows* to the more secure one (which we note $l_1 \gtrsim l_2$). Security labels will be the main workhorse of security proofs for TreeKEM (§6.5). The second tool, related to authenticity, are cryptographic predicates: for example, every participant will only sign messages that satisfy the (protocol-specific) signature predicate. When a signature verifies, we can then deduce that it was either computed by an honest participant, in which case the signature predicate holds on the message, or that it was computed by the attacker, in which case they know the signature key, hence must have performed some compromise depicted by the signature key security label. Signature predicates will also be a workhorse for security proofs in TreeKEM, mostly through the work of TreeSync [68].

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

6.4.2 Preliminaries

History of a group. In our security theorem, we consider everything from the viewpoint of a participant p belonging to a TreeKEM group G. This participant has seen the group evolve over time, resulting in several epoch secrets being established throughout the group's lifetime. At each epoch, participant p logs an event containing the information on their local group

state at this epoch: the epoch secret K_n, the group roster $\mathrm{p}_1^{(n)}, \ldots, \mathrm{p}_m^{(n)}$ (containing p), the joiners $\mathrm{j}_1^{(n)}, \ldots, \mathrm{j}_q^{(n)}$ (contained in the group roster), the tree T_n (whose leaves form the group roster), and whether the commit is add-only. For example in Figure 6.2, assuming we start at epoch 0, we have $\mathrm{p}_1^{(0)} = a_0$ and $\mathrm{p}_1^{(1)} = a_1$, however, $\mathrm{p}_5^{(0)} = \mathrm{p}_5^{(1)} = \mathrm{p}_5^{(2)} = e_0$ and $\mathrm{p}_5^{(3)} = e_1$. Participants that didn't update since they joined have a special tag in T_n. When that is the case, we write $\mathrm{stale}_n(p')$; for example in Figure 6.2f, $\mathrm{p}_2^{(3)} = b_1'$ and $\mathrm{stale}_n(\mathrm{p}_2^{(3)})$. Naturally, joiners of this epoch didn't update since they joined, so $\mathrm{stale}_n(\mathrm{j}_i^{(n)})$. If we did not create the group, we have been invited in it by participant p_{inv} at epoch n_0. Furthermore participant p records the time at which they verify signatures: we write $\mathrm{T}(\mathrm{p}_i^{(n)})$ the time of verification of the signature of leaf node of $\mathrm{p}_i^{(n)}$, and $\mathrm{T}(\mathrm{j}_i^{(n)})$ the time of verification of the signature of key package of $\mathrm{j}_i^{(n)}$.

State storage. We consider a fine-grained model, where different parts of the state may be compromised independently. For instance, to account for a deployment that may use higher-security storage (e.g. HSMs) to store (long-term) signature keys, we can let the private node keys stored by a participant be compromised, without necessarily compromising the signature keys. Furthermore, we consider that different signature keys can themselves be compromised independently, just like initialization keys and epoch secrets. However, we consider that all nodes private keys are compromised together, since compromising one reveals all node private keys on the path from the compromised participant to the root.

State identifiers. We write $K_n@p$ to identify the state of participant p that stores the epoch secret K_n of group G at epoch n. We write $\mathrm{Sig}(\mathrm{p}_i^{(n)})$ to identify the state that stores the signature key of $\mathrm{p}_i^{(n)}$. We write $\mathrm{Init}(\mathrm{j}_i^{(n)})$ to identify the state that stores the initialization key of $\mathrm{j}_i^{(n)}$. We write $\mathrm{Node}(\mathrm{p}_i^{(n)})$ to identify the state that stores the node keys of $\mathrm{p}_i^{(n)}$ for the current version of $\mathrm{p}_i^{(n)}$. For example in Figure 6.2, $\mathrm{p}_5^{(2)} = e_0$ hence $\mathrm{Node}(\mathrm{p}_5^{(2)})$ corresponds to the node keys of $\{e_0, x_0, y_0, w_0, w_1, w_2\}$, while $\mathrm{Node}(\mathrm{p}_5^{(3)})$ corresponds to the node keys of $\{e_1, x_1, y_1, w_3\}$.

Notations. We write $\mathrm{Att}_t(b)$ when the attacker knows the bytestring b at time t. We write $\mathrm{Compromise}_t(S)$ when the attacker has compromised the state identified by S before time t.

6.4.3 Security properties

We now describe the security properties we have proved on TreeKEM. We state confidentiality as a trace property: if the attacker knows some epoch secret, then some set of states must have been compromised at some time in the past. In turn, we will see in §6.4.4 that this trace property implies the desired security guarantees of TreeKEM, such as add-security, remove-security, forward secrecy and post-compromise security (see §6.2.1). For the purpose of stating security goals in this paper, we assume that some state stores the epoch secret, but in our code, this secret is never actually stored since it would break forward secrecy of TreeDEM.

We consider three scenarios: in the first scenario, we consider a participant in a group that has moved into a new epoch, in the second scenario, we consider a participant that has just joined a group, in the third scenario, we

consider a participant that has just created a group. These three scenarios cover all that may happen within an MLS group; indeed, advancing an epoch in the first scenario is done via a commit that may contain any number of add, remove, or other operations, and optionally a path update. These three scenarios come with different security guarantees.

Confidentiality theorem for new epochs. Suppose a participant p is in a group G at epoch n with epoch secret K_n, participants $\mathrm{p}_i^{(n)}$, PSKs psks_n and joiners $\mathrm{j}_i^{(n)}$. If $\mathrm{Att}_t(K_n)$, then one of the following cases hold:

(1) $\exists i.\mathrm{Compromise}_t(K_n@\mathrm{p}_i^{(n)})$: the attacker has compromised before t the state containing the epoch secret of one of the participants $\mathrm{p}_i^{(n)}$ in the current group.

(2) $\exists i.\mathrm{Compromise}_t(\mathrm{Init}(\mathrm{j}_i^{(n)}))$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker has compromised before t the initialization key used to invite the joiner $\mathrm{j}_i^{(n)}$ into the group at epoch n.

(3) $\exists i.\mathrm{Compromise}_{\mathrm{T}(\mathrm{j}_i^{(n)})}(\mathrm{Sig}(\mathrm{j}_i^{(n)}))$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker has compromised the signature key of one of the joiners $\mathrm{j}_i^{(n)}$ in the group at epoch n, namely the one that signed their initialization key. In that case, the compromise must have happened before we checked their key package signature. This is a variant of case (2) where the attacker is active.

(4) $\mathrm{Att}_t(K_{n-1})$ and $\mathrm{add\text{-}only}_n$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker knows the previous epoch secret, and the commit that led to epoch n is an add-only commit (as explained in §6.2.3).

(5) $\mathrm{Att}_t(K_{n-1})$ and $\exists i.\mathrm{Compromise}_t(\mathrm{Node}(\mathrm{p}_i^{(n)}))$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker knows the previous epoch secret, and has compromised before t the node keys stored by a participant $\mathrm{p}_i^{(n)}$ of the current group after they last issued a path update.

(6) $\mathrm{Att}_t(K_{n-1})$ and $\exists i.\mathrm{Compromise}_{\mathrm{T}(\mathrm{p}_i^{(n)})}(\mathrm{Sig}(\mathrm{p}_i^{(n)}))$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker knows the previous epoch secret, and has compromised the signature key of a participant of the current group $\mathrm{p}_i^{(n)}$, namely the one that signs their leaf node is the tree. In that case, the compromise must have happened before we checked their leaf node signature. This is a variant of case (5) where the attacker is active.

(7) $\mathrm{Att}_t(K_{n-1})$ and $\exists i.\mathrm{Compromise}_t(\mathrm{Init}(\mathrm{p}_i^{(n)}))$ and $\mathrm{stale}_n(\mathrm{p}_i^{(n)})$ and $\mathrm{Att}_t(\mathrm{psks}_n)$: the attacker knows the previous epoch secret, and has compromised before t the initialization keys stored by a stale participant $\mathrm{p}_i^{(n)}$ of the current group. This possibility of compromise exists because the path secret is encrypted using the initialization keys of joiners. Note that p might not know what precise initialization key was compromised – it might be that p joined after $\mathrm{p}_i^{(n)}$, meaning p never saw the key package of $\mathrm{p}_i^{(n)}$. However, p knows it is *an* initialization key of $\mathrm{p}_i^{(n)}$ that got compromised (i.e., p knows i).

Confidentiality theorem when joining. Suppose a participant p joined a group G at epoch n with epoch secret K_n, participants $\mathrm{p}_i^{(n)}$, PSKs psks_n. If $\mathrm{Att}_t(K_n)$, one of the following cases hold:

(8) $\mathrm{Compromise}_{\mathrm{T}(p_{inv})}(\mathrm{Sig}(p_{inv}))$: the attacker has compromised the signature key of the participant that invited p. In that case, the

signature must have happened before we checked the signature in the GroupInfo part of the Welcome message (as explained in §6.2.4).

(9) Participant p_{inv} (who invited participant p in the group G) belongs to a group G at epoch n with epoch secret K_n, participants $\mathrm{p}_i^{(n)}$, PSKs psks_n and joiners that are a subset of $\{\mathrm{p}_i^{(n)} \mid \mathrm{stale}_n(\mathrm{p}_i^{(n)})\}$. In that scenario, we have all the hypotheses required to apply this theorem inductively on p_{inv}.

Confidentiality theorem when creating. Suppose a participant p created a group G (hence at epoch 0) with epoch key K_0. If $\mathrm{Att}_t(K_0)$, then:

(10) $\mathrm{Compromise}_t(K_0@p)$: the attacker has compromised before t the state containing the epoch secret of participant p (the only participant in the group).

