An Analysis of TLS 1.3 and its use in Composite Protocols

Jonathan Hoyland

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
Department of Mathematics
Royal Holloway, University of London

2018

Declaration

These doctoral studies were conducted under the supervision of Dr. Hague.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Jonathan Hoyland
January 7, 2019

Acknowledgements

I would like to thank my supervisor Matthew Hague for his unstinting support and his overwhelming generosity with his time. He let me pick my own path, and but for his patient guidance and excellent advice I would have become lost in the weeds. He motivated me through the challenges of research, and without his constructive criticism my work would have been much poorer.

I must also thank my co-authors, Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Cas for his penetrating insight, sage advice, and for taking time from his extremely busy schedule whilst he moved country to put together a paper at the last minute. Marko for his unwitting Angela Rippon impression, and Sam and Thyla for being an inspiration to actually finish and move on with life. Special thanks as well to Sam and Thyla for inviting me to collaborate with them on their analysis of TLS 1.3, helping me find my direction. I also must thank Sam for his heroic efforts with Tamarin, we wouldn't have made it without you.

Thanks to Royal Holloway and the EPSRC for funding me and supporting this programme.

I would also like to thank Nick Sullivan, Martin Thompson, and numerous other IETF-ers for taking the time to discuss my work, giving practical advice and feedback that helped me improve both my analysis and subsequent proposals.

Thanks also to Kevin Milner for his help in dissecting and explaining Tamarin's internals, and for providing the ideas that enabled me to push my analysis far further than I would otherwise have been able to.

Most of all, I thank my family, for seeing me through this process, keeping me on task and focussed on the end goal. Thanks to my father for telling me there is no such thing as a failed experiment, only a negative result, and for motivating me when my resolve wavered, and to my mother for keeping me company during the long hours of work that somehow ended up in this thesis.

Abstract

The TLS protocol is one of the most important protocols today. This thesis is a study of TLS 1.3 and its use in composite protocols. The TLS protocol is intended to enable secure end-to-end communication over insecure networks, including the Internet. Unfortunately, this goal has been thwarted a number of times throughout the protocol's tumultuous lifetime, resulting in the need for a new version of the protocol, namely TLS 1.3. Over the past four years, in an unprecedented joint design effort with the academic community, the TLS Working Group has been working tirelessly to enhance the security of TLS.

We provide a comprehensive, faithful, and modular symbolic model of TLS 1.3, and use the Tamarin prover to verify the claimed TLS 1.3 requirements. Our analysis reveals a previously unreported unexpected behaviour, which inhibits strong authentication guarantees in some circumstances. In particular, participants cannot always derive their authentication status. We also provide a symbolic model of Exported Authenticators (EAs), a protocol that is layered on top of TLS to create a composite protocol, using the Tamarin prover to verify the claimed requirements. Our analysis requires us to define new authentication properties that allow us to capture the guarantees claimed by EAs. The results of our analysis show the same issue appears in EAs. We thus propose Layered EAs, an extension to EAs that allows participants to derive their authentication status. We provide a symbolic model of LEAs, and use the Tamarin prover to provide a partial proof of the claimed requirements.

We also propose a protocol composition that layers TLS 1.3 on top of a multi-party authentication protocol. This allows us to construct a TLS channel where the key is agreed between multiple parties, whilst preserving authentication and integrity. We compare and contrast this composition with three controversial proposals designed to achieve similar confidentiality guarantees.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security protocols have become ubiquitous in modern day life, with authenticated end-to-end encryption becoming the norm. Whilst the design of protocols has changed dramatically over the last few decades we can trace the use of security protocols back to antiquity. From the use of seals in early Imperial China [Lan12] to prove authenticity[1] to the use of obscure hieroglyphs to encipher proper names in ancient Egypt [Kah74], people have been attempting to secure the written word for thousands of years. With the advent of mechanised, and later digitised cryptology the guarantees required of security protocols began to be formalised and studied with rigour.

Modern security protocols are constructed from a relatively small number of widely studied and well understood security primitives, such as symmetric encryption and hashing. By constructing a protocol from these primitives it is possible to achieve a wide variety of complex security properties.

1.1 Thesis overview

Over the course of this thesis we will discuss and develop three major themes.

1. The use of formal analysis in designing and securing protocols,
2. the construction of composite protocols, and
3. the interaction between the formal analysis community and the standards bodies that deploy protocols.

[1] A practice still common in East Asia today. For example in Japan and Korea legal documents will often require a stamp from a seal registered with the government.

Formal analysis is a set of techniques for analysing security protocols and the properties they achieve. We use these techniques to analyse a protocol called TLS 1.3 and a number of protocols based on it. We extend these techniques and push the boundaries of what it is currently possible to analyse. We extend a number of definitions of authentication properties that allow us to reason about a wider array of protocols, and develop new techniques for constructing new protocols from old protocols.

Of particular focus is the study of composite protocols. Composite protocols are those formed of a number of constituent protocols. We design and analyse a number of composite protocols based around TLS to achieve various complex effects. We make use of a technique called channel binding to bind the constituent protocols together, achieving new security properties. In particular we study compound authentication, an authentication property that describes the authentication relationship between the different layers of a composite protocol and extend the definitions to allow us to analyse a wider range of protocols and properties.

Over the course of this research we collaborated closely with the IETF, the premier internet standards body. The IETF produces and maintains widely followed standards for internet protocols. IETF standards are therefore of great interest to the academic community. Working with the IETF on draft standards allows the academic community to contribute their expertise to protocols that may see wide deployment, finding flaws before they become a problem in the real world.

1.2 Motivation

The Transport Layer Security (TLS) protocol is one of the most widely deployed protocols in the world, used hundreds of billions of times each day.[2] TLS provides a secure channel between a client and a server, and is used to protect everything from financial transactions to cat gifs. The TLS protocol has been refined over the years, with several versions currently deployed. TLS 1.2 is by far the most common version.[3] In 2014 the IETF

[2] `https://www.cloudflare.com/ssl/`

[3] `https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/`

began drafting TLS 1.3, the latest version of TLS. Because of TLS's ubiquity, importance, and complexity the IETF commissioned a number of formal analyses [Cre+16] [Bha+16b] [BBK17] [Cre+17a].

Formal analysis is the process of constructing a model of the protocol in a form that can be reasoned about logically, and analysing its properties. Using model checking software we construct proofs that the model has, or does not have, certain properties. By careful construction of the model it is possible to thereby obtain assurances of the security of the protocol design. The IETF used the results of these analyses to inform the construction of the protocol, undergoing multiple rounds of analysis and redesign.

1.3 Contributions

Our major contributions in this work are as follows.

- We prove that TLS 1.3 meets most of its security goals.
- We develop the literature on compound authentication to prove that EAs meet their security goals.
- We develop layered exported authenticators (LEAs), an extension to EAs with even stronger security goals.
- We provide a partial proof of the security of LEAs.
- We develop MLS with TLS, a composite protocol with complex confidentiality and authentication guarantees.

1.4 The structure of the thesis

In this thesis we first introduce some background material on formal analysis and the IETF's processes in Chapter 2, and proceed on to one of these analyses, namely [Cre+17a], in Chapter 3.

In Chapter 3 we begin by describing TLS 1.3 and how we modelled it. We then describe our analysis and results. Because TLS is used in so many places and for so many different things TLS 1.3 is a very complex protocol. In the course of our analysis we highlight a problem with post-handshake authentication, which extends to the main handshake. We show that a client never knows its authentication status, and thus cannot distinguish between a bilaterally authenticated connection and a unilaterally authenticated connection.

This leads us into Chapter 4, where we analyse `draft-sullivan`. `draft-sullivan` defines a protocol that runs over the top of TLS that is intended to supersede post-handshake authentication. `draft-sullivan` allows either party to add arbitrarily many identities to a TLS channel, as opposed to standard TLS that allows the client and server at most one identity each. This is useful in the case of Content Distribution Networks (CDNs), which represent many websites, each with a different identity. Layering this authentication on top of TLS means that rather than construct a new protocol, it is possible to run one protocol inside another. This leads us to the study of layered authentication protocols.

Layered authentication protocols are a type of composite protocol formed by nesting a number of authentication protocols one inside the other. Composite protocols can have different security properties to those of their constituent protocols. In Chapter 4 we develop the tools for analysing composite authentication protocols, extending the definitions of compound authentication to describe a larger array of protocols. We then use these new definitions to analyse `draft-sullivan`, and prove it has the desired security properties.

`draft-sullivan` however has the same issue as we discovered in TLS 1.3, and further, extends the ambiguity to the server. Because either party may silently reject an authentication attempt, neither party can know if its authentication attempt was successful. To address this issue, and to allow for more complex authentication properties, we propose `draft-hoyland`, which extends `draft-sullivan` to link runs of `draft-sullivan` together.

This linkage has a number of applications, from certificate pinning to session resumption. In Chapter 5 we discuss the design of this binding, and

the reasoning behind it. We then analyse the properties of the proposal, finding that whilst we can prove a partial result about `draft-hoyland`, producing a complete proof is elusive. Given its scope of potential applications and the strength of the partial results, we demonstrate that `draft-hoyland` is worthy of further study, which we leave for future work.

Both Chapter 4 and Chapter 5 examine composite protocols that layer protocols on top of TLS. In Chapter 6 we invert this pattern and study protocols which run TLS on top of a different base layer. This lets us produce very different guarantees.

We were motivated in this work by a contentious discussion around the use of TLS in enterprise environments. A number of enterprises decrypt TLS traffic as it passes over their network to detect attacks and to diagnose issues. In TLS 1.2 this could be achieved by using any of a set of modes that didn't provide a feature called forward secrecy. If a mode provides forward secrecy then a passive observer cannot decrypt the traffic, even if provisioned with the servers long-term keys (LTKs). TLS 1.3 deprecates all modes that do not provide forward secrecy.

The removal of these modes was highly contentious, leading to a number of proposals for achieving visibility into TLS 1.3 connections within a corporate network. Each of these proposals was flawed in a number of ways. In Chapter 6 we describe each of these proposals, and discuss their merits and drawbacks, before proposing a different option. By constructing a composite protocol that uses TLS as an inner layer and a protocol called MLS as its base layer we can construct a protocol that has a complex set guarantees that we claim resolves or militates against all the objectionable parts of earlier drafts. We describe these guarantees formally, and then work through each objection raised in the debates surrounding this issue. Finally, in Chapter 7 we provide some conclusions and final remarks.

Chapter 2

Background

In this chapter we will provide a history of TLS, its goals, and its development. In particular we focus on the design process at the IETF and attacks on the different iterations of the protocol. We discuss the changes in the development process for TLS 1.3, the latest version at the time of writing. We use this as motivation to introduce formal analysis, a key factor in TLS 1.3's development. We then proceed to introduce the various security primitives we work with throughout the thesis. Finally we introduce the Tamarin prover, a tool for carrying out formal analysis of protocols.

2.1 TLS

TLS is a protocol that allows two parties "to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery." [RFC8446, p. 1]. In short TLS is designed to provide a secure channel between a client and server over the internet. This achieved by means of a handshake protocol that negotiates a set of authenticated secret keys that are used to build a secure channel. The Transport Layer Security (TLS) protocol is the *de facto* means for securing communications on the World Wide Web, securing upwards of 70% of connections.[1] Initially released as Secure Sockets Layer (SSL) by Netscape Communications in 1995, the protocol has been subject to a number of version upgrades over the course of its more than 20-year lifespan. Rebranded as TLS when it fell under the

[1] `https://letsencrypt.org/stats/`

auspices of the Internet Engineering Task Force (IETF) in the mid-nineties, the protocol has been incrementally modified and extended.

In the case of TLS 1.2 and below, these modifications have taken place in a largely retroactive fashion; following the announcement of an attack [Ble98] [Vau02] [Moe04] [KPR03] [Can+03] [Bar04] [Bar06], the TLS working group (WG) would either respond by releasing a protocol extension (A Request for Comments (RFC) intended to provide increased functionality and/or security enhancements) or by applying the appropriate "patch" to the next version of the protocol.

One particularly notable attack was the renegotiation attack [SR09]. Discovered by Marsh Ray and Steve Dispensa in 2009, and rediscovered by Martin Rex the renegotiation vulnerability was a major flaw in TLS 1.2 and earlier. The renegotiation flow allowed an attacker to prepend data to a TLS connection. The renegotiation attack was a protocol design flaw. An attacker could cause both parties to complete seemingly secure runs of the protocol. Other attacks mostly relied on side-channels, causing thousands of failed connections, or forcing downgrade of the protocol to older versions with previously known security vulnerabilities. The renegotiation flaw, on the other hand, looked like a secure connection to both the client and the server, they just had different views of the history of the connection.

Attacks of this type are the most challenging to patch, because they are very hard to detect, and require changes to the core of the protocol. Downgrade and side channel attacks can be mitigated against by individual implementations of TLS without affecting the core functionality of the protocol. Downgrade attacks can be mitigated against by removing support for old versions of TLS, at the cost of reduced backwards compatibility. Side channel attacks can be mitigated against by limiting the cipher suites offered, or by implementing fixed time cipher suite mitigations. Core protocol changes need to be agreed at the IETF, a much slower process.

Prior to the announcement of the BEAST [DR11] and CRIME [DR12] attacks of 2011 and 2012, respectively, a reactive strategy was effective, given the frequency with which versions were updated, and the limited number of practical attacks against the protocol.

Post-2011, however, the heightened interest in the protocol, and the resulting flood of increasingly practical attacks against it [AP12] [AP13] [Bha+14] [JSS15a] [DR11] [DR12] [Mav+12] [Avi+16] [MDK14] [Beu+15] [Adr+15] [Man15] [GPV15] [BL16b] [BL16a] made this approach increasingly untenable. As Figure 2.1[2] shows, the rate of attacks dramatically increased after 2011.

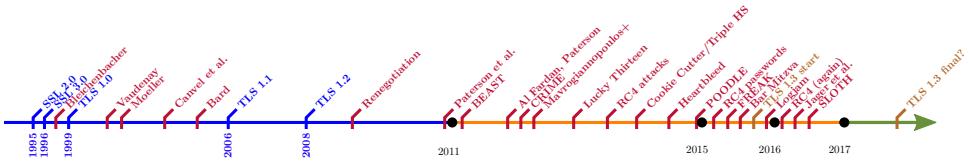


Figure 2.1: A timeline of the development of and attacks on TLS [PM16]

This increase in attacks and pressure to improve the efficiency of the protocol pushed the IETF to begin development of TLS 1.3. In an effort to reduce the likelyhood of serious attacks on the increasing important TLS protocol the IETF decided to take an analysis-before-deployment approach and actively solicited contributions from academia. This approach was successful in finding a number of problems in early versions of the specification draft [BBK17] [Cre+16]. The analyses that were performed were also able to show that the discovered flaws were fixed in later versions.

2.2 The IETF

The Internet Engineering Task Force (IETF) is a standards body that publish internet standards. It is formed of an open group of people who work together to come up with standards that "make the internet work better" [RFC3935]. The IETF publishes some of the most fundamental protocols on the internet, including the hypertext transfer protocol (HTTP), TLS, and the Transmission Control Protocol (TCP). The IETF have an open standards process, which means that anyone can contribute to the development of a standard. The work of the IETF is also transparent, happening on public mailing lists and at open meetings.

[2] Figure 2.1 is replicated from the talk associated with [PM16]. The talk was before TLS 1.3 was completed.

A key factor of the IETF's process is the principle of "rough consensus". Draft standards are brought to working groups at the IETF and, if they choose to accept the work, the drafts are discussed on the mailing list. When a draft standard has achieved so called "rough consensus" in the relevant working group the IETF moves the proposed standard to an RFC. "Rough consensus" means that all technical objections to a proposed standard have been addressed, and the working group is happy to move forward with the publication process. Addressing all technical objections does not mean that all the objections have been resolved, simply that they have been given a fair hearing. This process helps standards advance without becoming bogged down in endless minor objections, but also helps prevent serious issues from being overlooked.

In recent years there has been an increasing trend of looking to academia to assist with the development of high importance protocols, particularly security focussed ones. This trend of analysis-before-deployment has helped develop standards that are hopefully more secure than would have been developed otherwise. TLS 1.3 was heavily analysed in academia before the standard was completed. The IETF also requested a formal analysis of `draft-sullivan` before they advanced the draft through the publication process.

2.3 Formal analysis

Formal analysis is a group of techniques for analysing security protocols. The purpose of formal analyses of security protocols is to give mathematical proofs of their security. This gives a higher level of assurance than, for example extensive testing or fuzzing, because some attacks only appear in the presence of a malicious attacker.

A formal analysis encodes a security protocol into an algebraic form that can be reasoned about in a mathematically rigorous way. There are two main approaches to formal analysis, computational analysis and symbolic analysis. A computational analysis, which expresses the security properties required as an adversarial game, with the attacker as the adversary. A computational analysis proves that an attacker obtains a negligible advantage in winning the game. A symbolic analysis expresses a protocol as a formal grammar in

a process algebra and the security properties as decision problems over the grammar. The symbolic analysis gives a binary answer, saying the grammar satisfies the decision problem or it doesn't. A symbolic analysis is thus coarser, but for that reason is often more tractable.

Historically formal analysis and the requisite tools have not been expressive or powerful enough to analyse protocols as complex as TLS. However as the available computing power has increased and the tool support has improved the range of protocols that can be analysed has increased, making formal analysis a more and more feasible option.

2.3.1 Symbolic analysis

Symbolic analyses of security protocols encode a protocol into a process algebra, and encode the security requirements of the protocol into logical formulae that can be rigorously be shown to be true for a given process algebra.

A process algebra is simply a way of representing a concurrent computational process symbolically, with some fixed rules for manipulating algebraic expressions. These fixed rules represent primitive actions, such as communication or a computational step. Often this representation is a formal grammar that generates a record of protocol actions taken, called a trace. The security requirements are then encoded as decision problems over the grammar.

Symbolic analysis of security protocols was popularised by Burrows et al. [BAN90] through their work on the BAN logic. The BAN logic is described as a logic of authentication. The logic allows formal reasoning about the beliefs of protocol participants, and thus about authentication, although not about secrecy. The BAN logic is decidable [Mon99b], however it has limited expressiveness [Nes90] [GNY90] [BM94]. It has been shown that proving secrecy properties of protocols is undecidable except for in very restricted cases [Mit+99]. However many tools are effective in practice, in particular we use the Tamarin Prover tool to prove results about TLS.

To capture a wider range of protocols the protocols are simplified into a protocol model. In symbolic analysis this modelling process makes certain

assumptions, the major one being that cryptographic primitives are perfect, also called the perfect cryptography assumption. This is related to the Dolev-Yao attacker. Unlike in computational analysis, symbolic models give the attacker a fixed set of abilities it can use to attack the protocol. The Dolev-Yao attacker [DY83] is an attacker that can intercept, drop, modify, and send messages. This is also expressed as "the attacker controls the network". The attacker is not, however, given the ability to break cryptographic primitives. This means that if an attacker does not know a cryptographic key, for example, then it cannot derive any information about information encrypted with that key. In practice it is often possible to derive some information about an encrypted message without being able to decrypt it, for example through side-channel analysis. Symbolic analysis however, treats this type of attack as out of scope.

To improve the tractability of the decision problems other simplifications can be made by the modeller, such as limiting the number of nonces [BAN90] [Low96] [GL97], the depth of messages [Mea96] [Son99] [PS00] [BMV05], or the number of sessions or participants [Mil95] [Low96] [MSM97] [CJM98] [Mit98] [Mon99a] [BLR00] [RS03] . These techniques, however will sometimes fail to detect attacks, and as such are avoided. Some techniques, rather than restricting the protocol, over-approximate the protocol [Mea96] [Mon99b] [Gou00] [GK00] [HL01]. For example the fault-preserving simplifications proposed by Hui and Lowe [HL01] provide a number of transformations that simplify a protocol. These simplifications are designed such that if there is a fault in the original protocol, then the same fault exists in the simplified protocol, although the inverse is not true. This makes the results of such an analysis one-sided, i.e. if a flaw exists in the simplified model, it is not necessarily the case that it exists in the original. If, however, it is possible to prove a simplified protocol secure, then it proves that the original protocol is also secure.[3]

[3] If a flaw is found in the abstract model that doesn't exist in the full model, then we can increase the model's fidelity to remove the flaw and repeat our analysis. We can repeat this process until we find either a flaw in the protocol, or our model is proven to be secure. This is called a counter-example guided abstraction-refinement (CEGAR) cycle.

By using this type of approach a modeller can prove highly complex properties of sophisticated models. Furthermore, as compute power and tool support improves the fidelity of models goes up, as does the complexity of the properties that can be proven.

Tamarin, the protocol verification tool used throughout this thesis, takes a symbolic approach. A protocol is expressed as a multi-set rewriting system, and the security properties are expressed using a fragment of first order logic. The Tamarin prover then evaluates the logical formulae over the multi-set rewriting system, and if no attacks are found we achieve a high level of assurance that the protocol meets its security objectives.

2.3.2 Computational analysis

Computational analysis takes a finer grained, probabilistic approach. Computational analysis was introduced in the 1980's [SM84], and considers messages as bitstrings, rather than as symbols, which gives the model a much higher level of fidelity.

The protocol is expressed as an adversarial game in which the goal is to break one of the security guarantees, the computational approach calculates the attacker's advantage in winning the game. The game is expressed as a task that the adversary must complete, such as distinguishing between an encrypted message and a random value of the same length. Advantage, in this case, would be defined as the chance that it correctly selects the message less the chance that it incorrectly selects the random value. If the attackers advantage can be shown to be negligible, then the protocol is considered secure. A more complete description of computational analysis can be found in Katz et al. [KL07].

The advantage of computational analysis over symbolic analysis is that the adversary is defined as a probabilistic polynomial-time Turing machine, i.e. the adversary can perform any attack that can be performed efficiently, including making a limited number of guesses at things like keys. This differs from symbolic analyses where the attacker is given a fixed set of actions it may perform. The computational approach has a more powerful threat model.

To compute the attackers advantage the protocol game undergoes a series of transformations, until the game can be solved by appealing to standard cryptography assumptions. For example, the decisional Diffie–Hellman (DH) problem is considered hard. By assuming that the attacker can only obtain negligible advantage in solving the decisional DH problem, and that the game can soundly be transformed into a game which solves the decisional DH problem in a number of steps, we can compute the attackers advantage as the sum of the advantage accrued at each step, plus its advantage in solving the decisional DH problem. If the number of steps is small, and the advantage accrued at each step negligible, the attackers total advantage is also negligible.

Computational analyses are good at proving confidentiality properties, but proving authentication properties is more complex. Computational analyses define authentication in terms of matching conversations[4] [BR93] or session identifiers[5] [AFP05], which require that both parties saw the same set of exchanged messages, bar some negligible probability. The tool support for computational analysis is also much more limited, and thus proofs are often performed by hand, an error prone process.

2.3.3 Symbolic vs computational analysis

Symbolic analysis gives a coarser analysis than computational analysis, however the tool support for symbolic analysis is better, and analysis of complex authentication properties, a major part of this thesis, are easier under the symbolic model, and thus we use the symbolic model throughout this thesis.

2.4 Security primitives

In this section we will discuss the security primitives we use. Because throughout this thesis we take a symbolic approach to security primitives we do not address how these primitives are implemented, but merely describe the security properties we define them to have.

[4] Also called matching sessions or matching histories.
[5] Also called partnering.

2.4.1 Nonces

A nonce, or number used once, is a value chosen by a protocol participant at the start of a protocol. The value is included in messages sent by the protocol participant. If the participant sees the value in the response it can be sure that the response is fresh, i.e. that an old session has not been replayed. Depending on the context in which the nonce is used the nonce has different requirements. We assume that a nonce is fresh, i.e. that it has never been used before by any party. In practice nonces are selected from a large space, usually at least 64 bits. This makes collisions very unlikely, but not impossible. In most cases a nonce is also assumed to be unpredictable. Where this is not the case we will note this in the text.

2.4.2 Symmetric encryption

Symmetric encryption defines a function from messages to ciphertexts. Messages are also sometimes referred to as plaintexts. In practice this is a mapping a string of bits on to a string of bits. Symmetric encryption defines two functions, an encrypt function, E, and a decrypt function, D.

$E :: K \to (M \to C)$, takes two parameters, a key and a message, and outputs a ciphertext. $D :: K \to (C \to M)$, also takes two parameters, a key and a ciphertext, and outputs a plaintext. Every key defines a different relation between plaintexts and ciphertexts. We write E_k and D_k to notate E and D parametrised by k respectively. We define the relationship between E and D as follows.

$$D_k(E_k(m)) = m$$

A key used for symmetric encryption is called a symmetric key.

In practice symmetric encryption is defined by breaking a message into fixed length blocks and applying a blockwise cipher such as AES to each block, with some relationship between the blocks defined by a mode, such as Galois/Counter Mode (GCM). We assume the security of this construction as part of perfect cryptography, however various modes have well known attacks. We define such attacks as out of scope. Because we take a symbolic

approach, rather than define these functions, we simply assume encryption has the following properties.

- It is impossible to derive any information about m or k from $E_k(m)$ without knowing k, and if one knows k, then one can learn the entire plaintext.
- It is impossible to produce a ciphertext for a given message m without knowing k.
- There are no collisions in the ciphertext space. This means that we ignore the possibility that $E_k(m) = E_{k'}(m')$. This stems from our "perfect cryptography" approach.

A common restriction that we *do not* require is that E be non-strict. A non-strict function is one that if called twice on the same inputs, will give different outputs. For example this can be achieved by injecting randomness into the encryption process. We do not require E to be non-strict. This allows an attacker to compare two ciphertexts and decide if they are both encryptions of the same message under the same key.

Because we take a symbolic approach we represent ciphertexts as calls to E, rather than as elements in the ciphertext space. We sometimes use the notation $\{m\}_k$ as syntactic sugar for $E_k(m)$.

2.4.3 Asymmetric encryption

Asymmetric encryption operates in a similar manner to symmetric encryption, in that it maps pairs of messages and keys to cipher texts, however the decryption step is different. As the name implies, asymmetric encryption uses different keys for decryption and encryption. Instead of keys, we refer to key pairs, $(pk(sk), sk)$. A key pair has a public part, $pk(sk)$ and a private part, sk. A keypair used for asymmetric encryption is called an asymmetric keypair.

The encrypt and decrypt functions have the same types as in symmetric encryption, but the relationship between them is different. We define the relationship between E and D as follows.

$$D_{sk}(E_{pk(sk)}(m)) = m$$

This means that someone knowing $pk(sk)$, i.e. the public part of the key pair, henceforth the public key, can produce a ciphertext for any message, but only someone knowing sk, i.e. the secret part of the key pair, henceforth the secret or private key, can decrypt the ciphertext. We also place a further restriction on the relationship between E and D.

$$D_{pk(sk)}(E_{sk}(m)) = m$$

This means that someone knowing the secret key can produce a ciphertext that anyone can decipher, but only someone knowing the secret key can produce the ciphertext. This is called a signing operation, and it proves the author of the ciphertext knows the secret key. A ciphertext produced in this way is called a signature. A message with its signature appended is called a signed message.

From a protocol design point-of-view, when using asymmetric key pairs it is important to only use a given pair for signing or for encryption. If the same key is used for both operations it creates the risk that an attacker can use the signing mechanism as a decryption oracle. If the attacker has a message encrypted with a public key, and does not have the secret key to decrypt it, the attacker can ask the key owner to sign the ciphertext. If the key owner uses the same key for encryption and signing, the signature produced would be the plaintext of the message.[6]

We assume the following properties of asymmetric encryption.

- It is impossible to derive any information about the private key from the public key.
- The mapping from public to private keys is a bijection, i.e. there are no public keys with two corresponding private keys, and no private keys for which there are two corresponding public keys.
- It is impossible to derive any information about the plaintext from the ciphertext unless one knows the decryption key.

[6] Careful readers will note that this actually computes $E_{sk}(E_{pk(sk)}(m))$. In many asymmetric cryptosystems the E and D operations are the same, for example both may be modular exponentiation, with the only difference being intent.

2.4.4 Certificates

A certificate is a document proving the authenticity of something. In our context, a certificate is used to tie an identity to a public key. A certificate will have (1) an identity, (2) a public key, and (3) a signature (over the identity and the public key). If Alice receives a certificate that contains the identity "Bob", a public key $pk(sk_{bob})$, and it is signed with the private key of someone she trusts, then she can assume that "Bob", and only "Bob" knows the secret key sk_{bob}. We say that an actor *owns* a certificate if it knows the secret portion of the public key in the certificate.

2.4.5 Public key infrastructure (PKI)

The Public key infrastructure (PKI) is a system for issuing such certificates. A small number of certificate authorities issue certificates that have very wide acceptance, i.e. they are trusted by the majority of actors. These certificate authorities are considered roots of trust, or root certificate authorities. Some certificate authorities delegate their ability to sign certificates to other organisations by providing them with a secondary signing certificate. These organisations in turn delegate that ability. Certificates created by non-root certificate authorities need to provide a chain of certificates leading back to a root certificate. We discuss this more in Section 2.5.2.

In our work we mostly elide this, and assume that all parties agree on a certificate authority, and that the certificate authority is infallible. We consider the complexities of the public key infrastructure (PKI) out of scope.

2.4.6 Diffie–Hellman exchange (DHE)

The Diffie–Hellman exchange (DHE) is a method by which two parties with no shared secrets can establish a shared secret over a hostile network. There are two variants of the DHE that are used in the protocols in this work, finite field and elliptic curve, but when represented symbolically they operate in the same way. A DHE is an asymmetric key agreement protocol.

For finite field DH, a set of initial parameters is agreed, including a prime, p, and a generator for p, g.[7] A generator is a value such that for all positive integers n less than p there exists an integer a such that $n = g^a \pmod{p}$.

$$\forall n : n \neq 0 \pmod{p} \rightarrow \exists a : n = g^a \pmod{p}$$

If Alice and Bob wish to perform a DHE they each select a secret value, a and b respectively. These secret values are the private portion of the DH key pairs. Then Alice computes g^a which she sends to Bob. Bob computes g^b which he sends to Alice. g^a and g^b are the public portions of the DH key pairs. Alice can then compute $(g^b)^a$, and Bob can compute $(g^a)^b$. By the principle of associativity of multiplication we get:

$$(g^a)^b = g^{ab} = (g^b)^a \pmod{p}$$

Alice and Bob now share a value g^{ab}. The computational DH assumption says that given (g, g^a, g^b) it is computationally infeasible to compute g^{ab}. A passive observer therefore, does not know g^{ab}.

An active attacker however, can perform a man-in-the-middle (MITM) attack, pretending to Alice that it is Bob, and to Bob that it is Alice. To prevent this a DHE is usually run with an authentication protocol.

On a symbolic level the process works the same way for elliptic curve DHE, but in practice the elliptic curve version is much more computationally efficient.[8]

2.4.7 Long-term keys (LTKs) vs ephemeral keys

As we have discussed, keys can be classified as symmetric or asymmetric. However keys can also be classified as long-term keys (LTKs) or ephemeral keys. LTKs are keys used repeatedly across many sessions, and are expected to remain secure for a long time. Ephemeral keys, also known as session keys, are used only within a single session, and are generated anew with each protocol run.

[7] The term finite field refers to the fact that all operations are performed over $\mathbb{Z}_p$, i.e. are computed over the natural numbers modulo p, forming a finite field.

[8] An elliptic curve is a curve described by an equation of the form $y^2 = x^3 + ax + b$. The integer solutions of such a curve have special properties that make them useful in this scenario.

2.4.8 Hashing

A hash is a function that maps arbitrary length inputs onto a fixed length output. We assume that a hash function has the following properties.

Property 2.4.1. Pre-image resistance. Given the output of a hash function $h(x)$ it is infeasible to find an input x' such that $h(x') = h(x)$.

This means that an attacker learns nothing about x from $h(x)$, similar to the property of symmetric and asymmetric encryption, but also, the attacker learns nothing about other potentially colliding values. This can be thought of as a non-invertibility property, i.e. it is infeasible to find the input of the hash function given its output.

Property 2.4.2. Second pre-image resistance. Given a value x, it is infeasible to find a value $x' \neq x$ such that $h(x) = h(x')$.

This means that an attacker learns nothing about other potential values, x', from x. Because we assume perfect cryptography we assume these tasks are impossible.

Finally we require a collision resistance property.

Property 2.4.3. Collision resistance. For a given hash function h, it is infeasible to find two values $x \neq x'$ such that $h(x) = h(x')$.

The second and third properties are similar, with the key difference being that the attacker is allowed to choose both x and x' if the hash function has collision resistance.[9]

[9] For a more formal treatment of the definition of the properties of hash functions we refer the reader to Rogaway et al. [RS04].

We also require an independence property, i.e. given $y = h(x)$ the attacker learns nothing about $y' = h(f(x))$, for any efficiently computable function f where $f(x) \neq x$. When hashing a concatenation of values we notate $h(a +\!+ b +\!+ c)$ as $h(a, b, c)$

2.4.9 Message authentication codes (MACs)

A message authentication code (MAC) tag is a short value that is appended to the end of a message. This value is created by taking the message and a symmetric key, and combining them to create a value that proves that the tag was created by someone who knows the key. The tag can be verified by anyone who knows the key, but cannot be distinguished from random by anyone not in possession of the key. This provides a proof that the tag was created by someone who knew the message and the key.[10] Usually this is used to provide authentication and integrity of a message. An attacker should not be able to create a MAC tag unless it knows the key and the message.

2.4.10 Hash-based message authentication codes (HMACs)

One common method for implementing a MAC is using an hash-based message authentication code (HMAC). HMACs were initially proposed in work by Tsudik [Tsu92] because MACs based on block ciphers were slow in software and subject to U.S. export restrictions. By hashing the key concatenated with the message we produce a hash that can be computed by anyone knowing the key, but cannot be computed without knowing the key. We notate this HMAC(k, m), which is equivalent to $h(k, m)$. We use the HMAC notation to indicate intent.

2.4.11 Labels

Using MACs and fixed strings we can prevent the transplantation of messages from one context to another. By including a fixed string in the input to a MAC, e.g. $y = \text{HMAC}(k, \text{“context 1”}, m)$, an attacker who only knows

[10] This is related to the definition of signatures.

y cannot derive anything about HMAC(k, "context 2", m).[11] These fixed strings are called labels.

2.4.12 HMAC-based key derivation functions (HKDFs)

Labels can also be used to create a number of independent keys from a single master key. These independent keys are known as subkeys.

For example consider the case where two actors, Alice and Bob, share a secret key ms, but want to use different keys for sending messages and receiving messages. They could compute alice_write_key = h("Alice", ms) and bob_write_key = h("Bob", ms). Alice would then encrypt all messages she sends with *alice_write_key* and Bob would encrypt with *bob_write_key*. Whilst both subkeys are easily derivable from ms, an attacker who acquires one of the subkeys cannot derive the other. This method for computing subkeys is called a HMAC-based key derivation function (HKDF).

2.4.13 Key schedules

A key schedule, in this context, refers to the method by which all subkeys are derived. For example, TLS 1.3 uses an HKDF to derive 10 subkeys from each handshake, each used for different purposes.

2.4.14 Session transcript hashes

Another use of hash functions is to ensure agreement on the transcript of a run of a protocol. Both parties compute the hash of the transcript, and if at the end of the protocol both parties agree on the hash, then they agree on the entire transcript of the protocol.[12] This relies on the second preimage resistance of hash functions. This transcript hash is usually computed as an HMAC, to ensure authenticity.

[11] From the independence property of hash functions.

[12] In most cases this is computed incrementally with repeated hash functions. This is called a rolling transcript hash. This provides two benefits, (1) each actor only needs to store a updating hash output, rather than the complete transcript, and (2) the rolling hashes can be compared multiple times throughout the protocol.

2.4.15 Channel bindings

Channel bindings are a technique for securely layering or nesting security protocols inside one another. A channel binding is a string that uniquely identifies a protocol run, such that no two runs of a protocol with different parameters have the same channel bindings. This is related to the session transcript hash, in that a session transcript hash will be different for any two runs with different parameters. However, when channel bindings are used to securely layer multiple protocols they often need to include some extra components, such as channel bindings for earlier runs and shared secrets, depending on what security guarantees they need to achieve [BDP15]. For example the channel bindings produced by TLS are based not only on the sessions transcript hash, but also on the master secret. In Chapter 4 we discuss the different guarantees that can be achieved with different channel binding constructions. In Chapter 6 we develop this work further, and introduce channel bindings that layer two-party protocols on top of multi-party protocols.

2.5 Security properties

In this section we discuss the main security properties we use throughout the thesis. We introduce further properties as necessary as we proceed through the chapters.

2.5.1 Confidentiality

We describe a value as confidential if, at the end of a protocol run[13] between two honest parties, the attacker can not derive it. We describe a value as a shared secret if, at the end of a protocol run the value is known to both parties, and not to the attacker. We describe a key as a pre-shared key (PSK) if both parties are assumed to know the value at the start of the protocol run, and the attacker does not know it.

[13] We define a protocol run as a sequence of messages sent or received by an actor, as prescribed by the protocol definition, such that the parameters of the messages are consistent with the steps defined in the protocol definition. We say that an actor has completed a protocol run if they successfully completed the last step of the protocol in which they were involved. A protocol run may end unsuccessfully or continue to completion.

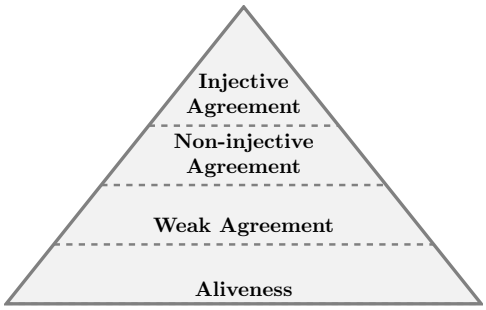


Figure 2.2: Lowe's hierarchy of authentication

2.5.2 Authentication

In this document we use Lowe's hierarchy of authentication [Low97]. Lowe defines a series of different authentication properties of different strengths, which we show in Figure 2.2. Lowe defines authentication for protocols run between an initiator A and a responder B. We list these definitions here, highlighting the progression between each definition.

The weakest guarantee Lowe defines is Aliveness.

Definition 2.5.1. Aliveness [Low97, p. 2] We say that a protocol guarantees to an initiator A aliveness of another agent B if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol.

Aliveness is a very weak property, that does not require recentness, or proof of who the responder thought they were communicating with. The presentation of a certificate can be thought of as a protocol with Aliveness. An attacker can replay a certificate as many times as they wish once they have seen a copy. Such a presentation doesn't prove the presenter owns the certificate, merely that the owner was once alive. This is useful in the case of certificate chains. A certificate chain is a list of certificates, starting at a root certificate, with each certificate signing the next in the list. The root certificate signs the first certificate, the first certificate signs the second, and so on. In the simple case, a website may present a chain of certificates

consisting of the certificate of a certificate authority, and its own certificate. Upon receiving this chain the client can reason about the Aliveness of the certificate authority.

Weak agreement is the next definition in the hierarchy.

Definition 2.5.2. Weak agreement [Low97, p. 3] We say that a protocol guarantees to an initiator A weak agreement with another agent B if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, **apparently with** A.

The only extra requirement over the Aliveness property is that the responder believes that it was running the protocol with the initiator. Proving a protocol has weak agreement requires A to reason about the beliefs of B.

As we will discuss in Chapter 3, TLS 1.3 only claims weak agreement in its peer authentication property. That TLS would apparently only require such a weak property is surprising. However this requirement is misleading. TLS 1.3 also requires other properties, such as requiring the client and server to establish the same key. Requiring the client and server to agree on some data is stronger form of authentication, namely non-injective agreement.

Definition 2.5.3. Non-injective agreement [Low97, p. 3] We say that a protocol guarantees to an initiator A non-injective agreement with a responder B **on a set of data items** ds **(where** ds **is a set of free variables appearing in the protocol description)** if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, **and** B **was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in** ds**.**

A modified version of this property is used in Chapter 4 to reason about EAs. Requiring non-injective agreement means that whilst EAs must not be forgeable[14], under some circumstances, it is permissible for them to be replay-able.

TLS 1.3 requires any two distinct sessions to produce a unique key. This uniqueness requirement gives us the final layer of the hierarchy, injective agreement. In cases where the context is clear we sometimes simply refer to this property as agreement.

Definition 2.5.4. Injective Agreement [Low97, p. 3] We say that a protocol guarantees to an initiator A agreement with a responder B on a set of data items ds if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and was acting as responder in his run, and the two agents agreed on the data values corresponding to all the variables in ds, **and each such run of A corresponds to a unique run of B.**

This is the property we prove for TLS 1.3. Lowe's hierarchy continues on with a number of variants of authentication properties, but these properties are sufficient for our purposes.

When speaking about messages, we say a protocol message is authentic if it was authored by its purported author.

2.5.3 Freshness

A related property to authentication, particularly to injective agreement is freshness. A protocol guarantees freshness of a run to a participant if whenever the participant completes a run of a protocol then its peer generated its responses specifically for the current session [Gon93].

[14] A message is forgeable if it can be created by an attacker.

This property is also related to recentness, i.e. if a protocol participant completes a run of a protocol then its peer was also running the protocol recently [Low97]. A protocol that has freshness and non-injective agreement also has injective agreement. There are other methods to achieve injectivity, but all the protocols we discuss in this thesis which require injectivity also require freshness.

Freshness can be achieved through challenge/response protocols [NS78] [BBF83] [OR87], counters [VK83], or timestamps [LG92]. Although using timestamps allows for a single message to prove freshness, timestamps require clock synchronisation between the protocols participants. A survey of different techniques for achieving freshness can be found in Gong [Gon93]. Freshness prevents certain classes of replay attacks known as classic replay attacks [Syv94], where an attacker replays a message recorded from another run of the protocol.

Freshness is often achieved through the inclusion of nonces in the messages, which would be classified as a truly random challenge/response under Gong's classification [Gon93], and this is the approach used throughout this thesis. If a nonce is echoed back in an authentic message by the peer, then the initiator can be sure the message was created in response to her message. If a predictable value is used in a challenge/response[15] the protocol is vulnerable to a wider range of attacks, and requires that the responder trust the challenger [Gon93].

2.5.4 Integrity

Another closely related property to authentication is integrity. Voydock et al. [VK83] define integrity of an encrypted message in terms of two properties. First that if an encrypted message is changed in transit it will be detected with high probability, and second that if an encrypted message is decrypted with the wrong key it will be detected with high probability. [16] Because we make the perfect cryptography assumption, we say that both these cases will be detected. We do not restrict this definition specifically to encrypted mes-

[15] Classified as an asynchronous counter in Gong [Gon93].

[16] By this they mean that with very high probability a message can be correctly classified as either modified or unmodified.

sages, but require these properties of any messages, or portions of messages, that claim to have integrity protection.

One way to achieve integrity with a MAC tag, as described in 2.4.9. Because the tag can only be created by protocol participants, if the message is tampered with then the tag will fail to verify, and the recipient can detect the tampering. For a protocol to achieve agreement on some parameters there needs to be some level of integrity.

2.5.5 Perfect forward secrecy (PFS)

Perfect forward secrecy (PFS) is one of the more complex properties we will discuss in the background. First proposed in Günther [Gün90], we say a protocol has perfect forward secrecy (PFS) with respect to an LTK, ltk, if the protocol's confidentiality guarantees are not broken if ltk is compromised after the session is complete. This is equivalent to saying that if the secrets of the protocol cannot be derived by a passive observer even if it knows ltk, then the protocol has PFS. This is usually achieved by agreeing an asymmetric ephemeral key, such as the key derived during a DHE. A passive attacker cannot, by definition, perform an active MITM attack. Therefore, because of the computational DH assumption, the attacker cannot derive the ephemeral session key.

2.6 Threat models

A threat model defines under what circumstances the properties are required to hold. In general this means defining the attacker's abilities. Dolev and Yao in their seminal paper [DY83] defined what came to be known as the Dolev-Yao attacker. The Dolev-Yao attacker is one that can read, write, intercept, modify, and delete any message. This is sometimes referred to as "the network is the attacker". The only restriction placed on its abilities is that it cannot break encryption. This is the "perfect cryptography" assumption.

A protocol can achieve different properties against different attackers. For example, a protocol that guarantees injective agreement against a Dolev-Yao attacker, might only achieve non-injective agreement against a stronger attacker. If an attacker is too powerful then it can be impossible to achieve

the desired properties, however if the modelled attacker is weaker than the attackers that will be attacking the protocol in practice then the protocol may have undiscovered flaws. This means that when defining the security properties a protocol requires it is also necessary to define under what threat model those properties are required to hold.

All the threat models we use in this thesis use the Dolev-Yao attacker as a base point. We then give the attacker the ability to compromise various keys by performing reveal actions. If all secrets are compromised then it is nearly impossible to achieve any security guarantees [CCG16]. We thus express the guarantees we require in terms of the attackers actions. For example we might say something along the lines "if A completes a run of the protocol and the attacker did not reveal the LTK of A, ltk_A, before the run completed, then the attacker never learns the ephemeral key k". In this way we can express the exact restrictions on the attacker, or more practically, exactly what the attacker needs to achieve to break the protocol.

2.6.1 The Needham-Schroeder protocol

In this section we briefly digress and introduce the Needham-Schroeder protocol, which we use as a running example throughout the thesis to introduce various concepts. We choose it as an example because it is a very well studied protocol in the formal analysis field. An analysis of the Needham-Schroeder protocol when introducing a new tool is almost de rigueur [BAN90] [Mil95] [Low95] [Mea96] [FHG98] [Son99] [GK00] [Bla01] [Cre08] [Mei13]. We will discuss the advent of tool-supported formal analysis, and attacks on the protocol. The Needham-Schroeder protocol[17] was published in 1978 [NS78], and claimed to provide mutual authentication between two parties.