Malicious participants. In previous MLS drafts, TreeKEM was vulnerable to attacks wherein a malicious participant could break the tree invariant and compute the epoch secrets after they are removed from the tree, hence breaking remove-security (e.g. "double-join attack" in [116, Fig 5 and Fig 8], or "attack on tree-signing" in [51, Fig 8]). In DY*, we model malicious participants as participants whose complete state is fully compromised: this has the same effect as if they were the attacker. Hence our security theorem accounts for malicious participants in its threat model, and we will see in §6.4.4 that it entails remove-security, making such attacks impossible. Indeed, the "double-join attack" [116, Fig 5] was since fixed by introducing the concept of blank nodes, and the "attack on tree-signing" [51, Fig 8] was since fixed by introducing the concept of parent hash and formally analyzed as part of the TreeSync sub-protocol [68].

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

6.4.4 Security corollaries

Using the TreeKEM security theorem in §6.4.3, we can now prove as corollaries the desired TreeKEM security guarantees stated in §6.2.1. What follows is manual reasoning: we are auditing our theorem statement to make sure it does indeed provide the security guarantees we want.

Add-security, remove-security. The security theorem implies that necessarily, one of the participants in the current epoch must be compromised. Indeed, each of the cases (1) to (3), (5) to (8) and (10) implies the compromise of a participant of the current group, because $\mathrm{j}_i^{(n)}$ and p_{inv} are participants of the current group. By induction, (4) implies a compromise of a participant in the group at epoch $n-1$ which is a subset of participants at epoch n because the commit is add-only. In case (9), by instantiating the theorem inductively on p_{inv} we deduce that a compromise must have happened in the current group roster. This implies that if no group participant at epoch n is compromised, then compromising any participant that was removed or that is not yet added provides no useful knowledge to the attacker.

Forward secrecy. The security property implies that some compromise of keys must have happened in the past, or not too far in the future. Indeed, in the case of compromise of a signature (cases (3), (6) and (8)), the compromise must have happened before we checked the signature, hence in the past. In cases (1) and (10), because participants delete epoch

secrets when moving to the next epoch, the compromise must happen before group participants move to the next epoch. In case (2), we notice that forward-secrecy relies on initialization keys being deleted quickly after processing a Welcome message. Not doing this undermines the forward-secrecy guarantees of MLS. We have found this was not part of the MLS deployment recommendations by the architecture document of MLS and notified the working group. In cases (4), (5) and (7), we do an induction on epoch $n-1$, and in case (9) we do an induction on participant p_{inv}.

Post-compromise security. The security theorem implies that a compromise cannot happen too far in the past. We can do a case analysis again. In cases (1) and (10) the compromise must have happened after $\mathrm{p}_i^{(n)}$ has computed the epoch secret K_n. Similarly, in case (5) the compromise must have happened after $\mathrm{p}_i^{(n)}$ has last issued a path update. In case (4), the group cannot heal from a compromise (unless psks_n is unknown to the attacker), hence we rely the healing of the previous epoch by doing an induction on epoch $n-1$. This means that if the group keeps doing add-only commits, there is no opportunity to recover from compromise (and implementations might need to adopt a policy encouraging updates to avoid this situation). In the cases of signature key compromise (cases (3), (6) and (8)) notice that such a compromise might have happened a while ago if the signature key is not rotated. This highlights that signature keys must be rotated regularly (so that it is changed before the attacker has the chance to forge a signature with it and perform an active attack) or stored securely e.g. in a hardware security module (HSM): this is recommended by the MLS architecture document. In cases (2) and (7), we note that to provide post-compromise security, the initialization key must not have been generated too long ago, otherwise this compromise may have happened far in the past. This highlights that key packages (hence initialization keys) must expire: adding a key package that was created too long ago could undermine post-compromise security. We have communicated to the MLS working group that this recommendation should be added to the MLS architecture document. The last case left to consider is (9), on which we do an induction on p_{inv}.

Lack of epoch authentication in Welcome. Note that in case (9) our theorem do not give the guarantee to the invitee (p) that that the inviter (p_{inv}) did invite them at this epoch. Indeed, in the presence of an active attacker, it may be possible that although the invitee successfully joined the group at epoch n, they were actually invited to join the group in a previous epoch (say, $n-1$). As discussed above, this cannot be used to affect the confidentiality guarantees of TreeKEM, hence is not a practical attack. We describe this more thoroughly in §6.A.

6.5 Proof methodology

We now discuss the methodology we used to prove the security theorem in §6.4.3. At a high level, we will use secrecy labels to prove that we only encrypt messages that are less secret than the key they are encrypted with, we will prove how secrecy labels evolve throughout the key derivations, and we will rely on the signature invariant when needed.

We describe our security proofs in the order keys are used in TreeKEM: we start with proofs on initialization keys (§6.5.1), then move on how they are used to encrypt the joiner secret in the Welcome message (§6.5.2),

then see how the key schedule produces a sequence of forward secret epoch secrets (§6.5.3), and finally dive into the tree invariant proofs (§6.5.4).

6.5.1 Security lemmas for initialization keys

The first key used by a participant in a group is its initialization key (sk_{init} in Figure 6.4), to process the Welcome message (§6.2.4). In this section, we present security lemmas for initialization keys that will be crucial in proofs associated with the Welcome message (in §6.5.2) and with the key schedule (in §6.5.3). As with the rest of the proofs, we describe security properties from the viewpoint of a participant, at a specific time point. In what follows, a crucial design choice is that we store each key in a separate state, which allows us to talk about the compromise of a particular key, instead of the compromise of the whole state of a participant.

Lemma for a participant's own key. Each participant generates its initialization key from fresh randomness, stores it in its private state and only uses it to decrypt Welcome messages. As such, we expect the only way for the attacker to obtain the key is to compromise the participant's state. We formally prove this fact, by showing any reachable trace falls into two categories: (1) either the initialization private key is currently unknown to the attacker (2) or the attacker has previously compromised the state storing the initialization private key.

Lemma for others' initialization keys. Each participant receives the initialization public key of each other participant in a key package (described in §6.2.4). To prevent the attacker from tampering with keys, the key package is signed by the corresponding participant. We expect that if the signature was computed by a honest participant, they have honestly computed their initialization key, otherwise the attacker must have compromised the signature key before we verified the key package. We formally prove this fact, by showing any reachable trace falls into three categories: (1) (2) as in the paragraph above or (3) the attacker has compromised the other participant's signature key before we verified the key package.

Using security labels. We encode the trace properties above using security labels, namely we prove that if we have verified a key package, then $\mathrm{Init}(\mathsf{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathsf{j}_i^{(n)}) \gtrsim \mathscr{L}(sk_{init})$ where $\mathsf{j}_i^{(n)}$ is the joiner we are considering and $\mathscr{L}(sk_{init})$ is the label of the initialization private key in their key package. This labeling property allows us to prove that any reachable trace falls into one of the three categories mentioned above, and is composable with rest of the security proofs.

6.5.2 Security lemmas for Welcome

The Welcome message consists of two parts: first, the GroupInfo object (gi_n in Figure 6.4) is signed, and the joiner secret (js_n in Figure 6.4) is encrypted with the initialization keys of joiners (sk_{init} in Figure 6.4). We now see the security proofs related to these two cryptographic computations.

Encrypting the joiner secret. To prove that it is safe to encrypt the joiner secret with the initialization keys of joiners, we must prove that the joiner secret is *less* secret than the initialization private key (as explained in §6.4.1). We prove this by combining the theorem on secrecy label of initialization keys in §6.5.1 and the theorem on key schedule that we will prove in §6.5.3. Formally, we prove that $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathcal{L}(sk_{init})$ by transitivity using the chain $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathrm{Init}(\mathsf{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathsf{j}_i^{(n)})$ (proved in §6.5.3) and $\mathrm{Init}(\mathsf{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathsf{j}_i^{(n)}) \gtrsim \mathcal{L}(sk_{init})$ (proved in §6.5.1).

Signing GroupInfo. We use the epoch secret (es_n in Figure 6.4) to derive the confirmation tag (ct_n in Figure 6.4) and combine it with the group context (gc_n in Figure 6.4) to form the GroupInfo object (gi_n in Figure 6.4). Further, the GroupInfo is signed by the inviter. In doing so, the inviter attests that they are in a group with group context gc_n (which includes the epoch number, group identifier, a hash of the tree, etc), and with an epoch secret es_n that produces the confirmation tag ct_n.

Verifying GroupInfo. When the joiner verifies the GroupInfo signature, they deduce that either the attacker knew the inviter signature key before they have verified the GroupInfo (and the attacker is doing an active attack), or that the inviter is in a group with the same group context and with an epoch secret that produces the same confirmation tag. Finally, by collision resistance for the hash function, we deduce that we must have the same epoch secret. In doing so, we have proved the cases (8) and (9) of the security theorem in §6.4.3.

6.5.3 Security lemmas for the key schedule

The key schedule (Figure 6.4) derives a stream of secrets through extraction (xtr) and expansion (xpd). We prove two things about the key schedule. First, we prove that the epoch secret (es_n) combines the security of the commit secret (cs_n) and the previous epoch secret (es_{n-1}). Second, we prove that the joiner secret is less secret than the private key of joiners (sk_{init}). The first goal is easily proved using the semantics of extraction in DY*, we therefore focus on the joiner secret.

Labeling of the joiner secret. The joiner secret can be trivially compromised if the attacker knows is_n and cs_n. If not, the attacker can gain access to js_n if they compromise the initialization key sk_{init} of a joiner at epoch n. The joiner secret js_n is thus the first secret in the key schedule that is revealed to the attacker when they compromise sk_{init}. This fact means that the joiner secret is less secret than the secret that directly precedes it in the key schedule, hence that interesting proofs must happen in the expansion with the group context that produced js_n. In DY* the security label of the output of KDF.expand may be weaker than the label of its input, and it may depend on additional inputs of the KDF (here, the group context). Indeed, the group context contains the transcript hash, which contains the proposals adding the key packages of joiners in the group. This allows us to say that the label of the joiner secret is weakened using the label of the joiners' initialization keys. More precisely, we prove that $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathrm{Init}(\mathsf{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathsf{j}_i^{(n)})$ which is then used in §6.5.2 to prove that it is safe to encrypt the joiner secret to each new joiner.

Note that the use of group context in the key derivation here is important for security, without it it would be possible that two group participants

who don't agree on the key packages added at this epoch still compute the same joiner secret, which would break the security theorem.

6.5.4 Formally proving the tree invariant

The tree invariant follows the same principle as earlier (§6.5.1): first, we consider the security invariant from our own point of view, then when receiving an update from another participant, we consider the possibility that their signature key might have been compromised. To establish our security lemma, we rely on a tree invariant.

The tree invariant. The tree invariant captures the security guarantees offered by TreeKEM; we show that this invariant is preserved through every step of the protocol, which ultimately allows us to conclude that the cryptographic tree state of TreeKEM (§6.2.2) is secure. The tree invariant is a disjunction that captures the two points of view we mentioned earlier, and states that if the private key of a (possibly internal) node n is known by the attacker, then either $\text{Compromise}_t(\text{Node}(p))$ or $\text{Compromise}_{\text{T}(p)}(\text{Sig}(p))$, where p belongs to the subtree rooted at n and is not an unmerged leaf for n. The former disjunct captures the fact that a participant may simply have been compromised; the latter disjunct captures that we may have been the victim of an active attack, in which the attacker injects a malicious path update that is signed with another participant's compromised signature key. Note that the $\text{T}(p)$ in subscript indicates a temporal relation: the signature key must have been compromised *before* we verified that participant's leaf node.

Concretely, we prove this invariant by relying on DY* secrecy labels, described below.

Lemma on the sender side. We use labels to track the usage of path secrets throughout the specification of a commit. Every base cryptographic operation in DY* is annotated with labels in its type; this means that every usage of the path secret forces us to reason about the set of compromises by the attacker that would lead to knowledge of this path secret.

The label of each refreshed path secret (i.e., the output of the KDF) flows towards all of the sub nodes secrets. Looking back at Figure 6.2b, performing a KDF expansion with the path secret of t_1 produces the path secret of u_1. Because the label of u_1 covers participants a_1, b_0, c_0, d_0, it is weaker than the label of t_1 that covers participants a_1 and b_0 only. The path secret of u_1 is encrypted with v_0's node secret – this is a safe thing to do, because the label of u_1 is weaker than the label of v_0 (that covers c_0 and d_0), which is imposed by encryption in DY*: the label of the key must be stronger than the label of the message.

At this stage, we have almost obtained the tree invariant: if the attacker knows the path secret of u_1, it must have compromised a participant p that is one of a_1, b_0, c_0 or d_0. Because we chose the label of path secrets to be the same as that of node secrets, and combined with the tree invariant that previously held upon entering the function, it means that $\text{Compromise}_t(\text{Node}(p))$ or $\text{Compromise}_{\text{T}(p)}(\text{Sig}(p))$, and the invariant is re-established.

Lemma on the receiver side. We reuse an earlier formalization (and proofs) of TreeSync [68], in order to use TreeSync as a signature mechanism specialized for TreeKEM. This follows the same logic as with signing

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

the initialization key, except this time the committer authenticates every subtree rooted at nodes they have modified (from their leaf up to the root), using TreeSync to do it efficiently with one signature in their leaf node.

The authentication covers the entire subtree, that is, the new node public keys, and all of their intended recipients, as they appear in the tree invariant. Using the semantics of DY*, we know that if the participant is honest, then the tree invariant is guaranteed by the signature (there was no compromise). If there is a compromise, it must be the case that the signature key was compromised before we checked the signature. This is one of the cases accounted for by the tree invariant, meaning that the invariant is re-established.

6.6 Discussion

We have presented a machine-checked security proof for a bit-level precise, executable, interoperable specification of TreeKEM. The specification is written in 1.3k lines of F* code, and our security proofs are in 11k lines of F*, relying on the DY* framework. The full development is available online at the URL below, along with instructions for running the code and verifying the proofs as well as pointers to these:

```
https://github.com/Inria-Prosecco/treekem-artifact
```

Anyone can download, run and test the code. Reading the proof statements requires some knowledge of functional programming and formal logic, extending the proofs requires knowledge of the F* proof assistant.

Benefits of Machine-Checked Proofs. MLS is a large protocol and even its TreeKEM component is quite complex. It maintains a dynamic tree data structure with some unusual features such as unoccupied leaves, blank nodes, filtered nodes, unmerged leaves, etc. It defines novel cryptographic mechanisms for encapsulating secrets to trees of public keys. It defines new serialization formats for trees, paths, and various messages and cryptographic inputs.

Ensuring that a formal specification of TreeKEM captures all these notions correctly can be hard and is greatly aided by being able to execute and test the specification. Furthermore, when proving properties about the protocol, it is easy to forget various corner cases, but a machine-checked proof keeps us honest and ensures that we account for anything that may arise in an execution of TreeKEM. Indeed, we believe that a pen-and-paper proof for the full TreeKEM protocol at this level of detail would be hard to write and even harder to check for correctness.

Symbolic vs. Computational Proofs. Our proofs in this paper rely on a symbolic (i.e. Dolev-Yao) model of cryptography, where the public key encryption is treated as a perfect black-box that can only be broken if the attacker knows the private key. In contrast, the classic pen-and-paper proofs of TreeKEM in prior work [51, 114, 115] operate in a computational model of cryptography, where public key encryption is modeled in terms of a probabilistic polynomial-time adversary. Both symbolic and computational models have their strengths and weaknesses [36]. Computational cryptographic assumptions are more precise, but symbolic models yield better proof tools and hence can handle more protocol details and finer-grained key compromise.

[51]: Alwen et al. (2022), *On The Insider Security of MLS*
[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*
[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*
[36]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

For example, let us compare our work to the most recent pen-and-paper proof for TreeKEM [51]. This paper proves a computational security theorem for an abstract model of TreeKEM draft 12, expressed as pseudocode. Like us, they consider an active attacker that can dynamically compromise participants, and consider malicious participants. However, they only allow coarse-grained compromise: the attacker can only compromise all keys held by a participant, unlike our work where the attacker can compromise the node secret keys stored by a participant without compromising their signature keys. Another difference is that they consider bad randomness as part of the threat model, which we do not.

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

We also note that in MLS draft 12, there is no distinction between the initialization key and the leaf node key. This hurts forward-secrecy in their theorem since new participants need to hold on to this medium-term key even after joining a group. The TreeKEM design in the published MLS standard was updated to separate the initialization key and leaf node key, which yields a stronger theorem in our case.

The main proof in [51] shows that the attacker cannot distinguish the epoch secret from fresh randomness when a safety predicate (i.e. a trace invariant) holds. Much of the high-level logic in their proof and ours is the same; the main differences arise in the different treatment of public-key encryption, in our handling of low-level cryptographic formats (ignored in their proofs), and in our proofs being oriented towards being machine checkable.

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

Guidance for Erasing Keys, Removing Members. Our security theorem clearly specifies which key compromises could affect the confidentiality guarantees of TreeKEM. This provides useful guidance for MLS implementations and deployments. For example, initialization keys must be deleted after a Welcome message is decrypted or when they expire, and this is important for both forward and post-compromise security. Our theorem also includes cases for participants that have joined the group but not yet updated their encryption keys. Such *stale* participants affect the security of the whole group. They could be identified and potentially removed from the group after a period of inactivity. We have proposed to add these recommendations to the MLS architecture document, and checked that the main MLS implementations (mlspp, OpenMLS and mls-rs) properly erase initialization keys upon Welcome decryption.

Experimenting with Protocol Improvements. TreeKEM is designed with many defense-in-depth mechanisms; some were needed for our proofs (e.g. the use of group context), and some made our proofs simpler (e.g. the authentication of subtrees in TreeSync). Others we did not need (e.g. the use of group context in HPKE encryption), and this may indicate potential future optimizations in the protocol.

Some changes to the protocol would have simplified our proofs. For example, the transcript hash input format is defined as a concatenation, which makes it harder to prove that it is unambiguous. Using a length-prefixed format would have simplified this proof. As another example, our proofs for path secret derivation would have been much simpler if the sibling tree hash were used in the key derivation.

Our executable specification and machine-checked proofs provide a good basis for experimenting with different optimizations and variations of MLS. Running the specification makes it easy to compare the impact of optimizations on message size and computation time. Rerunning the

proofs ensures that the new protocol satisfies the same properties as MLS, and maybe provides new security guarantees.

Future Work. Two natural directions for future work would be to develop a machine-checked proof of TreeDEM (and hence complete the verification effort for the MLS standard) and to investigate the post-quantum security of TreeKEM.

Acknowledgments

We are indebted to Franziskus Kiefer and Raphael Robert for proofreading a draft of this paper and providing precious feedback.

This work received funding from the French Government, managed by the ANR under grant agreements ANR-22-PECY-0006 and ANR-19-P3IA-0001.

6.A Lack of epoch authentication in Welcome

In TreeKEM (hence MLS), when an invitee joins a group through the Welcome procedure (§6.2.4), they do not have the guarantee that they were invited at this epoch: they may have been invited in a previous epoch. Indeed, an attacker can exploit the fact that during the Welcome process, only the GroupInfo is signed, but the encrypted group secrets are not signed, hence not bound to any epoch.