The protocol

We define two parties, Alice (A) and Bob (B), each of whom has an asymmetric key pair $(pk(sk_A), sk_A)$ and $(pk(sk_B), sk_B)$ respectively. $pk(sk_A)$ represents Alice's public key, and sk_A the corresponding secret key. For sim-

[17] The Needham-Schroeder protocol technically refers to a pair of protocols, a symmetric and an asymmetric version. In our example we will only discuss the asymmetric version.

plicity we will assume that Alice and Bob know each others public keys, $pk(sk_A)$ and $pk(sk_B)$ respectively. We will notate nonces n_X.

The protocol proceeds as follows.

$$A \rightarrow B : \{n_A, A\}_{pk(sk_B)}$$

Alice sends Bob a nonce n_A and her identity encrypted with Bob's public key.

$$B \rightarrow A : \{n_A, n_B\}_{pk(sk_A)}$$

Bob responds with Alice's nonce, and one of his own, both encrypted with Alice's public key.

$$A \rightarrow B : \{n_B\}_{pk(sk_B)}$$

Alice then responds with Bob's nonce, encrypted with Bob's public key.

The idea of the protocol is that because only Alice and Bob can decrypt the relevant messages n_A and n_B become shared secrets, and attacker cannot intercept them.

In 1990 Burrows et al. formally analysed the protocol using the BAN logic [BAN90], a manual authentication logic, and offered a proof that the parties mutually authenticated each other. In 1995, 17 years after the protocol was first proposed and 5 years after it had been formally analysed Lowe [Low95] found an attack on the protocol.

The attack

We introduce a new actor, the attacker (I), with its own public / private key pair $(pk(sk_I), sk_I)$. The attacker is allowed to act both as a legitimate actor and an attacker.[18] We introduce the notation I_X to indicate the attacker

[18] This is the description of the attacker given by Lowe, we offer a slightly different perspective on the attacker in the next section. Our perspective simply makes it easier to generalise Lowe's result to our use cases.

impersonating actor X. The attack proceeds as follows.

$$\begin{aligned} A \to I &: \{n_A, A\}_{pk(sk_I)} \\ I_A \to B &: \{n_A, A\}_{pk(sk_B)} \\ B \to I_A &: \{n_A, n_B\}_{pk(sk_A)} \\ I \to A &: \{n_A, n_B\}_{pk(sk_A)} \\ A \to I &: \{n_B\}_{pk(sk_I)} \\ I_A \to B &: \{n_B\}_{pk(sk_B)} \end{aligned}$$

Alice sends the attacker a request, which the attacker can decrypt because it is encrypted with its public key. The attacker re-encrypts the request with Bob's public key and forwards it on, impersonating Alice. Bob, on receipt of this message responds with a message encrypted by Alice's public key. Even though the attacker can intercept the message, it cannot decrypt it. The attacker forwards this opaque blob on to Alice, who can decrypt it. Alice, still believing she is in an entirely legitimate run with the attacker, decrypts Bob's message and encrypts Bob's nonce with the attacker's public key, and sends the result to the attacker. At this point Alice and the attacker have completed an entirely legitimate run. The attacker however, has now learned Bob's nonce. The attacker can now re-encrypt Bob's nonce with Bob's public key, and send it to Bob. Bob now believes that he has completed a run with Alice, however Alice does not believe she has completed a run with Bob.

Whilst this protocol does provide aliveness, as we can see from this attack it does not even provide weak authentication to Bob, that is, Bob has completed a run of the protocol ostensibly with Alice, but Alice has not completed a run of the protocol ostensibly with Bob.

Limitations of formal analysis

This attack seems to invalidate the proof offered in Burrows et al. [BAN90]. However the BAN logic doesn't consider that the attacker might act as a legitimate actor. If the attacker cannot act as a legitimate entity then this attack does not work, and the protocol is secure.

Alternatively, we could model this difference with a Dolev-Yao attacker extended with the ability to compromise long-term keys, as opposed to one who could act as a legitimate actor. The protocol is not intended to be secure against an attacker who can compromise all LTKs, but it should be secure against an attacker who can only compromise the LTKs of actors other than the principles, i.e. anyone other than Alice or Bob. However, if the attacker were to compromise the long term keys of some third actor, Charlie, then it can use those keys to attack Bob, by acting as Charlie to Alice. Thus an implicit assumption of the BAN logic is that all LTKs are secure. Burrows et al.'s proof is thus correct for the threat model it considers, however in practice requiring the LTKs of every actor to remain secure is too strong an assumption.

Lowe [Low96] then introduces a tool based analysis technique. Using a tool called FDR[19] he rediscovers his attack before introducing a fix, which he proves secure under his new formalism. The fix he provides slightly modifies the second message.

$$B \rightarrow A : \{n_A, n_B, B\}_{PK_A}$$

By adding Bob's identity into his response Lowe's fixes the attack on the original protocol. This tweaked protocol is sometimes known as the Needham-Schroeder-Lowe protocol.

Lowe's analysis only requires that the LTKs of Alice and Bob remain secure. This is a much more limited assumption, and thus requires a stronger proof of the security of the protocol. We use this example to highlights two things about formal analysis, first, that a proof is only as good as its model, and second, that it is important to consider the strongest possible attacker, even if the protocol is not necessarily designed to be secure against such an attacker.

In our work we take care to define the strongest possible attacker, and iteratively restrict its actions until we find what we term the security boundary, i.e. the point at which the security goals begin to hold. This gives us a very clear set of assumptions, defining exactly what threat model the pro-

[19] FDR is a refinement checker for Communicating Sequential Processes (CSP)

tocol protects against. By proving both positive and negative results about protocols we can build confidence in our models. A model that holds when it shouldn't, or that doesn't hold when it should is likely to have a flaw. By finding the security boundary we show that at the very least our model isn't vacuously true or false, removing a potential source of error.

2.7 Tamarin

2.7.1 The Tamarin prover

The Tamarin prover is a formal protocol analysis tool that allows the analysis of highly complex stateful protocols. Taking a symbolic analysis approach it implements a Dolev-Yao style network attacker, which can be extended by the user. It allows for the specification of detailed security properties and has state-of-the-art support for protocols with branches, loops, state, and equational theories. In addition to analysing trace properties, it also provides support for some classes of hyperproperties (diff-equivalence). Tamarin has been successfully used to analyse highly complex protocols such as TLS 1.3 [RFC8446]. For example an analysis of an early draft of the TLS 1.3 specification using Tamarin found a vulnerability in post-handshake authentication [Cre+16].

When analysing trace properties, Tamarin takes a protocol model, written as a multi-set rewrite system, and a series of security properties, written as first-order logical formulas, and attempts to prove the properties hold using a backwards search. The proof search essentially applies a constraint solving algorithm to the negation of the property – if no solution exists, this corresponds to a proof that the property holds, and if a solution is found, it represents a counterexample. Tamarin also has an extensive graphical user interface (GUI) which can be used to interactively construct proofs. This is very helpful for exploring partial proofs, and deriving which factors are key to security.

Multi-set rewrite systems

A multi-set is an unordered collection of elements, allowing repetitions. A multi-set rewrite rule r is a mapping from a multi-set F (called the antecedent) to a multi-set G (called the consequent). A multi-set rewrite system R is a collection of such rules. Starting from the empty multi-set, notated "$\cdot$", rules from the system can be repeatedly applied, to produce some multi-set, H.

2.7.2 The Tamarin specification language

A Tamarin model is specified by a list of multiset rewrite rules that model the state machines for the protocol and any special attacker capabilities, and security properties are specified using a fragment of first-order logic with quantification over timepoints. Note that within Tamarin's framework, all (security) properties are referred to as "lemmas". We introduce Tamarin's specification language with an example rule.

```
rule Example_Rule:
  [ !Key($A,sk), Fr(n) ]
--[ Send($A,n,sk) ]->
  [ Out(senc(n,sk)) ]
```

The rules are used to model a transition system, whose state is a multi-set of *facts*; this is initially the empty multi-set. Tamarin rules have the antecedents, or inputs, on the left-hand side (LHS), and their consequents, or outputs, on the right-hand side (RHS). Roughly speaking, a rule can trigger if the facts on its LHS are present in the current state, after which they are replaced by the facts on the RHS. In the middle are placed actions, which serve as the connection between the transition system and the property specification logic.

The actions of all the triggered rules form a tree called a trace. Actions specify observable events in every trace, and we express security properties as properties over this tree. Actions occur at symbolic times, for example we might write `Send(x,n,y)@j`, meaning that the `Send` action occurred at time `j`. Symbolic time, as opposed to regular time, forms a partially ordered set. Thus we might be able to say that some actions occurred before or after some other actions, but cannot necessarily derive a schedule of every action. When a symbolic time is referenced somewhere other than an action we distinguish it with a `#`, for example we might write `#i < #j`. This would be read as "the time `i` precedes `j`".

In our example, the rule takes as a pre-condition a pair of an actor, `$A`, and a key, `sk`; and a fresh value, `n`. This rule uses Tamarin's built in `Fr` function which outputs an unpredictable, unique value. Fresh values are always available in the state.

The identity `$A` is marked with a `$` character, which makes it a public value. The `!Key(x,y)` fact is marked with a `!` character, which makes it a persistent fact. This means that Tamarin will allow it to be consumed repeatedly, as opposed to only allowing it to be consumed once. The right-hand side of the rule outputs `senc(n, sk)` to the network. The `senc(message, key)` fact is a symbolic representation of symmetric key cryptography. We use the `Send($A, n, sk)` action to reason about the trace of this rule.

To continue our example we might now extend our attacker with a rule that allows them to reveal secret keys.

```
rule rev_sk:
  [ !Key($A, sk) ]
--[ Rev_sk($A, sk) ]->
  [ Out(sk) ]
```

This rule consumes a `!Key(x,y)` fact, and sends the key to the network. Because Tamarin assumes a Dolev-Yao attacker, sending a message to the

network is equivalent to revealing it to the attacker. This rule allows the attacker to reveal a secret key by performing the action `Rev_sk`.

We are now in a position to specify a simple security property.

```
lemma secret_key_confidentiality:
  "All actor nonce key #j #k.
      Send(actor, nonce, key)@j
    & K(nonce)@k
  ==>
  Ex #i.
      Rev_sk(actor, key)@i
    & (#i < #k)"
```

This property says that if a `Send` action occurs at time j, and the attacker learns the nonce at time k, then the attacker must have performed a `Rev_sk` action before learning the nonce. We could also add the requirement that `(#j < #k)` and prove that the attacker cannot learn the nonce before the actor chooses it.

A rule for creating `Key(x,y)` facts is also needed.

```
rule Create_key:
  [ Fr(~sk)
  , In(<$A>)
  ]
--[ CreateKey($A, ~sk) ]->
  [ !Key($A, ~sk) ]
```

The `Create_key` rule takes a fresh value `~sk` and a public identity, and outputs a `!Key` fact, pairing the identity with the key. The `Create_key` rule also triggers an action, `CreateKey`, such that we can reason about key creation.

This protocol is sufficiently simple that the Tamarin prover can solve it using its heuristic solver. For more complex protocols Tamarin provides an interactive prover that allows the user to guide the proof.

Chapter 3

Transport Layer Security

3.1 Introduction

The IETF started drafting the latest version of the protocol, TLS 1.3, in the spring of 2014. Unlike the development of TLS 1.2 and below, the TLS WG adopted an "analysis-prior-to-deployment" design philosophy, welcoming contributions from the academic community before official release. There have been substantial efforts from the academic community in the areas of program verification– analysing implementations of TLS [BKB16] [Bha+16b], the development of computational models– analysing TLS within Bellare-Rogaway style frameworks [Dow+15] [KW15] [Li+14] [Dow+16] [Fis+16] [Koh+14], and the use of formal methods tools such as ProVerif [Bla+16] and Tamarin [Sch+12] to analyse symbolic models of TLS [AM16] [Cre+16] [Hor16] [BBK17]. All of these endeavours have helped to both find weaknesses in the protocol and confirm and guide the design decisions of the TLS WG.

The TLS 1.3 draft specification however, was a rapidly moving target, with large changes being effected in a fairly regular fashion. This often rendered much of the analysis work 'outdated' within the space of few months as large changes to the specification effectively result in a new protocol, requiring a new wave of analysis.

The final specification [RFC8446] was published in August 2018. In this work we contribute to the last wave of analysis of TLS 1.3 prior to its official release. We present a tool-supported, symbolic verification of a near-final

draft of TLS 1.3, adding to the large effort by the TLS community to ensure that TLS 1.3 is free of the many weaknesses affecting earlier versions, and that it is imbued with security guarantees befitting such a critical protocol.

3.1.1 Chapter overview

Over the course of the chapter:

1. We develop a symbolic model of draft 20 of the TLS 1.3 specification that considers all the possible interactions of the available handshake modes, including PSK-based resumption and 0-RTT. Its fine-grained, modular structure greatly extends and refines the coverage of previous symbolic models that were successfully used to discover sophisticated interaction attacks, including that of Cremers et al. [Cre+16]. Our model effectively captures a new TLS 1.3 protocol, incorporating the many changes that have been made to the protocol since the development of these previous models. We also note that our model is highly flexible and can easily accommodate the removal of the 0-RTT mechanism, should the need arise.

2. We prove the majority of the specified security requirements of TLS 1.3, including the secrecy of session keys, PFS of session keys (where applicable), peer authentication, and key compromise impersonation resistance. We also show that after a successful handshake the client and server agree session keys and that session keys are unique across handshakes.

3. We uncover a previously unreported behaviour that may lead to security problems in applications that assume that TLS 1.3 provides strong authentication guarantees.

4. We provide a novel way of exhibiting the relation between the specification and our model: we provide an annotated version of the TLS 1.3 specification that clarifies which parts are modelled and how, and which parts were abstracted. This provides an unprecedented level of modelling transparency and enables a straightforward assessment of

the faithfulness and coverage of our model. We anticipate that this output will be of great benefit to the academic community analysing TLS 1.3, as well as the TLS WG as it provides a clear and easy-to-understand mapping between the TLS 1.3 specification and a TLS 1.3 model.

All our Tamarin input files, proofs, and the annotated TLS 1.3 specification that shows the relation between the RFC and the model, can be downloaded from [Cre+17b].

3.1.2 Related work

As mentioned, there has been a great deal of work conducted in the complementary analysis spheres pertinent to TLS 1.3. Of most interest to this work are the symbolic analyses presented in [Cre+16], [AM16] and [BBK17].

The work in [Cre+16] by Cremers et al. offered a symbolic model and accompanying analysis of draft 10 of the TLS 1.3 specification, using the Tamarin prover. Since then, there have been multiple changes made to the specification. These updates have included major revisions of the 0-RTT mechanism and the key derivation schedule. In draft 10, the sending of early data required a client to possess a semi-static (EC)DH value of the server. This particular handshake mode was removed and replaced by a PSK 0-RTT handshake mode– early data can now only be encrypted using a PSK. In fact, the PSK mechanism has been greatly enhanced since draft 10 with new PSK variants and binding values being incorporated in to the specification. Post-handshake authentication was officially incorporated from draft 11 onwards and a few drafts later, post-handshake authentication was enabled to operate with the PSK handshake mode. Another change to be incorporated after draft 10 was the inclusion of 0.5-RTT data - the server being able to send fully protected application data as part of its first flight of messages.

All of these changes have resulted in what is effectively a very different TLS 1.3 protocol, particularly from a symbolic perspective. As a Tamarin model aims to consider the interaction of all possible handshake modes and variants, changes to these modes, as well as the inclusion of new post-handshake combinations, results in a very different different set of traces to

be considered when proving security properties. Hence, this work presents a substantially different model to [Cre+16], and follows a far more fine-grained and flexible approach to modelling TLS 1.3.

The work in [AM16] is an analysis of TLS 1.3 by the Cryptographic protocol Evaluation towards Long-Lived Outstanding Security (CELLOS) Consortium using the ProVerif tool. Announced on the TLS WG mailing list at the start of 2016, it showed the initial (EC)DHE handshake of draft-11 to be secure in the symbolic setting. In comparison to our work, this analysis covers only one handshake mode of a draft that is now somewhat outdated.

The ProVerif models of revision 18 presented by Bhargavan et al. in [BBK17] include most TLS 1.3 modes, and cover rich threat models by considering downgrade attacks (both with weak crypto and downgrade to TLS 1.2). However, unlike our work, they do not consider all modes, as they do not consider the post-handshake client authentication mode. While they cover relatively strong authentication guarantees (which led to the discovery of an unknown key share attack), their analysis did not uncover the potential mismatch between client and server view that we describe in Section 3.5.2.

3.1.3 The final development of TLS 1.3

Our analysis is of draft 20 of the protocol, and the final draft before publication was 28. However unlike the change between draft 10 and draft 20, the mechanisms remained very stable over the final eight drafts, with most changes being typographical.

The most significant changes to the protocol were tweaks to make the draft more amenable to middleboxes. When running deployment tests it was found that some middleboxes would check the version number of a handshake and block versions of TLS other than TLS 1.2. Whilst this prevents the use of older, weaker versions of TLS, it also makes TLS 1.3 handshakes fail. The TLS WG thus decided to tweak the protocol so that it looked more like TLS 1.2, tricking the middleboxes, and adding a new version number field to the extensions.

We enumerate the non-trivial changes here, along with an estimate in the amount of effort required to model the change.

1. TLS 1.3 has a series of alerts and errors which indicate various failure conditions. In the final changes before publication a number of alerts were made either more or less specific, and some alert types were removed entirely. As we do not model the alert layer, these changes do not affect us.

2. The semantics of the `legacy_session_id` field in the `ClientHello` message were updated to improve middlebox compatibility. In version 18, unless the client had a session id set by a pre-TLS 1.3 server, this field was required to be of zero-length. In version 28, if the client is operating in so-called "compatibility mode", then the field is required to be non-empty, and if the client does not have a pre-TLS 1.3 session id it is required to use an unpredictable 32-byte value. Because we do not model earlier versions of TLS our model captures this as a blank field. Updating the field to include a random value would be very quick in terms of development work.

3. The `ServerHello` message was changed to add three fields to improve middlebox compatibility.

   (a) `legacy_version` - A field with a fixed value of `0x0303`.

   (b) `legacy_session_id_echo` - This field echoes the contents of the client's `legacy_session_id` field.

   (c) `legacy_compression_method` - A field with a fixed value of `0`.

   Changing the model to capture these changes would be very simple. Specifically it would require adding two fields of fixed value to the `ServerHello` message, and carrying one extra piece of state in the server. The `ServerHello` message was also changed to include the `version` field in the mandatory extensions. This would require minor refactoring of the code to move the `version` field from the beginning of the `ServerHello` message to the end.

4. The `HelloRetryRequest` message was changed to echo the structure of the `ServerHello` message, essentially performing the same transformation as was applied to the `ServerHello` message, but also adding an extra field labelled `Random`, which carries a fixed value equal to the SHA-256 of the string "HelloRetryRequest". This is also a trivial change to the model.

5. Support for RSASSA-PSS algorithms was changed. As we use abstract perfect cryptographic functions this change doesn't affect our model.

6. A nonce, `ticket_nonce`, was added to the `NewSessionTicket` message, to ensure uniqueness of tickets. Adding a nonce to a message is very simple in our model.

7. Messages sent in TLS 1.3 are broken down into typed records and, once record protection starts, the records are encrypted using authenticated encryption with associated data (AEAD). Record protection starts once a set of keys have been agreed between the participants, i.e. part way through the `ServerHello`. In the final version of the specification the record header was added to the additional data in the AEAD section of the record payload protection. Modelling this change would require a moderate amount of refactoring the code.

8. A new type of message, the `change_cipher_spec` message type, was added to the record protocol. This message type was added to improve compatibility with middleboxes, and both sides are required to ignore such messages during the handshake, unless they are malformed in which case they are required to abort the connection. Modelling this new message type would be relatively straight-forward.

9. The `signature_algorithms_cert` extension was made mandatory to implement. This extension allows a server to support a different set of algorithms for its certificate signatures and for the handshake signatures. Modelling this change would be simple, because we do not distinguish between cryptographic algorithms, assuming all to be perfect.

Whilst modelling the changes would be reasonably simple, re-checking the models would require a substantial amount of human effort. Further few of these changes are security valent. The adding of constants to various messages has no security impact, assuming that the protocol is secure without them, and the adding of an extra nonce to the `ClientHello` and `ServerHello` should make the protocol strictly more secure (or rather, removing a nonce should be a fault-preserving simplification [HL01], and thus our current proofs should also apply to the updated specification.). The addition of a nonce to the `NewSessionTicket` message also has limited relevance to our results due to a quirk of our model. The specification does not require the `ticket` field to be unique, however our model makes the simplifying assumption that it is, and models it as a nonce. Because we assume perfect cryptography, and thus longer nonces are no more or less secure, adding an extra nonce to a message (assuming it's always treated exactly the same) has no effect on the proof.

Adding a `change_cipher_spec` message that is ignored seems unlikely to affect the security of the protocol. The change to the AEAD structure should strictly improve the security of the protocol, and thus if we can prove the properties we want in the current model, we would expect them to hold in the more secure version. Finally the addition of the `signature_algorithms_cert` should make no difference to the results of our analysis, because of our perfect cryptography assumption. We model the list of supported signature algorithms in the handshake as a public constant, and would model the supported certificate signature algorithms in the same way. Thus, because removing public constant fields is a fault preserving simplification, if our model can be proven secure, then the extended model should also be secure.

At the time of our analysis most of the cryptographic mechanisms in the TLS 1.3 draft were stable, and other than fluctuations surrounding the 0-RTT mechanism [Mac17], we did not expect substantial changes to come.

3.1.4 Renegotiation and post-handshake authentication

In TLS 1.2 there is a feature called *renegotiation*, which allows an existing session to be renegotiated with different cryptographic parameters. A

common use case was for only requesting a client certificate when the client tried to access a protected resource, rather than requesting one for every client. A serious flaw was found with renegotiation in 2009 [SR09], that was patched in TLS 1.2 with extended master secret [RFC7627]. The vulnerability was caused because the new session was not cryptographically bound to the prior session. Renegotiation was deprecated in TLS 1.3 because its utility was deemed not great enough to justify the effort necessary to secure it. Instead a feature called post-handshake authentication was added to TLS 1.3, to cover the most common use case. Post-handshake authentication allows a client to add a certificate to a session after the handshake has been completed. In this chapter we analyse the post-handshake authentication mechanism, in combination with all the other modes of TLS 1.3.

In Chapter 4 we discuss a proposed extension to TLS 1.3 that supersedes the post-handshake authentication mechanism. We further this work in Chapter 5.

Between the tenth revision and the revision of the draft analysed in this chapter a large amount of work was produced on securing the PSK mechanism. PSKs established in an earlier session are a form of channel binding, a topic we discuss at length in Chapter 4, and revisit in Chapters 5 and 6.

3.1.5 Chapter organisation

This chapter is organised as follows. In Section 3.2 we describe the TLS 1.3 protocol and the security properties claimed in the specification. Section 3.3 describes our Tamarin model and provides a few Tamarin prover fundamentals. In Section 3.4, we describe our encoding of the security guarantees, followed by Section 3.5 where we describe our results. Section 3.6 covers the relationship between our model and the specification document, discussing how we provide a website that describes our model side-by-side with the specification, giving us unprecedented modelling transparency. We conclude in Section 3.7.

3.2 TLS 1.3

The TLS 1.3 protocol is made up of two main sections, the handshake and the record layer.[1] The handshake establishes the cryptographic context needed to create a secure channel, and the record layer provides a transport mechanism. In this section we provide a description of the TLS 1.3 handshake, and we outline the claimed security properties and guarantees of the protocol.

3.2.1 New mechanisms

The four years of effort that has gone into crafting and fine-tuning both the security and efficiency mechanisms of TLS 1.3 is readily apparent in the large structural departures from TLS 1.2. The two protocols have broadly similar goals but exhibit many differences. For example, a full TLS 1.3 handshake requires one fewer round trip before a client can transmit protected application data, and the new 0-RTT mechanism allows less sensitive application data to be sent by the client as part of its first flight of messages.

TLS 1.3 has three key exchange modes, namely, DHE, PSK exchange, and PSK coupled with DHE. These modes enable useful features like session resumption and the transmission of early application data. Additionally, there are a number of handshake variants that allow for group renegotiation and the sending of context-dependent, optional messages. Each of these variants has different properties and offers different security guarantees.

Furthermore, TLS 1.3 has three post-handshake mechanisms covering traffic key updates, post-handshake client authentication, and the sending of new session tickets (NSTs) for subsequent resumption via a PSK. The handshake protocol maintains a rolling transcript, on which both parties must agree. This transcript takes the form of a hash value of all of the handshake messages. Post-handshake messages, however, are not included in this transcript resulting in different security properties for the post-handshake mechanisms.

[1] There is also an alert protocol layer, which we did not examine.

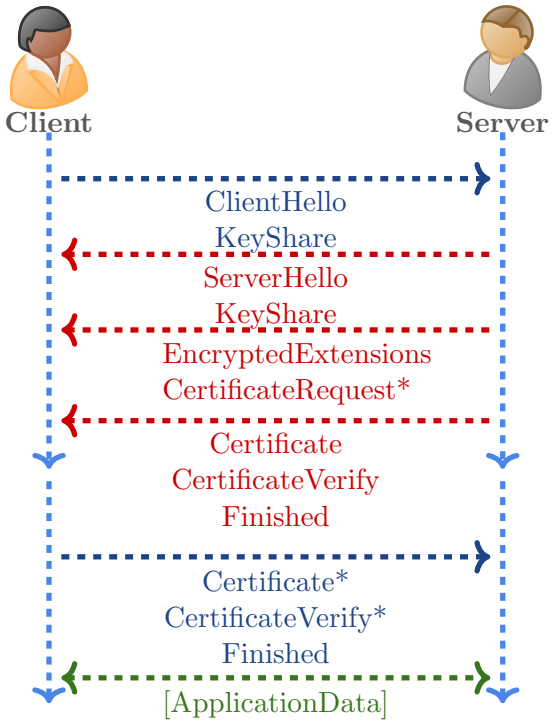


Figure 3.1: A full TLS 1.3 handshake (Section 3.2.2). Messages marked with a '*' do not occur in all modes.

TLS establishes a secure channel that is authenticated either unilaterally or bilaterally. A unilaterally authenticated channel is one in which only one party is authenticated, and the other is anonymous. A bilaterally authenticated channel is one in which both parties are authenticated. In the context of TLS only the client may be anonymous, the server must always authenticate itself.

We analyse all of the TLS 1.3 key exchange modes, handshake variants, and post-handshake mechanisms simultaneously, considering all possible interactions between them. We provide a brief description of these components as well as associated message flow diagrams.

3.2.2 The main handshake

The main handshake of TLS 1.3 consists of three flights of messages over one-and-a-half round trips. These flights are shown in Figure 3.1. The default

mode of TLS 1.3 allows for ephemeral DH keys to be established either over a finite field or using elliptic curves.

We now walk through a default DHE handshake.

3.2.3 The first flight

`ClientHello`

The first flight of messages consists solely of a `ClientHello` message. The `ClientHello` message contains two fields of interest, a nonce and a list of extensions. The message struct is defined in Figure 3.2. The list of extensions contains extra values that the TLS client wants to signal or negotiate with the server. In a vanilla handshake this list of extensions will include the `KeyShare` extension, which contains a DH key share.

```
struct {
ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
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;
```

Figure 3.2: Definition of `ClientHello` from [RFC8446, p. 28]

In Figure 3.2 we show the `ClientHello` struct, with fields of particular interest highlighted. Each field has a type and often a range of acceptable values. For example the `random` field has the type `Random`, indicating that it holds random data, in this case a nonce. The `extensions` field has a range of 8 to $2^{16} - 1$, i.e. $[8 - 65535]$. As we mentioned earlier the legacy version number has been preserved to improve compatibility with misbehaving middleboxes.[2] The `opaque` type means that the recipient is not expected

[2] There are specific rules about what a middlebox is allowed to do and what it is not allowed to do. This behaviour is not allowed. To try and prevent this sort of atrophying of

to understand or parse the contents of the field, but rather to treat it as a "blob" of data. In this case the type is used to indicate that the fields are actually unused, and are merely placeholders for values that would be present in a TLS 1.2 handshake. In later examples it indicates data that cannot be broken down, but can be compared, for example the output of a hash function. An actor cannot reverse the hash, and thus analyse the contents of the `opaque` type, but can recompute the hash from values it knows, and check to make sure that the values match.

3.2.4 The second flight

The second flight of messages is sent by the server, and contains the `ServerHello`, `EncryptedExtensions`, potentially a `CertificateRequest`, and a sequence of messages, `<Certificate, CertificateVerify, Finished>`. This latter pattern is at the heart of the security guarantees of both TLS 1.3 and EAs, which are the topic of Chapter 4 and Chapter 5.

`ServerHello`

The `ServerHello` is nearly a mirror of the `ClientHello`, with the server replacing the client's nonce with a nonce of its own, and the client's key share with its own. The server may also potentially send different extensions. We defer in depth discussion of extensions to Chapter 5.

`EncryptedExtensions`

Once the client has received the `ServerHello` it has enough information to compute a shared secret key. The client and server have completed an unauthenticated Diffie–Hellman exchange (DHE). The client and server compute a set of keys, called the handshake keys, which they use to encrypt the remainder of the handshake. This includes any extensions that aren't needed to compute the handshake keys. Because the handshake keys are unauthenticated an attacker may have performed a MITM attack, which would cause

the handshake in future substantial work has gone into "greasing" the protocol. Greasing involves sending messages that it is known the peer and any middleboxes will not understand, and making sure they respond appropriately. Usually this is done by sending values that have yet to be defined.

the remainder of the handshake to fail. However, if the remainder of the handshake succeeds, then the client and server can be confident ex post facto that the encrypted extensions, and the remainder of the handshake were confidential.

The handshake keys are computed using an HKDF, taking as input both the established DH key, but also the client and server hello messages. This means the keys depend on the both the client and server nonce. This is important, because the server's hello message does *not* include the client's nonce. Therefore until it receives the `EncryptedExtensions` the client doesn't see its nonce reflected back, and thus cannot establish freshness. However if it can decrypt the `EncryptedExtensions` it can indirectly establish freshness. If the client and server don't agree on the nonces used in the handshake so far they will compute different keys, and thus the decryption will fail. Thus if the client can decrypt the `EncryptedExtensions` the client knows that it computed the same handshake keys as the server, and thus that the server used the same value for the client nonce in its computation as the client did. This lets the client ensure that the handshake is fresh.

`CertificateRequest`

If the server wants the client to authenticate itself, producing a bilateral TLS channel, as opposed to a unilateral channel, then it sends a `CertificateRequest` message. The `CertificateRequest` message is defined by the struct in Figure 3.3.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

Figure 3.3: `CertificateRequest` definition from [RFC8446, p. 60]

It consists of a unique[3] `certificate_request_context` and a list of extensions that define the properties of the requested certificate. In the con-

[3] Unique within the context of a given connection.

text of a vanilla TLS 1.3 handshake the `certificate_request_context` has a length of zero, but in other contexts it is assigned to other values.

3.2.5 `Certificate`

The server now begins to send its `<Certificate, CertificateVerify, Finished>` message sequence. These messages bind the server's certificate to the handshake so far. The `Certificate` message is defined in Figure 3.4.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

Figure 3.4: `Certificate` definition from [RFC8446, p. 64]

In a vanilla handshake the `certificate_request_context` has zero length, but in other contexts it has other lengths. The `certificate_request_context` is paired with a certificate chain, chaining back to a certificate the client trusts.

The `CertificateVerify` message

The `CertificateVerify` message is designed to provide proof that the sender controls the relevant certificate. To do this the sender signs a value with the certificate's private key. To bind that signature to the session the value signed includes the rolling transcript hash, which includes the nonces. To make sure the `CertificateVerify` from the TLS handshake can't be used in another place, a label is used. For example if the server is sending a `CertificateVerify` in the context of a TLS 1.3 handshake the context string "TLS 1.3, server CertificateVerify" is included. The struct given in the TLS 1.3 draft is shown in Figure 3.5.

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

Figure 3.5: `CertificateVerify` definition from [RFC8446, p. 69]

The message consists of the choice of algorithm and a signature, which is computed as follows:

- The octet 0x20 repeated 64 times,
- a context string,
- a separator byte,
- the value to be signed.

The value that is signed is the hash of all the previous messages, referred to as the "Handshake Context", concatenated with the Certificate message. In a vanilla handshake the server uses the context string "`TLS 1.3, server CertificateVerify`" and when the certificate is being signed by a client in response to a `CertificateRequest` the context string "`TLS 1.3, client CertificateVerify`" is used.

The `Finished` message

The `Finished` message is simply an HMAC over the message transcript. This is intended to provide integrity protection for all the messages in the transcript.

3.2.6 The third flight

The client responds to the server's flight with `Finished` message, or, if the server sent a `CertificateRequest`, with a flight of `<Certificate, CertificateVerify, Finished>` messages.

3.2.7 Other modes

Pre-shared key (PSK)

In the event that a PSK has been established, a client and a server can begin communicating without a DH exchange or exchanging certificates. This is potentially attractive for low-power environments, however, without a DHE the connection loses PFS. In a PSK handshake, the server authenticates via a PSK, rather than a certificate.

When running in a PSK mode the client includes a special extension, `pre_shared_key`, in the `ClientHello`. The `pre_shared_key` extension contains a list of PSK identities and a list of PSK binders. The PSK binders are special values used to bind the PSK to the handshake. They are computed as an HMAC over the handshake up to that point, only excluding the list of binders itself.[4] For this reason the `pre_shared_key` extension is required to be the last extension. The HMAC is keyed with the PSK. The PSK binders are a type of channel binding. If the PSK is established out-of-band (OOB), for example in a prior non-TLS protocol run, then it is possible to get the guarantees of the non-TLS protocol in this mode. If the prior protocol run has PFS then it is possible to establish a PFS secrecy property for PSK mode. We discuss PSK binders in greater detail in Chapter 6.

PSK with DHE

In PSK modes By combining a PSK with DHE this mode maintains PFS whilst limiting the number of expensive public key operations that the server needs to perform. Neither the client nor the server needs to compute or verify a certificate signature.

Group renegotiation

It can be the case that the groups sent by a client are not acceptable to the server. In this case, the server may respond with a `HelloRetryRequest` message. This indicates to the client which groups the server will accept, and provides the client with the opportunity to respond with an appropriate key share before returning to the main handshake.

New session ticket (NST)

After a successful handshake, the server can issue an NST at any time. These tickets create a binding to a resumption-specific secret and can be used by the client as PSKs in subsequent handshakes.

[4] The binders cannot be dependent on themselves, i.e. one cannot use the output of the hash function as an input to the computation.

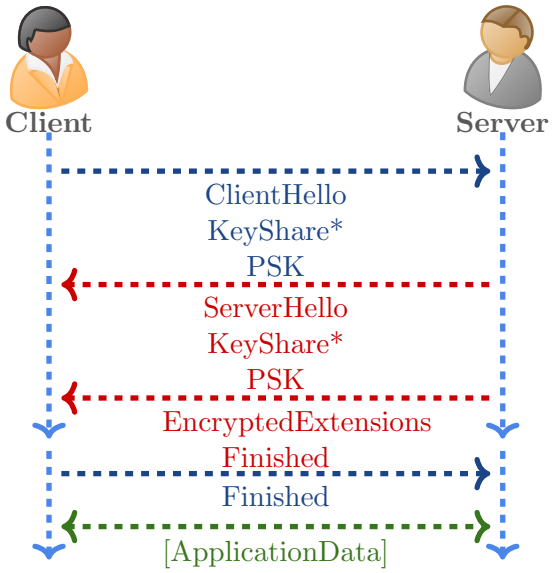


Figure 3.6: A PSK resumption handshake (Section 3.2.7)

Session resumption and PSK

This handshake variant allows a client to use a key established OOB to start a new session, or to use an NST established in a previous handshake to resume the session. This avoids the use of expensive public-key operations and in the case of a resumption, ties the security context of the new connection to the original connection. Note that a server may reject a resumption attempt made by a client, so the specification recommends that the client supplies an additional (EC)DHE key share with its PSK when trying to resume a session. Figure 3.6 depicts a PSK resumption handshake.

Zero round-trip time (0-RTT)

A client can use a PSK to send application data in its first flight of messages, reducing the latency of the connection. As noted in the TLS 1.3 draft specification, this data is not protected against replay attacks. If the communicating entities wish to take advantage of the 0-RTT mechanism, they should provide their own replay protection at the application layer. A 0-RTT handshake is depicted in Figure 3.7.

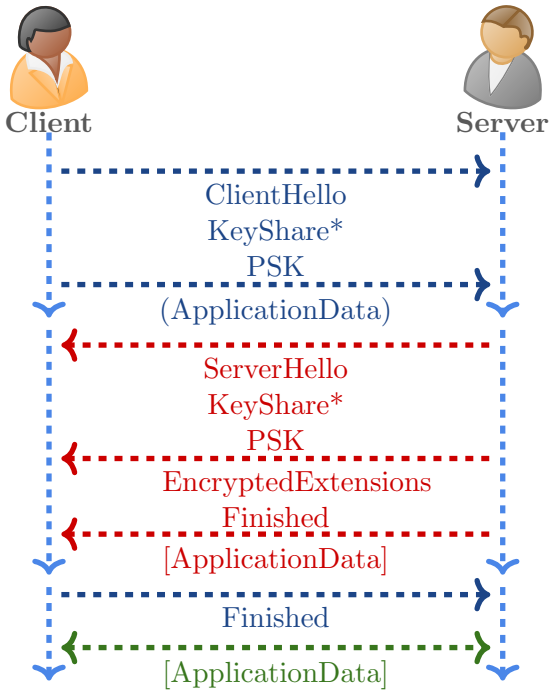


Figure 3.7: A 0-RTT handshake (Section 3.2.7)

Post-handshake client authentication

After a successful handshake, the server can send a `CertificateRequest` message. If the client responds with an acceptable certificate, then the server might authenticate the client. However, because the specification allows certificates to be rejected 'silently', the client cannot be sure of its authentication status in general. We discuss this in greater detail in Section 3.5.2.

Key update

After a successful handshake, either party can request an application data key update. Because the read and write keys for application data are independent, either party can immediately update their write key after requesting a key update. They must wait for a response from their peer before updating their read key, because the peer may have messages in flight under the old key. The write key on the other hand can be updated immediately because TLS provides in-order delivery of messages, and thus once the peer

has received the key update message it can be sure that all data that follows was encrypted under the new key.

Key derivation

A TLS 1.3 handshake will generate a set of keys on which both the client and server agree. The specification defines a key schedule which uses the repeated application of an HKDF [RFC5869] to combine the secret inputs with fixed labels so as to generate a set of independent keys.

The key schedule has two secret inputs, the (EC)DHE and the PSK. Depending on the handshake mode, either one or both of these will be used. The key schedule also includes the transcript hash in the key derivation. Because the transcript includes nonces, even if the secret inputs are repeated, the generated keys are guaranteed to be independent.

3.2.8 The security design of the TLS 1.3 handshake

Nonces

Both the client and the server send a nonce in their first message. A nonce is an unpredictable fresh value that is echoed back to the sender. When a client or server receives a nonce they created back from their peer they can be sure that their peer is active, and that an adversary is not just replaying messages from a previous session. In TLS 1.3 neither the client or server echo back the nonce directly. They instead compute a shared secret that is dependent on the nonces established. This means that if they agree on the shared secret at the end of the run, then they agree on the nonces used.

DHE

A DH key exchange establishes ephemeral session keys. The purpose of this key exchange is to prevent a passive adversary from being able to decrypt sessions even if it has compromised the long term (i.e. static) keys of one of the parties.

If the DH problem is hard then a passive adversary cannot derive the session secrets simply from observing the handshake because knowing g^x and g^y, which both appear in the handshake, gives the adversary no advantage in computing g^{xy}, the ephemeral key.

In TLS 1.2 there are cipher suites that did not have a DHE. These modes are commonly used in industry to monitor encrypted connections to a server. These were all deprecated in TLS 1.3. This was a controversial decision, and the only non-ephemeral mode of TLS 1.3 is the OOB-PSK mode. We discuss this at length in Chapter 6.

Transcript hashes

To ensure that messages are not modified in transit an accumulating hash of the transcript to date is computed by both sides. By ensuring both parties agree on this hash the two parties can be sure that they agree on the contents of all the messages, and that nothing has been modified by an adversary. Although the `ClientHello` doesn't include a transcript hash, because the hash is cumulative, agreeing on the hash by the end of the protocol gives the `ClientHello` message integrity protection ex post facto. In PSK modes the PSK binders provide integrity protection also. This protects any early application data.

This relies on hash functions having second-preimage resistance, i.e. that it is computationally infeasible to find two distinct inputs that hash to the same value. If an adversary can find such inputs then it could potentially cause the client and server to agree on the transcript hash, but have different views of protocol run. This would be a session synchronisation attack, where two different sessions output the same keying material. We discuss this at length in Chapter 4.

Labels

Another potential route of attack is that an adversary could takes values produced in one place and transplant them in another. This could be any value generated by an honest party, from entire blocks of messages, to single nonces, or even keys. For example the `<Certificate, CertificateVerify,`

`Finished>` message pattern can be produced in other contexts, see Chapters 4 and 5. If an attacker were able to transplant messages from one protocol to another it could potentially use some other protocol to acquire vaild messages that were intended for a different context, and by using them in the TLS handshake break the security guarantees. For example, by legitimately acquiring a `<Certificate, CertificateVerify, Finished>` message block from some other context it might be able to impersonate the Certificate owner. To prevent such messages being misused in a TLS run *labels* are included. These labels are simply fixed strings or numbers, but by ensuring different labels are used in every place, and that such messages have integrity protections, such as MACs, an adversary cannot transplant them from one place to another.

The same logic is applied to keys. To prevent an attacker who can acquire some keys from being able to derive other keys, keys are hashed with a contextual label before use. This renders the keys used in different contexts independent, i.e. knowing one key gives the adversary no advantage in deriving others.

This relies on hash functions having preimage resistance, i.e. that given the output of a hash function it is computationally infeasible to find a value produces that output.

3.2.9 Stated goals and security properties

The TLS 1.3 handshake protocol is intended to negotiate cryptographic keys by defining an authenticated key exchange (AKE). These keys can then be used by the record layer to provide critical security guarantees, including confidentiality and integrity of messages. As stated in Section 3.2.1, TLS 1.3 makes use of independent keys to protect handshake messages and application data messages: protection of the handshake messages starts with the server's `EncryptedExtensions` message, and in the majority of handshake modes, protection of application data messages occurs after the transmission of the server and client `Finished` messages, respectively. In the case of a 0-RTT handshake, early application data is protected with a PSK as part of the client's first flight of messages.

The TLS specification [RFC8446, Appendix E.1] lists eight properties that the handshake protocol is required to satisfy:

1. **Establishing the same session keys**. Upon completion of the handshake, the client and the server should have established a set of session keys on which they both agree.

2. **Secrecy of the session keys**. Upon completion of the handshake, the client and server should have established a set of session keys which are known to the client and the server only.

3. **Peer authentication**. In the unilateral case, upon completion of the handshake, if a client `C` believes it is communicating with a server `S`, then it is indeed `S` who is indeed executing the server role. An analogous property for the server holds in the bilateral (mutual) authentication case.

4. **Uniqueness of session keys**. Each run of the protocol should produce distinct, independent session keys.

5. **Downgrade protection**. An active attacker should not be able to force the client and the server to employ weak cipher suites, or older versions of the TLS protocol.

6. **Perfect Forward Secrecy (PFS)**. In the case of compromise of either party's long-term key, sessions completed before the compromise should remain secure. This property is not claimed to hold in the PSK key exchange mode.

7. **Key compromise impersonation (KCI) resistance**. Should an attacker compromise the long-term key of party `A`, the attacker should not be able to use this key to impersonate an uncompromised party in communication with `A`.

8. **Protection of endpoint identities**. The identity of the server cannot be revealed by a passive attacker that observes the handshake, and the identity of the client cannot be revealed even by an active attacker that is capable of tampering with the communication.

We model six out of the eight required properties, omitting downgrade protection and the protection of endpoint identities. Also, as stated previously, 0-RTT mechanisms allow for replay of early data across sessions. We discuss the reduced 0-RTT security properties as well as the properties described above more fully in Section 3.4.

The draft specification refers to RFC 3552 [RFC3552] for an informal description of the TLS 1.3 threat model. This model assumes a Dolev-Yao attacker [DY83]– an attacker that can perform MITM attacks by being able to replay, insert, delete, and modify messages at will. We consider a strictly more powerful attacker, as we will explain in Section 3.4.1.

3.3 Modelling the protocol

3.3.1 The Tamarin prover

The Tamarin prover [Sch+12] is a symbolic modelling and analysis tool for security protocols. Its specification language facilitates the construction of highly detailed models of security protocols, their security requirements and powerful Dolev-Yao-style attackers. The verification algorithm of Tamarin is based on constraint solving and multiset-rewriting techniques, which allows its users to prove intricate security properties in complex protocols exhibiting branches and loops. Moreover, it offers state-of-the-art symbolic Diffie-Hellman support. Tamarin inherently supports non-monotonic state and it includes an extensive graphical user interface that enables the visualisation and interactive construction of proofs.

These features make Tamarin a good fit for the modelling and in-depth analysis of highly complex protocols such as TLS 1.3. In particular, the support for branching allowed us to model the decisions that the protocol participants can make during execution, the loops were instrumental in covering repeated connections within a single session, and the main security aspects of TLS 1.3 critically depend on Diffie-Hellman key exchange. The non-monotonic state support enabled us to model branching without having to resort to custom-tailored hacks or having to rely on the considerable over-approximation where all branches can be considered simultaneously.

Lastly, the visualisations of attacks found by Tamarin provided us with a way to quickly identify potential problems, with either the protocol or our model– the graphical user interface was a great asset in guiding our TLS 1.3 verification workflow.

We defer the details of our Tamarin model to Section 3.4, and note differences to other TLS 1.3 models in the next section.

3.3.2 A comprehensive model

Using Tamarin's modelling framework we devised a comprehensive symbolic model of TLS 1.3 that captures the specified protocol behaviours, as well as unexpected behaviours that arise from a complex interaction of an unbounded number of sessions. Our model captures these behaviours in the presence of a powerful adversary.[5]

Other TLS 1.3 analyses consider the constituent parts of TLS 1.3, viewing these as separate protocols, and proceed to tie the individual proofs together with a compositionality result. For instance, [BBK17] considers the resumption mechanism as a separate protocol in which both the client and the server take as input a symmetric value—the PSK. If the PSK remains unknown to the adversary in every execution of the resumption protocol, a gap remains to be filled before concluding that the full handshake always completes without the adversary knowing the PSK. This gap is filled by a manual compositionality proof. In our work, there is no need for such manual proofs; composition is trivially satisfied by our comprehensive model, as Tamarin considers all the possible interactions in proving each property.

Although our model undoubtedly draws from the Tamarin models described in [Hor16] and [Cre+16], we opted to model TLS 1.3 with a significant increase in fidelity to the draft specification. Such an approach resulted in an improved ability to capture the full functionality of TLS, as well as a broader class of realistic attacks. This class includes the coverage of complicated interaction attacks, such as the post-handshake client authentication attack in [Cre+16]. Additionally, by closely matching our model to the specification and allowing for an almost line-per-line comparison, we achieve full

[5] We defer discussion of our adversary capabilities to Section 3.4.1.

transparency regarding which parts of the specification we abstract away from, and which assumptions our modelling process relies on. We discuss the relation between our model and the RFC in detail in Section 3.6.

Not only is our model more comprehensive than the Tamarin models that precede it, it also incorporates the many changes to the TLS 1.3 specification that have materialised since the development of these models. In the following sections, we describe our modelling process, pointing out enhancements over the previous models.

3.3.3 Closely modelling the specification

As with previous models [Cre+16], we employ the use of Tamarin rules to model state transitions within the TLS 1.3 protocol. However, our state transitions are far more fine-grained and modular in comparison to [Cre+16], modelling the effective change in state as a result of transmission, receipt and processing of cryptographic parameters. For instance, a basic, initial TLS 1.3 handshake invokes up to 21 different rules and the associated state transitions before post-handshake operations can commence. These state transitions are depicted in Figure 3.8, and correspond to message flights and cryptographic processing as described in Section 3.2.1, Figure 3.1. The full state diagram can be found Appendix A.

We provide an example of one of our rules in Figure 3.9. This rule describes the sending of data. Although Tamarin provides a communication primitive in its domain-specific language we define our own primitive here.

There are two facts that are consumed on the left hand side of the rule. `SendStream(~tid, $actor, $peer, auth_status, app_key_out)` means that a connection exists. This connection has the ID `~tid` and runs from `$actor` to `$peer`. The `auth_status` indicates whether or not the the actor has been authenticated. Finally, `app_key_out` is the write key of the sender.

The `Send(~tid)` and `SendData(~tid, $actor, $peer, auth_status, ~data)` are the send events triggered by the firing of the rule. The `Send(~tid)` event simply records that something has been sent on the channel with ID `~tid`. This simpler action allows us to write easier to read rules when not

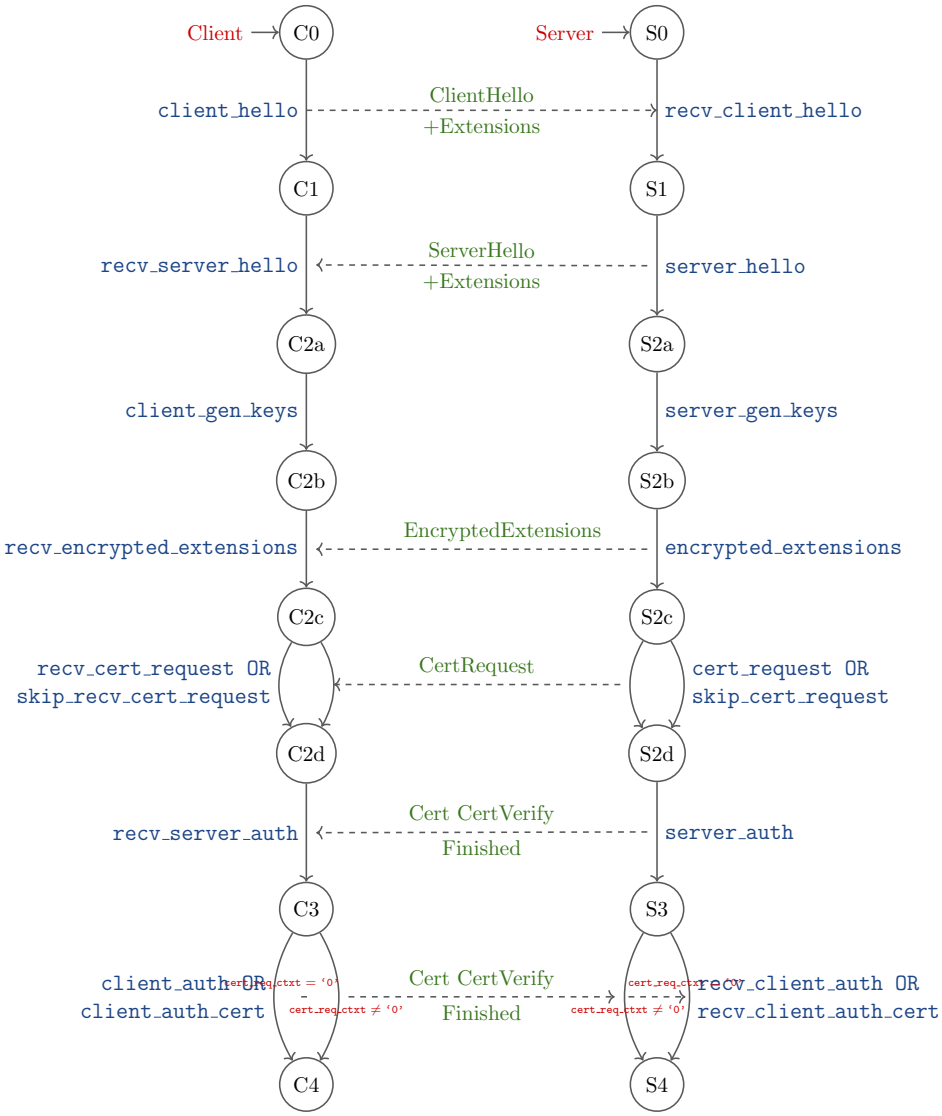


Figure 3.8: Partial state diagram for full TLS 1.3 handshake. Tamarin rules are indicated in blue. The messages exchanged between entities are given in green. Our full model contains many more transitions. We omit these here for the sake of simplicity.

```
1  rule send:
2      [ SendStream(~tid, $actor, $peer, auth_status,
       ↪  app_key_out),
3        Fr(~data)
4      ]
5    --[ Send(~tid),
6        SendData(~tid, $actor, $peer, auth_status, ~data)
7      ]->
8      [ SendStream(~tid, $actor, $peer, auth_status,
       ↪  app_key_out),
9        Out(senc{data_record(~data)}app_key_out)
10     ]
```

Figure 3.9: The `send` rule of our Tamarin model of TLS

all the fields are important. The `SendData` event records that data has been sent on the channel, along with the data that has been sent.

On the right hand side of the rule we repeat the `SendStream` fact so that the stream is not consumed and may be used again by subsequent rules. Note that this is not a persistent fact, but a fact that is consumed and recreated. This means we can always be sure of the order of any two send messages. A persistent fact would allow for multiple messages to be sent concurrently. The `Out(senc{data_record(~data)}app_key_out)` fact captures that data has been sent to the network using Tamarin's `Out` primitive.

Tamarin's `In` and `Out` primitives capture receiving and sending a message from and to the network respectively. However, this primitive describes an entirely unsecured channel with a Dolev-Yao attacker. The TLS record layer defines how messages are to be transported across such a channel in such a way as to create a secure channel. As can be seen in line 9 we use the `Out` primitive inside our `send` rule. In this case we wrap the data inside a special `data_record` frame. This allows the recipient to distinguish between handshake messages and data messages.[6] We then encrypt the framed data with a key we refer to as the `app_key_out` key. This key refers to the write key of the sender.

[6] Alert protocol messages are wrapped in yet another frame type.

We also note the extensive use of macros in our model, which is enabled by the `m4` preprocessor and allowed us to cover most of the specification, whilst syntactically keeping our model close to it. For example, our `ClientHello` message is a macro that expands to:

```
handshake_record('1',
    ProtocolVersion,
    ClientRandom,
    '0', // legacy_session_id
    $cipher_suites,
    '0', // legacy_compression_methods
    ClientHelloExtensions)
```

which reflects almost exactly how it is written in our Tamarin files. `ClientRandom` is itself another macro, defined to be the value of the client nonce `nc`. In Tamarin's syntax, constants are enclosed by single quotes. Constructing the model in this fashion enables a direct syntactic comparison to the specification. In Section 3.6 we show this comparison, and link to our website where we perform this comparison for our entire model. Previous Tamarin models also employ macros, but the connection to the specification is much less evident. For instance, in Cremers et al. [Cre+16] `ClientHello` is defined to be the pair of values `nc,pc`, representing the client's nonce and "parameters", which serves as a placeholder for handshake values that are abstracted away.

In our model we have tried to define cryptographic components in a way that is reminiscent of imperative programming. As in the specification, we compute the handshake secret using the function `HKDF-Extract(gxy,es)`, and the handshake keys are computed by applying a `Derive-Secret` function to this value. This is not strictly necessary due to the assumption of perfect cryptography, but it makes it easier to connect our model to the specification.

3.3.4 Advanced features

In our model we capture a number of complicated interactions and logic flows inherent to the TLS 1.3 handshake, greatly improving on preceding

models, adding features to the model which we consider to be 'advanced'. In particular we discuss group negotiation and handshake flows.

Group negotiation

We model the client and the server as having a limited ability to negotiate the group used in the Diffie–Hellman key exchange.

In Tamarin, any value can be used as a group generator. Typically, the fixed (public) constant `'g'` is used, which represents all parties agreeing to use a single group ahead of time. On receiving a key share and storing it in the variable `gx`, we simulate checking that the element resides in this group by pattern matching the value as `'g'^x = gx`. Intuitively, this corresponds to checking that $\exists x \,.\, g^x = gx$.

In Tamarin's syntax, variables that are always instantiated with public values are prefixed by `$`. In our model, the client starts with a pair of public values `$g1,$g2` that represent two supported groups, and offers these to the server along with a corresponding key share for `$g1`. Similarly, the server starts with a supported group `$g`. The model allows the server to return a `HelloRetryRequest` to the client, enforcing that `$g` is not equal to `$g1`, and expects the client to return instead a key share that matches `$g2`.

This interaction enables a much greater coverage of DH key exchange with respect to previous models, and opens up the possibility of future extensions to this work. One such extension would be to model a weak group by permitting the adversary to reveal the corresponding DH exponents.

Handshake flows

One of the most complex elements inherent to modelling TLS 1.3 is the vast number of possible state machine transitions. After a session resumption, the server can choose between using the PSK only, or using the PSK along with a DH key share. Alternatively, the server might reject the PSK entirely, and fall back to a regular handshake, or request that the client use a different group for the DH exchange. Additionally, there are several complex messages that can be sent in the post-handshake state: client authentication requests, new session tickets, and key update requests.

Since all of the above interactions can happen asynchronously, the resulting model becomes very complex and requires sophisticated handling logic. A number of complicated protocol flows, involving any number of sequential handshake modes and post-handshake extensions can, and will, transpire and we deal with this eventuality by modelling all possible handshake modes in a very modular fashion.

For example a client may connect unilaterally to a server, acquire a NST, upgrade the connection to a mutually authenticated bilateral connection using post-handshake authentication, then reconnect sending data in 0-RTT, before performing a second post-handshake authentication. Each of these stages has differing and complex security claims and guarantees. By creating a very modular model we avoid having to explicitly model every combination of modes and exchange modes.

For example the rules defining a 0-RTT handshake are entirely separate from the rules establishing post-handshake authentication, but by carefully maintaining the necessary state throughout we ensure that they can interact with each other correctly, despite neither knowing of the other's existence. Other models are, by and large, not capable of capturing complicated protocol flows. This also enables us to add and remove modes for testing very easily.

3.4 Encoding the threat model and the security properties

3.4.1 Threat model

We consider an extension of the Dolev-Yao (DY) attacker [DY83] as our threat model. The DY attacker has complete control of the network, and can intercept, send, replay, and delete any message. To construct a new message, the attacker can combine any information previously learnt, e.g., decrypting messages for which it knows the key, or creating its own encrypted messages. We assume perfect cryptography, which implies that the attacker cannot encrypt, decrypt or sign messages without knowledge of the appropriate keys. In order to consider different types of compromise, we additionally allow the attacker to do the following:

- compromise the long-term keys of protocol participants,
- compromise their pre-shared keys, whether created OOB or through a NST, and
- compromise their DH values.

Note that TLS 1.3 is not intended to be secure under the full combination of all these types of compromise. For example, session key secrecy can be broken by an attacker who eavesdrops on the communication and compromises the DH values of a single protocol participant.

A natural approach is to weaken the attacker model by adding realistic constraints until either the claimed security goals of the protocol are achieved, or the corresponding attackers become weaker than the ones we expect to face in practice. By starting from a strong attacker and explicitly stating the restrictions on its actions we can avoid the problem with the BAN logic model of the Needham-Schroeder protocol, namely that there was an implicit assumption that all LTKs keys were uncompromised. This workflow requires us to express, with high granularity, exactly what needs to be protected and when each of the claimed TLS 1.3 properties can be expected to hold.

We now give our formal definitions of the TLS 1.3 security properties mentioned in Section 2, noting where each property is covered in our model.

3.4.2 Security properties

We encode the claimed security properties of TLS 1.3 as lemmas in the specification language of Tamarin. Here we discuss the relationship between the lemmas we prove in the model, and the desired properties in the specification. We note that there is some overlap between the different handshake security goals of TLS 1.3 expressed in Section 3.2.9. For example, the requirement for PFS is effectively a modifier to the requirement for secret session keys. Where possible, we will prove these properties via distinct lemmas to aid in the comprehension of the model. However, it is also possible to combine many of the properties into a single, more complex lemma.

```
lemma secret_session_keys:
  "All tid actor peer write_key read_key peer_auth_status #i.
      SessionKey(tid, actor, peer, <peer_auth_status, 'auth'>,
      ↪ <write_key, read_key>)@i
    & not (Ex #r. RevLtk(peer)@r & #r < #i)
    & not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i)
    & not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i)
    & not (Ex resumption_master_secret #r.
    ↪ RevealPSK(actor, resumption_master_secret)@r)
    & not (Ex resumption_master_secret #r.
    ↪ RevealPSK(peer, resumption_master_secret)@r)
    ==> not Ex #j. K(read_key)@j"
```

Figure 3.10: secret_session_keys (Section 3.4.2). The `SessionKey` action is triggered at the end of a handshake. `Rev*` actions occur when the attacker reveals a key. The `K` action indicates that the attacker knows the value.

Establishing the same session keys

The definition of this first property is taken from Canetti et al. [CK01], where it is referred to as a consistency property. However, there is ambiguity in the circumstances that are necessary and sufficient for two protocol participants to establish the same keys. An answer to this question is typically given through the well-established practice of defining *session partnering* [BR93] [CK01] [LLM07]. One possible way to do so is to assign session identifiers in terms of a value (or pair of values) on which the two parties agree. We opted for the least restrictive session identifier, namely the pair of nonces generated by the client and the server. Therefore, if a *partnered* client and server complete the handshake, then they must agree on session keys.

Secrecy of the session keys

The `secret_session_keys` lemma is used to prove property 2 (secret session keys) in Section 3.2.9.

The `secret_session_keys` lemma we prove in Tamarin appears verbatim in Figure 3.10. The intuition for this lemma is that if an actor believes it has established a session key with an authenticated peer, then the attacker does not know the key. However, given the capabilities of the attacker, this will not hold without imposing some restrictions. This is why the additional

clauses (lines 4-8) are required. In Tamarin a lemma is a property on traces, i.e. potential runs of the protocol as described by the model. The additional clauses effectively exclude some traces from consideration, in particular those where the attacker performs some action which is beyond the scope of our threat model. The `Rev*` actions all refer to the attacker compromising a key.

Line 2 simply lists the variables over which we (universally) quantify, each given a name to indicate its purpose.[7] The five conditions stated in the lemma are generally repeated across all lemmas, and encapsulate the basic assumptions we make about our adversary. We describe them in more detail here: The first (line 4) imposes the restriction that the long-term signing key of the peer is not compromised.[8] This restriction can additionally be understood to signify that the actor is communicating with an honest peer, since the adversary can effectively simulate a party when in possession of its long-term key. We quantify over the time the attacker compromises the key, excluding traces where the attacker learns the peer LTK before time "`#i`". Time "`#i`" is the time where the actor has performed a `SessionKey` action, which corresponds to the actor completing a successful run of the protocol. Furthermore, it should be noted that the attacker is still allowed to compromise the peer's LTK *after* the session key is established. Hence we show that the session keys achieve PFS with respect to the LTK.

The second and third clauses (lines 5 and 6) ignore traces where the attacker reveals any DH exponents generated by the client or the server from before the session key was established. The attacker may reveal exponents that are generated after the session key is established.

The last two clauses (lines 7 and 8) specify that the adversary cannot compromise a PSK associated with either the actor or the peer. Note that the attacker is restricted from revealing these PSKs even after the session key has been established, which corresponds to the proviso in the specification

[7] These names have no significance to Tamarin, given α-renaming, but they do have implicit types, which are significant to Tamarin's solver.

[8] We remind the reader that both the client and the server are equipped with long-term signing keys, and the corresponding public key certificates, for the purposes of authentication.

that the PSK-only exchange mode does *not* provide PFS. We discuss this in more detail towards the end of this section.

Peer Authentication

The specification defines this property somewhat informally, as a form of authentication whereby both parties should agree on the identity of their peer. Looking at this more formally through the lens of Lowe's hierarchy of authentication [Low97], this definition corresponds to weak agreement, described in Definition 2.5.2. In particular, we note that this does not imply recentness—the requirement that the peer is currently running the protocol—nor does it specify whether any other values should be agreed upon.

We initially model this property via our `entity_authentication` lemma. Entity authentication is modelled in two parts so as to capture the distinction between the bilateral (mutual) and unilateral authentication cases. Authentication in the unilateral case means that if a client completes a TLS handshake, apparently with a server, then the server previously ran a TLS handshake with the client, and they both agree on certain data values of the handshake, including the identity of the server and the nonces used. Note that this is already a stronger property than the peer authentication property listed in specification, which doesn't require agreement on any values. This is because the same session keys property is actually an authentication property. Here we prove non-injective agreement on the nonces, which additionally provides recentness since both parties contribute a fresh nonce to the handshake. The unilateral entity authentication lemma we prove appears in Figure 3.11.

The intuition for this lemma is that if a client believes it has agreed on a pair of nonces with a server, then the server was, at some point prior, running the protocol with those nonces. The `CommitNonces` action occurs when the client completes a run of the protocol with the relevant nonces. The `RunningNonces` action happens when an actor, in this case the server, has seen both the client and server hello messages. We also require the client to have committed to the server's identity using the `CommitIdentity` action. We again find the necessary restrictions on the attacker to achieve this prop-

```
lemma entity_authentication [use_induction, reuse]:
  "All tid actor peer nonces client_auth_status #i.
      CommitNonces(tid, actor, 'client', nonces)@i
    & CommitIdentity(tid, actor, 'client', peer,
    ↪  <client_auth_status, 'auth'>)@i
    & not (Ex #r. RevLtk(peer)@r & #r < #i)
    & not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i)
    & not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i)
    & not (Ex resumption_master_secret #r.
    ↪  RevealPSK(actor, resumption_master_secret)@r & #r <
    ↪  #i)
    & not (Ex resumption_master_secret #r.
    ↪  RevealPSK(peer, resumption_master_secret)@r & #r < #i)
  ==> (Ex tid2 #j.
          RunningNonces(tid2, peer, 'server', nonces)@j
        & #j < #i)"
```

Figure 3.11: entity_authentication (Section 3.4.2).

erty. The property can only hold if the attacker does not acquire any of the secrets prior to the client agreeing on nonces. While one might expect that only the legitimacy of the signing key is necessary for authentication, if the adversary is able to obtain the PSK through compromising cryptographic material, or the PSK directly, then the adversary is able to resume a session and impersonate the peer.

In addition to entity authentication, we consider a transcript agreement property, where the value agreed upon is a hash of the session transcript. This provides us with near-full agreement. However, there are a couple of notable omissions. Firstly, the protocol technically continues after the initial handshake, for example post-handshake authentication can occur after the initial handshake has completed. None of these delayed handshake messages are included in the session transcript. Secondly, we observed that the actors do not necessarily agree on the current authentication status of the handshake. Whilst each actor can be certain of the authentication status of its peer, the client cannot be certain what the server believes its authentication status is. We cover this case in more detail in Section 3.5.2.

Finally, we also prove an injective variant of mutual transcript agreement, which TLS naturally achieves by agreeing on fresh nonces. Hence, we show that TLS achieves a relatively strong authentication notion: mutual agreement on a significant portion of the state with recentness.

Uniqueness of the session keys

We prove in the straightforward way that for any two session keys generated, if they match then they must be from the same session. This holds without any restriction on the adversary, since it is a straightforward consequence of the actor generating a fresh nonce for each session. We do not prove anything about whether two session keys are related, since this trivially follows from the assumption of perfect cryptography.

Downgrade protection

The specification cites the work by Bhargavan et al. [Bha+16a] for downgrade protection. This definition is not directly equivalent to any of Lowe's classical agreement methods; it only requires that both parties negotiate the *same* configuration parameters that they would do without the presence of an adversary. Specifically, we observe that in our model agreeing on the parameters (in the sense of non-injective agreement) is sufficient to achieve this, but not necessary. In the case of a single handshake we prove agreement on the transcript, and thus the attacker cannot add or remove any parameters. However the common method for causing downgrades is to break handshakes that use one set of security parameters, which cause one party to use another set of, usually weaker, parameters to try and achieve compatibility. Because we do not have any dependencies between handshakes in our model this attack cannot occur. Therefore, *within our model* we prove that TLS achieves downgrade protection through our authentication lemmas.

However, we note that this does not accurately capture the spirit of downgrade protection, due to the fact that we assume all cryptographic primitives are perfect and we do not model previous versions of TLS. Further we don't model the real behaviour of trying different sets of parameters after a handshake failure, assuming that the client and server will always

```
1  lemma secret_session_keys_pfs:
2      "All tid actor peer role write_key read_key
       ↪ peer_auth_status psk_ke_mode #i.
3        SessionKey(tid, actor, peer, <pas, 'auth'>,
         ↪ <write_key, key_read>)@i
4       & running(Mode, actor, role, psk_ke_mode)@i
5       & not (Ex #r. RevLtk(peer)@r & #r < #i)
6       & not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r <
        ↪ #i)
7       & not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r <
        ↪ #i)
8       & not (Ex rms #r. RevealPSK(actor, rms)@r  & #r < #i)
9       & not (Ex rms #r. RevealPSK(peer, rms)@r  & #r < #i )
10      & not (psk_ke_mode = psk_ke)
11    ==> not Ex #j. K(read_key)@j"
```

Figure 3.12: secret_session_keys_pfs (Section 3.4.2). We highlight the differences with the secret session keys lemma, see Figure 3.10.

treat every handshake independently. We simply note that this property holds almost vacuously in our model, and make no claims about downgrade protection in the real world.

Forward secrecy with respect to long-term keys

The PFS property was briefly mentioned in the context of the long-term signing keys and the secrecy of session keys. However, in those cases, we did not cover the requirement for forward secrecy with regards to the PSK. We have an additional lemma `secret_session_keys_pfs` which captures that, in either a full DHE or PSK-DHE handshake, the secrecy of the session keys does not depend on the PSK remaining secret after the session is concluded.

To achieve this, we modify `secret_session_keys` depicted in Figure 3.10, to create `secret_session_keys_pfs`, depicted in Figure 3.12. We highlight the salient differences between the two definitions. By adding a condition for the key-exchange mode, `not psk_ke_mode = psk_ke` (line 10), we can loosen the restrictions on the adversary such that the `RevealPSK` action is only forbidden for time points before the session keys are established (lines 8 and 9). In proving this lemma, we show that the session keys are forward secure

after a DHE, even if the PSK is later compromised. This is the definition of PFS with respect to PSKs.

Key Compromise Impersonation (KCI) resistance

Observant readers will notice that the only restriction on compromising long-term keys is that the peer's LTK must not be compromised. None of our security properties rely on the actor's LTK being hidden from the adversary.[9] Applying this fact to the authentication properties, therefore, additionally shows that the protocol, as given in the draft specification, achieves KCI resistance.

3.5 Analysis and results

In this section we provide a detailed description of our analysis, including a discussion of our results and an exploration of an authentication anomaly uncovered by our work.

In general we find that TLS 1.3 meets the properties outlined in the specification that our modelling process was able to capture. We show that TLS 1.3 enables a client and a server to agree on secret session keys and that these session keys are unique across, as well as within, handshake instances. Our analysis shows that PFS of session keys holds in the expected situations, i.e., in the (EC)DHE and PSK+(EC)DHE handshake modes. We also show that TLS 1.3, by and large, provides the desired authentication guarantees in both the unilateral and mutual authentication cases. The situation in which this is not the case is covered in the section to follow.

We remind the reader that our model does not truly cover downgrade protection, or the protection of endpoint identities at this time. A treatment of downgrade protection across TLS protocol versions would require modelling the earlier versions of TLS in a way that is consistent with the TLS 1.3 model as developed here. To consider the downgrade protection of cipher suites, we would need to relax our current assumption of perfect cryptography through rules that, for instance, allow for an attacker to learn the

[9] A minor exception to this is that the adversary cannot use the actor's long-term key to impersonate the actor to themselves since in this case, the actor is also the peer.

payload of a particular kind of encrypted messages without knowing the key. In spite of the fact that these additional considerations would substantially complicate the model and the proof process, our model is perfectly suited to their inclusion and could form the basis of future work.

3.5.1 Positive results

We now present our results for TLS 1.3, commenting on our proof methods and findings.

Proof strategies

For models as complex as TLS 1.3, proving lemmas in Tamarin is a multi-stage process, and proving complex lemmas directly is often infeasible. For protocol models of this size the proof trees can become very large. Tamarin provides a number of features that allow complex proofs to be broken down into more manageable sections. Writing sublemmas provides hints to the Tamarin constraint solving algorithm, allowing it to solve complex sections of a larger proof directly, making the overall proof more manageable. For the TLS 1.3 model, we used several types of lemmas. Helper lemmas can be used to quickly solve repetitive sections of a larger proof without repeatedly unrolling the entire subtree. Source lemmas provide hints to the Tamarin engine about the potential sources of messages, reducing the branching of a proof tree.[10] Inductive lemmas instruct Tamarin to prove the lemmas inductively, allowing us to break out of loops in the protocol, which otherwise can produce infinite proof trees. Proving the main properties of TLS 1.3 required many helper lemmas, of all of these types.

The Tamarin engine can also use heuristics to auto-prove lemmas, which proved invaluable in quickly re-proving large sections of properties after making changes to the model. By investing time in writing auto-provable sublemmas, we could flexibly incorporate changes made to the specification without having to restart our analysis from scratch.

[10] We discuss Source lemmas in great depth in Chapter 5.

The more complex lemmas used in our analysis of TLS 1.3, however, required manual proving in the Tamarin interactive prover. We note that by manual proving in this context we mean manually guiding the Tamarin prover through a proof by using the Tamarin graphical user interface.

Using the `m4` preprocessor to generate restricted subsets of the model we were able to prototype lemmas in a simpler environment without expending unnecessary effort. To give an indication of the number of helper lemmas required, and the relationship between all of our lemmas, we have constructed a 'lemma map', displayed in Figure 3.13. The map also indicates which lemmas were auto-proved by Tamarin, and which ones needed manual guidance for Tamarin to prove them.

In total, the modelling effort represents approximately 3 months worth of work. However, the vast majority of that is the process of writing lemmas to break down the overall proving effort into smaller, autoprovable chunks. With these lemmas in place, proving the entire model takes about a week of work, and significant computing resources. The model itself takes over 10GB RAM just to load, and can easily consume 100GB RAM in the course of a proof. In one instance, an automatically-computed proof was almost 1 million lines long. Once the proofs have been produced, they can be verified in the space of about a day, although still requiring a vast amount of RAM.

Findings

We summarise our results in Table 3.1. For each property discussed in Sections 3.2 and 3.4, we indicate our findings. We use $*$ to indicate that the property holds in most situations. Cases in which the property does not hold to the expected degree, are covered in sections to follow. We also list the applicable Tamarin lemma(s).

3.5.2 Possible mismatch between client and server view

During the development of our model, and in particular the analysis of the post-handshake client authentication, we encountered a possible behaviour that suggested that TLS 1.3 fails to meet certain strong authentication guarantees.

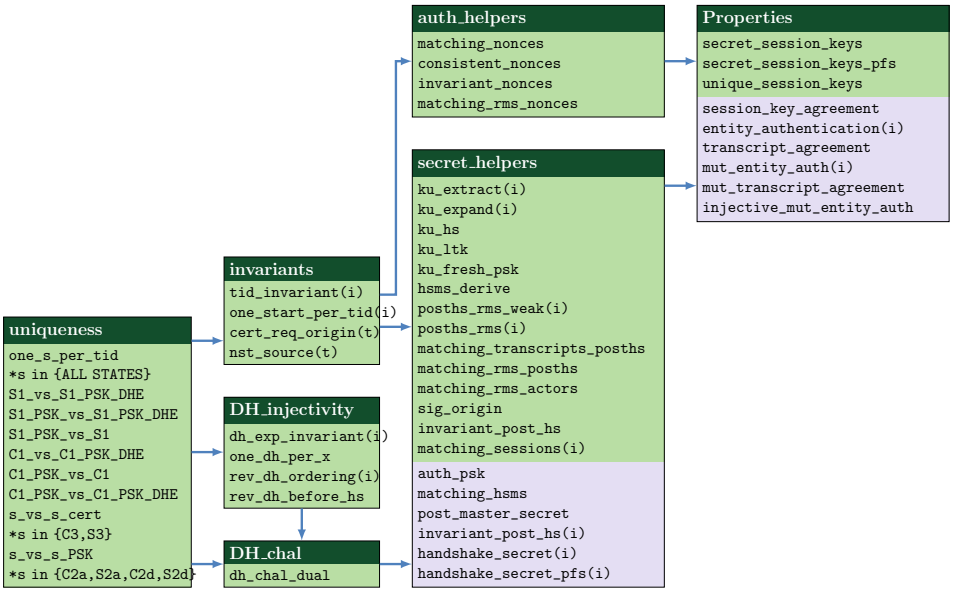


Figure 3.13: Lemma Map. Lemma names with a purple background indicate where manual interaction via the Tamarin visual interface was required. The remaining lemmas were automatically proven by Tamarin, without manual interaction. An arrow from one category to another implies that the proof of the latter depends on the former. The **Properties** box contains the main TLS 1.3 properties.

Property proven	**Lemma(s)**
(1) Same session keys	`session_key_agreement`
(2) Secret session keys	`secret_session_keys`
(3) Peer authentication*	`entity_authentication` `mutual_entity_authentication`
(4) Unique session keys	`unique_session_keys`
(6) Perfect forward secrecy	`secret_session_keys_pfs`
(7) Key compromise impersonation	`entity_authentication` `mutual_entity_authentication`

Table 3.1: TLS 1.3 Tamarin results

While there are many definitions of authentication, the common thread among strong authentication guarantees is that both parties share a common view of the session, i.e. that they agree on exchanged data, keys, etc. During our analysis of the post-handshake client authentication, it became apparent that the client does not receive any explicit confirmation that the server has successfully received the client's response. Due to the asynchronous nature of the post-handshake client authentication, the client may keep receiving data from the server, and will not be able to determine if the server has received its authentication message. As a consequence, the client cannot be sure whether the server sent the data under the assumption that the client is authenticated.

We formally modelled this property by adding a variable to the client and the server that records the current status of the connection, and in particular, if the connection is unilaterally or mutually authenticated. We discovered that even when the server asks for a post-handshake client authentication, and the client responds, the client cannot be sure that the server considers the channel to be mutually authenticated.

A discussion with the TLS 1.3 working group revealed that a similar problem exists within the main handshake. During the main handshake, the server can request a client certificate, and may decide to reject the certificate (for example because it violates certain domain-specific policies),

but still continue with the connection as if the certificate were accepted.[11] Therefore, the client cannot be sure (after what appears to be a main handshake with mutual authentication) that the server considers the client to be authenticated. Thus, this phenomenon leaves the client in the dark about whether or not the server considers it to be authenticated, even though the server asked for a certificate and the client supplied it.

To see why this may become a problem at the application level, consider the following application. Imagine a client and a server that implement TLS 1.3, where the server has the following policy: any data received over a mutually authenticated connection are stored in a secure database; all data received over connections where the client is not authenticated are stored in an insecure log. The client connects, the server requests a certificate, which the client duly provides, but the server rejects and continues regardless. Since the server rejected the certificate, it continues to store incoming messages in the insecure log. However, the client may assume it has been authenticated, and start sending sensitive data, which ends up in the insecure log.

The TLS working group has decided not to fix this behaviour for TLS 1.3, and has not introduced any mechanism that informs the client of the server's view of the client's authentication status. The TLS WG did not consider this risk sufficiently high to implement an acceptance mechanism at the TLS layer. If a client wants to be sure that the server considers it to be authenticated, this needs to be dealt with at the application layer. We anticipate that some client applications will incorrectly assume that sending a client certificate and obtaining further server messages indeed guarantees that the server considers the connection to be mutually authenticated. As we have shown, this is not the case in general, and may lead to serious security issues despite there being no direct violation of the specified TLS 1.3 security requirements.

In Chapter 4 we analyse the Exported Authenticators specification, which extends the certificate logic to the application layer. This behaviour is replicated at that layer. In Chapter 5 we introduce an extension to the EA

[11] This highlights the difficulty of analysing a specification as long and as complex as TLS 1.3. Despite a team of five people studying the specification in great detail for a number of years, none of us were aware that this behaviour was permitted.

specification that allows an actor to prove to its peer that it has accepted a given certificate.

3.6 The relation between our model and the TLS 1.3 specification

While there have been many academic analyses of various revisions of TLS 1.3 [BBK17] [Cre+16] [Dow+15] [AM16] [KW15] [Li+14] [Dow+16] [Fis+16] [Koh+14] [Hor16], they all (explicitly or implicitly) consider only part of the specification. Most analyses, even those that claim to be "complete" do not consider all possible modes, and many manual cryptographic analyses consider modes only in isolation (and not their interaction). This is caused by the inherent complexity of analysing TLS 1.3 and is not a problem in itself; rather, it justifies the need for multiple approaches.

However, we are of the opinion that readers, regardless of whether or not they are experts in the field, should be able to easily deduce the exact coverage of a given analysis. To ensure this, we provide an unprecedented level of transparency concerning the relationship between our model and the RFC (the draft specification) by creating a website [Cre+17b] that contains an annotated version of the RFC. We show an excerpt of our website in Figure 3.14.

In the excerpt, the left-hand side is a direct copy from the RFC, and the right-hand side contains our annotations. For example, they show how the concrete data structures of TLS 1.3 are mapped into abstract term structures. Additionally, we annotate the prose, describing the possible behaviours so as to indicate which Tamarin rules model them. The annotations also show exactly which details we do not model (and often list the reasons why).

We used these annotations ourselves during the development of our model to keep track of the parts of the specification that we had already modelled, and how we modelled them, which also simplified the task of keeping track of updates to the specification, something which proved incredibly useful given the rapid pace at which the draft specification would undergo changes.

```
Certificate Verify

This message is used to provide explicit proof that an endpoint possesses the
private key corresponding to its certificate and also provides integrity for
the handshake up to this point. Servers MUST send this message when
authenticating via a certificate. Clients MUST send this message whenever
authenticating via a Certificate (i.e., when the Certificate message is
non-empty). When sent, this message MUST appear immediately after the
Certificate message and immediately prior to the Finished message.

Structure of this message:

%%% Authentication Messages

   struct {
       SignatureScheme algorithm;
       opaque signature<0..2^16-1>;
   } CertificateVerify;
The algorithm field specifies the signature algorithm used (see for the
definition of this field). The signature is a digital signature using that
algorithm that covers the hash output described in namely:

    Hash(Handshake Context + Certificate)
---snip---
```

```
We compute the (server) signature as:

messages = <messages, Certificate>
signature = compute_signature(~ltkS, server)
where compute_signature expands to:

sign{<'TLS13_server_CertificateVerify', h(messages)>}
Since messages contains the handshake transcript up until that point, this is
valid for Handshake Context. We do not attempt to add the padding prefix
specified in the specification since it would have no purpose given our
assumption of perfect crypto.

The CertificateVerify message is simply defined as:

define(<!CertificateVerify!>, <!handshake_record('15', $sig_alg, signature)!>)
We do not currently model using different signing algorithms or their effects
on security.

The peer validates the CertificateVerify message by recomputing the signature
input, and enforcing the action Eq(verify(signature, sig_messages, pk(~ltkS)),
true) which makes the trace invalid if the verification fails (implying the
peer terminates the connection if receiving an invalid signature).

Note that an alternative way to model this in Tamarin would be to provide the
peer with the long-term key ~ltkA and pattern match the signature as an
expected message. While this can (probably) be shown to be equivalent and is
potentially more efficient for Tamarin, we believe using explicit verification
is clearer.
```

Figure 3.14: An excerpt of our website, showing how we annotated the specification. The full version can be found at [Cre+17b].

Our annotated RFC has a number of desirable features:

- Readers can check which parts we abstracted, and how, without having to reinvent the mapping between the Tamarin model and the RFC themselves. In other words, one can read through our website to see what is covered, and how it is covered, without having to understand Tamarin's formalism.
- If the specification is updated or changed, we can immediately track where the model should be changed.

We encourage other analyses of TLS 1.3 to follow a similar transparent approach, which would help the community to better understand which details from the specification might still need to be covered. We envision this will enable a faster convergence of confidence in all the details of the standard.

3.7 Conclusions

In this chapter we modelled the draft 20 of the TLS 1.3 specification within the symbolic analysis framework of the Tamarin prover, and used the tool to verify the majority of the security guarantees that TLS 1.3 claims to offer its users.

We focus on ruling out complex interaction attacks by considering an unbounded number of concurrent connections, and all of the TLS 1.3 handshake modes. We cover both unilateral and mutual authentication, as well as session key secrecy in all of the TLS 1.3 handshake modes with respect to a Dolev-Yao attacker. We also capture more advanced security properties such as perfect forward secrecy and key compromise impersonation. Our Tamarin model covers substantially more interactions than previous analyses due to its modularity.

Besides verifying that revision 20 of the TLS 1.3 specification meets the claimed security properties in most of the handshake modes and variants, we also discover an unexpected authentication behaviour which may have serious security implications for implementations of TLS 1.3. This unexpected

behaviour, at a high level, implies that TLS 1.3 provides no direct means for a client to determine its authentication status from the perspective of a given server. As a server may treat authenticated data differently to unauthenticated data, the client may end up in position in which its sensitive data gets processed as non-sensitive data by the server.

During the course of our analysis we also developed a line-by-line modelling aide that accurately captured which parts of the specification we were able to model, and which parts were abstracted. This artifact allows us to easily assess the faithfulness and coverage of our model, and also makes our model highly amenable to all kinds of extensions, especially with respect to the security properties and threat model. We expect that this artifact may serve as a comprehensive informational aide to academic researchers and well as the TLS Working Group.

Chapter 4

Exported Authenticators

4.1 Introduction

The use of TLS to protect traffic is becoming more and more ubiquitous[1]. As the push to encrypt more and more of the internet increases the more use cases TLS is expected to cover.

A draft was presented to the IETF that would allow certificates to be added to a TLS connection after it was established. The draft, `draft-sullivan` [Sul18a], describes a proposed optional feature for implementations of TLS 1.3 [RFC8446] and implementations of TLS 1.2 with extended master secret support [RFC7627]. The draft defines a protocol that binds additional certificates to an established TLS channel using exported authenticators (EAs). This allows either party to prove they have additional identities to those established in the TLS handshake.

The draft aims to supersede, and offer more flexibility than, both the renegotiation feature in TLS 1.2 and earlier, and the post-handshake client authentication feature in TLS 1.3.

The draft describes Exported Authenticators (EAs) as:

> "[...] a mechanism in Transport Layer Security (TLS) to provide an exportable proof of ownership of a certificate that can be transmitted out of band and verified by the other party."

[1] See `https://letsencrypt.org/stats/`

> `draft-sullivan` [Sul17, Abstract]

Informally, the goal of exported authenticators (EAs) is to allow one side of a TLS connection to prove to the other side that it controls a given certificate. In this case, an exportable proof of ownership means that the security of the proof is not weakened if the EA is public [RFC5056, p. 18].

This has two major use cases, providing a mechanism for post-handshake client authentication that interacts better with higher level protocols, such as HTTP/2, and allowing content distribution networks (CDNs) to send data from multiple domains they control over a single TLS connection. CDNs host content for services like Netflix, and deliver their content from locations often geographically closer to consumers. Amongst other benefits, this reduces lag for the client, and the amount of data that needs to be sent over long distance connections. Whilst this connection coalescing improves efficiency, it also dramatically changes the authentication guarantees of TLS.

Securely adding certificates to a connection is non-trivial, because unless there is a strong cryptographic binding between the TLS channel and the EA messages an attacker could potentially copy them from one channel to another. This could allow them to perform a credential forwarding attack, and thereby falsely claim to another party that they control a certificate, improperly authenticating themselves. `draft-sullivan` uses so-called *channel bindings* to protect against this. However, as we will show later, channel bindings are surprisingly difficult to get right.

4.1.1 Chapter overview

Our main contributions in this chapter are as follows:

1. `draft-sullivan` describes a number of informal authentication goals and use-cases. We specify the formal security properties that `draft-sullivan` requires to achieve its authentication goals.

2. We extend existing work on compound protocols, defining compound authentication for a larger class of protocols. Previous work describes compound authentication for protocols with both a key-agreement component and an identity tied to a LTK. We extend the definitions in the literature to describe new forms of compound authentication for protocols with either no key-agreement component or with no identity tied to a LTK.

3. We prove that `draft-sullivan` meets its security goals, proving that the TLS layer and the EA layer authenticate each other under a strong threat model. We prove this by building a symbolic model of `draft-sullivan`, and proving that it meets its security goals using the Tamarin Prover.

4. We analyse the security boundary for EAs to identify its precise assumptions and guarantees. Notably, our analysis reveals a security dependency between multiple EAs sent on the same channel. Using our extended definitions, we prove that under strong threat models multiple EAs sent on the same channel fail to authenticate each other, but under weaker threat models they do. While this does not violate any of the intended properties of EAs, it paves the way for establishing stronger properties.

4.1.2 Related work

Whilst `draft-sullivan` itself has not been analysed before, there is a large body of related work. `draft-sullivan` defines a layered protocol that uses channel bindings to securely add extra certificates to a TLS channel.

Channel bindings were proposed by Asokan et al. [ANN02], who found numerous flaws in layered protocols, and the requirements on channel bindings began to be standardised in RFC 5056 [RFC5056]. The properties needed to prove channel bindings secure[2] were proposed by Bhargavan et al. [BDP15], work which we extend here.

[2] We will specify precisely what we mean by 'secure' later in the chapter.

The work by Bhargavan et al. [BDP15] allows analysis of channel bindings for certain classes of protocol, which we extend to cover protocols like `draft-sullivan`.

As we mentioned earlier, `draft-sullivan` is intended to replace the functionality of renegotiation in TLS 1.2 and earlier. The renegotiation feature in TLS 1.2 and earlier was flawed [SR09]: an attack on the channel bindings broke the authentication guarantees. The feature has been deprecated in TLS 1.3 because it was considered to add too much complexity.

`draft-sullivan` is also intended to supersede post-handshake authentication in TLS 1.3. Cremers et al. [Cre+16] performed a formal analysis of a proposed version of the TLS 1.3 specification using the Tamarin prover, which found a vulnerability in post-handshake authentication that attacked the channel bindings.

The work by Bhargavan et al. [BBK17], also using formal analysis, found separate attacks on the channel bindings used for 0-RTT in draft 13 of the TLS 1.3 specification.

4.1.3 Motivation

`draft-sullivan` is more complex than the renegotiation mechanism that it intends to replace. Renegotiation allowed one certificate to be added, but not multiple certificates to be active at once. Renegotiation in TLS 1.2 was found to have a major vulnerability in its channel bindings, allowing an attacker to prepend data to a client's TLS connection. The feature was deprecated from TLS 1.3 because it was thought that the complexity it brought to the protocol brought more risk than the potential gains.

The draft is also more fully featured than post-handshake client authentication, which allows only the client to add a single certificate. However even the design of the simpler post-handshake mechanism was non-trivial. An early draft of TLS 1.3 [Res15] was found to have a vulnerability. Cremers et al. [Cre+16] performed a tool supported formal analysis, and found a credential forwarding attack where an attacker could use resumption to enable it to perform an improper post-handshake authentication. This attack was

highly complex involving 18 messages over 3 modes, and attacked a flaw in the channel bindings implementation.

If the proposed authentication mechanism was insecure the consequences would be significant. Not only could an attacker use this mechanism to impersonate some service, they could even potentially impersonate some group of services to clients or servers.

Cloudflare, a major CDN who claim nearly 10% of all internet requests[3], and the major browser vendors have all shown interest in implementing `draft-sullivan`. This combination of vulnerabilities in similar past protocols and the potential for widespread deployment led the IETF to delay the draft pending formal analysis. For TLS 1.3 the TLS WG adopted an analysis-prior-to-deployment philosophy, explicitly asking for input from academics. Following the success of formal analysis in raising issues that might otherwise have been missed, the trend of academia being consulted on the construction of complex protocols has continued [PM16]. As protocols become more complex, both in terms of features and security requirements, attacks become correspondingly more complex. Formal analysis, therefore, has become more important for finding problems before the protocol is widely deployed.