The attacker must be active, hence we suppose they control the network. The group is at epoch n, with a tree similar to Figure 6.2a, with E and F blanked. The attacker proceeds as follows:

- A invites E in the group and commits to epoch $n + 1$. The attacker do not transmit the Welcome message to E.
- A invites F in the group and commits to epoch $n + 2$.
- The attacker compromises the initialization key of F, and use it to decrypt the encrypted group secrets (i.e. joiner secret of epoch $n + 2$ and path secret of node X).
- The attacker re-encrypts the group secrets with the initialization key of E.
- The attacker sends a Welcome message to E, containing this re-encrypted group secrets and signed GroupInfo for epoch $n + 2$ (obtained when inviting F).
- E successfully joins at epoch $n + 2$ although it was invited at epoch $n + 1$.

This works by compromising an initialization key, but it works the same if F is a malicious participant (recall that we model malicious participants as participants whose state is fully compromised, see last paragraph of §6.4.3).

As explained in §6.4.4, this cannot be used to attack confidentiality guarantees of TreeKEM.

Final words

Related work 7

In this section, we give a broad overview of the work related to this thesis: we analyzed MLS (§7.1), by performing a computer-aided analysis of a secure messaging protocol (§7.2) on an executable specification (§7.3). To do so, we developed tools to analyze cryptographic protocols (§7.4).

7.1 Analysis of MLS

To recall, we analyze TreeSync (Chapter 5) and TreeKEM (Chapter 6) in the final version of MLS (RFC 9420 [21]) in the symbolic model against an insider attacker, meaning that the attacker may be a participant within messaging groups. MLS was standardized throughout 20 drafts, many of the related work analyze intermediate drafts.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

Many works study variations of MLS with different trade-offs, in here we choose to focus on the works that participated in the *proactive* analysis of MLS.

7.1.1 Pen & paper analysis

Alwen et al. [114] analyze draft 6 on pen & paper in a game-based framework, against a passive attacker. They only consider TreeKEM's tree (see §6.2.2) without its key schedule (see §6.2.3), and notice that in itself, the tree only provides weak forward secrecy, as we previously explained in §6.2.3. Whereas to provide strong forward secrecy, we consider that the key schedule is an essential part of TreeKEM, they propose an alternative solution to provide strong forward secrecy within the tree, by using Updatable Public-Key Encryption (UPKE), however this proposition did not make it into the final version of MLS.

[114]: Alwen et al. (2020), *Security Analysis and Improvements for the IETF MLS Standard for Group Messaging*

Alwen et al. [115] analyze draft 11 on pen & paper in a game-based framework, against a passive attacker (that can tamper with messages, but only if they end up being rejected by recipients, hence this is slightly more expressive that [114]). They analyze the Secure Group Messaging (SGM) properties of MLS, which correspond to confidentiality properties (forward secrecy, post-compromise security) and authenticity properties. They also modularize MLS into three components: Continuous Group Key Agreement (CGKA), Pseudo-Random Function / Pseudo-Random Number Generator (PRF-PRNG) and Forward-Secure Group Authenticated Encryption with Additional Data (FS-GAEAD). This modularization compares to ours as follows: FS-GAEAD corresponds to TreeDEM, PRF-PRNG corresponds to the key schedule of TreeKEM (see §6.2.3), CGKA corresponds to the tree of TreeKEM (see §6.2.2). The welcome subcomponent of TreeKEM (see §6.2.4) is baked-in the assembly of the three above components to form a Secure Group Messaging (SGM) protocol. They do not consider TreeSync, because it is only useful against insider attackers. This is the first work to analyze MLS the actual *secure messaging* guarantees of MLS, instead of focusing on its continuous group key agreement sub-protocol, and obtained a positive result.

[115]: Alwen et al. (2021), *Modular Design of Secure Group Messaging Protocols and the Security of MLS*

Cremers et al. [130] look at the healing post-compromise across groups

[130]: Cremers et al. (2021), *The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter*

in abstract secure group messaging systems, and on MLS draft 11. They notice that messaging groups as in MLS perform worse than implementing groups using pairwise Signal channels, because with pairwise Signal channels, the healing of a larger group implies the healing of every subgroup it contains. They further notice it is difficult to heal from the compromise of a signature key, to solve this problem they investigated the design space of signature key rotation policies and reported their finding in the MLS architecture document.

Brzuska et al. [52] analyze draft 11 on pen & paper in a game-based framework, against an active attacker. They only consider the key distribution mechanism of MLS, which is composed of TreeKEM's tree (see §6.2.2) in combination of its key schedule (see §6.2.3). In doing so, they abstract away how participants join a group or process commits, as well as the overall authentication guarantees provided by other MLS components. They proposed an improvement to the key schedule that was integrated in the final version of MLS. This change was crucial to our proof of TreeKEM, we discussed it at length in §6.5.3.

[52]: Brzuska et al. (2022), *Security Analysis of the MLS Key Derivation*

Alwen et al. [51] analyze draft 12 on pen & paper in the Universal Composability framework, against an insider attacker, meaning that the attacker is a participant within a messaging group. Their analysis covers both TreeSync and TreeKEM, but they conduct their analysis monolithically. They found several attacks that were fixed in the final version of MLS. They propose several variants of the parent hash mechanism of TreeSync, including the one we study in Chapter 5 (named "tree parent hash"). They discarded this variant as being "not workable due to other mechanisms of MLS", although the working group eventually chose this option (notably after the work we did in Chapter 5).

[51]: Alwen et al. (2022), *On The Insider Security of MLS*

Cremers et al. [139] analyze the final version of MLS (namely, RFC 9420 [21]) on pen & paper in the Universal Composability framework, against an insider attacker. Their analysis cover both TreeSync and TreeKEM (in our modularization), and also "external operations" such as external proposals or external commits that we do not cover in Chapter 6.

[139]: Cremers et al. (2025), *ETK: External-Operations TreeKEM and the Security of MLS in RFC 9420*

Other works have studied group key establishment, but none of them supports both asynchronous messaging and dynamic groups, which is a prerequisite for MLS. Nevertheless, We refer to [128] and [129] for surveys.

[128]: Manulis (2006), *Security-Focused Survey on Group Key Exchange Protocols*

[129]: Poettering et al. (2021), *SoK: Game-Based Security Models for Group Key Exchange*

7.1.2 Computer-aided analysis

A few works try to use (semi-)automated tools to obtain machine-checked symbolic security proofs for abstract models of TreeKEM.

Bhargavan et al. [116] analyze simplified models of early designs of TreeKEM in the draft 7 of MLS using a symbolic model in F* and compare its security and performance with alternative designs. In the process, they found a "Double Join Attack", and proposed a fix which corresponds to the creation of TreeSync, although they do not consider it as an independent protocol. This work served as inspiration for DY* [43], and ultimately this thesis, in which we analyzed a later (hence more complex) design of TreeKEM in all its details.

[116]: Bhargavan et al. (2019), *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*

Cremers et al. [131] analyze in Tamarin (symbolic model) a simplified version of TreeKEM in the draft 10 of MLS and prove forward secrecy (but not post-compromise or remove security). Note that we can prove

[131]: Cremers et al. (2023), *Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis*

this property without TreeKEM's tree (see §6.2.2), indeed it is achieved by TreeKEM's key schedule (see §6.2.3) regardless of the actual secrecy of the "commit secret" (which is what allows add-only commits).

7.2 Computer-aided analysis of messaging protocols

Several works use computed-aided techniques to analyze messaging protocols other than MLS.

Kobeissi et al. [54] analyze the Signal Protocol using both ProVerif and CryptoVerif. They only manage to analyze a small number of messages with this method: two messages using ProVerif (symbolic guarantees) and one message using CryptoVerif (computational guarantees). This analysis shows the limitations of *fully automatic* verification.

[54]: Kobeissi et al. (2017), *Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach*

Bhargavan et al. [43] analyze the Signal Protocol as a case-study of DY*. They provide symbolic guarantees for an unbounded number of messages. They manage to perform this proof by writing protocol invariants by hand, as we described in Chapter 2.

[43]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

Linker et al. [140] analyze PQ3 [141], a modification on the Signal Protocol [53] to have post-quantum post-compromise security in Apple's iMessage. They use Tamarin (symbolic guarantees) to prove forward secrecy, post-compromise security, and resistance against harvest-now-decrypt-later attacks, for an unbounded number of messages. They manage to perform this proof by writing by hand invariants and lemmas in Tamarin.

[140]: Linker et al. (2024), *A Formal Analysis of Apple's iMessage PQ3 Protocol*

Bhargavan et al. [142] analyze PQXDH [143], a recent post-quantum initial key agreement for Signal, using both ProVerif (symbolic guarantees) and CryptoVerif and (computational guarantees).

[142]: Bhargavan et al. (2024), *Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging*

Cremers et al. [144] analyze Sesame [145] using Tamarin and notice that although Signal's Double Ratchet [53] does provide post-compromise security, this strong guarantee is not lifted to Sesame [145] because of how it merges multiple Double Ratchet sessions to encrypt messages in a conversation. They managed to attack post-compromise security in practice with a clone attacker.

[144]: Cremers et al. (2023), *Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations*

Cremers et al. [146] then establish an impossibility result, and show using Tamarin that a messaging system fundamentally cannot provide post-compromise security if it also needs to be resilient again some type of state loss (that may happen in practice).

[146]: Cremers et al. (2024), *Impossibility Results for Post-Compromise Security in Real-World Communication Systems*

7.3 Analysis of executable specifications

Several works analyze executable specifications of protocols, as we did with TreeSync (Chapter 5) and TreeKEM (Chapter 6).

Bhargavan et al. [134] analyze using F* a reference implementation of TLS 1.2, and provide computational guarantees.

[134]: Bhargavan et al. (2013), *Implementing TLS with Verified Cryptographic Security*

Ho et al. [100] analyze Noise [147], a family of 59 secure channel protocols specified using a custom protocol description language. They do so by writing a generic interpreter for Noise protocols, proving the security of the interpreter *once and for all* using DY* [43], and finally derive

[100]: Ho et al. (2022), *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*

efficient C implementations from the interpreter, hence for all 59 protocol instantiations.

Arquint et al. [63] analyze a reference implementation of Wireguard [148] using a new technique they developed to do symbolic analysis in the Gobra program verifier [64]. We already discussed their approach toward protocol verification in §2.2.11.

[63]: Arquint et al. (2023), *A Generic Methodology for the Modular Verification of Security Protocol Implementations*

7.4 Tools for analyzing cryptographic protocols

There exists a variety of tools to analyze cryptographic protocols; we refer the reader to [36] for a thorough survey.

[36]: Barbosa et al. (2021), *SoK: Computer-Aided Cryptography*

Computational tools. Several tools give computational guarantees (as do traditional pen & paper proofs), they rely on various techniques with different trade-offs.

EasyCrypt [37] allows defining cryptographic systems and security statements in a game-based style similar to the pen & paper proofs ones, and proving them using tactics that operates on a logic called "probabilistic Relational Hoare Logic" (or "pRHL"), or using an SMT solver.

[37]: Barthe et al. (2011), *Computer-Aided Security Proofs for the Working Cryptographer*

CryptoVerif [38] allows defining cryptographic protocols as processes, and proving their security in an automated fashion: CryptoVerif will try to find the game-hops automatically, and when it fails, a custom tactic language allows the user to specify which game-hops to perform.

[38]: Blanchet (2007), *CryptoVerif: Computationally sound mechanized prover for cryptographic protocols*

Noticing that CryptoVerif is particularly geared toward proving the security of protocols, and that EasyCrypt is more geared toward proving security of cryptographic primitives, authors of both tools collaborated on a translation from CryptoVerif to EasyCrypt [149], so that security assumptions on cryptographic primitives that cannot easily be proved in CryptoVerif can be translated to and proved using EasyCrypt.

[149]: Blanchet et al. (2024), *CV2EC: Getting the Best of Both Worlds*

Squirrel [39] allows defining cryptographic protocols as processes, and proving their security using tactics that operate on the Bana-Comon logic [40], which proves security against a "Computationally Complete Symbolic Attacker": the goal is to obtain computational guarantees while doing proofs that look as symbolic as possible.

[39]: Baelde et al. (2021), *An interactive prover for protocol verification in the computational model*

Symbolic tools. Several tools give symbolic guarantees.

ProVerif [41] allows specifying protocols as processes that run in parallel, and proving security properties automatically, which can be either reachability properties or equivalence properties. ProVerif users can write lemmas or inductions to help the proof search.

[41]: Blanchet et al. (2016), *Modeling and verifying security protocols with the applied pi calculus and ProVerif*

Tamarin [42] allows specifying protocols as state machines using multiset rewriting rules, and proving security properties automatically, which can be either reachability properties or equivalence properties. Tamarin users can write lemmas or inductions to help the proof search, and use a Graphical User Interface (GUI) to do the proofs by hand if needed.

[42]: Meier et al. (2013), *The TAMARIN prover for the symbolic analysis of security protocols*

8 Conclusion

Un bon repas ne se termine que lorsque l'on est écœuré.

Luc Chabassier, *after a raclette party*

The initial goal of this thesis was to develop tools to analyze secure group messaging protocols, and use them to proactively analyze MLS [21] during its standardization; furthermore, we set out to do so on a bit-precise, executable, interoperable specification.

[21]: Barnes et al. (2023), *The Messaging Layer Security (MLS) Protocol*

8.1 Impact on MLS

Although we did not manage to analyze MLS in its entirety, we successfully analyzed a significant part of it, namely its sub-protocol in charge of authenticating the group state (TreeSync [68], Chapter 5) and its sub-protocol in charge of continuously establishing a secret group key (TreeKEM [135], Chapter 6), thereby paving the way to a complete analysis. The additional challenge of performing our analysis on a bit-precise, executable, interoperable specification was fruitful: this led to a more precise analysis which allowed us to catch subtle attacks that were missed in previous analysis, namely the signature ambiguity attack we discussed in Chapter 5, and propose fixes to the working group.[1]

[68]: Wallez et al. (2023), *TreeSync: Authenticated Group Management for Messaging Layer Security*

[135]: Wallez et al. (2025), *TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security*

1: `https://github.com/mlswg/mls-protocol/pull/526`

Interactions with the MLS working group. We had an excellent synergy with the MLS working group which was highly reactive and appreciative of any design flaws found by various teams, and gladly accepted various design improvements. Not all design improvements we proposed stemmed from our analysis of MLS, they sometimes stemmed from our *specification* of MLS. Indeed, in the perspective to analyze our specification, we set out to find the most elegant way to specify MLS, we then reported our findings to the MLS working group that sometimes enjoyed our way to specify MLS, and updated the MLS RFC to match our specification: this happened for example with the message framing part of MLS,[2] that belongs to TreeDEM sub-protocol which we did not analyze yet.

2: `https://github.com/mlswg/mls-protocol/pull/523`

Modularization of MLS. One of the ways we elegantly specify MLS is by specifying it *modularly* into three sub-protocols named TreeSync, TreeKEM and TreeDEM (see Chapter 5). This modularization was key to our success in analyzing MLS, because this allows us to perform bite-sized analysis. We proposed to the MLS working group to refactor the MLS protocol document around this modularization, however, although the MLS working group enjoyed our modularization and the change would only be editorial, they unfortunately declined our suggestion because the change was deemed to be too big, and too close from the final standardization of MLS (although history has shown the MLS standardization would happen two years after we proposed this change). Nevertheless, the modularization exists and can be relied on to better understand MLS: this is possible thanks to the working group accepting a small change (roughly 30 lines) to strengthen the parent-hash mechanism[3] that in turn allowed us to extract TreeSync as an independent, self-contained sub-protocol of MLS.

3: `https://github.com/mlswg/mls-protocol/pull/527`
`https://github.com/mlswg/mls-protocol/pull/713`
`https://github.com/mlswg/mls-protocol/pull/731`

8.2 Insights for Protocol Design and Analysis

Testing protocol changes. We believe that machine-checked analysis is useful to test protocol modifications: with pen & paper proofs, testing where the proof breaks when the protocol is modified requires a human to tediously re-check the proof, whereas in our case, we delegate this job to a computer which precisely points if and where the proof breaks, only then a human must take over to repair the proof. This is not just some wishful thinking, we were able in practice to test changes to MLS and repair proofs after such changes: for example, we proposed to the working group an overhaul of the unmerged leaves mechanism of TreeSync[4] in order to reduce the amount of data stored inside the tree and make unmerged leaves simpler to understand, we updated our proofs to account for this change and it required only a day of work. Unfortunately this change did not make it into MLS, because it introduced extra metadata into the public tree, that is, the epoch at which participants last issued a path update.

4: `https://github.com/mlswg/mls-protocol/pull/752`

Executable specifications. Doing our analysis on a bit-precise, executable, interoperable specification had several benefits. First, we have been able to find subtle attacks in MLS, such as the signature ambiguity attack (see Chapter 5) that previous analysis did not catch because they were not accounting for the precise message formats used by MLS. Second, we have been able to do interoperability testing with other implementations, and found bugs in three implementations (including ours): we found that OpenMLS was switching the info and ad fields of HPKE; that mlspp extracted joiner_secret twice in the key schedule instead of extracting it with psk_secret; and that we were serializing the Proposal tag on 8 bits instead of 16 bits (after failing to account for this change when updating our specification from draft 11 to draft 12).

Improving DY*. The complete analysis of TreeKEM (Chapter 6) was beyond the capabilities of DY*, this led us to do multiple improvements to DY* (Chapter 3), which quickly became pervasive: for example, the modular invariant technique (§3.1) is now *the* way to write invariants in DY*, and the revamp of the label framework (§3.2) ultimately changed our way to understand what *truly* is a label.

Toward better message formats in cryptographic standards. After finding the signature ambiguity attack in MLS (see Chapter 5), we spent some time to think about this general class of message formatting attacks and cross-protocol attacks, in particular how it could have been easily avoided through a better discipline in the design of message formats. This ultimately led to the work of Comparse [67] (Chapter 4), where we give conditions on message formats that are sufficient to avoid these type of attacks, and necessary to soundly abstract message formats away in security proofs. Furthermore, our message format conditions shed light on the importance to design future-proof message formats: indeed, for example, a signature key in version 1 of a protocol may end up being also used in version 2 of the protocol in the future; we can avoid cross-protocol attacks by designing a message format that can be extended through protocol versions. We have done so for MLS, but in the end, we firmly believe all protocol designers should have these conditions in mind when creating new protocols to systematically avoid these problems.

[67]: Wallez et al. (2023), *Comparse: Provably Secure Formats for Cryptographic Protocols*

8.3 Limitations and Future Work

There are still parts of MLS we didn't analyze yet, such as TreeDEM, and also more arcane features of MLS such as external commits, external proposals, group reinitialization, and subgroup branching. Extensions of MLS are being standardized, they could also benefit analysis.

Although our specification is executable and has been successfully integrated in a prototype version of Skype (see Chapter 5), it is certainly slower than other MLS implementations which were designed with efficiency in mind (whereas we had *provability* in mind). To bridge this gap, we could use *program verification* techniques to prove that an efficient implementation adheres to our specification, thereby lifting the security guarantees we proved on our specification to guarantees on this efficient implementation.

The conditions on message formats we designed in Comparse (Chapter 4) are intended for protocol analysts, but are currently not well-suited for an audience of protocol designers. Therefore, it would be useful to create a document that would provide actionable guidelines on the design of message formats in cryptographic standards; although we could argue that every standard should use Comparse to obtain machine-checked proofs that their message formats are secure, we recognize that this is a researcher fantasy, and that we could better reach the audience of protocol designers by writing a document tailored for them, with guidelines that are easy to apply.