In this work, we prove `draft-sullivan` meets its security goals. To do this we develop a symbolic model of `draft-sullivan` and provide formal specifications of its security goals. We then analyse the resulting model and security properties using the Tamarin Prover [Sch+12].

This chapter details that analysis, which was presented at the IETF 101 meeting in London, allowing the draft to be progressed. The draft has now entered last call.

4.1.4 Chapter organisation

This chapter is organised as follows: In Section 4.2 we discuss background and related work and in Section 4.3 we introduce `draft-sullivan` EAs, and enumerate the informal security claims. In Section 4.4 we use a framework for analysing channel bindings to develop the informal security claims of `draft-sullivan` into formal security properties. In Section 4.5 we develop a formal

[3] `https://www.cloudflare.com/careers/departments/engineering/`

model for `draft-sullivan` in the framework of the Tamarin prover, and describe in Section 4.6 how we formalized the security properties within its framework. In Section 4.7 we discuss our results in the context of compound authentication, and conclude in Section 4.8.

4.2 Background

4.2.1 Formal analysis

Up to this point our approach to formal analysis has constructed security properties in terms of the security of primitives. For example we consider the guarantees we get in situations where an attacker can acquire a secret key. We could continue in this vein and simply analyse `draft-sullivan` and TLS as a single complex protocol. However, this tells us little about the relationship between the `draft-sullivan` protocol and TLS, only about their security as a single piece. We thus introduce a framework that lets us reason about the security properties of `draft-sullivan` in terms of the security of TLS, and vice versa. This lets us describe the guarantees we get in situations where, for example, an attacker can compromise TLS.

We use a channel bindings analysis framework [BDP15] to formalise the security properties we need. We then create a symbolic model of the protocol, encode the security properties in the framework of the Tamarin prover [Sch+12], and use it to prove that the properties hold for our model.

Channel bindings are used to secure a wide range of protocols, including various modes of TLS. Using a channel bindings analysis framework we are able to formalise the properties that protocols using channel bindings need to achieve their goals.

4.2.2 Channel bindings

Channel bindings are values produced at the end of a protocol run than can be used to cryptographically bind runs of authentication protocols together, in order to achieve stronger combined authentication properties. By agreeing on the channel binding of one run in the course of another it is possible to reason about the authentication status of both runs in terms

of the other. `draft-sullivan` uses the TLS exporter interface to construct channel bindings that are included in EAs to bind them securely to the TLS layer, allowing us to reason about the security of the EA in terms of the security of the TLS channel, and vice versa.

One of the earliest uses of channel bindings was to secure legacy protocols by running them inside more secure outer protocols. The naïve approach of simply running the legacy protocol over the secure layer is not necessarily more secure than simply running the legacy protocol.

For example, when discussing tunnelling insecure legacy protocols over unilateral TLS, Asokan et al. show that a MITM attack can occur even though TLS protects against MITM attacks. Consider the case where the inner protocol is sometimes also used outside the TLS tunnel. This scenario can occur in legacy environments where some equipment only supports insecure protocols. In this case the attacker can intercept an un-tunnelled run of the inner protocol, and then start a tunnelled session with the server, forwarding the messages from the client to the server, and unwrapping the responses and passing them back to the client. From the server's perspective it is running a protected version of the protocol, but the connection is being MITMed. There are numerous examples of insecurely layered protocols in Asokan et al. [ANN02], which discusses this topic at length. We illustrate this problem by returning to our example of the Needham-Schroeder protocol.

Example

Recall that the Needham-Schroeder protocol intends to provide bilateral authentication, but that it fails to properly authenticate the initiator (Alice) to the responder (Bob). Consider an application where the Needham-Schroeder protocol is run over unilateral TLS to protect it.[4] We assign Alice the role of the client, and Bob the role of the server. We can see that the unilateral TLS run does not protect the Needham Schroeder run.

[4] If bilateral TLS is used the protected version achieves bilateral authentication irrespective of the Needham-Schroeder run.

The attacker induces the client to run an unprotected version of the Needham-Schroeder protocol, and connects to the server over unilateral TLS. By simply passing the messages between the unprotected client and the protected server, the attacker can authenticate itself to the server using the flaw discussed in Section 2.6.1. Further, even if the protocol is only run over TLS, if the attacker can convince the client to perform a run with it, acting as a legitimate actor, the attack is still possible.

A solution

Asokan et al. proposed a solution to this type of attack, which was to cryptographically bind the inner and outer protocols, using channel bindings. By binding the two protocols together the insecure protocol is protected by the secure protocol. If the binding is well designed, values taken from protected runs of the insecure protocol would not be useable in unprotected runs.

Informally a channel binding is a unique 'name' for a protocol run. If two parties participate in a protocol run, and agree on the 'name' of that run, then they know they both participated in the same protocol run. Proving agreement on the channel binding prevents MITM attacks where the attacker maintains a separate session with each of the honest parties.

Formally, we define channel bindings as follows[5]:

Definition 4.2.1. Channel binding. A channel binding is a value produced at the end of a protocol run, such that no two protocol runs with different parameters produce the same value.

For example the channel bindings used by TLS 1.3 are based on a transcript hash. By comparing the hash of all the handshake messages, any discrepancy in either party's view of the transcript is captured, as it is infeasible to find two different transcripts that give the same hash. However,

[5]This definition is loosely based on the definition of unique channel bindings as defined in RFC 5056 [RFC5056], reformulated to ease discussion of channel bindings under Bhargavan et al.'s framework.

a simple transcript hash is not a robust channel binding. There are two obvious places where discrepancies in the parties' views might occur.

1. TLS 1.3 has post-handshake messages that are not included in the transcript, and therefore even if the parties disagree on the result of a post-handshake protocol they will compute the same transcript hash.

2. TLS 1.3 PSKs are not negotiated in the clear, but referenced by `PSK IDs`. If two parties agree to use the PSK with `PSK ID`, x, but, for example, the client thinks x relates to key y and the server thinks it relates to key z, then they will have different views of the handshake, but agree on the transcript hash.

Thus TLS 1.3 computes a more complex channel binding, that includes extra information to make the channel binding robust.

Example continued

Consider a case where, for some reason, the message format of the Needham Schroeder protocol cannot be changed, preventing us from using Lowe's fix.[6] If Alice and Bob at the end of the protocol agree on both the 'name' of the TLS channel and the Needham-Schroeder run, then the protections of the TLS run apply to the Needham-Schroeder run. This means that Bob knows that he agrees with Alice on the identity of the server in the TLS channel, i.e. that Alice believes she is talking to him.

One way to achieve this is for Bob to respond not with Alice's nonce in the second message, but with a hash of Alice's nonce and a TLS channel binding, *cb*.

$$A \to B : \{h(n_A, cb), n_B\}_{PK_A}$$

Because the attacker cannot decrypt the message it must pass it on to Alice. If Alice isn't running a protected version of the protocol then she will reject the message because the returned nonce is incorrect. If Alice is running a protected version of the protocol then she will reject the message because the returned hash is computed using a different TLS run.

[6] This might, for example, be caused by middlebox incompatibility, as in TLS 1.3, see Section 3.1.3

Channel binding complexity

Today, many authentication protocols are layered on top of each other and the layers bound using channel bindings, for example user authentication in SSH. By careful construction of the channel binding of each layer, and constructing the channel binding of each layer based on the channel binding of the previous layers it is possible to prove strong authentication properties of composite protocols. By strongly binding each layer together it is possible to achieve stronger guarantees than each of the protocols achieve alone. However channel bindings are difficult to construct correctly, leading to many attacks [ANN02] [BDP15] [SR09] [Cre+16] [BBK17].

In our case, if EAs were not bound to a specific TLS run, a client might connect to a malicious server and request an EA for a certificate the malicious server doesn't control. The malicious server could then connect to an honest server and request the appropriate EA, and pass it back to the client, effectively using the honest server as an oracle. The client would then receive a valid EA, but would not be communicating with its author.

Various works have studied channel bindings extensively, deriving a number of design principles. Channel bindings and their security goals are described in RFC 5056 [RFC5056]. Bhargavan et al. [BDP15] extend this work, and derive formal security properties that achieve these goals.

We will use and extend these works to look at the forthcoming EAs [Sul18a] specification and derive formal properties that we can prove hold.

4.2.3 Channel synchronisation

As mentioned in the previous section, when a weak inner protocol is protected by wrapping it in a strong outer protocol the protections of the strong outer run can sometimes be defeated. By forwarding values from an unprotected run of the weak protocol to a protected run, the attacker can attack the protected inner run. This is a credential forwarding attack.

Ostensibly it would seem that protocols that are always run inside another protocol, such as `draft-sullivan`, are immune from Asokan's attack. Whilst we show that `draft-sullivan` is indeed not vulnerable to this attack, the reasoning is subtle. Bhargavan et al. [BDP15] demonstrate that if

there is an attack on the channel bindings then an attacker can defeat this protection. If an attacker can exploit a flaw in the channel binding design to find two different runs of the strong outer protocol with the same channel binding, or 'name', then Asokan et al.'s credential forwarding attack is again possible.

Asokan et al.'s suggested mitigation for his attack works by making the channel binding produced by the protected inner run dependent on the channel binding of the outer run. If the protocol participants cannot get the inner channel binding to match then the protocol fails. This means that even if the same parameters are used for two runs of the inner protocol, if they are run on top of different outer protocol runs they will produce different channel bindings.

One way to achieve this dependency on the outer channel binding is to make the values output by the inner protocol dependent on the channel binding of the outer protocol. This is called an implicit channel binding. This can also be done explicitly by achieving agreement on the inner binding in some other way, for example by explicitly adding an authenticated copy of the channel binding into the last message. This is called an explicit channel binding. Explicit bindings require changes to the message formats of the inner protocol however, and are uncommon in retro-fitted protocols.

Channel bindings protect runs by causing a mismatch in the values of the inner protocol. However if two different runs of the outer protocol can be forced to have the same channel binding then protocols run inside each of the outer protocols will produce values transplantable between the two outer runs. This is a channel synchronisation attack.

A channel binding collision can occur when the channel binding is improperly constructed. In an improperly constructed channel binding not all relevant parameters are included in the channel binding's construction. For example the renegotiation attack [SR09] was possible because the context of previous handshakes was not included in renegotiated handshakes. Thus an ordinary handshake would have the same channel binding as a renegotiated handshake with the same parameters. This meant that the server could believe it was participating in a renegotiation, whilst the client believed it was

participating in an ordinary handshake. This allowed an attacker to prepend data to a client's connection.

A channel synchronisation attack uses a vulnerability in a channel binding to create a difference between the initiator's view of the protocol and the responder's view of the protocol. An attacker who can attack both the channel bindings and the inner protocol can successfully attack the composite protocol.

To briefly return to our running example, if an attacker can create two TLS sessions, one with Alice, where Alice is acting as the client and a separate one with Bob, where Bob is acting as the server, but where both sessions have the same channel binding, then our channel binding version is still attackable because Bob will compute the same value for $h(n_A, cb)$ as Alice, even though they are referring to two different TLS sessions.

Because many composite protocols rely on the security of outer layers to protect inner layers finding two different sets of parameters that synchronise the outer layers channel bindings often leads to an attack.

The failure to include all necessary information in the channel binding was behind the flaw found in post-handshake authentication in revision 10 of the TLS 1.3 draft [Cre+16], and the attack on 0-RTT in revision 12 of the TLS 1.3 draft [BBK17]. In the attack in [Cre+16] the channel binding of a resumed session was not dependent on the PSK used to authenticate the resumption, and thus an attacker could resume one session with the client, and another, different session with the server, and cause the resumed sessions to have the same channel binding. The attacker, now acting as a MITM, could forward a post-handshake authentication between the two channels, allowing it to act as an authenticated client to the server.

In TLS 1.3 a post-handshake authentication run does not change the keys used to encrypt the data sent on the channel, i.e. the authentication is not directly bound to the data. Thus after the post-handshake authentication the MITM attacker can insert data onto the channel and have it accepted as authentic, even though it never learns the client's private key.

In the attack in Bhargavan et al. [BBK17] the channel binding of a 0-RTT handshake was not dependent on the PSK used to authenticate it. A

0-RTT handshake sends authenticated data in the first flight of messages. A malicious server could decrypt that data, and re-encrypt it under a different PSK. The malicious server could then forward the re-encrypted data to any other server with which it shared a PSK, and have it appear as if it was coming from the original sender.

Both these attacks occur because the PSK of a previous handshake was not factored in to every sub-protocol that TLS 1.3 offers. Although the attacks, when viewed from this perspective, are similar, both were found with machine-aided formal analysis.

Although here we describe attacks on inner protocols, as we will discuss in Section 4.4.5, well constructed channel bindings can be used to protect outer layers as well as inner layers.

4.2.4 `draft-sullivan`

`draft-sullivan` was designed to offer features currently not available in the TLS 1.3 specification. TLS 1.3 deprecates its historical renegotiation mechanism, and instead provides a mechanism for post-handshake authentication, in which a server can ask a client to authenticate at any point after the normal handshake, thereby upgrading a unilaterally authenticated connection to a mutually authenticated one. `draft-sullivan` aims to supersede the functionality of renegotiation and post-handshake authentication. `draft-sullivan`'s EAs are designed to be more flexible than post-handshake authentication, allowing for authentication from either side (server or client, as opposed to only the client) and support for multiple certificates per side. It also aims to provide more control over when authentication happens to higher layers, and to provide more context for that authentication.

This is useful to HTTP/2 connections that want to reactively ask for a certificate when a client tries to access a particular resource. Due to the multiplexed nature of HTTP/2, a client might have multiple outstanding requests. Therefore a client cannot necessarily determine which request triggered the servers request for post-handshake authentication. Furthermore in TLS 1.3 post-handshake client authentication doesn't cause any change in the data channel, so the HTTP/2 layer doesn't know whether a given piece

of data was sent before or after the authentication. This makes reliably determining the authentication status of any given piece of data impossible.

`draft-sullivan` does not require state changes in the TLS state machine, and provides better context to higher layers, so an HTTP/2 connection would be able to tightly couple authentication with specific requests and responses.

The HTTP/2 specification explicitly prohibits the use of renegotiation with HTTP/2 [RFC7540, p. 67]. This means even in TLS connections that allow renegotiation, `draft-sullivan` is necessary.

As well as providing features that have proven a source of bugs in the past [SR09] [BDP15] [BBK17] [Cre+16], `draft-sullivan`'s unusual layering structure makes it an interesting protocol to analyse. Most layered protocols have a single run of the inner protocol within the outer protocol. `draft-sullivan` is unusual in that it allows multiple independent runs within a single TLS session. Each of these runs shares keying material from the TLS layer but is not otherwise cryptographically bound to other EAs on the same channel.

`draft-sullivan` defines two protocol flows for sending EAs. Both flows are heavily based on messages from the TLS 1.3 draft. This is because the message formats for TLS 1.3 were carefully designed to provide strong guarantees and have been well studied.

To prevent messages in similar formats being mistaken for one another, `draft-sullivan` and TLS.[7] include fixed strings in their key schedules that unambiguously and securely identify the context in which the message was created, effectively giving messages created in different contexts different types. These fixed strings are known as context strings in `draft-sullivan`, and constitute labels as defined in Section 2.4.11 In this particular case, they ensure that while EAs and TLS share keying material, (parts of) messages from one cannot be reused in the other.

[7] Both TLS 1.2 and TLS 1.3.

4.3 Exported Authenticators

In this section we define the EAs from `draft-sullivan`. The draft defines two protocol flows, "Requested Certificates" and "Spontaneous Certificates", which we describe in turn below. We also enumerate the security goals of `draft-sullivan`, which we formalise in Section 4.4.

4.3.1 Requested certificates

The first protocol flow follows a request-response paradigm. As shown in Figure 4.1a the initiator, which can be either the client or the server, sends a `CertificateRequest` message, as defined in [RFC8446, p. 60], and the responder replies with a series of messages. The responder sends the sequence `Certificate, CertificateVerify, Finished`. This series of messages is defined in the TLS 1.3 specification [RFC8446, p. 64, pp. 69-72], see Section 3.2.4 for a discussion of this message pattern.

`CertificateRequest`

The `CertificateRequest` message comes from the TLS 1.3 specification [RFC8446, p. 60], see Figure 3.3, and consists of a unique certificate request context (CRC) and a list of extensions that define the properties of the requested certificate. `draft-sullivan` says the CRC `SHOULD` also be unpredictable, which we follow.

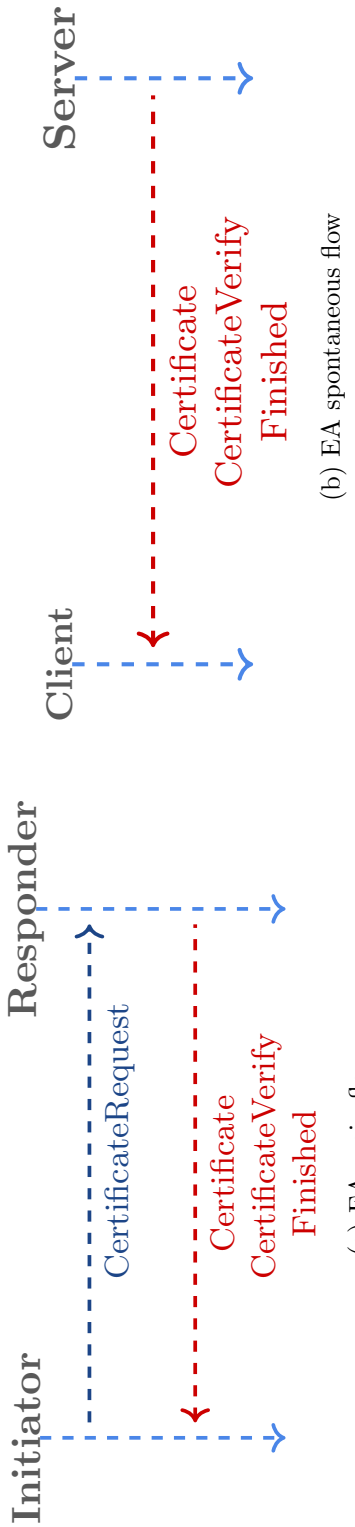
The response messages are defined as follows.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

Figure 3.3: `CertificateRequest` definition (repeated from page 61)

`Certificate`

The `Certificate` message also takes the form described in the TLS 1.3 specification [RFC8446, p. 64], see Figure 3.4. The message's CRC is carried



(a) EA main flow

(b) EA spontaneous flow

Figure 4.1: `draft-sullivan` protocol flows

over from the `CertificateRequest` message, and is paired with a certificate chain.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

Figure 3.4: `Certificate` definition (repeated from page 62)

`CertificateVerify`

The `CertificateVerify` message also follows the same structure as the TLS 1.3 draft, but signs a different value to ensure signatures cannot be transplanted from a TLS handshake to an EA handshake, or vice-versa. The struct given in the TLS 1.3 specification [RFC8446, pp. 69-71], see Figure 3.5, consists of the choice of algorithm and a signature.

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

Figure 3.5: `CertificateVerify` definition (repeated from page 63)

In TLS 1.3 the signature contains a context string and the value to be signed. In TLS 1.3 the value that is signed is the hash of all the previous messages, referred to as the "Handshake Context", concatenated with the Certificate message.

When the certificate is being signed by a server the context string used is "`TLS 1.3, server CertificateVerify`" and when the certificate is being signed by a client is "`TLS 1.3, client CertificateVerify`".

In `draft-sullivan` the signature contents are formatted in the same way, but the context string and the value to be signed are different. The context string used is "`Exported Authenticator`". The value that is signed is the hash of a value referred to as the "Handshake Context", and all previous messages.

In `draft-sullivan`, unlike in TLS 1.3, the "Handshake Context" does not refer to a hash of all the previous messages. The "Handshake Context" in this case refers to a special value computed using TLS's exporter interface. For convenience we will refer to `draft-sullivan`'s "Handshake Context" as the `HC Exporter`. The exporter interface is used to construct channel bindings, called exporters in TLS 1.3, for TLS sessions. The exporter interface is defined in the TLS 1.3 specification [RFC8446, p. 97], which we replicate in Figure 4.2.

```
TLS-Exporter(label, context_value, key_length) =
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
    ↪  "exporter", Hash(context_value), key_length)
```

Figure 4.2: The Exporter interface from [RFC8446, p. 97]

The exporter interface takes a label, and returns an exporter that is bound to the label, and the master secret and transcript of the *underlying TLS connection.*[8] An exporter is a value that uniquely identifies a TLS connection, and is unique and independent for each label.

To compute the `HC Exporter` the label is set to the string

`"EXPORTER-client authenticator handshake context"`

or

`"EXPORTER-server authenticator handshake context"`

depending on whether the `CertificateVerify` is being generated by the client or the server respectively. This makes `CertificateVerify` messages specific to a particular TLS connection.

[8] In `draft-sullivan` all `context_values` are of zero length.

`Finished`

In the same way that the `CertificateVerify` message follows the TLS 1.3 specification but with different parameters, so does the `Finished` [RFC8446, pp. 71-72] message. The `Finished` message consists of a single HMAC. The key used for the HMAC in `draft-sullivan` is different to the key used in TLS 1.3. In both TLS 1.3 and `draft-sullivan` the input to the HMAC is a transcript of all prior messages, even if they have yet to be sent. In `draft-sullivan` the `HC Exporter` is also included.

In TLS 1.3 the key used for the HMAC is the `finished_key` [RFC8446, p. 72]. `draft-sullivan` instead uses the `Finished MAC key`, which is computed using the exporter interface. The label in this case is set to

`"EXPORTER-client authenticator finished key"`

or

`"EXPORTER-server authenticator finished key"`

depending on whether the `Finished` message is being generated by the client or the server respectively. The other parameters remain the same as for the `HC Exporter` computation.

Both the `CertificateVerify` and the `Finished` messages rely on exporters from the underlying TLS layer.

4.3.2 Spontaneous certificates

The second protocol flow defined by `draft-sullivan` defines *un-requested* or spontaneous authenticators, see Figure 4.1b. The spontaneous flow is intended to allow the server to provide certificates that it knows that the client will require before the client requests them, effectively pre-loading the certificates.

For example if a CDN hosts both `thepiratebay.org` and `netflix.com`, and a user connects to The Pirate Bay (TPB) over TLS, the CDN will present a certificate for TPB. If a page on TPB was to contain thumbnails from Netflix, the CDN could spontaneously send a certificate for Netflix, allowing it to send the thumbnails over the same connection.

The message in this flow is much the same as the response to the `CertificateRequest` message in the Requested Certificate flow. One of the key differences is that only the server is allowed to initiate a spontaneous certificate flow, whereas in the requested certificate flow either party may take the role of the initiator.

A second difference is caused by the lack of `CertificateRequest` message. As a result, the server does not receive values for the `certificate_request_context` or the `extensions`. The server must therefore construct appropriate replacements for these values. The `extensions` define what types of certificate the initiator will accept in response, for example what signature algorithms to use.

Sending certificates that the client cannot process, for example if it doesn't support a particular signature algorithm, or won't accept loses all the benefits of pre-loading, and further is an expensive failure. To prevent this, the specification requires that the extensions used must be a subset of the `extensions` argument from the underlying `ClientHello` message in the TLS handshake. This gives the server a set of values that it knows the client *can* accept, even if the client might have chosen a different set of values when it requested the relevant certificate.

Historical development

Until the fifth version of `draft-sullivan` [Sul17] in a spontaneous flow the `certificate_request_context` was left out of the subsequent messages. This gave the spontaneous certificates a somewhat different set of security guarantees to the requested certificate flow. For example, if there is no `certificate_request_context` the client cannot detect replays. This difference of guarantees makes the analysis of this flow more challenging. However, this is not necessarily a security issue. For example in `draft-bishop` [BST17], which discusses using EAs to authenticate individual HTTP/2 streams for a single origin server, authenticators sent by the server are considered permanent for the lifetime of the session. A replay of an EA, therefore, has no effect.

Following consultation with the authors of the specification, in the sixth version of `draft-sullivan` [Sul18b] the server was allowed to choose the

`certificate_request_context` arbitrarily. This change makes it possible for a client to distinguish replays if the server uses a new value each time, but its main purpose was to allow for an extended version of EAs, which we analyse in Chapter 5.

At our request in the seventh version of `draft-sullivan` [Sul18a] the server is required to use a value unique to the session for the certificate request context. By requiring uniqueness only within the session it makes it possible for the client to determine freshness in half a round trip. To achieve global freshness in half a round trip the client would need to remember not only every nonce that had ever been sent to it on any channel, but also all nonces used amongst other parties. However because the nonces only need to be fresh within the session[9], the client simply needs to keep a record of nonces used on the TLS channel; a much more manageable task.

4.3.3 Security goals

`draft-sullivan` sets out a number of security goals.

1. "Authenticate one party of a Transport Layer Security (TLS) communication to another using a certificate after the session has been established."

2. "This proof of authentication can be exported and transmitted out of band from one party to be validated by the other party."

3. "Endpoints that are authoritative for multiple identities - but do not have a single certificate that includes all of the identities - can authenticate with those identities over a single connection."

4. "The application layer protocol used to send the authenticator SHOULD use TLS as its underlying transport to keep the certificate confidential."

5. "Authenticators are independent and unidirectional."

6. "The signatures generated with this API [...] cannot be transplanted into other protocols."

[9] The TLS channel provides global freshness.

In the next section we will formalise these goals into security properties that can be formally reasoned about.

4.4 Channel bindings

4.4.1 Methodology

To prove that `draft-sullivan` meets it security goals we take a formal analysis approach. We transform the protocol and the goals into a form we can formally reason about.

We take the work of Bhargavan et al. [BDP15] as a starting point. In particular, Bhargavan et al. suggest a number of properties that protocols based on channel bindings need to satisfy to be free from credential forwarding attacks, and in particular channel synchronisation attacks. We start from those properties. However, these properties have limited ability to describe protocols such as `draft-sullivan`. This leads us to extend the work of Bhargavan et al. to allow us to describe properties for a broader range of protocols. Using our extended properties we show that if `draft-sullivan` has these properties then it meets its goals.

We encode the protocol and the properties into the Tamarin specification language, which lets us use the Tamarin Prover [Sch+12], a protocol analysis tool, to prove that the protocol has the properties we want, and therefore meets its security goals.

4.4.2 Channel bindings security

Recall that a channel binding is a unique 'name' for a run of a protocol, and that by agreeing on the 'name' of the lower layer in an upper layer we can bind the lower layer to the upper. By constructing the 'names' of higher protocol layers based on the names of lower protocol layers agreement on the 'name' of the upper layer binds the run of the higher layer to a specific run of the lower layer. This gives us the ability to reason about the security guarantees of a lower layer run in the context of a higher layer. Further, by constructing the 'names' of higher protocol layers based on secrets es-

tablished in lower layers we can reason about the security guarantees of a higher layer run in the context of a lower layer.

Channel bindings provide a handle for reasoning about protocols in which an authentic channel is established inside another authentic channel. The goal of such reasoning is to prove that the actors at the end-points of the inner channel are the same as the actors at the end-points of the outer channel.

Bhargavan et al. formalised the security properties that composite protocols need to achieve. They define two properties, agreement and compound authentication, that together demonstrate that a layered protocol is secure, i.e. free from credential forwarding attacks.

Informally "Agreement", in this case, means that both parties agree on the identity of their peer and some data sent in the protocol.

Compound authentication, again informally, is the property that, in a series of layered authentications, as long as a single protocol run with an honest peer credential remains uncompromised then the same peer participated in all the protocol runs.

Intuitively this would mean that if the certificate in an EA was uncompromised or the master secret of the corresponding TLS channel was uncompromised, then the author of the EA and the TLS channel peer are the same actor.

4.4.3 Reasoning about channel bindings

To capture these properties formally Bhargavan et al. [BDP15] provide a framework for expressing the layers of a composite protocol in a uniform manner, which we enumerate here. This provides the language we will use to reason about `draft-sullivan`.

To define two-party authentication protocols Bhargavan et al. define the two principles or actors, $a, b \in Actors$, each of which has access to a set of public credentials,[10] $Creds := \{c_1, c_2, \ldots, c_n\}$. For each public credential,

[10] For example a certificate.

c_i, there is a corresponding secret, s_i, proving ownership of the credential.[11] We define the set $CredentialSecrets := \{s_1, s_2, \ldots, s_n\}$.

Bhargavan et al. instantiate six variables at the end of each sub-protocol run, called a protocol instance, l, as follows [BDP15, p. 3]:

- $p \in Actors$: the actor,
- $l \in \mathbb{N}$: a fresh locally unique session identifier,
- $role \in \{initiator, responder\}$: whether the actor played the role of initiator or responder (rather than client or server),
- params: public parameters each of which may be unassigned ($\perp$),
  - $c_i \in Creds \cup \perp$: the initiator credential
  - $c_r \in Creds \cup \perp$: the responder credential
  - $sid \in \mathbb{N} \cup \perp$: a global session identifier
  - $cb \in RunName$: a channel binding for the current instance
  - $cb_{in} \in RunName$ a channel binding for the previous / outer instance
- secrets: session specific secrets, with the following distinguished field:
  - $sk \in SessionSecrets \cup \perp$: an authentication key, which may be unassigned ($\perp$)
- $complete \in \{0, 1\}$: a flag that indicates whether the instance completed its role.

Note that sk is a symmetric key, and further that $CredentialSecrets \cap SessionSecrets = \emptyset$. When $c_i = \perp$ and $c_r = \perp$ we call the instance *anonymous*. When only one of them is assigned to $\perp$ we call the instance *unilateral*. When both c_i and c_r are assigned we call the instance *bilateral*. When $sk \neq \perp$ we call the instance *key-generating*. We call a peer credential, c_i, *honest* if $c_i \neq \perp$ and the corresponding secret, s_i, is unknown to the attacker.

[11] For example the private key of a certificate.

4.4.4 Threat model

The threat model in Bhargavan et al., and the model we use, is an extended Dolev-Yao attacker. This attacker is extended with the ability to compromise credentials with the event `Compromise(`c_i`)`, and local secrets with the event `Leaked(`sk`)`. Thus we say a protocol instance l has an honest peer credential if:

$$\text{Let } c_p = \begin{cases} c_r & role = initiator \\ c_i & role = responder \end{cases}$$
$$\text{in } (\neg\exists \# j \cdot Compromise(c_p)@j) \wedge c_p \neq \bot$$

We say the session secrets of l as have not been leaked if:

$$(\neg\exists \# k \cdot Leaked(sk)@k) \wedge sk \neq \bot$$

We assume the existence of a secure PKI that honestly issues certificates.

4.4.5 Security properties

Bhargavan et al. define two security properties that layered protocols need to have to achieve their security goals.

The first is Agreement. Bhargavan et al. use a variant of Lowe's non-injective agreement.

Definition 4.4.1. Agreement (from Bhargavan et al. [BDP15, Definition 1]). If a principal a completes protocol instance l, and if the peer's credential in l is honest, and if the session secrets of l have not been leaked, then there exists a principal b with a protocol instance l' in the dual role that agrees with l on the contents of *params* and any shared session secrets (most importantly sk).

In particular, l and l' must typically agree on each other's credentials, the session identifier *sid* and channel binding *cb*, and any negotiated cryptographic parameters. We do not explicitly state the confidentiality goal for secrets, but many derived authentication properties such as compound authentication implicitly depend on the generated sk being confidential.

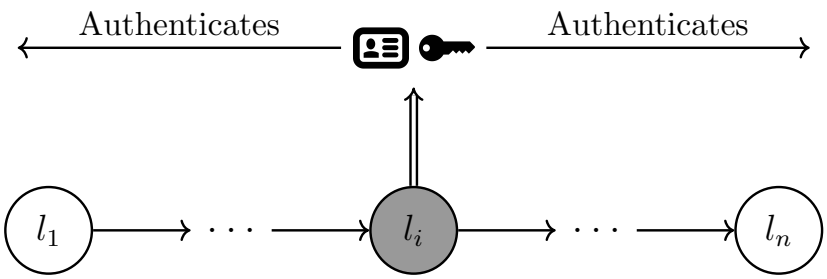


Figure 4.3: Compound Authentication says that if an instance l_i has an uncompromised secret key and an uncompromised identity then past and future instances are authentic.

Although Bhargavan et al. require agreement on the session identifier, they do not necessarily require injective, i.e. one-to-one, agreement on the number of sessions, two of a's protocol instances may pair with a single instance of b. This can happen when $sid = \perp$, as in the spontaneous certificate flow.

Definition 4.4.2. Compound Authentication (from Bhargavan et al. [BDP15, Definition 2]). If a principal a completes a compound authentication protocol consisting of protocol instances $\{l_1, \ldots, l_n\}$, such that some instance l_i has an honest peer credential and the session secrets of l_i have not been leaked, then there exists a principal b with protocol instances $\{l'_1, \ldots, l'_n\}$ such that each l'_j has the dual role to l_j and agrees with l_j on $params_j$ and sk_j.

This property says that if an actor, a completes a run of a composite protocol, ostensibly with b, and at least one layer of the protocol successfully authenticates b to a, then all layers of the protocol are authentic, see Figure 4.3. That layer can be said to authenticate all the others.

4.4.6 Expressing `draft-sullivan` in the channel bindings framework

To apply the security properties to `draft-sullivan`, we need to map each layer of the protocol onto the framework. To illustrate this mapping we first

assign each variable set by the framework to the relative element of the EA layer. The actors are the client and server, and the set of public credentials correspond to the set of certificates issued by the PKI. The secret keys of the certificates therefore correspond to the secrets that can be used to prove ownership of the corresponding credential.

The roles of initiator and responder go to the appropriate actor for each run. *params* is slightly different based on whether the EA is part of a Requested Certificate flow, Figure 4.1a, or a Spontaneous Certificate flow, Figure 4.1b.

For the requested certificate flow we define:
$params = (c_i, c_r, sid, cb, cb_{in})$ where

$$\begin{aligned} c_i &:= \perp, \\ c_r &:= Cert_r, \\ sid &:= \texttt{certificate\_request\_context}, \\ cb &:= EA, \\ \text{and } cb_{in} &:= \texttt{HC Exporter} \end{aligned}$$

where EA refers to the $\langle$`CertificateVerify`, `Finished`$\rangle$ messages, which are built with the TLS exporter and the private key of the responder's Certificate. Because the `CertificateVerify` contains the `certificte_request_context`, we know the channel binding must be unique within the TLS session.

In the spontaneous certificate flow we define:
$params = (c_i, c_r, sid, cb, cb_{in})$ where

$$\begin{aligned} c_i &:= Cert_i, \\ c_r &:= \perp, \\ sid &:= \perp, \\ cb &:= \text{EA}, \\ \text{and } cb_{in} &:= \texttt{HC Exporter} \end{aligned}$$

In this case the initiator is always the server, but there is no public global *sid*. The channel binding in this case is unique because the `Finished` message performs a hash over the ⟨`Certificate`, `CertificateVerify`⟩, which together include all the elements in `params`. Because we assume perfect cryptography any two different inputs to the hash will give different outputs

For both the EA flows we define the secret $sk := \perp$. The EA protocol doesn't establish any shared secrets, so there are no candidate values for sk. sk is an authentication key that both parties must agree on, and is implicitly required for compound authentication. We extend the work of Bhargavan et al., defining the security guarantees that can be achieved without a shared secret further in Section 4.4.7.

In both cases, because the authentication provided by EAs, per the specification, is not layered with other EAs, the outer channel is the TLS channel. We therefore define $cb_{in} :=$ `HC Exporter`, the channel binding exported from the TLS channel.

This means `draft-sullivan` does not try and prove joint authentication for two separate EAs. If we try to use `draft-sullivan` EAs to provide joint authentication, rather than authentication of each certificate individually they do not meet Bhargavan et al.'s definition of contributive channel bindings, see Section 4.4.10. We therefore would expect that they do not meet the security goal of compound authentication, see Definition 4.4.2.

We also need to define instances for the TLS layer. The actors are the client and server, as they are in the EA instances, because we are trying to prove they match. The roles are 'client' and 'server' respectively. For the TLS layer we define $params = (c_i, c_r, sid, cb, cb_{in})$ where

$$\begin{aligned} c_i &:= Cert_c \text{ or } \perp, \\ c_r &:= Cert_s, \\ sid &:= \perp, \\ cb &:= \texttt{HC Exporter}, \\ \text{and } cb_{in} &:= \perp \end{aligned}$$

It is worth noting that `HC Exporter` is derived from the `exporter_master_secret` which in turn is based on a complete transcript of the TLS handshake.

This binds all the parameters of the TLS session to the `HC Exporter`, so two sessions with different parameters must have a different channel binding. For TLS instances c_i is either $Cert_c$ or $\perp$ depending on whether the TLS channel is unilateral or bilateral.

Unlike EAs, TLS 1.3 does require injective agreement, however it doesn't operate on the basis of a public session ID, but based on matching sessions as defined in Canetti et al. [CK01].

Therefore we assign $sid := \perp$.

$cb_{in} := \perp$ because the TLS layer is the outermost layer we consider, so there is nothing for the TLS channel to bind to.

For the TLS layer we define the master secret as the shared secret, $sk := ms$.

4.4.7 Extending the definitions of Bhargavan et al.

If we try and formalise the properties we want from EAs using Compound Authentication we quickly run into problems.

Consider the case where a client connects to a server over unilateral TLS, and completes a run of the request-response flow as the responder. Under the definition of Bhargavan et al. the server cannot use the TLS layer to authenticate the EA because it does not have an honest peer credential. It also cannot use the EA to authenticate the TLS layer, because the session secrets of the EA layer are unassigned, and thus 'known' to the attacker.

Using this definition, we can only prove two properties.

1. If a client receives an EA on an uncompromised TLS channel, then the server controls the certificate in the EA.

2. If a server receives an EA on an uncompromised *bilateral* TLS channel, then the client controls the certificate in the EA.

This means that under this definition we cannot use EAs to reason about the security of the TLS channel, and that the server can only reason about EAs sent over a bilateral TLS connection.

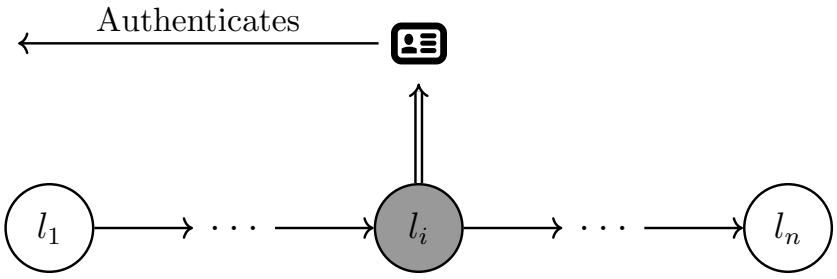


Figure 4.4: Outward Compound Authentication (OCA) says that if an instance l_i has an uncompromised identity then past instances are authentic.

We therefore extend the concept of Compound Authentication with two new definitions to capture stronger guarantees. Outward Compound Authentication, which authenticates outer layers / prior instances, and Inward Compound Authentication, which authenticates inner layers / subsequent instances.

Definition 4.4.3. Outward Compound Authentication. If a principal a completes a compound authentication protocol consisting of protocol instances $\{l_1, \ldots, l_n\}$, such that some instance l_i has an honest peer credential and the session secrets of l_i have been leaked or are $\perp$, then there exists a principal b with protocol instances $\{l'_1, \ldots, l'_i\}$ such that each l'_j has the dual role to l_j and agrees with l_j on $params_j$ and sk_j.

Intuitively, this means that if a protocol layer has an honest peer credential, then it can authenticate all outer layers, see Figure 4.4. This has a stronger threat model than standard compound authentication, because the attacker is allowed to compromise any session secrets, if there are any.

This stronger definition lets us capture the property we want from EAs, namely that if a client or server receives an EA with an uncompromised peer certificate then it was authored by the TLS peer.

If a composite protocol has outward compound authentication (OCA) for instance l_i then the same peer participated in all prior layers. Because we assume that the authentication at layer l_i was successful, i.e. the au-

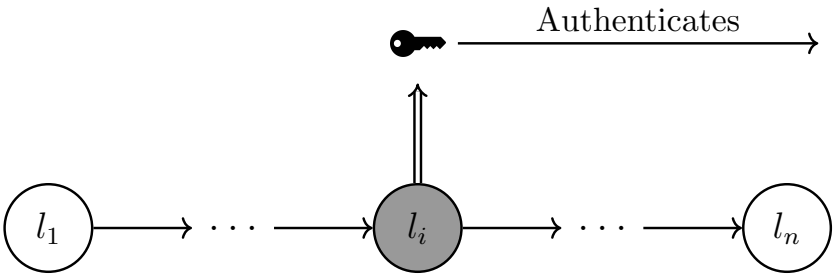


Figure 4.5: Inward Compound Authentication (ICA) says that if an instance l_i has an uncompromised secret key then future instances are authentic.

thentication at layer l_i was performed by an honest actor, we can be sure that there was no credential forwarding attack on prior layers. Consider for a contradiction that a credential forwarding attack exists on layer l_j where $j < i$, meaning the peer in layer l_j is dishonest. If the peer in layer l_i is honest, and the actor in layer l_i and the actor in layer l_j are the same actor, then the actor in layer l_j is honest. This is a contradiction.

We can similarly specify a property about unilateral and anonymous protocols.

Definition 4.4.4. Inward Compound Authentication. If a principal a completes a compound authentication protocol consisting of protocol instances $\{l_1, \ldots, l_n\}$, such that some instance l_i has no peer credential, or a leaked peer credential, but the session secrets of l_i are uncompromised, then there exists a principal b with protocol instances $\{l'_i, \ldots, l'_n\}$ such that each l'_j has the dual role to l_j and agrees with l_j on $params_j$ and sk_j.

This property says that once a secret key is established, the actor can be sure that any protocol layers run after it were indeed run by the same peer, see Figure 4.5.

If a composite protocol has inward compound authentication (ICA) for layer l_i, then the same peer participated in all subsequent layers. Because we assume that the protocol at layer l_i was successful, i.e. the authentication

at layer l_i was performed by an honest actor, we can be sure that there are no credential forwarding attacks on future layers. This follows the same logic as for OCA. Consider for a contradiction that there exists a credential forwarding attack on layer l_j where $j > i$, meaning that the peer in layer l_j is dishonest. Thus if the actor in layer l_i is honest, and the actor in layer l_i is the same actor as the actor in layer l_j then the actor in layer l_j must also be honest, a contradiction.[12]

This lets us describe the remaining property we want, namely that if a server receives an EA on an uncompromised *unilateral* TLS connection, then the TLS peer controls the certificate in the EA.

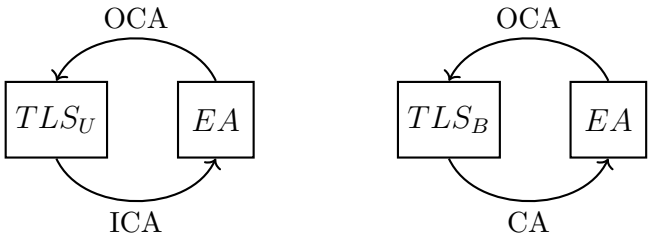
It may seem unintuitive that the actor cannot reason about earlier layers when the protocol has ICA, because many layered protocols use a key generating protocol as a base layer to achieve confidentiality as well as authentication. We illustrate this restriction through an example. Consider a protocol where a client uses his private key to sign a server's certificate, and sends this signed certificate to the server in the clear. The client then connects to the server using unilateral TLS and repeats the signed certificate. The server cannot be sure that it is talking to the client, and not some attacker who observed the certificate exchange in the clear. Whilst it knows that at some point the client wanted to communicate with it, it doesn't know that the TLS peer is that same client. However, as long as the TLS channel remains uncompromised, the server can be sure any compound authentication protocols run over it were indeed run by the TLS peer.

4.4.8 Formalising the properties

With these two new properties we can now work out what properties we want `draft-sullivan` to have.

We want to show that the EA layer has OCA, and that the TLS layer has compound authentication (CA), or ICA for the server in the unilateral TLS case.

[12] We assume here that an actor cannot become malicious during a composite protocol run.



(a) Unilateral TLS (b) Bilateral TLS

Figure 4.6: On a unilateral connection the server can authenticate the EA using the ICA property between unilateral TLS and EAs. On a bilateral connection the server can authenticate the EA using the compound authentication property between bilateral TLS and EAs.

Property 4.4.5. OCA of the EA layer with the TLS layer (using Definition 4.4.3). If an actor, a, with protocol instances $\{l_{TLS}, l_{EA}\}$ receives an EA, EA_p, in layer l_{EA}, and the attacker does not know the private key of the certificate in EA_p, i.e. l_{EA} has an honest peer credential, then there exists an actor, b, with instances $\{l'_{TLS}, l'_{EA}\}$ such that each l'_i has the dual role to l_i and agrees with l_i on $params_i$ and sk_i. Specifically EA_p was generated by the TLS peer and both parties agree on the TLS master secret.

Property 4.4.6. CA or ICA of the TLS layer with an EA layer (using Definition 4.4.2 and Definition 4.4.4 respectively). If an actor, a, with protocol instances $\{l_{TLS}, l_{EA}\}$ receives an EA, EA_p, in layer l_{EA}, and the attacker does not know the master secret of the TLS channel, i.e. the session secrets of layer l_{TLS} have not been leaked, then there exists an actor, b, with instances $\{l'_{TLS}, l'_{EA}\}$ such that each l'_i has the dual role to l_i and agrees with l_i on $params_i$ and sk_i. Specifically EA_p was generated by the TLS peer, and both parties agree on the TLS master secret.

It is interesting to note however, that in the ICA case this definition only assures the server that the EA is bound to a given TLS channel, but because the TLS peer is anonymous, if the EA's certificate has been compromised the TLS peer could be the attacker. This does however preclude an uncompromised client from acting as a signing oracle, i.e. signing any message a potentially malicious server requests.