It has been mostly enjoyable to use DY* to analyze TreeKEM (Chapter 6), especially to analyze its cryptographic core. However, this cryptographic core must in the end be connected to "outside world", for example to send and receive messages, retrieve and store state, etc. This takes the form of effectful functions which we then expose to the attacker, meaning that these effectful functions must also be analyzed. The analysis of such effectful functions currently feels less enjoyable, because the proof is relatively straightforward but nevertheless require non-trivial amount of work. We envision writing tactics to ease these proofs and allow users to focus only on its non-straightforward parts.

Symbolic analysis tools such as ProVerif [41] or Tamarin [42] allow users to define new symbols and equational theories, thereby allowing them to support new kind of cryptographic functions. In DY*, the set of supported cryptographic functions is fixed, baked-in the core of DY*, therefore extending this set would require to fork DY*, and potentially issue a pull-request to DY*. We envision it should be possible to use F*'s dependent types to create a modular type for `bytes`, so that users can extend it with custom cryptographic functions. We have a proof-of-concept supporting the fact that doing so is possible, but the resulting `bytes` type is cumbersome to use: more work would be needed to benefit this modularity without significant drawbacks.

Bibliography

Here are the references in citation order.

[1] A. Kerckhoffs. 'La cryptographie militaire'. In: *Journal des Sciences Militaires* (1883), pp. 5–38, 161–191 (cited on page 2).

[2] *The US 6812 Division Bombe Report Eastcote 1944*. https://www.codesandciphers.org.uk/documents/bmbrpt/bmbpg006.htm. 1944 (cited on page 2).

[3] W. Diffie and M. Hellman. 'New directions in cryptography'. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638 (cited on page 3).

[4] R. L. Rivest, A. Shamir, and L. Adleman. 'A method for obtaining digital signatures and public-key cryptosystems'. In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. DOI: 10.1145/359340.359342 (cited on page 4).

[5] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. 'SoK: Secure Messaging'. In: *IEEE Symposium on Security and Privacy (S&P)*. 2015, pp. 232–249 (cited on pages 5, 103, 126).

[6] Philip Zimmermann. *PGP Marks 10th Anniversary*. https://www.philzimmermann.com/EN/essays/PGP_10thAnniversary.html. 2001 (cited on page 5).

[7] Philip Zimmermann. *Author's preface to the book: "PGP Source Code and Internals"*. https://www.philzimmermann.com/EN/essays/BookPreface.html. 1995 (cited on page 5).

[8] Philip Zimmermann. *PGP source code and internals*. Cambridge, MA, USA: MIT Press, 1995 (cited on page 5).

[9] Philip Zimmermann. *Significant Moments in PGP's History: Zimmermann Case Dropped*. https://philzimmermann.com/EN/news/PRZ_case_dropped.html. 1996 (cited on page 5).

[10] Alison Dame-Boyle. *EFF at 25: Remembering the Case that Established Code as Speech*. https://www.eff.org/deeplinks/2015/04/remembering-case-established-code-speech. 2015 (cited on page 5).

[11] *U.S. Court of Appeals for the Ninth Circuit: Bernstein v. USDOJ*. https://archive.epic.org/crypto/export_controls/bernstein_decision_9_cir.html. 1999 (cited on page 5).

[12] *Narcotrafic : le Sénat autorise les services de renseignement à accéder aux messageries cryptées*. https://www.publicsenat.fr/actualites/parlementaire/narcotrafic-le-senat-autorise-les-services-de-renseignement-a-acceder-aux-messageries-cryptees. 2025 (cited on page 5).

[13] Alma Whitten and J. D. Tygar. 'Why Johnny can't encrypt: a usability evaluation of PGP 5.0'. In: *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999, p. 14 (cited on page 5).

[14] Nikita Borisov, Ian Goldberg, and Eric Brewer. 'Off-the-record communication, or, why not to use PGP'. In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*. WPES '04. Washington DC, USA: Association for Computing Machinery, 2004, pp. 77–84. DOI: 10.1145/1029179.1029200 (cited on page 6).

[15] Tarun Kumar Yadav, Devashish Gosain, and Kent Seamons. 'Cryptographic Deniability: A Multi-perspective Study of User Perceptions and Expectations'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3637–3654 (cited on page 6).

[16] Moxie Marlinspike. *Forward Secrecy for Asynchronous Messages*. https://signal.org/blog/asynchronous-security/. 2013 (cited on page 7).

[17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 'A Formal Security Analysis of the Signal Messaging Protocol'. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27 (cited on page 7).

[18] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 'The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol'. In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Cham: Springer International Publishing, 2019, pp. 129–158 (cited on page 7).

[19] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. 'A More Complete Analysis of the Signal Double Ratchet Algorithm'. In: *Advances in Cryptology – CRYPTO 2022*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Cham: Springer Nature Switzerland, 2022, pp. 784–813 (cited on page 7).

[20] Moxie Marlinspike. *Private Group Messaging*. https://signal.org/blog/private-groups/. 2014 (cited on pages 7, 104).

[21] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol*. RFC 9420. July 2023. DOI: 10.17487/RFC9420. URL: https://www.rfc-editor.org/info/rfc9420 (cited on pages 7, 20, 52, 74, 76, 98, 103, 104, 106, 109, 129, 132, 135, 155, 156, 159).

[22] C. E. Shannon. 'Communication theory of secrecy systems'. In: *The Bell System Technical Journal* 28.4 (1949), pp. 656–715. DOI: 10.1002/j.1538-7305.1949.tb00928.x (cited on page 8).

[23] Eli Biham and Adi Shamir. *Differential cryptanalysis of the data encryption standard*. Berlin, Heidelberg: Springer-Verlag, 1993 (cited on page 8).

[24] Mihir Bellare and Phillip Rogaway. 'Random oracles are practical: a paradigm for designing efficient protocols'. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 62–73. DOI: 10.1145/168588.168596 (cited on page 8).

[25] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica Vol. I*. Cambridge University Press, 1910 (cited on page 10).

[26] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 'Social processes and proofs of theorems and programs'. In: *Commun. ACM* 22.5 (May 1979), pp. 271–280. DOI: 10.1145/359104.359106 (cited on page 10).

[27] Kevin Buzzard. *Formalizing 21st century mathematics in Lean*. https://68nqrt.inria.fr/Slides/2021/KevinBuzzard.pdf. 2021 (cited on page 11).

[28] Leonardo de Moura and Sebastian Ullrich. 'The Lean 4 Theorem Prover and Programming Language'. In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635 (cited on page 11).

[29] The mathlib Community. 'The lean mathematical library'. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. DOI: 10.1145/3372885.3373824 (cited on page 11).

[30] Terence Tao. *A Maclaurin type inequality*. 2023. URL: https://arxiv.org/abs/2310.05328 (cited on page 11).

[31] Terence Tao. https://mathstodon.xyz/@tao/111287749336059662. 2023 (cited on page 11).

[32] Kevin Buzzard. *Fermat's Last Theorem — how it's going*. https://xenaproject.wordpress.com/2024/12/11/fermats-last-theorem-how-its-going/. 2024 (cited on page 11).

[33] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*. Cryptology ePrint Archive, Paper 2004/331. 2004. URL: https://eprint.iacr.org/2004/331 (cited on page 11).

[34] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Paper 2004/332. 2004. URL: https://eprint.iacr.org/2004/332 (cited on page 11).

[35] Shai Halevi. *A plausible approach to computer-aided cryptographic proofs*. Cryptology ePrint Archive, Paper 2005/181. 2005. URL: https://eprint.iacr.org/2005/181 (cited on page 11).

[36] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 'SoK: Computer-Aided Cryptography'. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021, pp. 777–795 (cited on pages 12, 77, 127, 151, 158).

[37] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 'Computer-Aided Security Proofs for the Working Cryptographer'. In: *Advances in Cryptology – CRYPTO*. Ed. by Phillip Rogaway. 2011, pp. 71–90 (cited on pages 12, 77, 127, 158).

[38] Bruno Blanchet. 'CryptoVerif: Computationally sound mechanized prover for cryptographic protocols'. In: *Dagstuhl seminar "Formal Protocol Verification Applied*. Vol. 117. 2007, p. 156 (cited on pages 12, 77, 127, 158).

[39] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. 'An interactive prover for protocol verification in the computational model'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2021, pp. 537–554 (cited on pages 12, 127, 158).

[40] Gergei Bana and Hubert Comon-Lundh. 'Towards Unconditional Soundness: Computationally Complete Symbolic Attacker'. In: *Principles of Security and Trust*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 189–208 (cited on pages 12, 158).

[41] Bruno Blanchet et al. 'Modeling and verifying security protocols with the applied pi calculus and ProVerif'. In: *Foundations and Trends® in Privacy and Security* 1.1-2 (2016), pp. 1–135 (cited on pages 12, 18, 19, 33, 39, 40, 44, 47, 77, 120, 127, 158, 161).

[42] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 'The TAMARIN prover for the symbolic analysis of security protocols'. In: *International conference on computer aided verification*. Springer. 2013, pp. 696–701 (cited on pages 12, 18, 19, 33, 39, 40, 44, 47, 77, 120, 127, 158, 161).

[43] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 'DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code'. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 523–542 (cited on pages 12, 15, 18, 20, 40, 56–58, 61, 68–70, 77, 96, 105, 120, 121, 127, 142, 156, 157).

[44] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. RFC 7457. Feb. 2015. DOI: 10.17487/RFC7457. URL: https://www.rfc-editor.org/info/rfc7457 (cited on page 13).

[45] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. 'On the Security of the TLS Protocol: A Systematic Analysis'. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–448 (cited on page 13).

[46] 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: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 98–113. DOI: 10.1109/SP.2014.14 (cited on page 13).

[47] Kenneth G. Paterson and Thyla van der Merwe. 'Reactive and Proactive Standardisation of TLS'. In: *Security Standardisation Research*. Ed. by Lidong Chen, David McGrew, and Chris Mitchell. Cham: Springer International Publishing, 2016, pp. 160–186 (cited on page 13).