Together these guarantees allow the server and the client to use either protocol layer to authenticate the other. It is interesting to note that despite the fact that neither the TLS layer, nor the EA layer achieves full compound authentication, i.e. both ICA and OCA, because there are only two layers, and the inner layer proves OCA and the outer layer proves ICA, we can reason about exactly the same cases as a protocol where all layers achieve CA. We discuss this further in Chapter 5.

4.4.9 Relating the properties to the goals

In Section 4.3.3 we lay out six goals for `draft-sullivan`, which we recap here. We then relate the goals to the properties we need to prove, see Table 4.1 for a summary.

1. An EA authenticates the sender to the receiver.
2. An EA can be send out-of-band and validated by the peer.
3. A peer can authenticate with multiple identities.
4. Certificates sent in-band should be confidential.
5. EAs must be independent and unidirectional.
6. EAs can not be transplanted into another protocol.

Goal		Properties
Goal 1	Authenticate TLS peer	OCA and either ICA (unilateral) or CA (bilateral)
Goal 2	Out-of-band validation	OCA
Goal 3	Multiple identities	Consequence of Goal 1 and Goal 5
Goal 4	Certificate confidentiality	Not a requirement
Goal 5	EA independence and unidirectionality	Consequence of Goal 1, OCA, and secret session keys
Goal 6	Transplantation protection	Perfect cryptography assumption

Table 4.1: Relationship between goals and properties

Goal 1 requires that we can authenticate one party of a TLS communication to another using a certificate after the session has been established. This is the main goal of the protocol. To show this, we need to prove that if an actor receives an EA bound to a TLS session, then the TLS session peer controls the certificate in the EA. This maps closely to the definitions in Section 4.4.8. OCA of EAs means that the receipt of an EA proves control of the end point, and CA of TLS[13] means that control of the end-point proves ownership of the certificate in a received EA.

Goal 2 requires that EAs transported out of band can still correctly verify. Proving OCA for EAs proves that the security of the EA is not dependent on the secrecy or the authentication of the TLS channel.

Goal 3 says that if multiple certificates are used they all authenticate a given connection. Proving Goal 1, and that multiple EAs sent on the same channel are independent, Goal 5, proves that each EA authenticates the TLS channel.

Goal 4 says implementations "SHOULD" use the underlying TLS layer to keep the certificates confidential. Because this is a should level requirement, as opposed to a must, we do not check this, but note that Cremers et al. [Cre+17a] prove that data sent over a TLS channel is confidential.

Goal 5 requires that authenticators be independent and unidirectional. Two authenticators can be said to be independent if the compromise of

[13] Or ICA for servers in unilateral TLS.

one does not give any advantage in the compromise of the other. Proving Goal 1 proves that to create an EA you need to know the private key of the certificate contained in it. Thus knowing some other certificate's private key provides no advantage in deriving an EA, assuming that the private keys of the certificates are themselves independent.

The only shared secret between two EAs on the same channel is the master secret of the TLS channel.[14] Thus we still need to prove that the master secret cannot be derived from the EA.

The final case is where the same certificate is sent over two different TLS channels. In this case we need to show that knowledge of a certificate's long term key, and knowledge of the master secret of a TLS channel, does not enable an attacker to produce an EA for another TLS channel without knowing the master secret of the other TLS channel. This is captured by the OCA property of EAs. This scenario is equivalent to a channel synchronisation attack.

Proving EAs are uni-directional comes from the agreement property on *params*, which includes the context strings, which indicate the direction of the EA.

Goal 6, which requires that the EA cannot be transplanted into other protocols, is unprovable without co-ordination between all protocols, however we can show that EA messages cannot be transplanted into the TLS handshake. Because the `CertificateVerify` message contains a context string a `CertificateVerify` message from an EA cannot be directly transplanted into a TLS handshake. Further, because we assume perfect cryptography, we assume that an attacker cannot modify the contents of an encrypted message without access to the key. The same reasoning applies to `Finished` messages. Thus this goal is achieved a priori by our model.

4.4.10 Achieving compound authentication

To achieve compound authentication Bhargavan et al. propose contributive channel bindings. Informally, this means that the channel bindings have to accumulate a contribution from each layer of authentication.

[14] Or more properly the TLS `exporter_master_secret`.

Bhargavan et al. hypothesise that if the channel bindings are contributive, and both parties agree on them, then, the compound protocol achieves the two security goals in Section 4.4.5, i.e. agreement and compound authentication. We call this the Bhargavan hypothesis.

Formally contributive channel bindings are defined as follows.

Definition 4.4.7. Contributive Channel Bindings (Definition taken from Bhargavan et al. [BDP15, p. 10]). If a compound authentication protocol consists of n protocol instances $\{l_1, \ldots, l_n\}$, the channel binding of l_n must be bound to the parameters and session secrets of all n instances $\{params_1, sk_1, \ldots, params_n, sk_n\}$, so that agreement on the channel binding guarantees compound authentication for the composite protocol.

Bhargavan et al. [BDP15] find flaws in a number of compound authentication protocols which do not use contributive channel bindings, and then fix the flaws by making the channel bindings contributive. This provides some evidence that the Bhargavan hypothesis is true, or at least a useful approximation of the truth.

We note that the channel bindings used in `draft-sullivan` are contributive when considering a single EAs, but that multiple EAs sent over the same channel do not contribute to each other. This means that we would not expect multiple EA to achieve OCA when considered as a single protocol run with many layers. This aligns with the results of our Tamarin analysis.

4.5 Tamarin model

`draft-sullivan` repurposes messages from the TLS 1.3 handshake to construct an authentication protocol to be used inside a TLS connection. We analysed `draft-sullivan` with the Tamarin analysis tool [Sch+12]. In this Section we describe the Tamarin model we used to verify and explore the guarantees of `draft-sullivan`.

Our channel bindings analysis gives us formal properties we can verify for our model in Tamarin. The Tamarin model we built proves that EAs have properties we describe in Section 4.4.8, and thus meets its security goals. We also examine stronger properties using Tamarin.

4.5.1 Abstraction

`draft-sullivan` is designed to be part of a layered protocol. It is cryptographically bound to the TLS layer. Therefore to accurately capture it in Tamarin we must model both the TLS layer, and the EA layer.

In Chapter 3 we construct a comprehensive model of TLS 1.3. Because of its accuracy, the model is inherently very complex. Extending it significantly to include `draft-sullivan`, may not feasibly enable analysis, or would at least require significant amounts of additional computing resource. To make the analysis tractable, we consider an abstracted version of TLS, whereby we run a version of the handshake without providing the attacker any rules to attack it. We matched the TLS 1.3 model we used in Chapter 3 very closely, using the same message formats and style such that integrating the two models would be as simple as possible, and we leave this for future work. We justify abstracting the TLS layer by noting that the security properties we rely on were proven to hold in the TLS 1.3 model in Chapter 3.

Further, our abstracted model does not cover resumptions on the basis that after a resumption the master secret changes, and thus from the perspective of an EA a resumption is indistinguishable from a separate session between the same participants. The authentication guarantees of an EA do not apply over a resumption boundary. In Chapter 5 we discuss a possible technique for bridging the resumption boundary, but here we stick to simplified model.

Because EAs only interact with the TLS channel through the exporter application programming interface (API), we do not give the EA layer access to the TLS master secret. The client and server only perform computations on exporters. This logical divide allows us to cleanly separate the two layers with a simple interface.

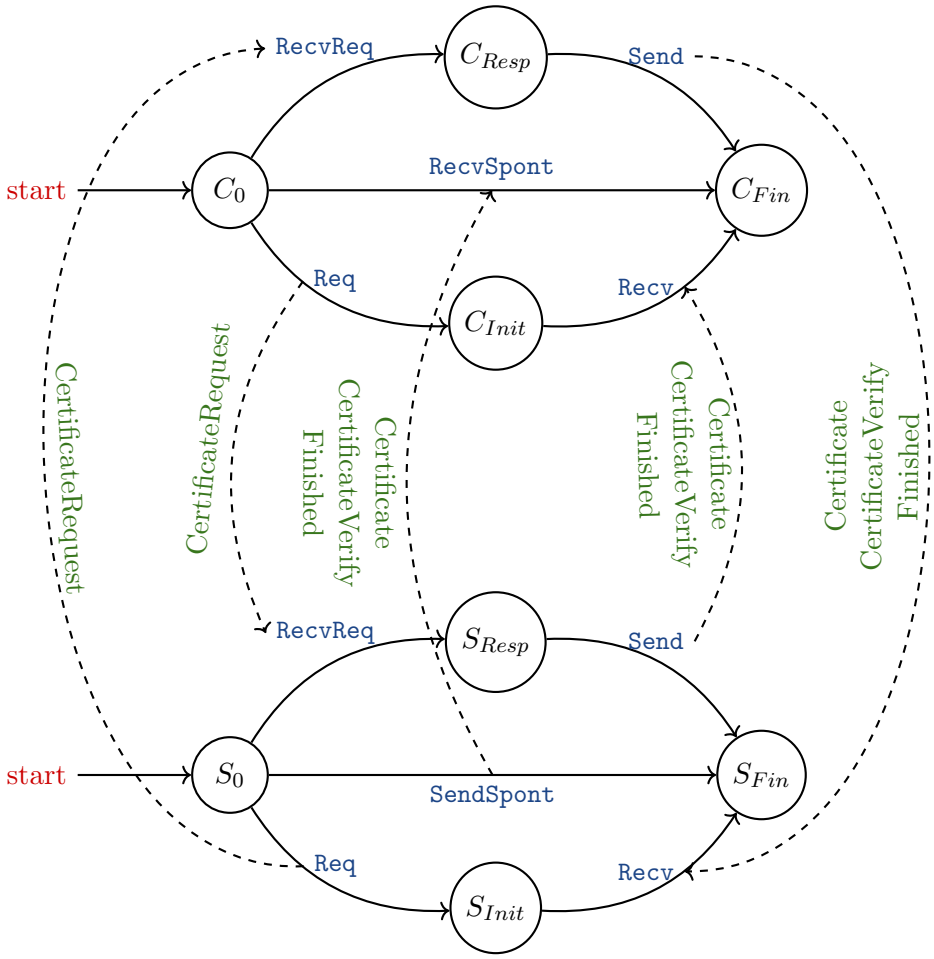


Figure 4.7: `draft-sullivan` state diagram

4.5.2 `draft-sullivan`'s state machine

The `draft-sullivan` protocol has four states for each actor,

1. the `initial` state,
2. the `initiator` state,
3. the `responder` state,
4. and the `finished` state.

These states are illustrated in Figure 4.7.

The state transitions of the client and the server in `draft-sullivan` can be described in approximately ten transitions, those denoted with solid black lines in Figure 4.7. We include an example transition below.

```
1  //Send an EA, as per the request/response flow.
2  rule C_Send:
3    let
4      certificate=pk(ltkA)
5      //Compute the CertificateVerify.
6      signature=compute_signature(ltkA,
       ↪ h(hccc, CertificateRequest, certificate))
7      //Compute the Finished.
8      verify_data = hmac(fmc,
       ↪ <hccc, CertificateRequest, Certificate,
       ↪ CertificateVerify>)
9    in
10   [ State(~cid, 'C1', $C, $S, 'client', auth_status)
11   , PendingReqR(~request_id, ~cid, 'client',
     ↪ CertificateRequest)
12   , !Exporters(ms, 'client', ~cid, hccc, hccs, fmc, fms)
13   , !Ltk($A, ltkA)
14   //This fact binds the Client to the additional
15   //identity it has been assigned.
16   , DelegateLtk($C, ltkA)
17   , Fr(~msg_id)
18   ]
19   --[ C_Send(~cid)
20   , Owns($C, ms, 'client', <$A, ltkA>)
21   , Instance(~cid, $C, $S, 'client')
22   , Fulfil(~cid, $C, 'client', certificate_request_context,
     ↪ <$A, pk(ltkA)>)
23   ]->
24   [ State(~cid, 'C1', $C, $S, 'client', auth_status)
25   , TLS_Send(~msg_id, ~cid, $C, $S,
     ↪ <Certificate, CertificateVerify, Finished>)
26   ]
```

This transition details a client `Send` transition. The `let` statement allows us to define shorthand names for more complex terms. For example line 4 defines the variable `certificate` to be `pk(ltkA)`, i.e. the public key of `ltkA`, which is a long-term key taken as an input in line 13. The

`compute_signature` and `hmac` calls are `m4` rewrite macros that rewrite the function to a form that can be analysed by Tamarin. Using `m4` in this way allows us to write rules in a imperative pseudo-code style which makes it easier to check for correctness against the draft, as opposed to complex and abstract facts.

The `State` fact represents the State of the actor, tracking the session ID, `~cid`; current protocol step, `'C1'`; actor identity, `$C`; peer identity, `$S`; role, `'client'`; and authentication status, `auth_status`, respectively. The `PendingReqR` fact represents the client's memory of receiving a request, assigning a `~request_id` to ensure uniqueness, even in the case of a repeated `CertificateRequest`. The `!Ltk` fact binds an identity, `$A`, to a LTK, `ltkA`.

The `Owns` action in line 20 indicates that an EA was created legitimately. As we will describe later in this Chapter this pairs with a `Recv` action, which captures an actor receiving an EA that validates. We will prove properties of the form "Given some pre-conditions, if an actor receives an EA that validates, then its TLS peer created a valid EA with the same certificate."

The `C_Send` transition is only possible if a request was previously received, and therefore cannot occur without an appropriate `PendingReqR` fact.

The transition also needs an established TLS channel, as captured by the `State` fact. The `Exporters` fact represents the client's memory of the exporter keys he has computed, the handshake contexts, `hccc` and `hccs` for the client and server respectively; and the finshed MAC keys, `fmc` and `fms` for the client and server respectively. We chose to model the exporters in this way to capture the fact that the same exporter keys are used for all EAs. The same fact is consumed for all runs of the EA protocol within a single TLS session.[15] The signature and the `Finished` message are both computed using the exporters, with no reference to the master secret.

Stasis of the model

`draft-sullivan` was designed to run inside a TLS channel and to provide authentication without changes to the TLS state machine. Our model

[15] Recall that the `Exporter` fact is marked with a `!`, and is thus persistent.

captures this stasis of the TLS state machine, by having a state fact that changes with the TLS handshake, but is left unchanged by `draft-sullivan` transitions.

Because runs of the `draft-sullivan` are explicitly allowed to interleave we model the `draft-sullivan` state machine as a bag-of-facts. We output "memory" facts that capture the state of each run of the protocol, any of which can then be consumed at any time. This means that rather than waiting for the state machine to arrive at some particular state to continue a run, an actor can pick up any run currently in progress and continue it at any point.

This structure highlights `draft-sullivan`'s unusual protocol structure, where there are multiple interleaving runs layered over a single run of the base protocol. As we will discuss in Chapter 5, this structure can make it difficult to reason about the ordering of runs of the EA protocol. In `draft-sullivan` this is not an issue, because the security properties attained by a given run is independent of any other run of `draft-sullivan`, however, in Chapter 5 we show how this can become an issue.

4.5.3 Closely modelling the specification

The EAs model is written in a similar manner to our TLS model. We opted for a high fidelity model of `draft-sullivan`, working to enable ready comparison between the draft messages and the wire formats.

Our abstracted TLS model only captures two modes of TLS, unilateral authentication and bilateral authentication.

TLS 1.3 has eight security guarantees, however not all properties hold in all modes. Rather than model all of these varying levels of security, our abstract version of TLS only provides a subset of those properties that apply in all modes. If we can prove the guarantees of `draft-sullivan` in conjunction with a strictly weaker abstraction of TLS we can be sure that the guarantees still apply when layered over a strictly stronger version of TLS. The properties we require of our abstract version of TLS are the first five guarantees of the TLS 1.3 specification, namely (1) that both parties establish the same session keys, (2) that those session keys are secret, (3) that each party can

authenticate its peer as appropriate, (4) that session keys are unique, and (5) that an attacker cannot perform a downgrade attack. Importantly, we do not require perfect forward secrecy (PFS).[16] All modes achieve the first five properties, but only modes with a DHE achieve PFS. By eliding the DH handshake, similar to the `PSK-only` modes of TLS 1.3, and not establishing a pre-shared secret, we create a strictly weaker abstraction of TLS.

By careful usage of the same message formats and modelling techniques, we have left open the possibility of merging the two models into one larger model, and we leave this for future work.

4.6 Encoding the threat model and security properties

4.6.1 Threat model

The threat model we used for our Tamarin analysis is an extended version of the Dolev-Yao attacker [DY83], as we used in Section 4.4.4. The attacker is given the abilities to leak the master secret from the TLS channel and the private key of certificates using the actions `Revms` and `RevLtk` respectively.

Because we abstract the TLS layer we provide the attacker with special actions to read and write to TLS channels for which they know the key. We write our security properties so that this is equivalent to the attacker being able to act as a legitimate host to one party to attack another. We achieve this by proving the attacker cannot compromise a session without compromising the master secret of that session, even if they have compromised the master secrets of other sessions.

4.6.2 Security properties

We analyse four security properties for `draft-sullivan`:

1. that the master secret of the TLS channel is confidential,
2. proof of certificate ownership,
3. certificate linking, and

[16] For completeness, the final two properties are KCI resistance and the protection of endpoint identities.

Property proven	**Lemma**
Secret session keys	`secret_session_keys`
OCA of EAs	`cert_ownership`
ICA of unilateral TLS	`cert_ownership`
Compound authentication of bilateral TLS	`cert_ownership`

Table 4.2: Relationship between properties and lemmas

```
lemma secret_session_keys:
  "All ms transcript #j #k.
      SessionKey(ms, transcript)@j
    & KU(ms)@k
  ==> Ex actor peer #i.
      Revms(ms, actor, peer)@i
    & (#i < #k)"
```

Figure 4.8: The secret session keys lemma, edited for consistency of style, states that if a session key is established, and the attacker knows it, then the attacker must have previously revealed the session key.

4. Outward compound authentication (OCA) between EAs.

As we will discuss in the following paragraphs, the first two lemmas are required to meet the goals of `draft-sullivan`.[17] As we discussed in Section 4.4.9, all the goals of `draft-sullivan` are achieved if we can prove the various compound authentication properties and the secrecy of the master secret. We show the relationship between the properties we need and the lemmas we prove in Table 4.2. The latter two lemmas examine more complex properties not claimed by the specification.

Master secret confidentiality

To check that `draft-sullivan` EAs do not weaken the security guarantees of the underlying TLS channel, we prove that the master secret cannot be derived from EAs. The lemma is shown in Figure 4.8.

[17] For reference, these are (1) authentication of the sender, (2) validation by the peer, (3) authentication of multiple EAs, (4) certificate confidentiality, and (5) independence and unidirectionality of EAs.

```
1  lemma cert_ownership[reuse]:
2    "All actor peer ms cert role #k.
3        Recv(actor, peer, ms, role, cert)@k
4      & not (Ex #i #j.
5          RevLtk(cert)@i
6        & (#i < #k)
7        & Revms(ms, actor, peer)@j
8        & (#j < #k)
9      )
10   ==> Ex role2 #h.
11       Owns(peer, ms, role2, cert)@h
12     & (#h < #k)
13     & not(role=role2)"
```

Figure 4.9: The certificate ownership lemma, edited for consistency of style, states that if an actor receives an EA (line 3) and the attacker has not compromised both the EA certificate (line 5) and the master secret of the TLS channel (line 7) then the EA was created by the peer (line 11).

Certificate ownership

To check that EAs meet their authentication goals we prove that if an actor completes a run of `draft-sullivan` and accepts an EA with certificate, *cert*, then the certificate was signed by the peer, or the attacker knows the private key of *cert* and the master secret of the TLS channel.

This is the logical conjunction of our OCA property, Property 4.4.5 and our CA/ICA property, Property 4.4.6. The lemma is shown in Figure 4.9.

As long as either the TLS channel or the private key of the certificate is uncompromised, then the recipient of an EA knows the TLS end-point it is talking to controls the certificate in the EA. This is a very strong guarantee.

Certificate linking

Because EAs sent on the same channel share key material it is important to define the relationship certificates have to each other. We prove properties about the relationship between EAs sent on the same channel. Certificate linking looks to prove that EAs are securely linked if the key material they share, i.e. the TLS channel master secret, is uncompromised. Specifically we

```
1  lemma cert_linking:
2    "All actor actor2 ms role role2 cert cert2 peer peer2 #j #k.
3        Recv(actor , peer , ms, role , cert)@j
4      & Recv(actor2, peer2, ms, role2, cert2)@k
5      & (#j < #k)
6      & not (Ex #g.
7          Revms(ms, actor,peer)@g
8        & (#g < #k))
9    ==> Ex role3 role4 #h #i.
10       Owns(peer, ms, role3, cert)@h
11     & not (role = role3)
12     & (#h < #j)
13     & Owns(peer2, ms, role4, cert2)@i
14     & not(role2 = role4)
15     & (#i < #k)"
```

Figure 4.10: The certificate linking lemma, lightly edited for consistency of style, states that if two EAs are received (lines 3 and 4) on the same uncompromised (line 7) channel, then both were created by the peer (lines 10 and 13). This applies whether they were sent by a single actor, or if each actor sent one.

construct a lemma that shows what is needed for an EA to authenticate another.

This property is the strongest compound authentication property that can be proven about the relationship between multiple EAs. Because EAs are non-key generating, i.e. have no shared secrets, we cannot consider the session secrets of an EA layer to be un-leaked.

Thus the only layer we can prove authenticates an EA layer is the TLS layer, but we can prove that multiple EAs sent on an uncompromised TLS channel are all authenticated by the TLS channel.

The lemma we prove is shown in Figure 4.10. It states that if two EAs are received on the same, uncompromised TLS channel, then both EAs are authentic. This applies whether they are received by the same actor, or whether each actor receives one. We prove this lemma to contrast the OCA lemma discussed in the next paragraph.

Outward compound authentication

We also look to prove whether multiple EAs sent on the same channel can authenticate each other, rather than just the TLS channel. This would allow us to consider multiple runs of `draft-sullivan` over a single TLS channel as a single layered protocol run with multiple layers, rather than multiple composite protocol runs with only two layers that run with some shared key material. Because compound authentication as defined in Definition 4.4.2 can't be used to reason about the security of an EA, because EAs are non-key-generating, we might want to prove that EAs have OCA and thus that later EAs authenticate earlier EAs.

We write a lemma that claims that if two EAs are sent on the same TLS channel, and the second has an uncompromised certificate, then the first is authentic. This corresponds to OCA between EAs, as discussed in Definition 4.4.3. However we prove we do not have OCA between EAs by way of counter-example.

4.7 Results

Our channel bindings analysis implies that if we prove `draft-sullivan` has the properties we lay out in Section 4.4.8 then it meets its security objectives. We proved that `draft-sullivan` has said properties. Additionally, we analysed a stronger set of properties considering multiple EAs sent on the same channel.

We proved that in a Tamarin model of `draft-sullivan` the lemmas we constructed to capture the properties laid out in Section 4.4.8 hold. Thus, EAs authenticate the underlying TLS layer, and the TLS layer authenticates EAs bound to it.

When we consider the properties of multiple EAs sent on the same TLS channel we get more mixed results. We were able to prove that if the TLS channel was uncompromised then multiple EAs sent on the same channel were authentic, i.e. the TLS channel can authenticate multiple EAs. EAs do not, however, authenticate one another.

```
1  lemma outward_compound_auth:
2    "All actor actor2 ms role role2 cert cert2 peer peer2 #j #k.
3        Recv(actor , peer , ms, role , cert )@j
4      & Recv(actor2, peer2, ms, role2, cert2)@k
5      & (#j < #k)
6      & not (Ex actor3 peer3 #f #g.
7          Revms(ms, actor3, peer3)@f
8        & (#f < #j)
9        & RevLtk(cert2)@g
10       & (#g < #k))
11   ==> Ex role3 role4 #h #i.
12       Owns(peer, ms, role3, cert)@h
13     & not (role = role3)
14     & (#h < #j)
15     & Owns(peer2, ms, role4, cert2)@i
16     & not(role2 = role4)
17     & (#i < #k)"
```

Figure 4.11: The outward compound authentication lemma, edited for consistency of style, extends the certificate linking lemma by adding a single restriction; that the attacker not compromise the second certificate (line 9, highlighted in red). Thus an attacker may compromise the master secret of the TLS channel, or the second certificate, but not both. This property does *not* hold.

4.7.1 Master secret confidentiality

We prove that the attacker cannot derive the master secret from any combination of runs of the EA layer.

4.7.2 Certificate ownership

Our certificate ownership property is the logical conjunction of OCA between an EA and its TLS channel, and ICA between the TLS channel and the EA.[18]

By proving certificate ownership we proved that EAs authenticate the TLS layer they are bound to, and that the TLS channel authenticates an EA bound to it. As detailed in Section 4.4.9, along with the master secret confidentiality lemma, this proves that `draft-sullivan` meets its security goals.

4.7.3 Certificate linking

As we mentioned in Section 4.4.5, the `draft-sullivan` specification says that joint authentication is hard to prove formally. Although EAs are independent, there is a security dependency between the inputs of EAs sent on the same channel. Because multiple runs of `draft-sullivan` within a single TLS session use the same exporter keys there is a security dependency between EAs.

Referring back to the definition of compound authentication, Definition 4.4.2, to prove that an actor is jointly authoritative over multiple certificates we would need to prove that both parties agreed that multiple EAs were authored by the same actor. By proving our certificate ownership property, we prove compound authentication between the TLS channel and each EA.

We can strengthen our definition of compound authentication of the TLS to cover multiple certificates. We prove that two EA sent over the same channel are both authenticated by the TLS channel, and thus, as long as the TLS channel is uncompromised, both EAs are authentic. This lemma

[18] Or regular compound authentication in the bilateral case

captures a stronger version of compound authentication than required by the security goals.

4.7.4 Outward Compound Authentication (OCA) between EAs

We strengthened our OCA lemma in the same way, considering multiple EAs sent on the same channel, and tried to prove that an EA authenticates other earlier EAs sent on the same channel. Our regular version of this property only requires that EAs authenticate the TLS channel.

We proved that the stronger OCA property does not hold using Tamarin, which produced a counter-example which violates the property. The counter-example produced is as follows. If the attacker compromises a TLS channel, forges an EA with a compromised certificate, and sends it to the client, and the server later sends an EA with an uncompromised certificate, the latter EA will be accepted by the client, violating the property.

Because prior EAs do not affect the production of later EAs they do not meet the definition of contributive channel bindings, and thus Bhargavan's hypothesis would suggest that they do not authenticate each other, a result borne out by our Tamarin analysis. This means that we cannot assume that if an EA is genuine, that other earlier EAs ostensibly sent by the same actor are genuine. We suggest a modification to EAs that claims this property in Chapter 5.

Our models are available online [Hoy18a], and contain a full counter-example along with proofs of all the other lemmas and their sub-properties. The repository also includes an extensive guide to the model.

4.8 Conclusions

In this work we showed that `draft-sullivan` achieves its main security goals. We formalised the goals of `draft-sullivan` into security properties, extending previous work on channel bindings, and used the Tamarin Prover to prove the protocol meets those properties.

Previous work on layered protocols had limited ability to reason about protocols like `draft-sullivan`. With layered protocols we look to prove that if a given layer is uncompromised then some other set of layers is authentic. This lets us reason about the security of one layer in terms of another. Formalising `draft-sullivan`'s goals required the definition of two new properties for composite protocols. Prior definitions could only reason about protocol layers that authenticate all other layers, but our new definitions let us reason about layers that only authenticate all past layers, or all future layers. This lets us capture the authentication guarantees that can be achieved by a wider class of protocols.

`draft-sullivan`'s security goals only define the relationship between individual EAs and the TLS channel, a composite protocol with two layers. We proved that if either layer is uncompromised then the other is authentic. However, because `draft-sullivan` allows for multiple protocol runs over the same TLS channel we also examined whether multiple EAs could achieve the same guarantees when layered, effectively considering all these runs as a single composite protocol. Our new definitions suggest that EAs do not have the necessary features to authenticate future EAs, but that they do have the necessary features to be able to authenticate prior EAs.

We proved that when considering these runs as a single composite protocol, whilst the TLS layer authenticates all the EAs, EAs do *not* authenticate all past EAs.

In practical terms, this means that if `draft-sullivan` is deployed in an environment where the threat model does not include an attacker who can break TLS channels, then all EAs bound to a TLS channel were created by one of the TLS endpoints.

However if the threat model does consider such attackers, then we cannot use EAs to reason about the security of prior EAs.

The threat model where an attacker can compromise the TLS channel is indeed a very strong threat model. However, we note that it is still common in industry to use static RSA keys to allow traffic sent to a server to be analysed by middle-boxes and passive taps. Our analysis shows that in this

case the client and the server cannot prove any compound authentication property between EAs.

Being able to prove and disprove properties increases the confidence in the accuracy of our Tamarin model. The properties which we proved and disproved precisely align with the cases that used channel bindings that were contributive, and those that did not. This provides more evidence for the Bhargavan hypothesis, that composite protocols that use contributive channel bindings achieve compound authentication.

Chapter 5

Layered Exported Authenticators

5.1 Introduction

In the previous chapter we introduced EAs, a draft-standard before the IETF. EAs allow a participant in a TLS channel to add additional identities to the channel. These identities are linked to certificates, and to add them to the channel the participant constructs a special message, an EA, that proves it controls the identity in the certificate. As discussed in the final sections of the chapter these additional identities are not jointly authenticated. This means that whilst the participant can prove it is authorised to act as each of the identities individually, it cannot prove that it is authorised to act as a group of them.

Compound authentication is a form of joint authentication. Compound authentication proves that for a series of authentication protocols all the identities are controlled by a single actor, as long as at least one of the authentication protocols is successful. EAs, as discussed in the previous chapter, have the relevant components to achieve a similar property, that we call outward compound authentication (OCA). This means that later authentication protocols authenticate earlier ones. OCA is a variant form of compound authentication. However, even though EAs have the components to achieve OCA they do not. In this chapter we describe an extension to EAs, which we call Layered Exported Authenticators, that allows a participant to prove joint authentication in the form of OCA.

5.1.1 Chapter overview

Our main contributions in this chapter are as follows:

1. We present `draft-hoyland`, a draft standard presented to the IETF introducing LEAs. LEAs extend EAs in a way intended to provide compound authentication between EAs.

2. We discuss the design of LEAs with an analysis of LEAs in terms of the Bhargavan framework.

3. We define authentication forests, a complex authentication property. Authentication forests allow a complex layering of authentication properties between a client and server, with each branch providing OCA of all ancestor nodes.

4. We provide a model of LEAs in Tamarin, along with an extended discussion of Tamarin's pre-loader and its treatment of source lemmas. Tamarin's pre-loader is not well documented, and we provide a detailed explanation of its operations. LEAs transform the multiple independent runs of EAs, a two layer composite protocol, into an arbitrarily deep composite protocol, and stretch the boundaries of what it is possible to reason about with the Tamarin prover.

5. We give a partial proof of the security of LEAs using Tamarin, proving that a LEA provides OCA of its immediate predecessor in those cases where we would expect it.

5.1.2 Motivation

Joint authentication is useful in a number of cases, for example it can be used to securely update a pinned certificate. Joint authentication can also be used as an explicit signalling mechanism, proving an explicit proof that a peer has accepted a particular certificate. Developing this extension to provide joint authentication to EAs allows us to examine more fully compound authentication and composite protocols, applying the definitions we introduced in Chapter 4.

5.1.3 Chapter organisation

In Section 5.2 we recall the definition of OCA and discuss the uses of OCA in EAs. In Section 5.3 we introduce `draft-hoyland`, a draft standard before the IETF, and we examine it under the Bhargavan framework in Section 5.4. We also discuss the development of `draft-hoyland` alongside `draft-sullivan`, and the ways in which both drafts have influenced each other. In Section 5.5 we describe various authentication structures that can be built with LEAs, in particular authentication forests.

We introduce our Tamarin model in Section 5.6 and then proceed on to a discussion of source resolution in Tamarin. We detail a set of experiments we ran to analyse the pre-loader, discovering properties of the pre-loader that were surprising to some of the authors of Tamarin. We also detail the lemmas we prove about LEA, which allow us to show that they do not weaken the security of EAs as well as a partial proof of the OCA property we want to achieve. Finally in Section 5.8 we discuss our results and some conclusions.

5.2 Background

5.2.1 Achieving Outward Compound Authentication (OCA)

In this Chapter we describe an extension to establish OCA for EAs. We recall here the definition of OCA.

Definition 4.4.3. Outward Compound Authentication. If a principal a completes a compound authentication protocol consisting of protocol instances $\{l_1, \ldots, l_n\}$, such that some instance l_i has an honest peer credential and the session secrets of l_i have been leaked or are $\perp$, then there exists a principal b with protocol instances $\{l'_1, \ldots, l'_i\}$ such that each l'_j has the dual role to l_j and agrees with l_j on $params_j$ and sk_j.

In this case instance l_1 is the TLS layer, l_2 is a run of the EA protocol, and instances $l_3 \dots l_n$ are runs of the LEA protocol. We establish in Chapter 4 that EAs achieve OCA with the TLS layer, but that subsequent runs of the protocol do not achieve OCA with each other. Unlike compound authentication and ICA which both require a protocol run to establish a shared secret, OCA requires a protocol run to establish an identity. Thus it should be possible for a protocol that only establishes identities to achieve OCA. The Bhargavan hypothesis suggests that for a composite protocol to establish compound authentication properties it must use channel bindings of a certain form, i.e. they must be contributive.

Informally a contributive channel binding is one which takes as input the channel bindings of all previous layers, amongst other things. The channel bindings of EAs only take as input the channel bindings of the TLS layer. To achieve compound authentication of any form, we thus need to include, at least, the channel bindings of all previous layers.

5.2.2 Use cases

LEAs have a number of potential use cases. For example, they could be used to securely update pinned certificates. When a client remembers a server's certificate from one connection to the next, we say the client has pinned the certificate. This can protect clients from threat models where an attacker can obtain a fraudulently issued certificate. If a server's certificate changes unexpectedly, a client will refuse to connect, protecting them from this attack.

Historically being able to obtain a mis-issued certificate was considered the preserve of a nation-state level attacker, however with the rise in both automated certificate provisioning and cloud based hosting, where many servers share resources such as IP address, fraudulently obtaining certificates has become easier.

Permanently pinning a certificate, however, is not practical, because servers do sometimes need to update their certificates for a variety of reasons, such as expiry or compromise. The method for updating a pinned certificate requires letting the pin expire whilst deploying a new certificate,

running multiple certificates with overlapping time-frames, to ensure that clients who connect rarely aren't locked out, and that there is never a point where no certificate is pinned. This makes updating certificates a slow process, which is problematic in the case of key compromise.

Using LEAs a server could explicitly update a pinned certificate at any point, even under a threat model when an attacker can obtain certificates fraudulently. A client could connect to a server supporting LEAs, using its pinned certificate. The server could send their new certificate in an EA, and then send a LEA authenticating their new certificate with their old certificate, which the client has pinned. The client can see that the server is jointly authorised as the owner of both certificates and can update its pinned certificate. Because older layers are authenticated by newer layers the LEA constitutes a proof that the owner of the old certificate created the EA, and thus controls the new certificate.

For an attacker to successfully attack this they need to achieve a number of things. First the attacker needs to compromise a TLS channel between the client and the server, or persuade the client to connect to the attacker, mis-identifying the attacker as the server. Because of the use of EAs these are not the same case, because in the latter case the attacker is not able to include EAs created by the server. To successfully attack the latter scenario, the attacker needs access to the server's private key, because we are discussing the case where the client already has the server's certificate pinned. To successfully attack the former scenario the attacker needs to trick the server into signing a fraudulent EA. Second, the attacker needs to fraudulently obtain a certificate for the server. This could occur if a government compelled a certificate authority to create such a certificate, or if they broke into a certificate authority to create one secretly, as happened in the case of DigiNotar.[1] However it could also happen through attackers exploiting poorly configured or abandoned servers and automated certificate issuance mechanisms to acquire such a certificate. Using LEAs raises the bar for an attacker, whilst simplifying the requirements of the server. An attacker at least needs to be able to acquire both a fraudulent certificate and the pri-

[1] `https://www.bbc.com/news/technology-14789763`

vate key of a server to be able to fraudulently update a pin. A server who needs to update their certificate suddenly, for example because of key compromise, doesn't need to allow the pin on that certificate to expire. It can simply update the certificate in the pin, providing a proof that it is jointly authoritative over both. As long as the attacker hasn't fraudulently acquired a certificate *and* the server's private key, this is secure.

A second use case is providing a mechanism proving that an EA has been accepted. Currently there is no TLS layer mechanism for establishing whether or not a certificate, whether part of an EA or part of a post-handshake authentication flow, has been accepted. By cryptographically signing an EA that has been accepted, an actor can prove to its peer that the certificate has been accepted.

A third use case, which we will consider in future work, is to re-establish a chain of authentication across a resumption. Recall that if a client and server establish a TLS connection, and establish a number of EAs over the channel, and then perform a resumption, they must re-create all the EAs from the previous session if they wish to use them, i.e. EAs are not valid over a resumption. If they create an LEA binding to the last LEA in the chain of the previous session they could prove that they believed all the EAs from the prior connection were still valid. If both parties can agree that all prior EAs are still valid they do not need to re-create them for the new channel. This relies on the PSK remaining confidential, which, depending on the use case, may be an acceptable security / convenience trade-off. Our work does not cover this use case, but we plan to study it in future work.

5.2.3 Extensions

In both TLS and EA in nearly every message there is an "extensions" field, which contains a list of extensions. Extensions are additional pieces of data that can be added to a message to define some extra functionality. Extensions are defined in RFCs, and lists of extensions and the messages in which they can be used are maintained by the Internet Assigned Numbers Authority (IANA) [NSS18]. IANA is the organisation that coordinates various parts of the global internet, and in particular they maintain lists of unique codes used

by protocols. This stops multiple protocols, and in our case extensions, using the same values for signalling their type, helping prevent miscommunication.

Extensions are usually defined for multiple messages and form a request-response pattern, although sometimes they are simply indicators. For example the client can send the `server_name` extension to tell the server the name of the server it thinks it is contacting.

Some extensions are mandatory to implement in the absence of a specific application profile standard specifying otherwise, making them nearly ubiquitous. For example the Key Share extension, which carries the DH key shares in the `ClientHello` and `ServerHello` messages, is mandatory to implement, and required in every mode other than the `PSK-only` mode. Other extensions are entirely optional, and only supported when necessary.

Because the extensions field is already present in many messages, we can add the extra functionality provided by extensions without any changes to the underlying protocols. The use of extensions prevents fragmentation of the ecosystem into hundreds of different incompatible variants of a protocol each with slightly different functionality.

In early drafts of the TLS 1.3 standard a server was not allowed to send an extension that had not first appeared in the `ClientHello`, and a client receiving such an extension was required to abort the connection [Res16, pp. 34-35]. However in the final specification this requirement was changed. The requirement is now that if an actor receives the "response" part of an extension for which it didn't send the corresponding "request" then it must abort [RFC8446, p. 36]. This gives the server more flexibility. This change becomes important in Chapter 6.

5.3 Layered Exported Authenticators

Layered EAs were proposed to the IETF in `draft-hoyland` [Hoy18b] and presented to TLS WG at the IETF 102 meeting. We include `draft-hoyland` in Appendix B. The draft proposes adding an extension to the `Certificate` and `CertificateRequest` messages in the EA flows, see Section 4.3. The extension contains a reference to an EA that had previously been sent on the channel, which we will refer to as the requested binding, to differentiate

it from the requested certificate. A `CertificateRequest` containing this extension would be requesting two things, (1) that the responder validate in some way the requested binding, and (2) that the responder fulfil the request as normal.

If the responder includes the requested binding in the `Certificate` then it indicates that the responder believes the requested binding, i.e. the previous EA, was authentic. The aim is to achieve OCA between EAs. The draft's proposal is shown in Figure 5.1.

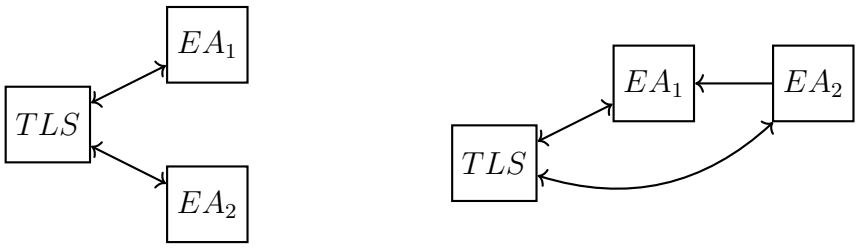
```
struct {
    opaque prev_certificate_request_context<0..2^8-1>;
    opaque binding[Hash.length];
} LayeredEA;
```

Figure 5.1: `LayeredEA` definition from [Hoy18b, p. 3]. The `[Hash.length]` notation indicates that the binding field has the same number of bits as the hash function associated with the underlying TLS connection.

The `prev_certificate_request_context` is the `certificate_request_context` of the requested binding. The `binding` is the `Finished` message of the requested binding.

The intention is that this extension constitutes a channel binding of the requested binding. If the responder includes the extension in the `Certificate` then both sides have agreed to this channel binding. Because the `Finished` message of an EA is a transcript hash, if the recipient of an EA can validate the `Finished` message then it knows that both sides agree, amongst other things, on the CRC, and the `Finished` message. By including this extension in the `Certificate` message the responder is claiming that they believe the EA referred to by the requested binding is valid. Because extensions, such as the one we propose, are included in the `CertificateVerify` message, in creating the `CertificateVerify` the respondent signs the requested binding with the certificate contained in the EA. This relationship is highlighted in Figure 5.2.

As can be seen in Figure 5.2b, the authentication property only goes from EA_2 to EA_1, i.e. the later EA authenticates the earlier EA, but not



(a) The TLS channel authenticates both EAs, and each EA authenticates the TLS channel, but neither EA authenticates the other.

(b) The TLS channel authenticates both EAs, and each EA authenticates the TLS channel, and EA_2 authenticates EA_1.

Figure 5.2: Compound Authentication in Exported Authenticators vs Layered Exported Authenticators.

vice versa. This is in line with our definition of OCA, because EAs do not establish secrets they can only authenticate older EAs, and not future EAs.

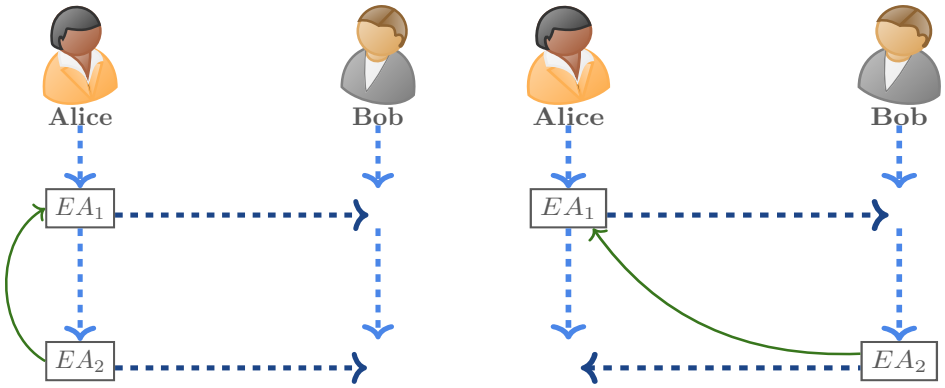
As we proved via a counter-example in Section 4.7.4, EAs sent on the same channel do not authenticate each other.[2]

5.4 LEAs under the Bhargavan framework

We here consider the authentication properties of LEAs under the Bhargavan et al. framework [BDP15]. The analysis proceeds very similarly to the analysis of EAs in Section 4.4.6, we simply have to adjust the `params` of LEAs in order to account for the different channel binding.

Consider a channel, t, on which an EA, EA_1, has already been sent. The parameters, $params_2$, of the second EA, EA_2, are slightly different to those of EA_1 in this case, specifically cb_{in}, highlighted in red, changes. The cb_{in} in this case is no longer the channel binging of the TLS channel, i.e. `exporter_master_secret`, but the requested binding.

[2] An attacker who can compromise a TLS channel could send an illegitimate EA, and later, legitimate EAs would be accepted.



(a) Alice first sends EA_1, and then EA_2, which she binds to EA_1. When Bob receives EA_2 he knows (1) that Alice claims to own the certificate in EA_2, and (2) that Alice claims to own the certificate in EA_1.

(b) Alice sends EA_1, and then Bob sends EA_2, which he binds to EA_1. When Alice receives EA_2 she knows (1) that Bob claims to own the certificate in EA_2, and (2) that Bob claims that EA_1 is valid.

Figure 5.3: Self-self bindings vs self-peer bindings

$$
\begin{aligned}
params_2 &= (c_i, c_r, sid, cb, cb_{in}) \\
&\text{where} \\
c_i &:= \bot, \\
c_r &:= Cert_{r_2}, \\
sid &:= \texttt{certificate\_request\_context}, \\
cb &:= EA_2, \\
cb_{in} &:= \text{requested binding}
\end{aligned}
$$

The property we want to prove has two cases, illustrated in Figure 5.3, depending on whether the requested binding refers to an EA created by the recipient of the new EA, or by the author's peer. If the peer signs a certificate that it created, it is claiming that it controls both certificates. If the peer signs a certificate it didn't create, then it is claiming that it received the referenced EA, and that the referenced EA passed its validation checks. Consider an actor, A, receives an EA, EA_2, of this type, i.e. one that is

linked to an EA, EA_1, which the peer, p, claims was authored by A. If A does not recognise EA_1, then it must reject EA_2. If A accepts EA_2 the peer can know that A also believes EA_1 is valid.[3]

Property 5.4.1. OCA of an EA layer with an EA layer *created by the peer* (using Definition 4.4.3). If an actor, a, with instances $\{l_1, \ldots, l_i\}$ receives an EA, EA_p, in layer l_i, and EA_p is bound to an EA, EA_m, received by a in layer l_j such that $j < i$, and the attacker does not know the private key of the certificate in EA_p, i.e. l_i has an honest peer credential, then there exists an actor b with instances $\{l'_1, \ldots, l'_i\}$, such that each l'_k has the dual role to l_k and agrees with l_k on $params_k$ and sk_k. Specifically b owns the peer credential in l_i and b generated EA_m.

Property 5.4.2. OCA of an EA layer with an EA layer *created by the actor* (using Definition 4.4.3). If an actor, a, with instances $\{l_1, \ldots, l_i\}$ receives an EA, EA_p, in layer l_i and EA_p is bound to a prior EA, EA_m, sent by a in layer l_j such that $j < i$, and the attacker does not know the private key of the certificate in EA_p, i.e. l_i has an honest peer credential, then there exists an actor b with instances $\{l'_1, \ldots, l'_i\}$, such that each l'_k has the dual role to l_k and agrees with l_k on $params_k$ and sk_k. Specifically b owns the certificate in EA_p and b received EA_m in layer l_j and believes EA_m is valid.

An LEA may be linked to an ordinary EA, but it might also be linked to another LEA, forming a chain. To prove that LEAs constitute a secure composite authentication protocol we need to show two things.

1. That $\text{EA}_2 \leftarrow$ requested binding, i.e. that EA_2 is dependent on the requested binding in such a way that if the recipient of the EA accepts EA_2 then the signer and recipient both agree on the requested binding.

[3] Because of the two generals' problem, both parties cannot simultaneously be sure that their last acceptance was accepted.

2. That EA_2 has OCA with all EAs in the chain. According to Bhargavan hypothesis[4] this can be achieved by showing the requested binding constitutes a contributive channel binding over all the EAs in the chain.

We recall the definition of a contributive channel binding.

Definition 4.4.7. Contributive Channel Bindings (Definition taken from Bhargavan et al. [BDP15, p. 10]). If a compound authentication protocol consists of n protocol instances $\{l_1, \ldots, l_n\}$, the channel binding of l_n must be bound to the parameters and session secrets of all n instances $\{params_1, sk_1, \ldots, params_n, sk_n\}$, so that agreement on the channel binding guarantees compound authentication for the composite protocol.

By linking multiple EAs into a chain we can produce a composite authentication protocol of arbitrary depth, so we need to show that the requested binding is dependent on all the previous layers. The base layer is always the TLS channel, so the requested binding needs to be dependent on the master secret of the TLS channel. Because the `Finished` message is an HMAC keyed with an exporter from the TLS channel this dependency is met, only someone who knows the exporter key can generate (or validate) the `Finished` message. This is what allows the TLS channel to authenticate EAs, even though they have yet to be produced.

No other layer generates secrets and so the requested binding is vacuously dependent on $sk_2 \ldots sk_n$.[5]

[4] See Section 4.4.10 for a full description of the Bhargavan hypothesis.

[5] Alternatively, we could say the EAs themselves are secret, in as much as an attacker cannot predict them. Thus the binding would have to be dependent on the `CertificateVerify` messages. The `Finished` messages are, by construction, dependent on the `CertificateVerify` messages.

Because `params` are defined to be public values we simply need to create a dependency on the `params` of the requested binding to create a dependency on the full chain. The `params` of the requested binding include a cb_{in} which is dependent on the EA referenced in the requested binding, i.e. the requested binding of the requested binding, or the `exporter_master_secret`, depending on whether the requested binding refers to an LEA or an EA respectively. Secret values, i.e. sk, need to be explicitly included every time to prove that the author still has access to the secret value, with public values, the author only needs to ensure the recipient agrees on the values, thus an inductive agreement property suffices. The `Finished` message portion of the requested binding includes all the `params` as part of the transcript hash, and thus is dependent on all the `params`, including the cb_{in}.

5.4.1 Development of `draft-sullivan` and `draft-hoyland`

Before `draft-hoyland` was formally proposed to the IETF work was done in collaboration with the authors of `draft-sullivan` and HTTP/2.[6] The initial proposal was to use a technique called "CRC smuggling", where the requested binding was packed into the CRC, along with a fresh value that provided the security of the CRC. This technique meant that LEAs could be implemented without changes to `draft-sullivan`. However as of the fifth draft of `draft-sullivan` [Sul17], spontaneous certificates did not include a CRC. This meant that one could not create a reference to a spontaneous EA. However this was one of the most likely use cases, a server sending a number of joint authentications to the client at the start of a connection.

In the sixth draft of `draft-sullivan` [Sul18b] the spontaneous certificate flow was changed such that the CRC could be arbitrary, as opposed to blank. This meant that a server that wished to use LEAs could simply include a CRC in all spontaneous certificates it might wish to bind to.

However, the CRC smuggling approach still had a problem. How would the recipient of an EA know whether the peer was aware of LEAs? The CRC smuggling proposal therefore suggested a simple transformation of the CRC in the response, to indicate that the author was aware of LEAs.

[6] Thanks to Nick Sullivan and Martin Thompson

This was considered an overly complex solution, and meant that implementations of LEA would be non-compliant with `draft-sullivan`, thus the decision was taken to instead add a new `LayeredEA` extension to the `CertificateRequest` and `Certificate` messages. The TLS specification requires that both clients and servers ignore unrecognised extensions. Thus a respondent who was aware of EAs but not LEAs, would simply ignore the extension. An author who was aware of LEAs would echo the extension back in the `Certificate` message.

This still left some unresolved issues.

1. A server who sent a number of linked certificates could not determine if the client understood or accepted the binding.

2. Because a server might always choose to use the same CRC in a spontaneous EA[7], a client could not distinguish between a replay attack and an honest server legitimately creating the same EA twice. An EA has no other source of randomness, within the context of a single TLS session, other than the CRC, so if given the same stimulating event, it may send the exact same EA.

3. A misbehaving or poorly implemented server might simply echo the extension back, rather than correctly ignoring it. This is particularly a risk in the case where a CDN is being employed. If the origin server, rather than giving its keys to the CDN, is simply signing blobs sent to it by the CDN, the origin server might sign a LEA even if it doesn't recognise the extension let alone the requested binding.

We discuss each of these issues in turn. Issue 1 is very similar to the mismatch issue in our TLS 1.3 analysis, see Section 3.5.2. In TLS 1.3 the client is never sure if the server has accepted his certificate, because there is no explicit acceptance signalling mechanism. In this case the issue occurs in the reverse. Because the client has no acceptance signalling mechanism, the server doesn't even know if the client has accepted the EA, let alone understood or processed the binding. The server can only know if the client

[7] Recall that until draft six of `draft-sullivan` this was compliant behaviour

has accepted its binding if the client binds to them in turn. To achieve this, the server would need to send a layered `CertificateRequest` to the client, with a requested binding to the last LEA the server sent. If it receives a bound response the server knows the client accepted all the certificates in the chain. This in effect becomes a signalling mechanism, and is a potential use case for LEAs.

Issue 2 was resolved by requiring that LEAs with duplicate certificate request contexts be rejected. We could thus ensure that an honest client would never accept the same LEA twice. Whilst a server might create the same spontaneous EA twice, by rejecting spontaneous certificates with the same certificate request, an honest client would never accept a repeated certificate. After discussion with the authors of `draft-sullivan` at the IETF 102 meeting, the seventh draft of `draft-sullivan` [Sul18a] required an honest server to never reuse a CRC for EAs on a given channel. A client still needs to maintain a list of EAs sent on a channel, or at least their certificate request contexts, to ensure freshness, because the spontaneous flow is a single message protocol, so nonces cannot ensure freshness in the usual way. However, because the scope within which nonces must be unique is restricted to a single TLS session, this is a viable approach.

Issue 3 was raised in discussions at the IETF 102 meeting. Because formal analysis generally does not consider misbehaving actors this was a particularly useful contribution.[8] The suggested resolution was to limit the request to the first half of the `Finished` message, and require the responder to include the second half. Although we treat the `Finished` message as an atomic symbol in practice it is the bit-string output of a hash function. A server that simply returns all extensions would return the first half of the `Finished` message, not the second. If two EAs had the same prefix this would introduce an ambiguity. However, because the hash function with the smallest output offered by TLS 1.3 is 256 bits [Res18, p. 131], it is infeasible to find a collision even on half of the hash output.[9] Although this requires the server to at least minimally process the extension, and to prove that it

[8] Thanks to Eric Rescorla for pointing this out.

[9] Specifically a 2^{-128} chance. This is not derivable from the properties we describe on hash functions, but stems from properties of well behaved hash functions like diffusion.

has a record of the previous hash, it doesn't obviously solve the problem caused by a misbehaving CDN treating the origin server as a signing oracle. Further analysis of this tweak is left for future work. An alternative proposal, enabled by the uniqueness of CRCs, was to simply include the CRC in the `CertificateRequest`, and the `Finished` in the `Certificate`. This has the advantage of an actor being able to request a chain of LEAs without having to wait to receive each LEA, and its attendant `Finished` message, before being able to request the next one. Analysis of this suggestion is also left to future work, but we note that this still establishes agreement on the channel binding, as the `Finished` message is dependent on the CRC, so even though the CRC is not included explicitly in the response, the requester can be sure that the sender agrees on its value.

5.4.2 Achieving full compound authentication between EAs

It is interesting to note that it is possible to achieve full compound authentication in this design by requiring the author of the EA to sign an EA with the private key of every EA it authored in the chain for each new EA it creates. This effectively creates a "secret" value, i.e. the chain of signatures, which cannot be guessed by the attacker in advance, and then immediately publishes it. Whilst this method might be theoretically interesting, it is not practical. The signing operation is expensive to compute, and the signature is large, making the EA difficult both to construct and to validate. Further this would require a change to the structure of the EA message, adding extra `Certificate` and `CertificateVerify` messages. Finally, in practice a TLS server might not have continuous access to the private keys of certificates, for example a CDN might not have direct access to the private keys of the origin server. The origin server might perform signing operations as necessary and pass them back to the CDN.

5.5 Authentication forests

Because at any time either party can initiate a request, and the server can send spontaneous certificates the parties may not agree on the order that they saw various EAs. When EAs are not linked this is not problematic,

because one EA can not affect the authentication status of another. However when building a chain of authentication this is more problematic.

If both parties are trying to build a single chain, and two requests pass each other in mid-flight then at least one party will have to make their request a second time to bind the response into the chain. Consider a TLS channel, t, on which an EA, EA_1, has been established. Let the server and client both send a layered request binding to EA_1, $R_S^{\mathrm{EA}_1}$, $R_C^{\mathrm{EA}_1}$, respectively. If both parties respond the authentication tree forks, with EA_{R_S} and EA_{R_C} binding to EA_1. To construct a single chain one party, for example the client, then must remake its request to bind to the new EA, $R_{C2}^{\mathrm{EA}_{R_S}}$, and the server must recompute its response, $\mathrm{EA}_{R_{C2}}$.

We propose two mechanisms by which such clashes can be resolved without either side having to compute an extra authenticator.

1. Separate the client and server LEAs into two distinct chains.
2. Require that the server EA always be placed first on the chain.

By separating the authentication chains of the client and the server we avoid any ordering issues where two EAs pass each other mid-flight. Each actor maintains the order of its own LEAs, and thus no excess LEAs are ever computed. This approach has a number of drawbacks.

1. If a client request and spontaneous server certificate cross in mid-flight the client still needs to re-request the certificate, although no extra computations are done by the server, it can simply ignore the request.
2. Neither party gets any confirmation that its EAs are being accepted. One of the advantages of LEAs is that the actor knows whether the peer has validated its EAs.

The second approach requires both sides to fulfil all requests before making one of their own, and further that the client remake any request if it receives a request between making a request and receiving a response. This approach also has drawbacks.

1. The server can prevent the client from ever successfully making a request.

2. Both parties could potentially spend a lot of time waiting.

An alternative to both these approaches is to accept authentication forests. By viewing authentication properties not as chains, but as trees, with each ordinary EA forming the root of a new tree, creating a forest. This allows for either party to construct highly complex authentication properties. These may have uses in complex CDN set-ups, where a CDN is authorised to act for multiple groups of servers, but not for all servers together.

5.6 Tamarin model

LEAs are simply EAs with an extra extension. We therefore simply extend our EA model from Chapter 4. Whilst the changes 'on the wire' are minimal, the only difference being an extra extension is added to the list, the client and server logic become notably more complex. The guarantees we require are also more complex.

5.6.1 Protocol changes

Our EA model treats the extension field as a 'blob', i.e. a field with no specific meaning. Specifically we declare a static public value `$certificate_extensions`. This is a fault-preserving simplification, assuming that extensions are not dependent on secret values. For example if an extension included the master secret as a parameter, modelling it as a static public value would not detect the mistake. However given that new extensions can be defined we assume some general unspecified notion of well-behaved-ness of extensions.

To extend our model to capture LEAs we redefine the certificate extensions to be a pair[10]:

```
1 <$certificate_extensions,
2         <prev_certificate_request_context,
3          prev_Finished>
4 >
```

[10] Lightly edited for clarity.

Because Tamarin implicitly types variables in rewrite rules, this change does not make the EA model incompatible with the LEA model. If we were to use the rules of the EA model for the server and the rules for the LEA model for the client, the models should still interact correctly. An actor that was not aware of this change would simply continue to treat the certificate extensions as a blob, correctly ignoring the extension it didn't understand.

5.6.2 Processing logic

To process the LEA variant we must duplicate all the processing rules, to allow for processing a LEA and an ordinary EA. This is necessary because to send a LEA there must either be a earlier LEA or an EA for it to bind to. Thus all the rules for EAs must remain in place to be able to run the layered section of the protocol. We split the rules into "free" and "bound" versions, to indicate whether they deal with EAs or LEAs respectively.

The extra processing logic we require is as follows:

1. We need to remember past authenticators and past bindings, such that we can reference them in future LEAs and requests.
2. We need a mechanism for deciding to request a binding.
3. We need to verify that requested bindings are in memory before signing them.
4. We need to be able to validate layered responses.

Remembering EAs and bindings

Only the first of these affects the "free" rules.

We extend the "free" rules to output a fact that acts as a reference to an EA that we call an EA handle.

The handle we use for server rules is as follows.

```
!EA_HandleS(~ea_id, ~sid, certificate_request_context,
↪  Finished)
```

We provide a sample rule using this fact in Figure 5.4. The rule is prefixed with a `!`, indicating to Tamarin that the fact is persistent, i.e. that it shouldn't be consumed on use. This allows an actor to reference a previous EA multiple times, allowing for a tree of authentication properties.

The `~ea_id` is a unique number used internally by the actor to reference the EA. Although the CRC is supposed to be a unique reference number for an EA because it is at least sometimes chosen by the peer an actor cannot be sure it is unique. A misbehaving or malicious peer could be duplicating CRCs, or an attacker could be inserting extra EAs into the channel. The `Finished` message is unique for a given transcript, however a misbehaving server might send the same spontaneous certificate twice, and thus a client cannot be sure that a given `Finished` message is unique without comparing it to all previous `Finished` messages. By using a number generated internally an actor can be sure the reference is unique, as indicated by the `~`-prefix.

As in both the TLS and EA models, the `~sid` indicates the server's thread ID. The `certificate_request_context` and `Finished` fields capture the information needed for a requested binding.

For the "bound" rules we need the EA handle, but we also need to add facts for remembering bindings.

```
!EA_Binding(~ea_id, ~prev_ea_id, ~sid, ms, Finished, binding,
↪  'local')
```

As in the EA handle fact, the EA binding fact is persistent. The binding fact contains the `ea_id` of both the current EA and the previous EA.

The binding fact also contains the master secret, *ms*, to simplify the lemma writing. Although the master secret is included in the `Finished` and `binding` arguments, having it separated out into an argument allows it to be referred to directly.

The `binding` argument is the previous `Finished` message, assigned a different name to prevent a name clash with the `Prev_Finished` in the certificate extensions.

The final parameter is the place where the binding was generated. If the actor performed the binding this value is set to 'local'. If the peer performed the binding this value is set to 'remote'. Figure 5.4 shows the `S_Send_Bound` rule that outputs this fact.

Requesting a binding

To minimise restrictions on the usage of LEAs, we simply allow the actor to bind to any EA in its memory, i.e. for which it has an EA handle.

Verification of LEA requests

When an actor receives a request before it sends a response it needs to ensure that the requested binding is valid. By only producing EA handle facts (1) after a received EA has been validated, or (2) when an EA is created by the actor, the simple presence of an EA handle in memory[11] is a sufficient check. This corresponds to the two potential meanings of a LEA, that the requested binding refers either to an EA that the actor accepted, or that the actor created. In Figure 5.4 we show a bound server send rule. The server requires an `EA_Handle` fact from the environment (line 14), and creates a new `EA_Handle` fact for the newly created EA (line 34).

Verification of LEAs

When receiving a LEA, simply consuming the relevant `!EA_Handle` fact is sufficient. If the fact is in memory it means that the actor considers the referenced EA valid. This follows the same logic as EA requests.

5.7 Proving the model

In this section we discuss the process of proving properties of our model. We prove a partial result about LEAs, proving that in most cases LEAs authenticate the EA they are bound to. The case that does not prove is in fact one we do not expect to hold. In particular, if an actor binds an LEA to an EA that it previously received it does not learn anything new about the

[11] Note that although the bag of facts is global, including the `tid` in the handle gives each actor a disjoint memory space.

```
1  rule S_Send_Bound:
2    let
3      binding = prev_Finished
4      certificate=pk(ltkA)
5      signature=compute_signature(ltkA, h(hccs, CertificateRequestB,
       ↪  CertificateB))
6      verify_data = hmac(fms, <hccs, CertificateRequestB,
       ↪  CertificateB, CertificateVerify>)
7    in
8      [ State(~sid, 'S1', $S, $C, 'server', auth_status)
9      , !StateInvS(~sid, cid, ms, $C, $S, 'server', certC, certS)
10     , PendingReqR(~request_id, ~sid, $S, $C, 'server',
       ↪  CertificateRequestB)
11     , !PendingReqRSBInvariant(~request_id, ~sid, $S, $C,
       ↪  CertificateRequestB)
12     , !ExportersS(ms, 'server', ~sid, hccc, hccs, fmc, fms)
13     , !Ltk($A, ltkA)
14     , !EA_HandleS(~prev_ea_id, ~sid, p_crc, binding)
15     , DelegateLtk($S, ltkA)
16     , Fr(~msg_id)
17     , Fr(~ea_id)
18     ]
19   --[ S_Send(~sid)
20     , Owns(S, ms, 'server', <$A, pk(ltkA)>)
21     , Instance(~sid, $S, $C, 'server')
22     , Source_In(~sid, ms, crc)
23     , Source_In(~sid, ms, p_crc)
24     , Source_In(~sid, ms, binding)
25     , Source_Out(~sid, ms, Finished)
26     , Fulfil(~sid, $S, $C, ms, 'server', crc, <$A, pk(ltkA)>)
27     , FulfilB(~sid, $S, $C, ms, 'server', crc, <$A, pk(ltkA)>,
       ↪  binding)
28     , Bind(~ea_id, ~prev_ea_id, ~sid, ms, Finished, binding,
       ↪  'local')
29     , Finished_In(~prev_ea_id, ~sid, ms, p_crc, binding)
30     , Valid_Finished(~ea_id, ~sid, ms, crc, Finished, 'local')
31     ]->
32   [ State(~sid, 'S1', $S, $C, 'server', auth_status)
33     , TLS_Send(~msg_id, ~sid, $S, $C, <CertificateB,
       ↪  CertificateVerify, Finished>)
34     , !EA_HandleS(~ea_id, ~sid, crc, Finished)
35     , !EA_Binding(~ea_id, ~prev_ea_id, ~sid, ms, Finished, binding,
       ↪  'local')
36   ]
```

Figure 5.4: The `S_Send_Bound` transition. The lines marked in black (9, 11) highlight invariant shortcuts, see Section 5.7.2 Experiment #9. The lines marked in red (14, 34-35) highlight memory facts, see Section 5.6.2. The lines marked in blue (22-25) highlight source actions, see Section 5.7.2 Experiment #10.

validity of the received EA. Although we do not expect this case to hold, that it does not makes it challenging to reason about the security of LEAs inductively. We thus prove a partial result, showing that, where expected, LEAs authenticate the EA they reference.

Even proving this partial result tests the limits of Tamarin. The arbitrarily deep layering of LEAs gives Tamarin difficulties. We discuss at length the internal mechanisms of Tamarin that struggle, before describing how we model the protocol, allowing us to prove the results we do achieve.

5.7.1 Source resolution in Tamarin

One of the difficulties we faced in building our LEA model was simply getting the model to load in Tamarin. Specifically we had issues with the way the Tamarin pre-loader determines where specific values could have come from. Whilst performing a backwards search Tamarin looks at the facts in the desired end state and attempts to reason about where they could have come from. To do this it enumerates the rules which output those facts. It then selects a rule that produces a given fact and replaces the fact with the inputs or preconditions to the selected rule. It proceeds on this basis until it reaches an impossible state, i.e. one where no possible rule could be applied[12], or it arrives at the empty state, i.e. it has found a sequence of rules that, starting from the empty state, reach the desired state. Because of the way lemmas are commonly written, generally if Tamarin can prove a state unreachable then the lemma is proven, and if Tamarin reaches the empty state then the trace is a counter-example.

To make the backwards search more efficient Tamarin pre-computes all the potential sources of each fact. This pre-computation step allows Tamarin to exclude many impossible traces from consideration. By applying some heuristics Tamarin is usually able to resolve all sources in the model[13], however, in some cases Tamarin's heuristics fail, and some sources remain

[12] If no rule can be applied in backwards search, then from a forwards search perspective, the state is unreachable.

[13] A resolved source is roughly one in which Tamarin can fully derive where everything came from. Often a single fact will have many sources, each fully unrolled to the empty state.

unresolved, these cases are said to have *partial deconstructions*. Cases where all sources have been resolved are said to have *complete deconstructions*. These unresolved sources dramatically increase the difficulty of proving, even in the human-guided prover, because they introduce many spurious cases that must be shown impossible. We describe a case as spurious if we can easily derive from meta-reasoning that it must be impossible without proving properties about it in Tamarin. For example Tamarin might not be able to exclude the possibility that an honest actor uses its key as a public nonce during the precomputation phase. In the case of our LEA model these spurious cases number in the hundreds for each proof, rendering a manual proof infeasibly time consuming. Further, because some unresolved sources may have cyclic dependencies, some spurious branches may be insoluble.

Usually these hard-to-resolve sources revolve around actions performed by the attacker. We illustrate the problem with our running example of the Needham Schroeder protocol, or rather with the Lowe variant.[14] This is the protocol used in the Tamarin manual [TAMARIN], and we reuse their example code, providing a fuller exposition of the functioning of Tamarin's internal mechanisms.[15]

One of the properties achieved by the Needham-Schroeder-Lowe protocol is nonce secrecy. By this we mean that an attacker who does not reveal the LTKs of the participants cannot learn the values of the nonces.

When loading the model Tamarin looks for all potential sources of knowledge for an attacker. If we load the protocol model Tamarin tells us that there are twelve sources with partial deconstructions. These all revolve around the attacker's `!KU` fact. The `!KU` fact is a special fact, that signals that the attacker has learned something new, e.g. `!KU(ltk)` indicates that the attacker has learned `ltk`.

The attacker's rules for deriving information from messages are split into two sets, those that operate on `!KU` facts, called `K` up rules, and those that operate on `!KD` facts, called `K` down rules. `K` up rules construct new

[14] Although the issue occurs in both variants the Lowe variant is easier to work with. In the non-Lowe variant Tamarin can find the attack even without resolving all the sources. In the Lowe variant Tamarin simply cannot decide the protocol one way or the other.

[15] We reproduce the code of their example in full in Appendix C.

objects from previously known objects, whereas `K` down rules deconstruct previously known objects, to learn their components. This split is used to prevent Tamarin considering traces where an attacker repeatedly constructs and then deconstructs the same object. This is achieved by requiring all `K` down rules to come before all `K` up rules. There is a special rule, `coerce`, see Figure 5.6b, that transforms a `!KD` fact into a `!KU` fact, but no rules that perform the opposite transformation.

In Figure 5.5 we see one of the partial deconstructions. We will describe this figure over the next few paragraphs. We enlarge portions of the figure in Figure 5.6. In this figure Tamarin is attempting to derive all places an attacker could derive some new piece of knowledge, i.e. solving for `!KU` facts. In this case, the attacker learns the responders nonce from the initiators second message.

For the deconstruction of a source to be *complete* Tamarin must be able to show that every antecedent can be unrolled until it reaches the empty set. Tamarin's heuristics stop it unrolling the same rule twice, to prevent it looping on cyclic derivations. The simplest way for Tamarin to show that a source can be fully unrolled is to unroll the antecedents until they reach the empty set. However Tamarin does not always need to fully unroll every rule. The `Fr` fact is always available, for example, so Tamarin can stop unrolling when it reaches a `Fr` rule, and know that it can be unrolled to the empty set.

As we can see in Figure 5.5 there are two `!KU` facts that haven't been unrolled.[16] A fact that hasn't been unrolled is marked with an oval, whereas unrolled rules are marked with rectangular boxes. Unrolled rules have three layers, along the top are the antecedent facts, in the middle are the actions, marked with a symbolic time, and on the bottom are the consequent facts. Tamarin uses backwards search, so these diagrams are read from the bottom upwards. Solid arrows indicate the direct sources of facts, and dotted arrows indicate unrolled facts that must precede a given rule.[17] For example, the rule at time `#vr.10`, see Figure 5.6c, shows the attacker decrypting a mes-

[16] These facts are enlarged in Figures 5.6d and 5.6e.

[17] Tamarin can also have actions that it knows must precede a given rule, although there are none in this example.

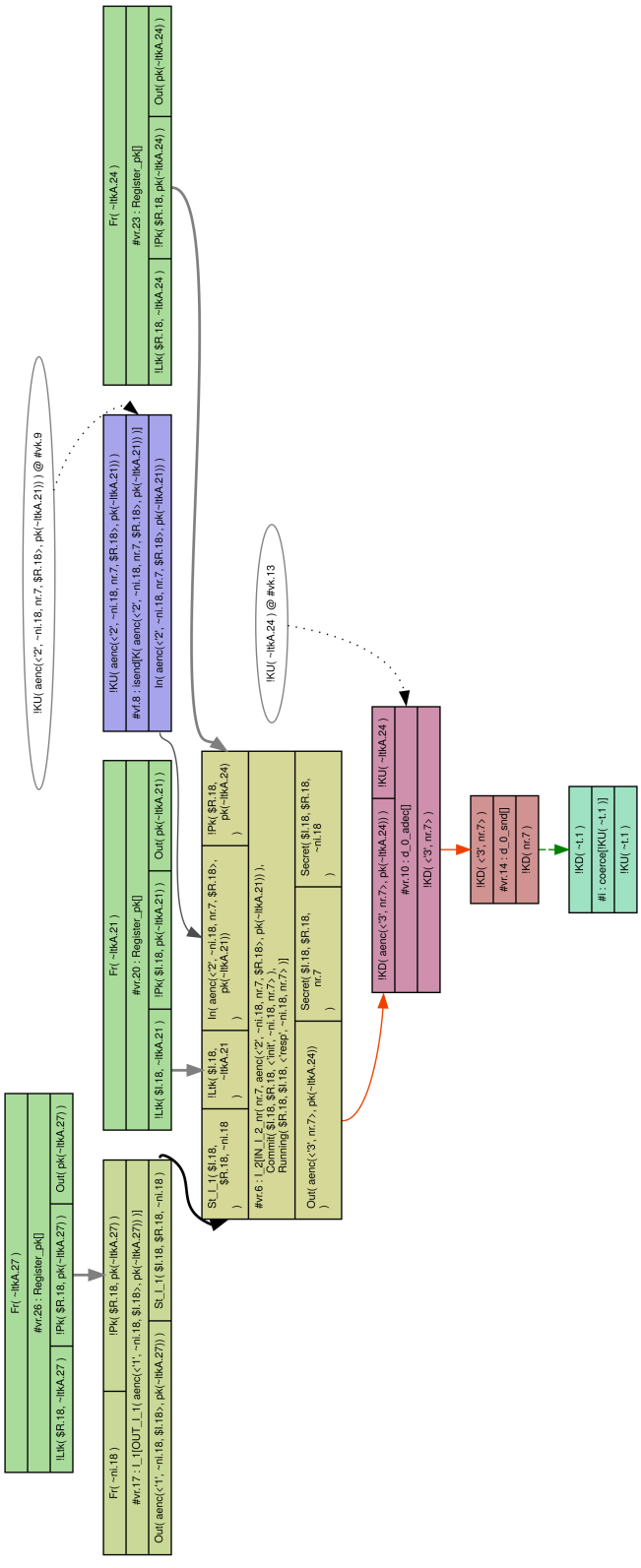
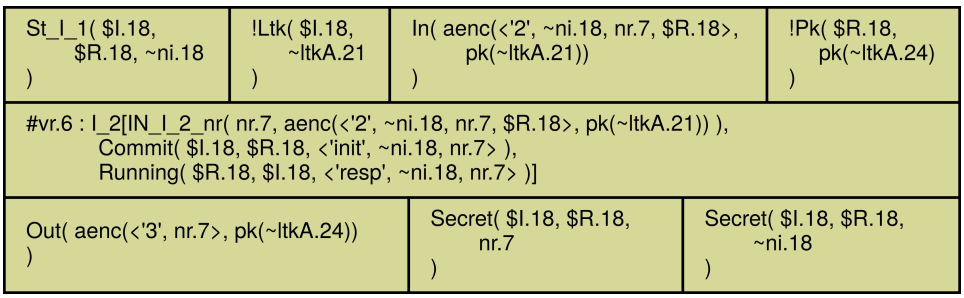
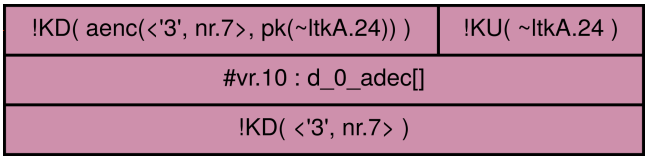


Figure 5.5: This is a diagram of one of the partial deconstructions found by Tamarin for the Needham-Schroeder-Lowe protocol. Reading from the bottom, we see that Tamarin is seeking the source of `!KU(~t.1)`. The green arrow indicates that Tamarin cannot rule out `!KD(nr.7)` as a source. We include enlarged versions of some key rules in Figure 5.6.

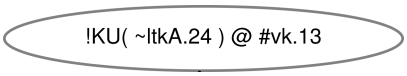


(a) The `I_2` rule captures the Initiator receiving a message from the Responder, and replying in turn.

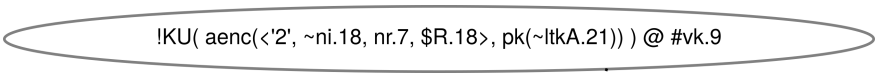
(b) The `coerce` rule tranforms a `!KD` fact into a `!KU` fact.



(c) The `decrypt` rule is a `K` down rule, breaking a large encrypted message, into it's constituent components.



(d) This unrolled `!KU` fact has type `!KU(~t.1)`.



(e) This unrolled `!KU` fact has type `!KU(aenc(t.1, t.2))`.

Figure 5.6: Enlarged rules

sage. Tamarin knows that at some point the attacker must have learnt the key to decrypt the message. Thus even though it doesn't know when, where, or how the attacker learns the key, Tamarin draws a dotted arrow from the point where the attacker learns the value (through a `!KU` fact) to where the attacker uses the value.

As we mentioned previously Tamarin can determine a source can be unrolled by unrolling it, however Tamarin can also use inductive reasoning to determine that a source can be unrolled to the empty state. When searching for the sources of all `!KU(~t.1)` facts, for each case Tamarin can either show that `!KU(~t.1)` can be reached without preconditions, or show that it can be reached if some other `!KU(~t.1)` fact can be reached. Assuming there is at least one case that can be reached without preconditions, cases with a dependency on an older `!KU(~t.1)` fact can be reached.[18] Thus when Tamarin is sourcing `!KU(~t.1)` and it reaches a `!KU` rule with arguments of the same type[19] it can apply the inductive hypothesis, and stop unrolling.

This makes Tamarin's heuristic of unrolling until it reaches the same rule very effective in general. However, as we can see in our case, when Tamarin reaches a `!KU` rule with a different type it also stops unrolling. In this case we have a `!KU` rule at time `#vk.9`, see Figure 5.6e, which takes an argument of type `aenc(t.1, t.2)` as opposed to `~t.1`. Thus Tamarin stops unrolling, but cannot apply the inductive hypothesis, and thus cannot be sure that every antecedent unrolls to the empty state. This gives us a partial deconstruction.

Partial deconstruction are problematic because it means Tamarin cannot derive any restrictions on the data the attacker learns. In our example Tamarin cannot derive any restrictions on what the value of `nr.7` might be. `nr.7` appears in the unrolled `!KU` fact at `#vk.9`, and passes through the blue `isend` rule, which indicates the attacker sending a message. This message is passed on to the `I_2` rule, shown in Figure 5.6a. The `I_2` rule represents the

[18] In this instance, an attacker can, without preconditions, learn a fresh value from a `!KU(~t.1)` fact, thus the base case is trivial.

[19] `!KU` rules with arguments of a different type can be considered overloaded rules, i.e. although they have the same name, because they have a different type, they are different rules.

initiator receiving the responder's $\{n_A, n_B, B\}_{pk(ltk_A)}$ message and sending its response. In this case `nr.7` represents the responder's nonce, n_B.

Though it is intended to be a nonce, there might be some complex interaction of rules that cause an honest actor to use some other value. If, for example, the attacker could trick the responder into using the private portion of their LTK as the nonce the attacker could learn their private key.

This means that every time we wish to prove the attacker has not learned an actor's LTK we must prove that it was never used as a respondent's nonce. If the attacker can acquire a message that appears to come from the responder in which the respondent uses the key as its nonce, then the initiator will repeat this value, potentially allowing the attacker to learn it. Because Tamarin uses backwards search it will look for the source of this responder message, and find two possible sources.

1. Either the responder sent it, or
2. the attacker knows the responder's secret key, and constructed the message.

We can exclude the first of these possibilities, as the respondent always chooses a fresh value for its nonce, but we must consider the second case. Because our nonce secrecy lemma requires that the attacker not perform a key revelation, Tamarin looks for other places the attacker might have learnt the key. However, as we have just shown, Tamarin believes that the attacker can learn keys from the initiator's second message. This creates an infinite chain of key derivations, where Tamarin decrypts a key using a key it learned earlier, which it learned using an even earlier key, and so on. From a forward search perspective it is clear that the attacker cannot learn anything new from this chain. For the attacker to create a message containing the key it must have already known the key. Thus it must have already learned the key from some other source.

To solve this problem Tamarin introduces source lemmas. Source lemmas are provided by the modeller and give hints to Tamarin's pre-computation step that allow it to resolve complex sources. Tamarin divides the pre-computation step into three phases:

1. Tamarin resolves all the sources it can, producing a set of sources called the raw sources, potentially with partial deconstructions.

2. Tamarin then attempts to automatically discharge any source lemmas using induction.

3. Tamarin applies the source lemmas to the raw sources, producing a new set of sources called the refined sources.

A source lemma might prove that all traces for a given rule, a are preceded by some action (or set of actions) b. Any raw source that contains a, but does not have b as a possible precursor is thus excluded as impossible. This can exclude sources both with partial deconstructions and complete deconstructions. The goal is to construct source lemmas such that we can exclude as impossible all partial deconstructions.

In Figure 5.7 we show a fragment of the source lemma Meier uses in his Needham-Schroeder-Lowe protocol model. This lemma states that either the attacker knows the nonce before the initiator receives it and thus that the attacker cannot learn the nonce from the initiators message, or the message was sent by an honest actor. Looking at Figure 5.5 we can see that this will split the partial deconstruction into two cases around the `I_2` rule, which contains the `IN_I_2_nr(nr, m2)` action.

The first is where the attacker knows `nr.7` before it sends the message to the initiator. Although a multi-set allows duplicates, Tamarin will not consider traces where an attacker learns a fact it has already learnt. This allows Tamarin to discount this trace.

The second is where the message was honestly sent by the responder. Because the nonce chosen by the responder is always fresh Tamarin can be sure that the value is not used elsewhere as a key.[20] This allows Tamarin to restrict the information it can learn from this source to fresh nonces. By excluding traces where the attacker learns a key from this source in pre-computation Tamarin removes the problematic loop.

[20]Rather, Tamarin can be sure that it is distinct from all other fresh values, including those used as keys.

```
lemma types [sources]:
" [...]
  & (
    All nr m2 #i.
      IN_I_2_nr(nr, m2) @ i
    ==>
    ( (Ex #j. KU(nr) @ j & j < i)
    | (Ex #j. OUT_R_1(m2) @ j)
    )
)"
```

Figure 5.7: A fragment of Meier's source lemma for the Needham-Schroeder-Lowe protocol model. This lemma says that if the initiator uses `nr` for the responder's nonce in message `m2` then either the attacker knew `nr` before the initiator received `m2` or an honest responder sent the message `m2`, triggering the `OUT_R_1(m2)` action.

5.7.2 Proving lemmas during pre-computation

In our LEA model we had many unrefined sources which introduced many spurious branches within proofs, to the extent that manually discharging them became infeasible. However, we struggled to write source lemmas that refined the sources. Some source lemmas did not refine the sources, and some source lemmas caused Tamarin to loop apparently indefinitely. All our work was done on a server with 32 cores and 500GB of RAM.

The way in which Tamarin discharges lemmas during pre-computation is not well documented. We thus applied the scientific method to try and construct source lemmas that refined all the sources.

Experiment #0: Problem statement

Our first attempt to load the model with no source lemmas produced 112 unrefined sources. The Tamarin interactive prover has a section called "Raw Sources". This section has diagrams detailing the precursors for all the sources Tamarin has computed. Sources are marked with the phrase "partial deconstructions" if there are pre-conditions that Tamarin cannot resolve. Careful exploration of these partial deconstructions indicated that there were three values that Tamarin could not fully deconstruct. The first was

the CRC, which was a problem in our EA model, that we resolved with a source lemma. The other two problematic values were the previous CRC and the previous Finished, i.e. the two items that make up the requested binding. The two values are created in different places, and thus require different source lemmas. The previous CRC was created in one of three locations:

1. The client request,
2. the server request or
3. the server spontaneous send.

The previous Finished, however is created in:

1. the client send,
2. the server send,
3. or the server spontaneous send.

To resolve these we wrote three lemmas, `crc_n_source`, `p_crc_n_source`, and `binding_source`. The `crc_n_source` lemma was taken from the EA model, and the other two lemmas followed the same style.

```
lemma crc_n_source[sources]:
  "All tid ms crc #k.
      CRC_In(tid, ms, crc)@k
  ==>
    ( Ex tid2 #i.
        CRC_Out(tid2, ms, crc)@i
      &  (#i < #k)
    )
    |
    ( Ex #j.
        KU(crc)@j
      & (#j < #k)
    )"
```

We added a `CRC_In` action every time a CRC was received from the network, and a `CRC_Out` action every time a CRC was created. This lemma thus claims that if a CRC is received over the network at time `#k` then either it was created by an honest agent at some earlier time `#i` or the attacker knew the value at some earlier time `#j`. Because an honest agent only uses fresh values as a CRC Tamarin can derive that the only new knowledge the attacker can learn from a CRC is a fresh nonce. This effectively restricts the type of the CRC, such that Tamarin can mark as impossible any path that requires a different type.

The other two lemmas proceed similarly, with the caveat that the binding is also required to be valid.

However when these lemmas were inserted the model would no longer load, exhausting all the available RAM and crashing.

Experiment #1: Lemma correctness

At the suggestion of the authors of Tamarin[21], our first attempt was to load our source lemmas not as a source lemmas, but as an ordinary inductive lemmas, and see if Tamarin's automatic prover could discharge them. Tamarin's pre-computation step does not use helper lemmas, so the lemmas had to be constructed such that they prove automatically without the aid of any helper lemmas.

After a small number of iterations this produced lemmas that would prove automatically under the 'S' heuristic.[22] These lemmas auto-proved in seconds when loaded as ordinary inductive lemmas. However when attempting to load these lemmas as source lemmas, Tamarin would exhaust all available memory and crash.

Experiment #2: Model subsets

Our second experiment was to exclude spontaneous certificates from our model. In all other cases the unresolved sources came from both the actor

[21] Thanks to Kevin Milner and Cas Cremers.

[22] The 'S' heuristic orders goals using the "smart" heuristic, and allows so called "loop breakers". Loop breakers are goals that Tamarin usually demotes because they are likely to cause a loop.

and the peer, i.e. if the previous CRC was created by one party, then the binding came from the other. This meant that both parties had contributed something to the agreed value. However this test had no impact on the result.

Further consultation with the authors of Tamarin revealed that, although the pre-computation step does not use helper lemmas, if there are multiple source lemmas they do interact. In the pre-computation phase Tamarin always unrolls actions if they appear on the LHS of a source lemma. When Tamarin found the source of binding, i.e. in a `Send` rule, the rule would have an unresolved CRC.[23] Tamarin would attempt to apply the `crc_n_source` lemma, which would unroll to a `Request` rule. If the request was a bound rule, i.e. a request for an LEA it would contain an unresolved binding. Tamarin would attempt to unroll this leading to a `Send` rule, creating an infinite loop.

Experiment #3: Restrict actions

In our third experiment we restricted the rules in which the actions occur to the minimum set of points possible in an attempt to prevent this loop from occurring. We cut the `prev_crc_n_sources` lemma entirely, merging it with the `crc_n_sources` lemma. However, limiting the set of actions enough to break the loop meant that Tamarin was unable to resolve the sources, because the reasoning chains became too long and Tamarin's heuristics stopped unrolling before the source was found. This meant that although the model loaded, the sources remained unresolved.

Experiment #4: Increase pre-computation

The authors of Tamarin suggested we try a development version of Tamarin with a more aggressive un-roller in the pre-computation phase.[24] This, it was hoped, would make the restricted actions sufficient to resolve the sources without re-introducing the looping behaviour. The more aggressive un-roller

[23] Except in the case of a spontaneous send. A spontaneous send creates its CRC in the same rule.

[24] This version of Tamarin can be found at `https://github.com/kmilner/tamarin-prover/tree/actionprecomp-and-annotes`.

however would unroll the unresolved sources much further, which caused many more case splits, and was still unable to capture the sources, resulting in more than 1000 partial deconstructions.

Experiment #5: Restrictions

Tamarin allows the user to define restrictions. These are effectively lemmas that Tamarin assumes hold at all times. They can be used to define relations such as equality.

```
restriction Eq_check_succeed: "All x y #i. Eq(x,y)@i ==> x = y"
```

This rule says that if two variables are present in an `Eq` action then they must be equal. This lets a user exclude cases where the terms are not equal from consideration. Using this feature we could thus attempt to resolve the sources in a two-step process.

1. Prove the `crc_n_source` and `binding_source` lemmas as regular inductive lemmas.
2. Reload the model with the two source lemmas as restrictions, effectively short-circuiting the partial deconstructions.

We attempted this, and found that Tamarin still found partial deconstructions. After discussion with the Tamarin authors we discovered that there was an undocumented restriction on restrictions. Tamarin only excludes sources that do not match restrictions if the RHS has no first order terms. The source lemmas have complex RHSs, and re-expressing them such that they have simple RHSs did not prevent the partial deconstructions appearing.

Experiment #6: Oracle

Our experiments had shown that each of our source lemmas could be made to load and resolve the sources they were designed to resolve individually, even in the unmodified Tamarin prover. By controlling the order of the preconditions of various rules we could make any of the source lemmas load

individually.[25] This technique worked by finding a path that resolved the source without touching on other unrefined sources. The problem was that choosing a path that, for example, avoided the rules that had unresolved bindings, letting the `crc_n_source` lemma resolve, would mean that when we tried to solve `binding_source` any rule on the path that intersected with the `crc_n_source` path would start to follow the same path as the `crc_n_source`. However this path was designed to avoid bindings, which meant Tamarin would search further and further up the tree, but never find the source. A path that was always finite for CRCs was infinite for bindings, and vice versa.

Solving at least one of the lemmas meant that we could remove at least some of the spurious cases. For example loading just the `crc_n_source` lemma resolved all but 28 of the 112 partial deconstructions.[26] We thus looked to methods we could use to more precisely control the order goals were solved.

Tamarin provides an oracle heuristic, which allows the user to provide a program that Tamarin will call to order the goals. If we could provide a different goal ordering for the two lemmas we would be able to solve both lemmas even though they needed different paths.

However, our experiments quickly showed that Tamarin does not honour the oracle heuristic during the pre-computation phase. This surprised some of the authors of Tamarin.[27]

Experiment #7: Identifying the pre-computation heuristic

We then ran a series of experiments to discern which of the heuristics Tamarin uses to resolve sources in the pre-computation stage. This experiment revealed that Tamarin does not honour any heuristic during the pre-computation stage, and uses its own undocumented heuristic. Discussion with the Tamarin authors revealed that, in contrast with the documenta-

[25] Controlling the unrolling through the order of the preconditions does not appear to be an intentional feature of Tamarin, but an artefact of the way the pre-computation phase operates.

[26] Because of state space explosion discharging even 28 spurious branches takes an infeasibly long time.

[27] Personal correspondence with Kevin Milner.

tion, Tamarin does not actually prove source lemmas in the pre-computation stage, but merely assumes them. It then uses those lemmas to guide its unrolling.

Experiment #8: Customising Tamarin

One of the authors of Tamarin produced a patch that would force Tamarin to only solve an action goal once whilst resolving sources. It was hoped that this would prevent sources from entering an infinite loop, terminating after one iteration. However it transpired that Tamarin will unroll rules for a number of different reasons, not just because it appears in the LHS of a source lemma. Thus the patch did not prevent the looping behaviour.