[48] 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: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 470–485. DOI: 10.1109/SP.2016.35 (cited on page 13).

[49] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 'Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate'. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 483–502 (cited on pages 13, 93, 127).

[50] 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*. 2017, pp. 1773–1788 (cited on pages 13, 93, 127).

[51] Joël Alwen, Daniel Jost, and Marta Mularczyk. 'On The Insider Security of MLS'. In: *Advances in Cryptology – CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part II*. Santa Barbara, CA, USA: Springer-Verlag, 2022, pp. 34–68. DOI: 10.1007/978-3-031-15979-4_2 (cited on pages 14, 104, 109, 113, 119, 122, 126, 129, 131, 146, 151, 152, 156).

[52] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. 'Security Analysis of the MLS Key Derivation'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 2535–2553 (cited on pages 14, 104, 108, 109, 126, 129, 131, 156).

[53] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. https://signal.org/docs/specifications/doubleratchet/. 2016 (cited on pages 14, 103, 157).

[54] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno 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)*. IEEE. 2017, pp. 435–450 (cited on pages 14, 127, 157).

[55] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. *The Messaging Layer Security (MLS) Architecture*. RFC 9750. Apr. 2025. DOI: 10.17487/RFC9750. URL: https://www.rfc-editor.org/info/rfc9750 (cited on pages 15, 104, 132).

[56] Danny Dolev and Andrew Chi-Chih Yao. 'On the security of public key protocols'. In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207 (cited on pages 17, 18, 96, 143).

[57] Roger M. Needham and Michael D. Schroeder. 'Using Encryption for Authentication in Large Networks of Computers'. In: *Communications of the ACM* 21.12 (1978), pp. 993–999 (cited on pages 17, 75).

[58] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 'Refinement Types for Secure Implementations'. In: *2008 21st IEEE Computer Security Foundations Symposium*. 2008, pp. 17–32. DOI: 10.1109/CSF.2008.27 (cited on page 19).

[59] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 'Modular verification of security protocol code by typing'. In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 445–456. DOI: 10.1145/1707801.1706350 (cited on page 19).

[60] Stephen C. Kleene. *Introduction to Metamathematics*. Princeton, New Jersey: D. van Nostrand, 1952 (cited on page 27).

[61] Mike Rosulek. *The Joy of Cryptography*. https://joyofcryptography.com. 2021 (cited on page 33).

[62] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. 'A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems'. In: *J. Autom. Reason.* 46.3–4 (Apr. 2011), pp. 225–259. DOI: 10.1007/s10817-010-9187-9 (cited on page 40).

[63] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. 'A Generic Methodology for the Modular Verification of Security Protocol Implementations'. In: *Computer and Communications Security (CCS)*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1377–1391. DOI: 10.1145/3576915.3623105 (cited on pages 41, 158).

[64] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer International Publishing, 2021, pp. 367–379 (cited on pages 41, 158).

[65] *Protocol Proof Ladder*. https://github.com/proof-ladders/protocol-ladder. 2025 (cited on pages 41, 42).

[66] Karthikeyan Bhargavan, Abhishek Bichhawat, Pedram Hosseyni, Ralf Küsters, Klaas Pruiksma, Guido Schmitz, Clara Waldmann, and Tim Würtele. 'Layered Symbolic Security Analysis in DY*'. In: *Computer Security – ESORICS 2023*. Ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis. Cham: Springer Nature Switzerland, 2024, pp. 3–21 (cited on page 49).

[67] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 'Comparse: Provably Secure Formats for Cryptographic Protocols'. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 564–578. DOI: 10.1145/3576915.3623201 (cited on pages 50, 75, 160).

[68] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. 'TreeSync: Authenticated Group Management for Messaging Layer Security'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Aug. 2023, pp. 1217–1233 (cited on pages 50, 76, 96, 98, 101, 103, 129, 131, 139, 143, 146, 150, 159).

[69] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A Wood. *RFC 9180: Hybrid public key encryption*. Tech. rep. Internet Research Task Force, 2022 (cited on pages 52, 104, 135).

[70] Thierry Coquand and Christine Paulin. 'Inductively defined types'. In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 50–66 (cited on page 59).

[71] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 'HACL*: A verified modern cryptographic library'. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017, pp. 1789–1806 (cited on pages 69, 123).

[72] Gavin Lowe. 'Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 1996, pp. 147–166 (cited on page 76).

[73] Catherine A. Meadows. 'Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches'. In: *Computer Security — ESORICS*. Springer Berlin Heidelberg, 1996, pp. 351–364 (cited on page 76).

[74] James Heather, Gavin Lowe, and Steve A. Schneider. 'How to Prevent Type Flaw Attacks on Security Protocols'. In: *Journal of Computer Security* 11.2 (2003), pp. 217–244 (cited on page 76).

[75] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446 (cited on pages 76, 84, 87, 89–91).

[76] S. Cantor, J. Kemp, R. Philpott, and E. Maler. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0*. 2005 (cited on page 76).

[77] *XML Signature Syntax and Processing Version 2.0*. W3C Recommendation. July 2015 (cited on page 76).

[78] *XML Encryption Syntax and Processing Version 1.1*. W3C Recommendation. Apr. 2013 (cited on page 76).

[79] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS)*. IETF RFC 7515. May 2015 (cited on page 76).

[80] Michael B. Jones and Joe Hildebrand. *JSON Web Encryption (JWE)*. IETF RFC 7516. May 2015 (cited on page 76).

[81] J. Schaad and August Cellars. *CBOR Object Signing and Encryption (COSE)*. IETF RFC 8152. July 2017 (cited on page 76).

[82] J. Schaad and August Cellars. *Protocol Buffers (proto 3)*. https://protobuf.dev. July 2008 (cited on page 76).

[83] ITU-T. *Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks*. Recommendation ITU-T X.509. Oct. 2019 (cited on page 76).

[84] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. 'A Cross-Protocol Attack on the TLS Protocol'. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 62–72. DOI: 10.1145/2382196.2382206 (cited on pages 76, 81, 82, 92).

[85] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. 'Three Lessons From Threema: Analysis of a Secure Messenger'. In: *Proceedings of the 32th USENIX Conference on Security Symposium*. SEC'23. USA: USENIX Association, 2023 (cited on pages 76, 83).

[86] Martin R. Albrecht, Sofia Celi, Benjamin Dowling, and Daniel Jones. *Practically-exploitable Cryptographic Vulnerabilities in Matrix*. Cryptology ePrint Archive, Paper 2023/485. https://eprint.iacr.org/2023/485. 2023. URL: https://eprint.iacr.org/2023/485 (cited on page 76).

[87] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 'Everparse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats'. In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, pp. 1465–1482 (cited on pages 80, 93, 97, 99, 100).

[88] Wikipedia contributors. *Malleability (cryptography) — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-May-2023]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Malleability_(cryptography)&oldid=1083763968 (cited on page 80).

[89] Pieter Wuille. *Dealing with malleability*. BIP 62. 2014 (cited on page 80).

[90] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 'Meta-F* : Proof Automation with SMT, Tactics, and Metaprograms'. In: *Programming Languages and Systems - European Symposium on Programming, ESOP*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 30–59. DOI: 10.1007/978-3-030-17184-1_2 (cited on page 90).

[91] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246 (cited on pages 91, 92).

[92] E. Rescorla, R. Barnes, H. Tschofenig, and B. Schwartz. *Compact TLS 1.3*. IETF Internet Draft version 8. Mar. 2023. URL: %5Curl%7Bhttps://www.ietf.org/archive/id/draft-ietf-tls-ctls-08.html%7D (cited on page 91).

[93] Yoav Nir, Simon Josefsson, and Manuel Pégourié-Gonnard. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*. RFC 8422. Aug. 2018. DOI: 10.17487/RFC8422. URL: https://www.rfc-editor.org/info/rfc8422 (cited on page 92).

[94] 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: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 5–17 (cited on page 92).

[95] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. 'A Cryptographic Analysis of the TLS 1.3 Handshake Protocol'. In: *J. Cryptol.* 34.4 (2021), p. 37 (cited on page 93).

[96] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. '(De-)Constructing TLS 1.3'. In: *Progress in Cryptology – INDOCRYPT 2015*. 2015, pp. 85–102 (cited on page 93).

[97] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. 'Multiple Handshakes Security of TLS 1.3 Candidates'. In: *IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 486–505 (cited on page 93).

[98] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 'Dependent Types and Multi-Monadic Effects in F*'. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2016, pp. 256–270 (cited on pages 93, 96, 105, 109, 131, 139, 142).

[99] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 'An In-Depth Symbolic Security Analysis of the ACME Standard'. In: *ACM SIGSAC Conference on Computer and Communications (CCS)*. ACM, 2021, pp. 2601–2617 (cited on pages 96, 121).

[100] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 'Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 107–124 (cited on pages 96, 121, 127, 157).

[101] *Comparse: Supplementary Material*. https://github.com/Inria-Prosecco/comparse-artifact. 2023 (cited on page 97).

[102] Kenton Varda. 'Protocol buffers: Google's data interchange format'. In: *Google Open Source Blog, Available at least as early as Jul* 72 (2008), p. 23 (cited on page 99).

[103] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 'Hardening Attack Surfaces with Formally Proven Binary Format Parsers'. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 31–45. DOI: 10.1145/3519939.3523708 (cited on pages 99, 100).

[104] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 'Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats'. In: *Proc. ACM Program. Lang.* 3.ICFP (2019). DOI: 10.1145/3341686 (cited on pages 99, 100).

[105] Marcell van Geest and Wouter Swierstra. 'Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data'. In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 30–40. DOI: 10.1145/3122975.3122979 (cited on pages 99, 100).