Experiment #9: Constructing shortcuts

In work on proving the security of Distributed Network Protocol 3 (DNP3) Cremers et al. [CDM17] present a novel approach to producing proofs of complex looping protocols in Tamarin. The DNP3 protocol has a state called "Security Idle" to which the protocol returns repeatedly. This is similar to the state machine of the LEA protocol, when considered as a single composite protocol. This gave them similar modelling difficulties, with many states leading to repeated loops. They identified invariants of the various transitions and created persistent facts that linked directly to their instantiation, i.e. the first point at which the invariant held. By prioritising solving these facts they could solve complex properties of this looping protocol skipping over the looping behaviour entirely.

By imitating this approach in our LEA model we were able to construct "short-cuts" to various sources, by producing persistent facts at the place they were generated and consuming them every time they are generated. This created a one-step resolution for many sources. This substantially decreased the amount of memory and time the model took to load, but did not resolve the underlying looping issue. The looping in our sources was caused by multiple sources interacting, not just the structure of the state machine. Figure 5.4 shows our `S_Send_Bound` rule, which uses two invariant facts. Line 9 captures the state invariant. Once the TLS channel is estab-

lished there are various aspects of the protocol state that do not change, such as the actor's role, and the server and client's thread identifiers. By emitting a persistent fact when a session is established and consuming it at every future state we can resolve all these variables in a single step. Line 11 captures the invariants associated with a pending request. Note that the variables are virtually identical to those in the non-persistent `PendingReqR` fact on line 10.[28] Having a non-persistent fact as an antecedent prevents the persistent invariant fact from being consumed multiple times, which would allow an actor to improperly respond to a request multiple times. Structuring the rules in this way allows us to take advantage of the invariant shortcut logic without introducing an over-approximation into our model. The `PendingReqRSBInvariant` fact is particularly crucial to resolve because the `CertificateRequestB` macro expands to include all three hard to resolve sources, namely the CRC, the previous CRC and the previous binding.

Experiment #10: Unified lemma ping-pong

Our final approach was successful in resolving all the partial deconstructions. The issue of the dual infinite paths seemed unresolvable. So we decided to write a single lemma that would resolve all sources simultaneously. Instead of having separate `CRC_In` and `Finished_In` actions we created a single `Source_In` action.

Because we were creating a single sources lemma there was no need to try and restrict the locations of `Source_In` actions, as we attempted in Experiment #3. Because source lemmas are proven inductively, if a rule with a `Source_In` action unrolls to find another `Source_In` action with the same parameters the lemma is considered to hold, by the inductive hypothesis. By placing `Source_In` actions at every point that a source was used, and a `Source_Out` action when the source was created, most cases of the unified source lemma were solved in a single step. In our `S_Send_Bound` rule, shown in Figure 5.4, we use three `Source_In` actions and one `Source_Out`

[28] The only difference is the lack of the 'server' field. This is established by the requirements on other antecedent facts, and by omitting this we can reduce memory usage. Tamarin stores strings in full at every occurrence, so strings in persistent facts are particularly memory intensive.

action (lines 22-25). Each of these sources resolves in a different rule. Whilst we cannot unroll each source to its parent rule in a single step, by adding `Source_In` actions liberally throughout the model we can resolve many of these using the inductive hypothesis. However this didn't address the problem of cyclic path dependencies.

A path that resolved CRCs efficiently didn't resolve bindings, and a path that resolved bindings efficiently didn't resolve CRCs. However, because we now had a single source lemma, we could design a path that ping-ponged between the two actors, visiting every use and every creation of every fact. By testing this lemma on larger and larger subsets of the model we were able to capture all the necessary cases on our path as efficiently as possible, varying the placement of the persistent invariant facts to control the path and prevent excess visits to non-resolving states. Loading the full model with the unified source lemma requires virtually all of the 500GB of RAM we had available. This is because the Tamarin is effectively verifying the sources of every potential binding and CRC in the pre-computation step, including previous CRCs, requiring it unroll two runs. Furthermore, because Tamarin does not make use of helper lemmas in the pre-computation step the analysis is not efficient.

5.7.3 Lemmas

LEAs have strictly stronger guarantees than EAs thus we need to prove that all the EA lemmas continue to hold. In Chapter 4 we proposed an OCA lemma that we would want to hold. However in our design for LEAs the suggested lemma doesn't hold, because not all LEA are linked to each other. Each LEA links directly to at most one other EA. By linking LEAs in this way we can construct complex authentication structures, namely trees. This gives our LEAs design more flexibility, as opposed to linking all LEAs into a chain.

Although the binding fact we define in Section 5.6.2 only captures the direct link between LEAs, we want to reason about any pair that are on the same path. We therefore define a fact that captures the transitive closure of the binding relation.

```
1 !EA_BindStar(~chain_id, ~tid, ms,
  ↪  ~base_ea_id, base_crc, base_cert,  base_binding,
  ↪  ~ea_id,      crc,      cert,       binding)
```

To create a `!EA_BindStar` fact we take an `!EA_Binding` fact, and pair it with a new `~chain_id`. The requested binding becomes the *base*, and the EA becomes the *tip*. We then consume `!EA_Binding` facts that have the tip as their requested binding, taking their EA as our new tip. This allows us to build a chain from any given EA, even one that is not itself layered, to any LEA that links to it. Constructing the closure in this way would appear to be inefficient, generating many useless `chain_id`s, however because Tamarin uses backwards search, starting from the tip and working backwards to the base, generating the closure in this way will only generate a single `~chain_id`.

The lemma we would like to prove is shown in Figure 5.8. This lemma is very similar to the OCA lemma we disprove in Chapter 4. The key differences are on lines 4, 6, and 7. Lines 4 and 6 simply capture another action, `ValidFinished`, that occurs in `Recv` rules, note they occur at the same times, `#i` and `#j`, as their respective `Recv` actions, marked in blue. This simply allows us to refer to other variables in the rewrite rule, and does not impose any new restrictions. We could equivalently express the two actions in a single action, but for our purposes it is more convenient to split them in two. Line 7, marked in red, imposes the requirement that the EAs in the two `Recv` messages be in the same chain.

To extend our result beyond the guarantees of EAs we needed to add some new helper lemmas.

Our initial helper lemmas are similar to those used in our EA and TLS models, simply ensuring various consistency properties, such as the requested binding associated with a particular `~ea_id` remaining constant. We needed fewer of these lemmas than we might have otherwise expected, because a side benefit of the strategy of using persistent invariant facts is to make it easier for Tamarin to automatically derive such properties. A downside of this strategy is that the diagrams produced by Tamarin during a proof, a key tool for understanding the bag of facts and what Tamarin has derived, are harder to read. See Figure 5.5 for an example diagram. This is because

```
lemma closure_outward_compound_auth[use_induction, reuse]:
  "All actor actor2 role role2 cert cert2 peer peer2
   ↪ chain_id tid tid2 ms crc finished  p_crc binding
   ↪ location location2 o_ea_id ea_id #i #j #k.
    Recv(actor , peer , ms, role , cert)@i
  & Valid_Finished(o_ea_id, tid, ms, p_crc, binding,
   ↪ location)@i
  & Recv(actor2, peer2, ms, role2, cert2)@j
  & Valid_Finished(ea_id, tid2, ms, crc, finished,
   ↪ location2)@j
  & BindStar(chain_id, tid, ms,
   ↪ ea_id, finished,
   ↪ o_ea_id, binding, location)@k
  & (#i < #j)
  & not (Ex actor3 peer3 #f #h.
      Revms(ms, actor3, peer3)@f
    & (#f < #i)
    & RevLtk(cert2)@h
    & (#h < #j)
  )
==> Ex role3 role4 #d #e.
    Owns(peer, ms, role3, cert)@d
    & not(role=role3)
    & (#d < #i)
    & Owns(peer2, ms, role4, cert2)@e
    & not(role2=role4)
    & (#e < #j)"
```

Figure 5.8: OCA of LEAs

(1) they have many more edges, because every fact has an extra edge to the persistent fact's source; (2) the extra edges are longer, because they all have direct edges to facts generally near the top of the graph, as opposed to just immediate neighbours; (3) there are lots of overlapping edges, because many rules have edges to a small number of rules.

The main helper lemmas we prove are single step versions of our main lemma, i.e. given a LEA prove that its requested binding is valid. There are four cases for a single step binding, (1) self-self, (2) self-peer, (3) peer-self, and (4) peer-peer.

Case (1) is the case where both the LEA and the requested binding were created by the actor. Case (2) is the case where the LEA is created by the actor, but the requested binding was created by the peer. Case (3) is the inverse of this case, where the LEA is created by the peer, but the requested binding was created by the actor. Case (4) is the case where both the LEA and the requested binding are created by the peer.

In each case we try and prove that, assuming the second EA is authentic, the first must also be authentic, even if the attacker knows the secret key of the first.

Proving that an actor can be sure that an EA it created is authentic, the self-self case, is trivial. An honest actor will not sign any requested binding it doesn't recognise, and our model does not capture any form of forgetful signing. We can prove this without any restrictions on the attacker. Even an attacker that can compromise all keys cannot convince an honest actor it signed a message it did not sign. We leave the self-peer case for the moment, and continue on to the peer-self case.

The peer-self case is similarly trivial. Assuming that the peer certificate is uncompromised, if we receive a LEA signed with that certificate we can be sure that the peer thinks the requested binding is valid. Further if the actor does not recognise the requested binding, i.e. it does not believe it created the EA referenced in the requested binding, then it will reject the LEA.

The peer-peer case is somewhat harder to prove. We include the lemma in Figure 5.9. The lemma is very similar to the closure OCA lemma in Figure 5.8. The key differences are in the LHS of the implication. They prove

```
1  lemma peer_peer_bind[reuse]:
2    "All actor peer role cert cert2 ea_id prev_ea_id tid ms
     ↪  finished prev_crc prev_finished #i #k.
3        Recv(actor, peer, ms, role, cert)@i
4      & Valid_Finished(prev_ea_id, tid, ms,
       ↪  prev_crc, prev_finished, 'remote')@i
5      & Recv(actor, peer, ms, role, cert2)@k
6      & Bind(ea_id, prev_ea_id, tid, ms,
       ↪  finished, prev_finished, 'remote')@k
7      & (#i < #k)
8      & not (Ex #f #g.
9          Revms(ms, actor,peer)@f
10       & (#f < #k)
11       & RevLtk(cert2)@g
12       & (#g < #k))
13   ==> Ex role2 #h #j.
14       Owns(peer, ms, role2, cert)@h
15     & (#h < #j)
16     & not(role = role2)
17     & Owns(peer, ms, role2, cert2)@j
18     & (#j < #k)"
```

Figure 5.9: The peer-peer binding lemma, edited for consistency of style, claims that if an actor receives two EAs, and one is bound to the other, then if the second is authentic then so is the first.

the property for a narrower scope, the cases captured by the LHS are a strict subset of those capture by the LHS of the OCA lemma. The key differences are on lines 4 and 6. Both cases require the last field, highlighted in red, to be 'remote'. This is the location field, and signifies where the actor believes the `Finished` message was generated. In this lemma we are only interested in EAs that, from the actors perspective, were created remotely. The other key difference is also on line 6. We use the `Bind` action, as opposed to the `BindStar` action in the OCA lemma. The `BindStar` action represents the transitive closure of the `Bind` action. These restrictions restrict the captured cases to a direct binding of an LEAs sent by the peer to an EA or LEA also sent by the peer.

We now return to the self-peer case. The self-peer case is much more problematic. The self-peer case is not captured by the closure OCA lemma,

which reasons about a pair a received EAs, and the self-peer case is in fact not true. The self-peer case tries to prove that if an actor binds to an EA that it received, then if the actor's key is uncompromised the received EA is authentic.

It is easy to see that this case *does not* hold. An attacker who has compromised the peer's private key can send a forged EA to the actor, and the actor can still sign it, i.e. the actor will not detect that the EA is a forgery. If the actor sends the LEA to its peer the LEA will be rejected, because the peer would not recognise the forged EA, however, the actor never knows this. Unless some higher layer tells the actor the LEA was rejected, it can never know that the EA was not recognised by the peer.[29]

Given that this case is not required to prove the closure OCA lemma it is tempting to ignore it. However, without this case it is impossible to construct a proof of the closure OCA lemma by induction over messages. This is because the closure OCA property is not inductive over messages. This can be easily demonstrated.

An actor who receives an EA, and sends back a chain of n LEAs has learned nothing about the authentication status of the LEAs. However, the moment it receives an LEA bound to the end of its chain, it knows that all the LEAs it sent were accepted.

We therefore need a new strategy. The closure OCA lemma appears to be inductive over the receipt of messages. Our peer-peer binding lemma functions as a base case of this lemma. We thus need a way to reason about the inductive step, where there are a number of LEAs sent before another LEA is received. Because there is a state change from knowing nothing about the peer's view of the LEAs' authentication statuses to knowing all of them are valid simultaneously we attempt to construct a proof that would give a result in that way.

Receiving a second LEA implies that all intervening LEAs are considered valid by the peer. We thus need a way to refer to, in a lemma, the intervening messages. Usually we would achieve this by inductively unrolling the

[29] If the peer binds to the LEA the actor knows the peer accepted it, but if the peer doesn't bind to it then its status is still ambiguous to the actor.

requested bindings until we had covered all the LEAs we wished, i.e. the base case. However, as explained, in this case this is ineffective. Looking to prior work in Tamarin, in particular [Sch+14], we find that Tamarin can be used to reason about lists.

Tamarin provides a feature called multi-sets. These are a variables with an associative, commutative + operator. For example you could have a multiset, m containing 1, and then refer to $m + 1$. This is slightly different to the tuples we have previously described. With tuples you can store a fixed number of values, and by nesting, produce list-like structures. However using nested tuples you can only refer to the head of the list in a lemma. With a multi-set we could store $m := crc_1 + crc_2 + \cdots + crc_n$, and then write a lemma of the form $\forall crc \cdot \text{EA\_Handle}(ea_{id}, tid, crc, binding) \& (\exists x \cdot crc + x = m) \rightarrow \text{Owns}(\dots)$. We briefly experimented with constructions of this form, but rapidly ran into difficulties resolving sources. Consultation with the authors of Tamarin suggests that multi-sets are particularly prone to these sorts of difficulties, and are particularly taxing on RAM. We thus leave the completion of this proof for future work.

5.8 Results and conclusions

In this Chapter we have introduced and analysed LEAs. Our research did not find any problems with LEAs. Our analysis with the Bhargavan framework suggests that our construction is sound, and our proofs of single steps of the binding lemmas show that at least received LEAs achieve OCA with the immediately preceding EA, as do LEAs when sent by the same actor. We also introduced authentication forests, and discussed possible use cases of various patterns of compound authentication.

Our model taxed the limits of what is possible with Tamarin, and highlights some unusual or unexpected behaviours of the model checker in extremis. Our work however leaves open a number of questions. Whilst a conceptually small step, proving that LEAs provide closure OCA is still an unknown. Our work also leaves open what the effects of splitting the hash function, as proposed by the IETF, would be. Further, because our LEA

model builds off our EA model, it is also an open question whether we can prove something using a composite model.

Our work does however, provide a solid basis for continuing to consider the security guarantees of LEAs, in conjunction with the IETF.

Chapter 6

Messaging Layer Security with Transport Layer Security

6.1 Introduction

The use of TLS is becoming ubiquitous on the public internet, but TLS is not restricted to the world wide web. TLS is also heavily used on enterprise and private networks.[1] Enterprises often secure connections between different sites and within data-centres with TLS. For example, one common use of TLS in the enterprise is to satisfy the requirements of the Payment Card Industry (PCI) Data Security Standard (DSS) [PCI DSS]. The PCI DSS is an industry standard for protecting credit card information, and one requirement is the use of "strong cryptography" to protect certain communications, such as credit card transactions. Compliance with PCI DSS is required by both VISA[2] and Mastercard[3], and in some jurisdictions is even enshrined in law [NRS-603A]. According to the latest edition of the standard TLS 1.1 and newer versions of TLS are sufficient for this purpose, assuming they are appropriately configured.

To detect attacks in their networks companies may use intrusion detection systems (IDSs) and intrusion prevention systems (IPSs). These devices

[1] Throughout this section we will make a distinction between public and private networks, however in practice there is no technical difference between them. The separation is entirely notional.

[2] `https://www.visaeurope.com/receiving-payments/security/`

[3] `https://www.mastercard.us/en-us/merchants/safety-security/security-recommendations/service-providers-need-to-know.html`

sit on the corporate network and monitor traffic. If something suspicious is detected the device might do anything from raise an alert to block the suspicious connection. Some IDSs and IPSs simply rely on metadata for their analysis, for example profiling connections based on their destination, point of origin, or size; whereas others make use of deep packet inspection (DPI). DPI is when the device analyses the payload of the connection. When the payload is unencrypted, such as in a vanilla HTTP connection, this process is simple, however when the payload is encrypted the devices ability to perform this type of analysis is curtailed. For convenience we shall refer to all devices that analyse traffic in transit, as opposed to at the endpoint, as middleboxes. Middleboxes can be inline, i.e. directly intercept and forward traffic, or passive, i.e. receive a copy of all data sent to and from the server.

One solution used in industry for performing DPI on TLS connections is to configure the server to use TLS 1.2 with a static RSA cipher suite. Using a static RSA cipher suites means that a passive observer who knows the server's private key can decrypt all traffic sent to and from the server, we discuss this in more detail later in this section. By providing middleboxes with a copy of the server's private key they are able to perform DPI without needing to be inline on every connection. Using TLS in this way on the public internet is controversial because it can be used for censorship and anti-competitive behaviour, and breaks the security guarantees TLS clients expect. Enterprise, however consider this use case vital, because, for example, it allows them to block malicious connections to vulnerable devices on their network that are impossible to update, for example because patches do not exist.

The TLS 1.3 standard removes all static RSA cipher suites as insecure. All cipher suites in TLS 1.3 are forward secret. We recall the definition of perfect forward secrecy (PFS) from Section 2.5.5. We say that a protocol is forward secure with respect to a LTK if an attacker that compromises the LTK after the handshake is complete cannot decrypt the session. This is equivalent to saying that passive attacker that knows the LTK cannot decrypt the session. An active attacker may be able to MITM a session for which it knows the LTK, but a passive attacker cannot.

Because some industries have regulatory or contractual requirements to monitor various traffic, removing all static cipher suites was controversial. Further, some organisations wish to do more than the regulatory minimum in terms of security, and plan to upgrade from TLS 1.2 to TLS 1.3 before they are forced to, however with static cipher suites unavailable they do not have a clear upgrade path. A number of proposals for achieving visibility of TLS 1.3 connections have been made. Over the course of this chapter we will examine some of the more significant proposals, looking at advantages and disadvantages of each, before proposing and evaluating our own solution.

6.1.1 Chapter overview

Our main contributions in this chapter are as follows:

1. We describe the problem space and current approaches to achieving visibility of the contents of a TLS connection.

2. We analyse and evaluate three proposals that would enable visibility of TLS 1.3, each of which has serious drawbacks.

3. We make a new proposal, based on a composite protocol approach. We construct a protocol that layers TLS 1.3 on to a multi-party protocol called MLS. This approach avoids many of the pitfalls affecting the other proposals.

4. We introduce pairwise channel bindings, channel bindings that authenticate two members of a multi-party protocol to each other in other protocol layers.

5. We suggest a new cipher suite that provides authentication and integrity guarantees to the TLS 1.3 record layer independent of the confidentiality guarantees.

6. We provide an extended discussion of our proposal, comparing and contrasting it with the earlier proposals.

6.1.2 Related work

There is an extended body of work on decrypting TLS messages in flight [Nay+15] [Gre+17] [HD18], each of which has been analysed extensively. We will discuss three such proposals and objections brought to them.

The first proposal we will discuss is multi-context TLS (mcTLS) [Nay+15]. In mcTLS the middleboxes act as a series of active MITM, each decrypting and re-encrypting the traffic, passing it from one to the other, until all have seen it, and it is finally passed to the client or server as appropriate. mcTLS provides fine-grained access control to the connection, offering differing levels of visibility to different middleboxes, but it requires that all middleboxes be on path. Requiring all middleboxes to be on path introduces practical difficulties, from network choke-points and latency to reliability and resilience.

The second suggested mechanism for achieving visibility into TLS 1.3 connections is by using static DH keys. This was proposed by Matt Green in `draft-green` [Gre+17]. If a server always uses the same DH key share, g^y, then a passive observer who knows the private portion of the key share, i.e. y, can decrypt the session. This mechanism doesn't require changes to the TLS 1.3 specification, simply being a configuration change, but it makes the server's key share into a long term secret. TLS 1.3 is designed to have PFS with respect to long term secrets, however if static DH keys are used this property no longer holds. An attacker who can learn y can passively decrypt sessions. Although this might seem unavoidable for passive decryption, the mechanism we propose in Section 6.8 maintains the PFS property of TLS.

The third suggested mechanism mirrored the static RSA method of TLS 1.2. The static RSA key exchange mechanism makes use of a technique called key wrapping. Key wrapping is when one key is encrypted with another. In static RSA the client encrypts a pre-master key with the server's public key. The server, on receipt of this message decrypts the pre-master key and computes the master key. `draft-RHRD` [HD18] defines an extension to TLS 1.3 which wraps the session secrets with a key that the server agrees with the middleboxes ahead of time.

The most comprehensive analysis and critique of this work is by Farrell [Far18]. Farrell deals with `draft-green` and `draft-RHRD` specifically, and

visibility drafts in general, enumerating an extensive list of problems. Our analysis is heavily based on this work.

The IETF also has a number of relevant policy positions. RFC 2804 [RFC2804] defines wiretapping, and states the IETF position against making any provision for it in IETF developed protocols. RFC 7258 [RFC7258] defines pervasive monitoring, and requires all IETF developed protocols to mitigate against pervasive monitoring where possible.

6.1.3 Chapter organisation

In Section 6.2 we provide some background to the IETF's objections to the prior drafts, and on alternative approaches to achieving the stated goals of enterprise [Fen18]. In Section 6.3 we introduce mcTLS, and give a brief discussion, before introducing `draft-green` in Section 6.4, along with a brief discussion. In Section 6.5 we introduce the third proposal, `draft-RHRD`, and give a brief discussion. We use these three sections to motivate a new approach in Section 6.6. Our new approach constructs a composite protocol layering TLS 1.3 over a protocol called MLS, which we introduce in Section 6.7. We define and analyse pairwise channel bindings in 6.9, before defining a number of variant constructions in Sections 6.10-6.12.

In Section 6.15 we give an extended discussion of the various issues raised with the various proposals including our own, based on the work by Farrell [Far18], before concluding in Section 6.16.

6.2 Background

The attempts to re-introduce visibility to TLS 1.3, after the WG had decided to remove all non-ephemeral modes have been highly controversial. Members of the TLS WG were worried that any functionality that allowed for inspection of TLS connections would be rapidly repurposed to allow state surveillance and censorship. A number of countries in both historic and modern times have carried out widespread surveillance of their populace, something the IETF is explicitly against. Despite industry assurances that such mechanisms would only be used inside data centres and corporate networks, TLS WG members worried that there was no way to enforce this,

as from a technical perspective there is no clear distinction between a large corporate network and that of a small country.

In its mission statement the IETF explicitly states that it is not value-neutral [RFC3935], and that it is committed to values such as openness and fairness. In RFC 7258 [RFC7258] the IETF explicitly lists pervasive monitoring as an attack that should be militated against in IETF protocols. The IETF also explicitly does *not* consider requirements for wiretapping as part of the process for producing standards [RFC2804]. Together, these documents[4] position the IETF against any changes to TLS 1.3 that might enable wiretapping or pervasive monitoring. Both pervasive monitoring and wiretapping are given specific definitions in their respective documents [RFC7258], [RFC2804]. Thus any suggested change to TLS 1.3 must at least not exacerbate, if not militate against these two attacks.

6.2.1 Alternative approaches

There are a number of alternative approaches to performing the necessary security functions other than decrypting the TLS connections in transit. We briefly summarise these here, and the discussion around them. For some enterprises these are sufficient, and no further work is necessary. However, some of the largest and most heavily regulated enterprises have issues with all of these, prompting the works described in the latter sections of this chapter.

Meta-analysis

Many middleboxes are able to perform the majority of their functions only being able to see the unencrypted portions of a TLS connection. This method however makes diagnosis of specific connection issues more challenging.

Remaining on TLS 1.2

Another proposal was for corporate networks who could not sacrifice visibility in the short or medium term to remain on TLS 1.2. This is not a long-term solution however, because eventually TLS 1.2 will be considered

[4] Numerous other related documents exist, but this subset are the most relevant.

too insecure even for use on a trusted network. A counter-argument raised was that the PCI DSS only proscribed TLS 1.0 and Secure Sockets Layer (SSL) in May of 2018, almost 20 years after the publication of TLS 1.0, so the prospect of TLS 1.2 being deprecated is particularly remote. Furthermore, although a draft deprecating TLS 1.1 and earlier [MF18] was discussed by the TLS WG at the IETF 102 meeting, it was not accepted as a WG item, strengthening the argument that the deprecation of TLS 1.2 is a long way off. It was also discussed that enterprise would like to take advantage of the greater speed and security of TLS 1.3, and thus a solution that allowed them to upgrade would be preferable.

Split TLS

In a corporate environment adding an extra certificate to the root-of-trust for all devices would allow for split TLS. This effectively creates a new certificate authority controlled by the company. A middlebox at the network gateway can create certificates for any server and sign them with the company certificate authority and have them accepted by devices inside the network. Split TLS refers to the practice of terminating all connections at the network edge, and creating a new connection to the destination. By forcing all connections through a gateway middleboxes would be able to see all traffic as it traversed the edge of the corporate network.

Gateways with access to internal server's certificates can also split incoming TLS connections. This is a common practice on the web, allowing for load balancing incoming connections across a number of servers. This solution requires installing an extra certificate on every device on the corporate network, and requires the gateway to decrypt and re-encrypt every connection. Further this solution does not assist with traffic of analysis within the corporate network, whether internal to internal, or between the internal node and the gateway. This creates a potential choke-point at the gateway, both for traffic and analysis.

Endpoint analysis

Installing security software at the endpoints may prove sufficient for some networks, particularly with logging enabled. This solution however has two major drawbacks. Endpoints might be lightweight devices, only powerful enough to perform their designated functions. They may not be able to run either security software or logging. Further if a device has a vulnerability for which the patch is yet to be deployed, or even non-existent, then attempting to do security analysis of the connection after it has been processed may prove too late.

Export keys

A server could, using some other protocol, export keys to the relevant middleboxes. This would give the middleboxes full visibility, however doing this in a real-time way is practically difficult at large scale and doesn't allow analysis of packets before the are processed by the server. Further this would act as a form of key revelation which goes against best practice [BCP200].

6.3 Multi-context TLS

mcTLS is a proposal by Naylor et al. [Nay+15]. Motivated by the increase in TLS usage, and the inelegance of the then current solution space they proposed an in-line mechanism whereby endpoints could explicitly authorise different middleboxes to see different parts of the connection. mcTLS is not simply aimed at security focussed middleboxes, but also load balancers, caches, and other network management functions. mcTLS is a modification of TLS 1.2, but we include it here because it has been proposed as an alternative to visibility in TLS 1.3, and provides finer-grained control of network access than static RSA. mcTLS can grant read and/or write access to various subsets of traffic. Specifically it can restrict middlebox access to headers and/or content for requests and/or responses. This allows for very tightly controlled access, tailored to each middlebox.

The European Telecommunications Standards Institute (ETSI) has published a draft standard based on mcTLS [CYBER-27-2], called the Transport layer Middlebox Security Protocol (TLMSP).

6.3.1 Mechanism

mcTLS operates by extending the TLS 1.2 handshake. After the server's second flight, the middlebox sends a flight of messages to the client and server that is structurally the same as the server's second flight.[5] This is used to establish cryptographic contexts between the client and the middlebox, the server and the middlebox, and the client and the server.

The middlebox requests a "context"[6] that defines what traffic it can access. The traffic is broken into requests and responses, and into headers and content. A middlebox could ask, for example, to have read and write access to request headers, and read access to response content.

6.3.2 Discussion

A formal analysis of mcTLS by Bhargavan et al. found an attack [Bha+18]. It was shown that malicious clients could collude with middleboxes or servers to attack other middleboxes. For example, a client could re-insert something removed by a middlebox before it arrived at the server. This has serious implications for security, as an attacker can circumvent protections provided by middleboxes, and furthermore, convince the middlebox that such protection had been effective. Bhargavan et al. in the same work proposed a fix that could be shown to be formally secure. However this highlights that defining a secure visible version of TLS 1.3 is non-trivial.

Some criticisms specific to mcTLS are that every middlebox becomes a point of failure. Because every middlebox is on-path, if one middlebox becomes unavailable then so does the entire connection. If use of a particular middlebox is required by policy then this turns a denial-of-service (DoS) attack on a middlebox into a DoS attack on the entire network. This brittle structure is high risk to deploy in large environments.

[5] In the case of multiple middleboxes the middlebox sends this message to its two neighbours, whether they be the client, the server, or another middlebox.

[6] Not to be confused with a cryptographic context.

Another criticism is that it modifies the TLS handshake in a non-trivial way, making the guarantees it achieves non-obvious and invalidates formal analyses of the handshake. Further it is based on TLS 1.2, meaning that it does not address the desire for visibility in TLS 1.3.

6.4 draft-green

Green proposed a configuration of TLS 1.3 that would allow for visibility of TLS connections. The proposal mimicked the static RSA setup used with TLS 1.2 by using static DH key shares. The proposal was unable to achieve consensus at the IETF, and was not accepted as a work item. Further the TLS WG did not achieve consensus to work on any draft that enabled any kind of visibility into TLS connections. The European Telecommunications Standards Institute however has published a draft standard [CYBER-27-3] that defines a protocol they call enterprise TLS (eTLS), which uses static DH.

6.4.1 Mechanism

The proposal, presented to the IETF as `draft-green`, required servers to repeatedly use the same key share for every TLS 1.3 handshake. Usually a server would use a fresh, ephemeral key, g^y, for each handshake. When using `draft-green` the server would instead select the same key share each time. By sharing the private portion of the DH key, y, with all middleboxes allowed to access traffic, said middleboxes can compute the key by observing the handshake.

6.4.2 Discussion

`draft-green` was proposed to the IETF before the IETF 99 meeting, where it was presented by Fenter, Green, and Housley. As can be seen from the minutes of the meeting, the proposal was controversial [Tur17]. Fenter presented a number of use cases, (1) packet analysis, (2) fraud monitoring, (3) IDSs and IPSs, (4) malware detection, (5) incident response, (6) regulatory requirements, (7) layer 7 distributed denial-of-service (DDoS) protection, and (8) performance management. Diagnostics was also mentioned as a key use

case. Identifying problems without being able to trace them is harder, and thus might extend issues that could otherwise be resolved quickly.

Green then presented a security analysis. According to Green, with the exception of forward secrecy, `draft-green` is cryptographically secure. The pros of `draft-green` were given as (1) it involves no significant protocol changes, (2) it uses well understood cryptography, and (3) it is detectable.

In opposition, Farrell presented TINFOIL [Far18]. TINFOIL is a list of objections to interception technologies being made compatible with TLS 1.3. We discuss the general objections to interception technologies in Section 6.15.2, we limit discussion here to those objections specific to `draft-green`.

One criticism specific to static DH approaches is that they are not implementation robust. This means that a poor implementation can lead to attacks. If something is implementation robust, then a poor implementation should fail, rather than be successfully attacked. For example Jager et al. [JSS15b] show that an invalid curve attack can be used on certain implementations of TLS 1.2 to derive the private portion of the server's DH key share. When a client connects to the server in TLS 1.2 using elliptic curve cryptography with DH it usually sends the server a point on the negotiated curve. By carefully choosing a point not on the elliptic curve, but on a related one that is chosen for having certain weak properties, the attacker can make the server compute an easily invertible value, with some non-negligible probability. This is called an invalid curve attack, and a well implemented server will check that the incoming point is on the claimed curve, but at Jager et al. show, a number of implementations do not check this. If the attack is successful, the attacker can compute the private portion of server's DH key share. If the server uses static DH values, i.e. it reuses the same DH values for more than one connection, then the attacker can use that value to impersonate the server by MITMing connections.

Although this attack is on TLS 1.2, TLS 1.3 also uses a DHE and may well share libraries with TLS 1.2. This attack serves to highlight that using static DH can introduce vulnerabilities.

Another criticism of this approach is that a client cannot tell on its first connection whether this mechanism is being employed, and thus at least some clients are unaware that their connections are not private, when they have a reasonable expectation that they will be.

One criticism of draft-green in comparison with mcTLS is that a middlebox that possesses the connection keys can undetectably modify the connection in either direction. This means that whilst draft-green achieves its goal of weakening the confidentiality goals of TLS 1.3, it has the side effect of breaking the authentication and integrity guarantees. It shares this criticism with draft-RHRD, which we introduce next.

6.5 draft-RHRD

draft-RHRD[7] was created to address criticisms of draft-green. It was presented as a draft to the TLS WG at the IETF 101 meeting. The reopening of the visibility issue after the IETF did not achieve consensus to work on visibility drafts was again contentious [Tur18].

6.5.1 Mechanism

draft-RHRD takes a similar approach to the static RSA mode of TLS 1.2. Before TLS connections begin the server is provisioned with the public portion of a DH key share, g^m. The private portion of this key share, m, is given to all middleboxes.

The client adds an extension to the ClientHello called Visibility. When sent by the client the extension is empty. By including this extension the client acquiesces to the connection being observed.

The Visibility extension is also included in the server's reply. When sent by the server the extension includes a value that is opaque to the client. The server sends the extension in the unencrypted portion of the ServerHello.

The opaque value is formed of three parts, and completes a DHE with all the middleboxes. The pieces are as follows.

[7] This is sometimes pronounced "rehired", for its visual similarity to that word.

1. The public key with which the server was provisioned, g^m.
2. An ephemeral key share[8] generated by the server, g^n.
3. The master secret[9] of the connection, ms, wrapped with the key the server has established with the middleboxes g^{mn}, i.e. $\{ms\}_{g^{mn}}$.

Any middlebox knowing m can unwrap the master secret, and decrypt the connection.

6.5.2 Discussion

This suggestion is an improvement over `draft-green` because the client must explicitly opt-in. However, in practice, this opt-in may not be optional. Because the extension is included in the client and server's unencrypted extensions an outside observer can detect whether the extension is in use, and block connections that do not use it. Further the observer can monitor whether the public key the server sends is one it knows, and block connections for which it does not know the key. It can do this even without computing the wrapping key.

Another criticism of `draft-RHRD`, which applies equally to `draft-green`, is that neither the client nor the server knows the identities of observers. Further the mechanism relies on a form of key escrow, or key revelation. This is considered bad practice [BCP200]. Although the mechanism is ostensibly similar to that employed in static RSA, it differs in a number of key aspects. Static RSA is secure under a threat model where an attacker cannot acquire the server's long-term keys, and is a key exchange mechanism. In `draft-RHRD` the key is established by TLS 1.3, and then revealed to a third party. Another criticism of `draft-RHRD` is that introducing extensions that weaken the security of TLS is outside the charter of the TLS working group.

[8] Note this key share is different from the key share the server intends for the client, which we denote g^y.

[9] More accurately the early secret and the handshake secret, from which the master secret is derived, are sent. This ensures that the middleboxes can decrypt the entire handshake.

6.6 A new approach

The three proposals we have discussed so far, mcTLS, `draft-green`, and `draft-RHRD`, all modify the way TLS works to transform a two-party end-to-end protocol into a multi-party protocol. This approach means that all the formal analyses that have been performed of TLS 1.3 are not valid if one of these suggestions is in use. Our suggestion takes a different approach.

We take a multi-party protocol and layer TLS 1.3 on top of it, to construct a composite protocol. This makes clear the exact relationship between TLS 1.3 and the lower layer protocol. In the previous two chapters we discuss EAs and LEAs, which layer protocols on top of TLS 1.3. With this suggestion we layer TLS 1.3 on top of another protocol. By applying the same channel bindings logic we believe that we can construct a composite protocol that it is possible reason about formally.

The multi-party protocol we choose is MLS [Bar+18], a protocol being developed at the IETF to construct a common base for group messaging applications. MLS is under active development, and thus any analysis depends on the exact end state of MLS. However, we base our construction on research into asynchronous ratcheting trees (ARTs) [Coh+17], which form the suggested base for the key-exchange phase of MLS, and has had some formal analysis. Where details of MLS have yet to be defined we use the details suggested in ART.

The intuition for how this would work is that the client and server would establish a group with any requisite middleboxes, and use the key from this group to derive an out-of-band pre-shared key for a TLS 1.3 handshake. By using a multiparty protocol for the establishment of the OOB PSK means that rather than trying to construct a multi-party protocol from a two-party protocol, we instead use a multi-party protocol to establish the multi-party key, then use a two-party protocol for the two party portion of the protocol. We also provide a mechanism for ensuring that middleboxes cannot break the integrity or authenticity of the connection. Further by using a standard mode of TLS 1.3 we do not invalidate analyses of the specification.

Constructing our layering from two protocols that have some formal analysis and constructing a contributive channel binding, in line with the Bhargavan hypothesis, gives us some confidence that our construction is reasonable, although we leave a formal analysis for future work.

6.7 MLS

MLS defines a multi-party key agreement protocol, that is designed to define the security guarantees for group messaging systems à la WhatsApp. The purpose is to establish a shared key between a group of people suitable for encrypting messages between them. MLS has various group management features such as adding and removing people from the group. We make use of these features to ensure that before each connection between a client and server they explicitly agree again on the middleboxes involved. MLS uses two tree constructions that we introduce here. ARTs are used to establish secret keys, Merkle trees are used to efficiently commit to a group of participants.

6.7.1 Asynchronous ratcheting trees

Asynchronous ratcheting trees (ARTs) are described in work by Cohn-Gordon et al. [Coh+17], and we follow their notation. An ART is a left-balanced binary tree whose root is the shared secret key, and whose leaves are DH secret keys.[10] Each non-leaf node contains links to its two children and some ancillary data. In particular, if its two children contain x and y then the node contains g^{xy}. The root key, therefore, is constructed of a tower of exponentials.[11] We call the key at the root of the ART the root key.

Each node in the ART contains a DH secret key, which is then used as a private key.[12] We can construct a second tree, T, in which each node corresponds to a node in the ART, and contains the public key of private key stored in the corresponding node, i.e. if a node in the ART contains x the corresponding node in T contains g^x, we call T the tree of public keys.

[10] Note that we use the term secret keys to refer to the result of a DHE, g^{xy}, as opposed to the private portion of a DH key share, i.e. x or y.

[11] Recall that all these exponentiations are modulo some prime p, and thus the key remains in the keyspace.

[12] Excluding the root node.

Because we assume the DH problem is hard[13], we can publish T and an attacker gains no advantage in deriving any of the private keys.

If an actor, Alice, wishes to establish a shared key with a group, Bob, Charlie, ...; she acquires ephemeral DH key shares from all prospective participants, including herself[14], $EK_0, \ldots, EK_n$. She then creates a new DH key, called the setup key, suk. Using the setup key and the key shares she computes a shared key, $\lambda_i := (EK_i)^{suk}$, with each participant. We refer to the node constructed from an actor's DH key share as the actor's node. Using these keys as the leaves of the tree she can now compute the root secret. Because the root secret contains contributions from all the participants each can be sure it has sufficient entropy.

Alice now sends a four part message to all the participants. The message contains the following.

1. The list of ephemeral keys used, $EK_0, \ldots, EK_n$;
2. The public portion of the setup key, g^{suk};
3. A tree, T, containing the public key of each node in the ART; and
4. A signature of items 1 - 3.

Let EK_i represent Bob's ephemeral key share, and let ek_i represent the private portion of EK_i. On receipt of Alice's message Bob can compute $\lambda_i := (g^{suk})^{ek_i}$. Knowing just λ_i and T Bob can compute the root secret as follows.

To compute the root secret Bob must extract the co-path of λ_i from T.[15] The co-path of a node is defined as the node's sibling, the sibling of its parent, and so on, until the root. The sibling of λ_i in T is g^{λ_j}.[16] Using these two values Bob can compute $g^{\lambda_i \lambda_j}$, the secret value of his parent node in the ART. Using the copath of his parent node in T, Bob can now compute the secret of his grandparent and so on, until he computes the secret of the root.

[13] See Section 2.3.2.

[14] For simplicity, we here describe the unauthenticated version of this computation. For the authenticated version see [Coh+17, p. 20]

[15] Note that Alice can compute the secret directly during construction of the ART.

[16] Where $j = i \pm 1$

Bob can only compute the secrets of his direct ancestors, not the secrets of his siblings. Using this mechanism all members of the group share a key, but only those whose key shares were included in the ART can compute the root secret.

Cohn-Gordon et al. use a technique proposed by Marlinspike [Mar13] to achieve asynchronous key establishment. Users of the protocol sign a number of key shares with their public certificate, and send them to an untrusted server. When an actor, Alice, wishes to communicate with another actor, Bob, she can go to the untrusted server and request pre-keys for Bob. This allows Alice to complete a DHE with Bob even if he is offline. By posting her setup message to the server, when Bob comes back online he can retrieve the setup message and compute the keys, even if Alice is offline. We call the server untrusted because even a malicious server cannot achieve any malign outcome beyond a simple DoS. This design is such that even if the server is legally compelled to attack its users, it is technically incapable of doing so.

The use of an untrusted server is of particular interest in our case, because not all middleboxes will be available all the time. Unlike mcTLS, where if one middlebox goes down so does the entire network, with asynchronous key establishment a middlebox that is added to a session whilst it is offline, will be able to decrypt that session when it again comes online.

6.7.2 Merkle trees

The second part of the MLS key establishment protocol involves Merkle trees. A Merkle tree is a tree of elements, whose key property is that the size of a proof that an element is in the tree grows logarithmically in the number of leaves in the tree. In MLS Merkle trees are used to efficiently commit to a set of identity keys. These identity keys correspond to the identities of the participants. Whilst ARTs establishes keys between a group of participants Merkle trees provide a robust way of committing to the identities of those members.

Merkle trees were proposed by Merkle [Mer88]. Originally Merkle trees were constructed with DES operations, but they are now constructed with hash algorithms. A leaf node in a Merkle tree is constructed $h(1 \mid\mid e)$, where

h is a hash function, || is the concatenation operation, and e is the element in the tree. A parent is constructed from its left and right children, $h(2$ || left child || right child).

To prove an element, e, is in the tree it is sufficient to provide e and all nodes on its co-path. The verifier can then compute the path from e to the root. If the computed root node matches the expected root node then the verification succeeds. Because it is infeasible to find two different values that hash to the same value it is sufficient for the verifier to store just the root node. Thus if two parties agree on the root of the Merkle tree then they both agree on all its elements.

In MLS the Merkle tree contains identities for all the participants, and the leaf nodes of the Merkle tree correspond to the leaf nodes of the ART.

6.8 Layering MLS over TLS

We now begin to compose the MLS and TLS 1.3 protocols, see Figure 6.1 for a sketch of the complete protocol. We use the multi-party key exchange defined by Cohn-Gordon et al. [Coh+17] to agree a key between all the participants. Before a TLS session the client performs an MLS run adding all the participants in the protocol, including the server and any observers.

At the end of an MLS run all participants agree on the root of the ART and the Merkle tree, amongst other things. We refer to the leaf key shared between the client and server as λ_{cs}. Because we define the client to be the group initiator we can be certain this exists.

6.9 Channel binding

We now construct a channel binding that will uniquely identify an MLS run. We will construct what we term a *pairwise channel binding*.

Definition 6.9.1. Pairwise Channel Binding. A pairwise channel binding is a value produced at the end of a multi-party protocol run, such that no two protocol runs with different parameters produce the same value, and no two distinct pairs of participants in the protocol produce the same value.

In our case, we construct a channel binding that uniquely identifies an MLS run and designates two participants, the client and the server. This definition allows us to capture the multi-party to two-party transition, distinguishing the client and server from the middleboxes.

Our work in Chapter 4, in particular the definitions of ICA and contributive channel bindingss (CCBs), see Definitions 4.4.4 and 4.4.7, tells us that a channel binding that authenticates future / later runs needs to include contributions based on the shared secrets established during the run. To this end we include contributions from the root secret. To establish a pairwise channel binding, and thus to authentically identify two participants as the client and the server we also include a contribution from λ_{cs}. λ_{cs} can only be computed by the someone who knows the setup key suk or ek_S[17], i.e. the client or the server.[18],[19] The MLS draft [Bar+18, pp. 16-17] defines the minimal elements that each participant needs to maintain of the MLS session state. We use this as a basis for our channel binding, because we know that the server will be certain to know these values. A participant must store:

1. Its index in the identity and ratchet trees;
2. The private key of its key share, used to construct the leaf keys;
3. The private key associated with its identity key in the identity tree;
4. The current epoch number;

[17] Recall that ek_S is the private portion of the key EK_S

[18] This use may invalidate the computational proof of ART, but because it is only used to key an HMAC, it might not.

[19] We assume the ART is bound to the identity tree by the end of the run, but if not the channel binding must also be signed.

5. The group ID (GID);
6. The co-path of its leaf in the identity tree;
7. The co-path of its leaf in the ART;
8. The message encryption secret;
9. The current add key pair; and
10. The current init secret.

Because we do not use the messaging functionality of MLS the message encryption secret is not relevant to our use case. The epoch number, add key pair and the init secret relate to group management functions, which we do not discuss, save to note that we assume that once the group management features have been developed and have stabilised they will be analysed and will achieve a reasonable level of security. Once MLS is complete, if the group management functions are not useful for our purposes we can simply construct a new group for each new set of participants.

The group ID (GID) is simply a value that uniquely identifies groups, which remains constant when adding and removing members.