[106] Sebastian Mödersheim and Georgios Katsoris. 'A Sound Abstraction of the Parsing Problem'. In: *2014 IEEE 27th Computer Security Foundations Symposium*. 2014, pp. 259–273. DOI: 10.1109/CSF.2014.26 (cited on pages 99, 100).

[107] *Cheerios*. https://github.com/uwplse/cheerios. 2016 (cited on pages 99, 100).

[108] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 'Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System'. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 413–429 (cited on pages 99, 100).

[109] Qianchuan Ye and Benjamin Delaware. 'A Verified Protocol Buffer Compiler'. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 222–233. DOI: 10.1145/3293880.3294105 (cited on pages 99, 100).

[110] Moxie Marlinspike and Trevor Perrin. *Signal Specifications*. https://signal.org/docs. 2016 (cited on pages 103, 129).

[111] Melissa Chase, Trevor Perrin, and Greg Zaverucha. 'The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption'. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020, pp. 1445–1459 (cited on pages 104, 126).

[112] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 'On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees'. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018, pp. 1802–1819 (cited on pages 104, 126, 132).

[113] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS)*. Research Report. Inria Paris, May 2018 (cited on pages 104, 126, 131, 132).

[114] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 'Security Analysis and Improvements for the IETF MLS Standard for Group Messaging'. In: *CRYPTO*. Vol. 12170. Lecture Notes in Computer Science. Springer, 2020, pp. 248–277 (cited on pages 104, 108, 109, 126, 129, 131, 151, 155).

[115] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 'Modular Design of Secure Group Messaging Protocols and the Security of MLS'. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021, pp. 1463–1483 (cited on pages 104, 108, 109, 126, 129, 131, 151, 155).

[116] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS*. Research Report. Inria Paris, Dec. 2019 (cited on pages 104, 109, 113, 122, 127, 129, 131, 146, 156).

[117] *TreeSync: Supplementary Material*. https://github.com/Inria-Prosecco/treesync. 2022 (cited on pages 105, 110, 112, 115).

[118] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. 'Collective Information Security in Large-Scale Urban Protests: the Case of Hong Kong'. In: *USENIX Security Symposium*. USENIX Association, Aug. 2021, pp. 3363–3380 (cited on page 106).

[119] Moxie Marlinspike. *Disappearing messages for Signal*. https://signal.org/blog/disappearing-messages/. 2016 (cited on page 106).

[120] Richard Barnes. *Remove without double-join (in TreeKEM)*. https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik. 2018 (cited on page 109).

[121] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cited on page 111).

[122] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. 'Evercrypt: A fast, verified, cross-platform cryptographic provider'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2020, pp. 983–1002 (cited on pages 118, 123).

[123] Forrest Voight. *CVE-2012-2459 (block merkle calculation exploit)*. https://bitcointalk.org/?topic=102395. Aug. 2012 (cited on page 118).

[124] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. 'Haclxn: Verified generic SIMD crypto (for all your favourite platforms)'. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020, pp. 899–918 (cited on page 123).

[125] Jérôme Vouillon and Vincent Balat. 'From bytecode to JavaScript: the Js_of_ocaml compiler'. In: *Software: Practice and Experience* 44.8 (2014), pp. 951–972 (cited on page 124).

[126] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 'Formally verified cryptographic web applications in webassembly'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2019, pp. 1256–1274 (cited on page 124).

[127] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 'Bringing the web up to speed with WebAssembly'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017, pp. 185–200 (cited on page 124).

[128] Mark Manulis. *Security-Focused Survey on Group Key Exchange Protocols*. Cryptology ePrint Archive, Paper 2006/395. `https://eprint.iacr.org/2006/395`. 2006. URL: `https://eprint.iacr.org/2006/395` (cited on pages 126, 156).

[129] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. 'SoK: Game-Based Security Models for Group Key Exchange'. In: *Topics in Cryptology – CT-RSA*. Ed. by Kenneth G. Paterson. Springer International Publishing, 2021, pp. 148–176 (cited on pages 126, 156).

[130] Cas Cremers, Britta Hale, and Konrad Kohbrok. 'The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter'. In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 1847–1864 (cited on pages 126, 131, 155).

[131] Cas Cremers, Charlie Jacomme, and Philip Lukert. ' Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis '. In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, July 2023, pp. 200–213 (cited on pages 127, 129, 131, 156).

[132] Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David Basin. 'Automated Verification of Group Key Agreement Protocols'. In: *2014 IEEE Symposium on Security and Privacy (S&P)*. 2014, pp. 179–194. DOI: `10.1109/SP.2014.19` (cited on page 127).

[133] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 'Formally verified cryptographic web applications in webassembly'. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2019, pp. 1256–1274 (cited on page 127).

[134] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 'Implementing TLS with Verified Cryptographic Security'. In: *IEEE Symposium on Security and Privacy, (S&P)*. 2013, pp. 445–459 (cited on pages 127, 157).

[135] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 'TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security'. In: *2025 IEEE Symposium on Security and Privacy (SP)*. 2025, pp. 4375–4390. DOI: `10.1109/SP61157.2025.00228` (cited on pages 129, 159).

[136] David Balbás, Daniel Collins, and Phillip Gajland. 'WhatsUpp with Sender Keys? Analysis, Improvements and Security Proofs'. In: *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part V*. Springer-Verlag, 2023, pp. 307–341 (cited on page 129).

[137] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. 'Continuous group key agreement with active security'. In: *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*. Springer. 2020, pp. 261–290 (cited on pages 131, 132).

[138] *MLS test vectors*. `https://github.com/mlswg/mls-implementations/blob/main/test-vectors.md` (cited on page 139).

[139] Cas Cremers, Esra Günsay, Vera Wesselkamp, and Mang Zhao. *ETK: External-Operations TreeKEM and the Security of MLS in RFC 9420*. Cryptology ePrint Archive, Paper 2025/229. 2025. URL: `https://eprint.iacr.org/2025/229` (cited on page 156).

[140] Felix Linker, Ralf Sasse, and David Basin. *A Formal Analysis of Apple's iMessage PQ3 Protocol*. Cryptology ePrint Archive, Paper 2024/1395. 2024. URL: `https://eprint.iacr.org/2024/1395` (cited on page 157).

[141] Apple Security Engineering and Architecture. *iMessage with PQ3: The new state of the art in quantum-secure messaging at scale*. `https://security.apple.com/blog/imessage-pq3/`. 2024 (cited on page 157).

[142] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. 'Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging'. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 469–486 (cited on page 157).

[143] Ehren Kret and Rolfe Schmidt. *The PQXDH Key Agreement Protocol*. `https://signal.org/docs/specifications/pqxdh/`. 2023 (cited on page 157).

[144] Cas Cremers, Charlie Jacomme, and Aurora Naska. 'Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations'. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1235–1252 (cited on page 157).

[145] Moxie Marlinspike and Trevor Perrin. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. `https://signal.org/docs/specifications/sesame/`. 2017 (cited on page 157).

[146] Cas Cremers, Niklas Medinger, and Aurora Naska. *Impossibility Results for Post-Compromise Security in Real-World Communication Systems*. Cryptology ePrint Archive, Paper 2024/1886. 2024. URL: `https://eprint.iacr.org/2024/1886` (cited on page 157).

[147] Trevor Perrin. *The Noise Protocol Framework*. `https://noiseprotocol.org/noise.html`. 2018 (cited on page 157).

[148] Jason Donenfeld. 'WireGuard: Next Generation Kernel Network Tunnel'. In: Jan. 2017. DOI: `10.14722/ndss.2017.23160` (cited on page 158).

[149] Bruno Blanchet, Pierre Boutry, Christian Doczkal, Benjamin Grégoire, and Pierre-Yves Strub. 'CV2EC: Getting the Best of Both Worlds'. In: *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. 2024, pp. 279–294. DOI: `10.1109/CSF61375.2024.00019` (cited on page 158).

RÉSUMÉ

Les applications de messagerie sont de nos jours utilisées de façon généralisée pour communiquer, en particulier en utilisant les *conversations de groupe* pour relier les personnes d'un cercle social. C'est une menace potentielle pour la vie privée, par exemple si les serveurs de l'application de messagerie ont accès au contenu des conversations. Pour résoudre ce problème, les applications de messagerie modernes fournissent du *chiffrement de bout en bout,* ce qui veut dire que les messages sont chiffrés par l'appareil de l'expéditeur et déchiffrés par l'appareil du destinataire, de manière à ce que les serveurs de messagerie ne puissent pas connaître le contenu des messages. Un tel chiffrement de bout en bout est de façon plus générale un protocole cryptographique, dont la conception est notoirement sujette aux erreurs. Cela soulève la question suivante: *les applications de messagerie sécurisée sont-elles réellement sécurisées ?* Dans cette thèse, nous développons une nouvelle méthodologie permettant d'analyser le protocole de messagerie de groupe Messaging Layer Security (MLS) en utilisant les méthodes formelles sur des spécifications précises à l'octet près dans le modèle symbolique et, ultimement, nous aidons à corriger des problèmes de conception dans MLS avant sa standardisation.

MOTS CLÉS

vérification formelle, messagerie sécurisée, cryptographie

ABSTRACT

Messaging applications are nowadays pervasively used to communicate with each other, in particular using *group conversations* to connect people within a social circle. This is a potential threat for privacy, for example if the messaging application servers were to have access to the conversation content. To address this issue, modern messaging applications provide *end-to-end encryption,* meaning that messages are encrypted by the sender device and decrypted by the receiver device, so that their content stays hidden from the messaging application servers. Such end-to-end encryption is a feature of cryptographic protocols, whose design is notoriously error-prone. This begs the following question: *are secure messaging applications actually secure?* In this thesis, we develop a novel methodology to analyze the secure group messaging protocol Messaging Layer Security (MLS) by using formal methods on bit-precise specifications in the symbolic model, and ultimately helped to fix design flaws in MLS before its standardization.

KEYWORDS

formal verification, secure messaging, cryptography