Based on this state, we propose the following construction for the channel binding for the client, server pair.

$$\begin{aligned} HMAC(\lambda_{cs}, <&epoch, \\ &group\_id, \\ &cipher\_suite, \\ &identity\_co - path_S, \\ &ratchet\_co - path_S >) \end{aligned}$$

6.9.1 Analysis under Bhargavan et al.'s framework

Examining this design under Bhargavan et al.'s framework shows this channel binding to be contributive by construction. As there is no prior layer we only need to show that the channel binding is dependent on the $params$[20] and the session secrets. The $params$ in this case are as follows.

[20] See Section 4.4.3

$params = (c_i, c_r, sid, cb, cb_{in})$ where

$$\begin{aligned} c_i &:= SUK, \\ c_r &:= id_S, \\ sid &:= GID, \\ cb &:= \text{as above}, \\ \text{and } cb_{in} &:= \bot \end{aligned}$$

Where id_S is the server's identity in the identity tree. The channel binding is dependent on λ_{cs}, which is dependent on SUK.[21] The identity co-path constitutes a proof that id_S is in the identity tree, and thus the channel binding is dependent on id_S. The GID is included directly in the channel binding, and there is no prior layer.

The session secrets established are the ratchet root key and λ_{cs}, both of which are included in the channel binding.

6.9.2 Channel bindings in TLS 1.3

A shortcoming of the TLS 1.3 specification is that it does not include an explicit API for adding channel bindings. This means there is no explicit way for the client to signal that it is aware, at the TLS layer, that it is being bound to lower layer. Channel bindings were originally proposed for use with insecure legacy protocols that could not be changed. This would seem to imply that this construction is acceptable, however this is not the case in general. In the legacy case it is implicitly assumed that it is general knowledge that the legacy protocol is insecure. Further it is assumed that the secure outer protocol strictly increases the security. Because TLS 1.3 is considered highly secure, and layering it with MLS weakens the confidentiality guarantee neither of these assumptions holds in our case.

The ideal solution would be for TLS 1.3 to have an input to the key schedule explicitly for channel binding, which given that it has an output for channel bindings would bring it in line with the channel bindings requirements in RFC 5056 [RFC5056, p. 6].

[21] Recall that SUK is the public portion of suk.

Another approach would be to assume that any application creating an MLS with TLS session is aware at the application layer. This solution is less than ideal, because whilst the author of the application may or may not be aware that this solution is in use, this is not necessarily signalled to the user.

We propose two technical measures[22] that ensure this data is available to, and understood by, the TLS layer, and require that the TLS layer be responsible for signalling this to the user. Specifically, to use MLS with TLS with these measures would require modifications to the TLS libraries, and thus we can require the implementers to provide the appropriate signalling. In practice, for reasons we discuss later, we do not expect MLS with TLS to be implemented in browsers, and thus the need for user signalling is limited.

The first of our proposed measures is to require the channel binding to include a truncated copy of the `ClientHello`, specifically everything before the `PSK_ID`s, including the client nonce.

$$\begin{aligned}\text{HMAC}(\lambda_{cs}, <&Truncate(\texttt{ClientHello}),\\ &epoch,\\ &group\_id,\\ &cipher\_suite,\\ &identity\_frontier_S,\\ &ratchet\_frontier_S >)\end{aligned}$$

This is a shorter truncation than used in the TLS 1.3 specification to truncate the `ServerHello`, which includes the `PSK_ID`s, but not the PSK binders. Because, as we describe in the next section, we include the channel binding in the `PSK_ID`, we must truncate earlier. This construction requires the channel binding to be computed at the TLS layer with access to all the fields identifying the MLS session, rather than allowing the caller to simply pass the TLS library the `PSK_ID` and its PSK. This would require support at the TLS layer. Our construction works effectively without this modification, and it may be preferable to not support MLS with TLS in libraries to make it is less likely to spread outside the data centre, but this is a trade-off between ensuring user visibility and controlling the spread of deployment.

[22] We introduce the first of these below, and the second in Section 6.12.

6.10 PSK-identifier

The client constructs the PSK identifier as follows.[23]

$$< group\_id,\ client\_index,\ server\_index,\ channel\_binding >$$

The server can identify the session and participants from the first three fields, and calculate the appropriate channel binding, to compare with the last field. The key used in the session is constructed from the secret portion of the root key. We propose the following construction.

```
HMAC(root key, ``mls with tls, oob psk'')
```

This allows us to use the key independently of any other usage of the ART root key, following the best practice of labelling keys.

6.11 Cipher suites

We propose a new cipher suite mode that simply appends a MAC tag to each TLS record before it is passed to a standard mode. If the MAC key is independent of the keys used for the AEAD then it would appear that this doesn't affect the security of the AEAD, because the security of AEAD is independent of the plaintext. In this scenario we use a key derived from λ_{cs} to MAC the messages. We propose the following construction.

```
HMAC(λcs, ‘‘mls with tls, mac key’’)
```

We refer to this as the record MAC key. This provides integrity and pairwise authentication of the messages to the client and server. The other participants can read the channel, but cannot write correctly formed messages to the channel or verify the MAC tag. This construction addresses a problem with `draft-green` and `draft-RHRD`, both of which sacrifice the authenticity and integrity guarantees of TLS 1.3 along with confidentiality.

[23] Because MLS is still under active development we use a symbolic style, and do not give specific types or ranges. When work on MLS is complete we will revisit this work and specify these details.

6.12 Participants extension

We further propose an extension to TLS, the participants extension. This extension is empty when sent by the client. The server returns the identity tree, the ART, the client and server nonce, along with a MAC tag keyed with the record MAC key. This makes the identities of the participants transparent to the TLS library. A TLS library that implements this extension must include the relevant signalling to the user that MLS with TLS is in use, and further exactly which identities are in the tree, and thus are monitoring the connection. This is the second of our two mechanisms for ensuring user visibility.

If the channel binding does not include the truncated `ClientHello`, i.e. uses the first construction we suggest, and the client does not support this extension at the TLS layer, the user may not know that they are in a visibility scenario. In this case sending the extension unprompted would correctly abort the connection.

Use of this extension makes the use of this mechanism visible on the wire. Prior to draft-18 of the TLS 1.3 specification clients that received extensions that they didn't send in the `ClientHello`[24] were required to abort with an "unsupported extension" alert [Res16, p. 35]. By the final specification this had changed such that an actor receiving the response portion of an extension they did not request was required to abort [RFC8446, p. 36]. We could thus alternatively define this extension such that the server sends the extension during the encrypted extensions as an indicator, without needing prompting from the client. This makes use of MLS with TLS indistinguishable to an adversary.[25] However a client unaware of MLS with TLS would simply ignore the extension in this latter design. When this indistinguishable design is in use we rely on the inclusion of the $Truncate(ClientHello)$ mechanism to ensure the client is aware of MLS with TLS.

[24] With the exception of the `Cookie` extension

[25] Whilst this is true from a symbolic perspective it may be possible to detect by analysing the length of the server's `EncryptedExtensions`.

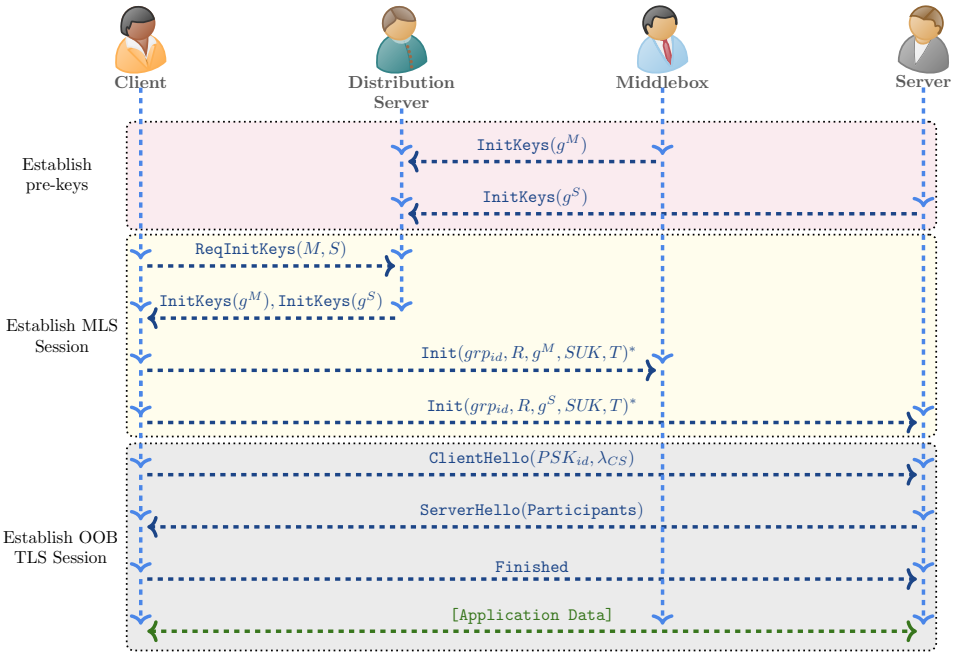


Figure 6.1: Sketch of the MLS with TLS protocol. The starred `Init` messages are currently undefined in the MLS specification, and thus we base the parameters off the ART protocol [Coh+17].

As with our earlier mechanism, this extension can be entirely elided with no effect on the efficacy of the protocol, and is only useful for ensuring user awareness.

6.13 Sketch of the complete composition

In this section we provide a sketch of the complete protocol composition, in the form of parametrised messages.

The actors in this sketch are the client, C, the server, S, the distribution server, D, and a middlebox, M.

$$M \to D : InitKey(pk(sk_M), g^M)$$
$$S \to D : InitKey(pk(sk_S), g^S)$$
$$C \to D : ReqInitKey(M, S)$$
$$D \to C : InitKey(pk(sk_M), g^M), InitKey(pk(sk_S), g^S)$$
$$C \to S : Init(grp_{id}, Roster(C, S, M), g^S, SUK, T)$$
$$C \to M : Init(grp_{id}, Roster(C, S, M), g^S, SUK, T)$$
$$C \to S : ClientHelloPSK(gid, C, S, \texttt{HMAC}(g^{CS'}, Roster(C, S, M)))$$
$$S \to C : ServerHello(Participants(Roster(C, S, M)))$$
$$C \to S : Finished$$

The proposed structure of the `InitKey` message when sent by an actor, A, is as follows. [26]

$$\langle g, g^A, pk(sk_A), sig\_alg, signature \rangle$$

where *signature* is a signature of the message signed with A's LTK.

$$signature := \{g, g^A, pk(sk_A), sig\_alg\}_{sk_A}$$

The `ReqInitKey` message does not yet have a specified form.

The `Init` message does not yet have a specified form, so we have based the parameters on the form in Cohn-Gordon et al. [Coh+17]. The form in Cohn-Gordon et al. is as follows.

$$\langle index(A), [pk(sk_X) \mid X \in Actors], g^A, SUK, copath(A) \rangle$$

Because the MLS protocol does not currently have a mechanism for instantiating a group with more than one member, groups are created by adding each member one by one. This has the longer form:

$$C \to S : \{grp\_id, epoch, roster, T, transcript, init\_secret\}_{g^S} \quad (\texttt{Welcome})$$
$$C \to D : \langle g, g^S, pk(sk_S), sig\_alg, signature \rangle \quad (\texttt{Add})$$
$$S \to D : copath(g^{S'}) \quad (\texttt{Update})$$

[26] This structure has slightly simplified for clarity. The first two fields, g and g^A, are in fact a list of groups and a corresponding list of public keys respectively.

Where the messages sent to the distribution server, D, are forwarded to all members of the group (including S).

The `Welcome` message has all the information S needs to join the group, and is encrypted with S's `init_key`. The `Add` message, which has the same structure as the `InitKey` message is sent to all group members, including S, and is used to update the group state to include S. S then immediately sends an `Update` message, updating its leaf key to a new key constructed from a new DH public key, $g^{S'}$.

6.14 Proposed usage

We don't anticipate that MLS with TLS will receive support in TLS libraries, particularly we anticipate that MLS with TLS will not be supported by browsers. We consider this a vital lynchpin in ensuring MLS with TLS is confined to data centres and enterprise networks.

Because MLS with TLS requires support on both endpoints, and we anticipate limited support we propose MLS with TLS be used in conjunction with split TLS. Split TLS refers to the practice of terminating a TLS connection at a middlebox and creating a new connection with the destination.

For connections that both start and end inside the enterprise network where the client does not support MLS with TLS we propose that the client's TLS 1.3 connection be terminated near the client, with the middlebox creating an MLS with TLS connection on to the server. The middlebox near the client could be an end-point agent[27] on the machine, or a proxy near the client on the network.

If neither the server or the client support MLS with TLS then the connection can be split twice, once near the client, and once near the server. For connections that leave the network, the middlebox must terminate at the edge. Similarly, TLS 1.3 connections that enter the network from outside can be terminated by a middlebox authorised to act as the server, and MLS with TLS run on the internal network.

[27] An end-point agent is a piece of software running on the client or server that performs some extra function. In this case splitting the TLS connection.

This setup has the advantage that it requires the terminating middleboxes to be able to act as certificate authorities for the internal network. Establishing a certificate authority company-wide is possible by requiring the addition of a certificate to the root of trust for each device, however obtaining a certificate authority certificate that is in the "standard" root of trust is quickly detected, and can lead to the issuing authority being shut down.

Famously, Symantec, formerly one of the oldest and largest certificate authorities, was found to be mis-issuing such certificates and was removed from the roots of trust shipped with Chrome[28] and Firefox[29], which together capture the majority of the browser market. This meant that Symantec certificates became much less useful, and consequently much less valuable. This forced Symantec to sell off their PKI business.[30] This means that whilst this mechanism will be effective within corporate environments, any attempts to deploy it across the internet will be detected.

This configuration has advantages over using split TLS alone. First, with split TLS the connection is only decryptable at active middleboxes, this means that either there are only a small number of points where a connection can be read, or there is a high degree of latency as each middlebox decrypts and re-encrypts the connection. With MLS with TLS forming the majority of the connection, middleboxes can be added at virtually any point, with no increase in latency beyond connection setup. Further, because MLS uses pre-keys such that participants do not have to be simultaneously online, the increase in setup time for each new participant is very small as the client only needs to make one round trip to the keying server. This configuration has similar advantages over mcTLS, although it does not offer the fine-grained control over what content can be seen by the middlebox.

[28] `https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html`

[29] `https://blog.mozilla.org/security/2018/03/12/distrust-symantec-tls-certificates/`

[30] `https://www.theregister.co.uk/2017/08/03/symantec_q1_2018/`

6.15 Security considerations

In this section we discuss the security goals of our layered protocol. The TINFOIL document [Far18] is the best summary of all the objections raised against the other drafts. We thus enumerate our security goals, and then proceed systematically through the objections raised. In Section 6.15.3 we discuss shortcomings of our proposal that have not been earlier addressed.

6.15.1 Security goals

We here enumerate the security goals of MLS with TLS. These goals are very similar to the goals of TLS 1.3, and we do not examine them in great detail.

1. Secret session keys: No-one outside the group of participants can read the session.
2. Authentication: No messages will be accepted except those written by the client or the server.
3. Channel binding: The MLS session will be agreed upon by all participants.
4. Transparency: Both the client and server agree on all participants.
5. Protection of endpoint identities: The TLS handshake does not reveal the identities of the participants to a passive outside observer.
6. PFS: Compromise of a participant does not reveal the contents of messages from a previous MLS epoch.
7. Post-compromise security (PCS): After an MLS Update an attacker who cannot derive the updated root key cannot compute any derived secrets.

6.15.2 General concerns

We here enumerate the arguments against all earlier proposals, and discuss how they apply to the various proposals, including our own. We highlight which criticisms apply to which proposals in Table 6.1

	Weakening confidentiality	Weakening TLS	Pervasive monitoring	Secret monitoring	Hidden observers	Mass deployment	TLS complexity	Two-party vs Multi-party	Trasparent and private	Authentication and integrity	Formal analysis	Standardising "broken crypto"	Revising threat models	Multi-party forward secrecy	Participant identity protection	Wireshapping	TLS charter violation	Key revelation	Points of attack
mcTLS	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
draft-green	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓	✗
draft-RHRD	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
MLS with TLS	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗

Table 6.1: Objections to middlebox visibility proposals

[a] mcTLS is not work before the TLS WG
[b] There has been interest in performing a formal analysis on the composition of MLS with TLS
[c] Multi-party forward secrecy policies are an area of active development in MLS [Gil18]

Weakening confidentiality

It is argued that all proposals weaken the confidentiality of TLS 1.3, and thus are prima facie a bad idea. This proposal increases the number of parties who can read the TLS channel. This weakens the confidentiality guarantee of TLS and replaces it with the guarantee from MLS. Alternatively, if one considers all participants except the client as extensions of the server, i.e. the server is defined as a collection of end-points, or at least as not the adversary, then a textualist reading of the definition of the secrecy of TLS session keys holds [CK01, Def 1. part 2], and it can be argued that the confidentiality guarantee remains unchanged.

All the proposals weaken the confidentiality guarantees of TLS 1.3, as this is in fact the goal, but only mcTLS and MLS with TLS make these new guarantees explicit.

Weakening TLS

It is argued that all proposals of this type weaken TLS, which is vastly important given its ubiquity and the sensitivity of the data it protects. mcTLS, `draft-green`, and `draft-RHRD` all modify TLS in ways that weaken its security guarantees. `draft-green` and `draft-RHRD` in particular break guarantees other than confidentiality. mcTLS is based on TLS 1.2, which makes direct comparison of security guarantees impossible, but using mcTLS and remaining on TLS 1.2 is certainly weaker than moving to TLS 1.3.

If used without the extension and cipher suite our proposal does not require any changes to the TLS protocol, and therefore can be considered a valid use of TLS 1.3. We thus argue that this cannot weaken the guarantees of TLS 1.3, although perhaps TLS 1.3 has weaker guarantees than intended.

The two proposed changes we make to TLS 1.3, using the extension and the cipher suite, improve the security guarantees achieved between the client and server, and being independent of the values protected by TLS (bar the OOB-PSK) do not weaken TLS.

Pervasive monitoring

Pervasive monitoring is a threat-model we must consider [RFC7258]. `draft-green` is the most amenable to pervasive monitoring, requiring only changes on the server end. A government could compel a server to implement `draft-green` and without any cooperation from the client monitor all connections to the server. `draft-RHRD`, given its detectability on the wire, is also problematic in this regard. A government could simply drop all connections without the `Visibility` extension, effectively forcing all connections to be monitorable. This criticism can also be made of mcTLS, although as a modification of TLS 1.2 it could also be argued that this is an improvement on the status quo. Further with mcTLS the monitoring devices must be visible to both endpoints, mitigating this risk at least partially.

In our case monitoring requires co-operation from both the client and the server, and, unless the extension is sent by the client, cannot be detected without active participation in every session by the monitor. Even a monitor that just blocks any connection that doesn't use MLS with TLS, assuming it could detect its usage, would have to participate in the MLS handshake to know whether on not it was involved in the session. Further, even if it had been involved initially, the monitor would have to continue to check on each handshake to determine whether it had been removed from the group. Pervasive monitoring through `draft-green` [Gre+17] would be cheaper and simpler to administer and enforce, requiring only co-operation from servers, and operates as a panopticon where clients cannot determine whether a given session is being monitored, and if it is, by whom. Further, because this proposal requires an end-point agent, assuming it is not implemented in core TLS, it would be simpler for a government to simply install an end-point agent that performs monitoring directly, for example the endpoint agents employed in Xinjiang province by China [McC17]. We suggest, therefore, that this proposal does not greatly increase the risk of mass surveillance.

Secret monitoring

A criticism that does apply to our proposal, in particular the use of split TLS with MLS with TLS could allow for secret monitoring.

Use of our proposal requires the client and server to be aware that it is in use, and to explicitly agree to use it. However, if the extension is not used and the channel binding does not include the truncated `ClientHello`, an endpoint agent could keep the fact that this mechanism is in use secret from the client. This is problematic, and thus we propose that both the second channel binding and the indistinguishable extension are used. However, an endpoint agent could still keep this secret by MITMing the connection on the client, and running MLS with TLS over the wire. In this threat model however, an end-point agent could simply push all the plain-text in the clear or publish the master secret. Further this threat model assumes that the end-point agent is a malicious root of trust, which breaks all PKI based protocols.

Hidden observers

A criticism of both `draft-green` and `draft-RHRD` is that even if participants know they are being monitored, they do not know by whom. With mcTLS and with our proposal the client and the server must agree on all participants, including certificates for each. By using the extension the client and server must be explicitly aware of all participants at the TLS layer. To a non-participating observer or a passive participant, the list of participants in the session is secret. A passive participant does not know if it, or any other participant, has been excluded from the session.

Mass deployment

A criticism of `draft-green` and `draft-RHRD` is that they could see mass deployment, even if they are only intended to be used inside the data centre. Because this protocol requires MLS to be run in conjunction with a TLS connection it can only be used where both endpoints support it. If this functionality is not included in browsers or other major libraries of TLS 1.3 its use will require the use of an endpoint agent, limiting its distribution to areas under a single administrative domain.

TLS complexity

A criticism of mcTLS and `draft-RHRD` is that they increase the complexity of TLS and thus increase the chances of an attack. This proposal can be implemented without modification of TLS, although it uses the OOB PSK mechanism in a way that explicitly changes the confidentiality guarantees of TLS. By pinning the confidentiality guarantees of the proposal on MLS, and specifically ARTs, which have some formal analysis, we can ensure that the minimum level of confidentiality is at least that of MLS. The same reasoning can apply to PCS and PFS. Therefore whilst this criticism applies to our proposal too, we argue that our proposal isolates TLS 1.3 from the complexity as much as possible, and further, that the complexity can be studied and that our proposal lends itself to formal analysis.

Two-party vs multi-party

A criticism of `draft-green`, `draft-RHRD`, and mcTLS is that they modify TLS to turn a two-party protocol into a multi-party protocol. This transformation is not well understood, and potentially invalidates all analyses of TLS by changing the basic assumptions.

Our proposal avoids this problem by layering TLS 1.3 on top of a multi-party protocol. TLS is a two-party protocol, so using a multi-party protocol to establish the necessary keys is more appropriate. By then producing a pairwise channel binding we only use TLS in a two-party manner. The transformation therefore is from a multi-party protocol to a two-party protocol, which is special case of a multi-party protocol.

Transparent and private

A design challenge was to design a protocol that is both transparent to the client and server, but also private from the network. `draft-green` is invisible to the network, but not transparent to the client. `draft-RHRD` and mcTLS are both transparent to the client, but also detectable on the network.

If the server sends the Participants extension unprompted then the use of this mechanism is invisible to network intermediaries, except via side-channels, but is explicitly agreed to by the client.[31] Further, a network intermediary who mandates that it be included in every session cannot passively detect compliance, but must actively participate in the session, staying up-to-date with the MLS epoch and decrypting the handshake to ensure that its DH share has been included in the ART. The participant cannot discern, without participation until the end of the $ServerHello$, whether in the latest MLS epoch it was excluded from the participants group.

Authentication and integrity

`draft-green` and `draft-RHRD` both allow middleboxes to derive the master secret of the TLS channel. Because this key is used for confidentiality, integrity, and authentication of the data, a middlebox that knows the key is able to subvert integrity and authentication, as well as decrypt the data. This means that both proposals weaken the TLS connection more than necessary.

mcTLS allows for fine grained control of reads and writes, and once the vulnerability found by Bhargavan et al. [Bha+18] was patched, provides accountable proxying, i.e. middleboxes can modify the channel in an integrity preserving way. By this we mean that any changes are detected by the endpoints, and attributed to the middlebox.

Our proposal preserves integrity and authentication by using a new cipher suite mode and a seperate key to provide authenticity and integrity. We thus improve on `draft-green` and `draft-RHRD`, but do not provide as much functionality as mcTLS.

Formal analysis

`draft-RHRD` has not been subject to any formal analysis, which is a serious criticism. Further the design invalidates the formal analyses of TLS 1.3 that have already been done. TLS is a sufficiently important protocol that formal

[31] Assuming the client is aware of MLS with TLS, i.e. the second channel binding proposal is used.

analysis is required. A proposal that modifies TLS's security guarantees without any formal analysis is problematic, but one that invalidates previous work is worse.

mcTLS has seen substantial formal analysis, although it is based on TLS 1.2 which limits its applicability. `draft-green` arguably doesn't change the way TLS 1.3 operates, or rather is already a valid mechanism. However it changes the base assumptions of TLS 1.3 in a way that invalidates some analyses, including our own from Chapter 3.

Our proposal has complex guarantees and complex mechanisms. It needs rigorous formal analysis before deployment. Because it only applies to a single mode of TLS 1.3, and builds off the work of MLS which has already seen some formal analysis, this work may well be tractable. Using MLS in this way might invalidate the computational analysis of ART, but an initial survey would suggest that extending the computational proof to this case would be trivial. Because this proposal doesn't *require* any changes to TLS the prior analyses should still hold, with the caveat that all participants bar the client are considered to be part of the server. The channel binding between the two layers is contributive, and thus assuming the Bhargavan hypothesis the composition is secure. Composing the analysis of MLS with that of TLS 1.3 and considering the effect of the new cipher mode and extension will require substantial amounts of effort, which we leave for future work.

Standardising "broken" crypto

A criticism of `draft-green` and `draft-RHRD` is that they standardise "broken" cryptography. Standardising weaker forms of cryptography has been the cause of vulnerabilities in TLS, even many years after the weaker forms were considered deprecated. The LOGJAM [Adr+15] and FREAK [Beu+15] attacks both compromise TLS connections by tricking them into using so called "export grade" security. "Export grade" security limited the number of bits American software could use in its keys if it was being exported outside the USA. Although this policy was terminated in the late 1990's cipher

suites with very short keys were still widely supported when these attacks were found in 2015.

This was used as an argument against adoption of the two standards, although in the case of `draft-green` there is nothing explicitly preventing the server using static DH keys in the specification, and thus standardisation is not strictly necessary. Further, in case of `draft-green`, the comparison with "export grade" security is not strong. Every session has a strong key, and if only a single session is run, it is indistinguishable from vanilla TLS 1.3. Sessions are only weaker in aggregate.

`draft-RHRD` suffers this criticism more. Because `draft-RHRD` introduces a new extension it needs to be standardised in order to interoperate between TLS 1.3 implementations. Further, a single session with the `Visibility` extension is weaker than one without, and standardisation would be standardising weaker cryptography.

mcTLS does not suffer this criticism as it was proposed whilst TLS 1.3 was still in the early stages of development, and improves the security of TLS 1.2. Further mcTLS is not an IETF standard.

We argue that this criticism does not apply to our proposal either. Our proposal combines two other standards in a new way, and defines new security guarantees. With the exception of the confidentiality guarantee of TLS it does not violate any of the guarantees of TLS 1.3, and for the confidentiality guarantee offers the guarantee used in MLS. Different guarantees are not inherently broken, in the same way that MLS is not considered "broken" because multiple participants can read a group message. Further, we assert that we do not break cryptography in any meaningful way. In "export grade" cryptography keys have an intentionally limited number of bits, and in `draft-RHRD` keys are intentionally exported, in comparison MLS with TLS, to the best of our knowledge, does not use any constructions or practices that are out of line with current best practice.

Revising threat models

A strong argument against including any of the proposals in a TLS 1.3 library is that any application that called the library would have to revise its

threat model to consider the visibility case. Given the thousands of applications that make use of TLS as a base layer, and that may simply upgrade to TLS 1.3 when their libraries get updated this is entirely infeasible.

We therefore suggest that our proposal not be included in such libraries, but that it be activated by a separate call to a different library. Any application taking this extra step has clearly done so with full knowledge of this suggestion, and with the intent to use it. Further, our proposed deployment, pairing MLS with TLS with split TLS mitigates this issue, because split TLS is currently a threat model with TLS 1.3, and therefore must already be considered by developers.

Multi-party forward secrecy policy

A criticism raised of `draft-green` and `draft-RHRD` is that there is no mechanism by which the client or server can determine the forward secrecy policies of other participants, and this applies equally to our proposal. For example a logging device may just log all traffic and not update their epoch. This would allow the logging device to decrypt historical logs, at the cost of forward secrecy for the entire connection. Establishing these sort of policies is an area of active development for MLS [Gil18].

Participant identity protection

TLS 1.3 guarantees endpoint identity protection, and it is not clear how that guarantee would extend to middleboxes. In `draft-green` and `draft-RHRD` the middleboxes are passive and thus cannot be identified by an outside attacker, but in mcTLS an active attacker could collect certificates from all participants. This suggests that whilst the client's certificate must be protected from active attackers, the certificates of middleboxes only need to be protected from passive attackers.

If our protocol is implemented as written, even an active adversary cannot enumerate the participants added to a connection without being one of the participants added to the connection by the client. In practice however, to aid participant discovery, the client will probably add a single "discovery box", that will then add all appropriate participants. This construction

means that clients do not need to maintain an up-to-date list of middleboxes, just to be able to find a discovery box. In this case the participants will have the same degree of identity protection as the server, i.e. protection from passive adversaries.

Wiretapping

There was much debate whether the various proposals constituted wiretapping under the definition in RFC 2804 [RFC2804]. The RFC defines wiretapping as follows.

> "Wiretapping is what occurs when information passed across the Internet from one party to one or more other parties is delivered to a third party:
>
> 1. Without the sending party knowing about the third party
> 2. Without any of the recipient parties knowing about the delivery to the third party
> 3. When the normal expectation of the sender is that the transmitted information will only be seen by the recipient parties or parties obliged to keep the information in confidence
> 4. When the third party acts deliberately to target the transmission of the first party, either because he is of interest, or because the second party's reception is of interest."
>
> RFC 2804 [RFC2804, pp. 3-4]

For a protocol to be said to enable wiretapping all four conditions must be violated.

The second clause has an unusual construction. Why worry specifically about knowledge of delivery of the message, rather than knowledge of the third party? This question is addressed in the work of Foucault [Fou95] when discussing Panopticism. Panopticism is a reference to the work of Jeremy Bentham on the Panopticon.

The Panopticon is a prison design by Jeremy Bentham [Ben91]. It describes a ring of cells surrounding a central watchtower. The watchtower is designed such that a warden within can see into any cell, but no prisoner can see into the watchtower. This means that whilst the warden cannot watch every cell at once, no prisoner knows whether or not he is being watched. Famously critiqued by Foucault in his book Discipline and Punish [Fou95], Foucault writes

> "He who is subjected to a field of visibility, and who knows it, assumes responsibility for the constraints of power; he makes them play spontaneously upon himself; he inscribes in himself the power relation in which he simultaneously plays both roles; he becomes the principle of his own subjection"
>
> Discipline and Punish [Fou95, pp. 202-203]

In short, because the prisoners know they *could* be being watched, they self-regulate their behaviour as if they *are* being watched. In the same way, the recipient of a message who knows that it could be being read, must regulate its behaviour as if it is being read. This increases the power of the observer over the observed.

This raises significantly different problems from those raised by monitoring someone who is unaware they are being observed. Someone who does not know they are being observed suffers a violation of their privacy, someone who knows they are being observed suffers a violation of their autonomy.

`draft-green` was criticised for enabling wiretapping. It clearly violates conditions 3 and 4. On at least the first connection it is undetectable to the sender, and with minor modifications is always undetectable[32], which violates condition 1.

[32] For example, by sharing the seed of a pseudo-random number generator with observers the server could generate a virtually infinite stream of keys that the observers know, but which it is nearly impossible to detect.

A server configured to use a static DH key has no ability to determine whether the monitor is listening. Even though it knows that a monitor could listen, it does not know whether the plaintext is delivered to the third party, violating condition 2.

`draft-RHRD` suffers from very similar criticism. The violation of condition 3 is not as clear, but if we consider the case where an adversary blocks all connections without the `Visibility` extension, the situation becomes that of the Panopticon. The violation of condition 1 comes from the lack of sender knowledge of who the observers are.

mcTLS does not constitute wiretapping, as the client and the server both are always aware that all middleboxes are actively participating in the channel. This means that conditions 1 and 2 are not violated.

We argue that our proposal does not constitute wiretapping. Not only is the client aware that the third party exists, it has a certificate from each middlebox, thus condition 1 is not violated. We argue that condition 2 is not violated, or at least it cannot be easily reduced to the case of the Panopticon. Consider the case where a middlebox, if it detects it does not have access to a connection, blocks said connection. This is the reduction we use for `draft-RHRD`. The server could detect whether the middlebox is observing a given connection by removing it from the group. If the middlebox is not actively processing MLS group updates then it will be unaware that it has been ejected from the group and allow the connection to proceed. Thus the server can test for message delivery, leaving condition 2 unviolated.[33] Condition 3 is also unviolated, because the client actively nominates the middleboxes that may view the connection, thus it cannot have an expectation that said middleboxes will not read it.

TLS charter violation

The TLS WG objected to `draft-green` and `draft-RHRD` being accepted as work for the WG on the grounds that it is outside the WG's charter. The TLS WG charter calls for improving the security of TLS 1.3, in particular with respect to privacy [Gro18]. `draft-green` is already permissible within

[33] This argument applies equally to the client.

TLS 1.3, and constitutes wiretapping, giving the WG a strong reason to object to accepting it as work. `draft-RHRD` breaks forward secrecy, uses key revelation and enables wiretapping, giving the WG grounds to object to it under BCP 200 [BCP200] and RFC 2804 [RFC2804] respectively. mcTLS was not raised with the TLS WG so this objection is not applicable.

Our proposal can be implemented under TLS 1.3 without any changes, and thus does not violate the charter in that form. Adding the cipher suite and extension, which would require work from the WG, strictly improve the security of a valid deployment of TLS, and thus, we argue, do not constitute a violation of the charter.

Key revelation

`draft-RHRD` uses key revelation, which is not best practice [BCP200]. Our mechanism uses a key agreement protocol, and there is no mechanism for key revelation. Further a participant added to the MLS session after the TLS handshake has begun is unable to read that session. Only after a second handshake with agreement from both the client and server can the new participant read the channel.

Points of attack

A criticism of all the proposals is that, by complicating TLS 1.3, they increase the area which can be attacked. Whilst our proposal increases the number of points of attack, particularly on the confidentiality of a TLS session, we believe with formal analysis these risks can be militated against. Formal analysis reduces the attack surface by forcing attacks to be on implementations rather than on the protocol design.

6.15.3 Specific concerns

In this section we discuss concerns specific to our proposal.

Composition complexity

Protocol composition is hard, and channel bindings in particular have often been attacked, including those designed for use in TLS 1.3 [Cre+16] [BBK17]. Before deployment this proposal would require rigorous formal analysis.

Session linkability

In our design the group ID field remains the same for repeated handshakes, even after MLS updates. This weakens the privacy guarantees achievable in TLS 1.3. It is unclear how this identity could be masked. If this is a problem in a particular environment then a new MLS group could be negotiated for each connection, at the cost of efficiency.

ART computational proof

The re-use of the leaf-keys from MLS potentially invalidates the computational proof of ART, but is required for the construction of a pairwise channel binding (i.e. one that distinguishes between pairs of participants in an MLS session). Therefore our proposal would need analysis as a composite protocol.

Malicious servers

A client that relies on an end-point agent to perform the MLS run, and uses the first channel binding construction, cannot at the TLS layer confirm that the identity tree sent by the server is the one used in the MLS run. If the second construction is used the client can verify the tree, if it trusts the end-point agent. Even using the second construction the client can still be tricked into using the wrong identity tree, but this relies on the end-point agent and the server maliciously collaborating. Without direct support for MLS with TLS in TLS libraries this seems hard to resolve.

6.16 Conclusions

In this chapter we propose a novel method for achieving limited visibility into the payload of TLS 1.3 connections using the OOB PSK mode. We propose two variants, one which requires no changes to TLS 1.3, and one that requires only changes that strictly improve security. By splitting our proposal into these two variants we can easily demonstrate that (1) MLS with TLS does not weaken TLS 1.3, although we do highlight that the OOB PSK mode has weaker guarantees than might have been expected and (2) that our proposed changes to TLS 1.3 meet at least some of the criteria for extensions to TLS, namely that they do not weaken security.

Our design is carefully constructed to aid formal analysis, which we leave for future work. Our design uses best practices, to minimise the risk of flaws. We construct our design of two pieces, asynchronous ratcheting trees (ARTs) and TLS 1.3 OOB PSK mode, both of which have been the subject of rigorous formal study [Coh+17] [Cre+16] [Cre+17a]. The Bhargavan hypothesis, that composite protocols layered with CCBs are secure, has been successful in finding and fixing flaws in composite protocols [BDP15]. This gives evidence that protocols constructed in this way are likely to be secure, and by constructing our composition in this way we minimise the risk of flaws.

In Section 6.15 we carefully compare our proposal to earlier proposals, and consider objections that may be raised. We show that MLS with TLS avoids many of the flaws that made prior proposals problematic, and where we cannot completely avoid the flaws, we discuss how we can militate against them.

We argue that MLS with TLS is a reasonable composition, and worthy of further study. Specifically we propose that a formal analysis of MLS with TLS be performed, which would ameliorate the remaining technical concerns. This analysis needs to await the completion, or at least the stabilisation of the MLS draft, but once this milestone has been reached, we expect the analysis to be feasible.

Chapter 7

Conclusions

In this thesis we study TLS and its use in composite protocols. A central theme has been the use of formal analysis both to evaluate security protocols, and also as an aid in designing them. A secondary theme has been the analysis of composite protocols. Protocols have become more and more complex in design as we try and achieve more and more complex effects. Rather than construct ever more complex protocols from the same base primitives, these complex protocols can be constructed by using other protocols as building blocks. We study how these protocols can be securely composed, and what effects they can achieve. A tertiary theme is the interaction between the IETF and the formal analysis community. By performing analysis on protocol designs before they are standardised flaws in the design can be fixed before they become a problem in the real world. This interaction has provided a fruitful area of practical research problems and allowed the formal analysis community to contribute their expertise to the production of real world protocols.

As protocols have become more complex analysing them has become more difficult. This has driven the development of ever more powerful analysis tools, and in Chapter 3 we show that it is now possible to analyse a protocol as complex as TLS 1.3 in a single piece. We analyse TLS 1.3 using the protocol analysis tool Tamarin [Sch+12], proving that it meets its claimed security guarantees. However we also find that there is an ambiguity on the client side, such that it can never know it is authenticated.

When presented to the TLS WG, this ambiguity was not considered severe enough to warrant changing the handshake itself. In a similar manner, a known replay attack on the 0-RTT mode of TLS 1.3 was not fixed in the protocol. Both problems are noted in the TLS 1.3 specification [RFC8446, p. 146, pp. 150-151], which requires that they be protected against at a different layer. This is because the costs of mitigating it were high, and the security benefit low. To mitigate either issue at the TLS layer would require an additional message be sent, adding latency to the connection. This highlights the trade-offs inherent in protocol design, in this case trading better security at the TLS layer for faster speeds, which will increase adoption.

Throughout this work we develop the understanding of composite protocols, and in particular the properties that relate one layer of the protocol to another. In Chapter 4 we analyse EAs, and their relationship to TLS. The literature on compound authentication, a property that defines the relationship between different layers of a composite protocol, was previously related to a fairly restricted set of protocols. We extend this work with two new properties, inward compound authentication (ICA) and outward compound authentication (OCA). These two properties allowed us to formalise and prove the security guarantees of EAs, whereas earlier techniques were only able to reason about a restricted set of results.

EAs define a two layer composite protocol, comprising of a TLS session and a single EA. In Chapter 5 we use the definition of OCA to extend EAs to LEAs, producing a n-layer composite protocol, using TLS as a base layer, and then layering EAs one atop the other. We describe different layering patterns, leading us to define authentication forests, allowing us to describe very complex authentication properties. Pushing the boundaries of Tamarin, we prove a partial set of results about our composite protocol. We leave completion of the proof for future work, along with an analysis of the extended set of use cases and design changes suggested at the IETF 102 meeting. By proposing work to the TLS WG we invert the pattern of protocols being proposed by the engineering community. Our proposal had a fairly niche set of use cases, but presentation to the TLS WG led to the suggestion of a number of other areas of application, most notably for extending authenti-

cation across a resumption. This again shows that the interaction between formal analysis community and the IETF is useful to both.

Chapter 6 inverts another of the patterns in the thesis. EAs and LEAs both build layers on top of a base TLS layer. We construct a composite protocol that uses a protocol called MLS as base layer, and TLS 1.3 as an upper layer. This construction lets us achieve complex properties, in particular it allows an authenticated and authorised group of third-parties to decrypt, but not modify, the contents of the TLS connection. This work was inspired by a contentious discussion in the TLS WG over whether to support this use case. Numerous technical objections were raised to modifying TLS 1.3 to support this use case. Our construction is designed such that TLS 1.3 may be used unmodified to achieve this effect, whilst avoiding or militating against all technical objections to previous proposals. This demonstrates the power and flexibility of composite protocols, allowing for the construction of a highly complex protocol, with complex properties which is also amenable to formal analysis, which we leave to future work.

Our work pushes the boundaries of formal analysis, extending the set of protocols we can reason about and advancing modelling techniques to allow current tools to analyse more complex protocols. We also provide more evidence for the Bhargavan hypothesis, showing that EAs, which use CCBs, achieve various compound authentication properties. Finally, our work shows that it is possible to construct complex composite protocols with nuanced properties that achieve effects that multiple earlier attempts based on modifying a single layer protocol had failed to achieve.

7.1 Future work

Throughout this thesis we propose a number of pieces of future work. We suggest that the EA model be integrated into the TLS model, removing the abstraction in the EA model, and providing a much more robust result. To achieve this result would require work on both the TLS and EA models to make them more memory efficient.

Another future line of work is to complete the proof of the LEA model. Currently the proof only holds for the immediately preceding EA, this means that whilst we can reason about a 3-layer TLS-EA-LEA composite protocol, we cannot generalise this result to an n-layer composite protocol. Integrating this LEA model with the TLS model would greatly improve the fidelity of the result. Further it would let us examine the use of LEAs across resumptions; a proposed use case of LEAs.

A third line of work we propose is a formal analysis of MLS with TLS. This work requires the MLS protocol to be closer to completion but would provide useful results, finding any potential vulnerabilities in the design and allowing progress to be made both in the MLS with TLS protocol and in the design of composite protocols in general.

Bibliography

[Adr+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*. Denver, Colorado, USA, 2015.

[AFP05] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. "Password-Based Authenticated Key Exchange in the Three-Party Setting". In: *Public Key Cryptography - PKC 2005*. Ed. by Serge Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 65–84. ISBN: 978-3-540-30580-4.

[AM16] Kenichi Arai and Shin'ichiro Matsuo. *Formal Verification of TLS 1.3 Full Handshake Protocol Using ProVerif*. TLS mailing list post. Internet Engineering Task Force, Feb. 2016.

[ANN02] N. Asokan, Valtteri Niemi, and Kaisa Nyberg. *Man-in-the-Middle in Tunnelled Authentication Protocols*. Cryptology ePrint Archive, Report 2002/163. `http://eprint.iacr.org/2002/163`. 2002.

[AP12] Nadhem J. AlFardan and Kenneth G. Paterson. "Plaintext-Recovery Attacks Against Datagram TLS". In: *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. 2012.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 2013, pp. 526–540.

[Avi+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. "DROWN: Breaking TLS with SSLv2". In: *25th USENIX Security Symposium*. Aug. 2016.

[BAN90] Michael Burrows, Martín Abadi, and Roger Needham. "A logic of authentication". In: *ACM TRANSACTIONS ON COMPUTER SYSTEMS* 8 (1990), pp. 18–36.

[Bar+18] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-barnes-mls-protocol-01. Work in Progress. Internet Engineering Task Force, July 2018. 37 pp.

[Bar04] Gregory V. Bard. "The Vulnerability of SSL to Chosen Plaintext Attack". In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 111.

[Bar06] Gregory V. Bard. "A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL". In: *SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal, Portugal, August 7-10, 2006, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*. 2006, pp. 99–109.

[BBF83] R. K. Bauer, T. A. Berson, and R. J. Feiertag. "A key distribution protocol using event markers". In: *ACM Transactions on Computer Systems* 1.3 (Aug. 1983), pp. 249–255.

[BBK17] K. Bhargavan, B. Blanchet, and N. Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 483–502.

[BCP200] Fred Baker and Brian E. Carpenter. *IAB and IESG Statement on Cryptographic Technology and the Internet*. Formerly RFC 1984. https://rfc-editor.org/rfc/rfc1984.txt. Aug. 1996.

[BDP15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. "Verified Contributive Channel Bindings for Compound Authentication". In: *ISOC Network and Distributed System Security Symposium – NDSS 2015*. San Diego, CA, USA: The Internet Society, Feb. 2015.

[Ben91] Jeremy Bentham. *Panopticon; or, The Inspection-House*. Vol. 2. 1791.

[Beu+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 535–552.

[Bha+14] 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, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. 2014, pp. 98–113.

[Bha+16a] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella-Béguelin. "Downgrade resilience in key-exchange protocols". In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 506–525.

[Bha+16b] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. *Implementing and Proving the TLS 1.3 Record Layer*. Tech. rep. `http://eprint.iacr.org/2016/1178`. INRIA, Dec. 2016.

[Bha+18] K. Bhargavan, I. Boureanu, A. Delignat-Lavaud, P. Fouque, and C. Onete. "A Formal Treatment of Accountable Proxying over TLS". In: *2018 IEEE Symposium on Security and Privacy (SP)*. May 2018, pp. 339–356.

[BKB16] K. Bhargavan, N. Kobeissi, and B. Blanchet. "ProScript TLS: Building a TLS 1.3 Implementation with a Verifiable Protocol Model". Presentation. Presented at TRON 1.0, San Diego, CA, USA, February 21. 2016.

[BL16a] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 456–467. ISBN: 978-1-4503-4139-4.

[BL16b] Karthikeyan Bhargavan and Gaëtan Leurent. "Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH". In: *Network and Distributed System Security Symposium–NDSS 2016*. 2016.

[Bla+16] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. *Proverif 1.96: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial List of Figures*. 2016.

[Bla01] B. Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules". In: *Computer Security Foundations Workshop, IEEE(CSFW)*. June 2001, p. 0082.

[Ble98] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*. 1998, pp. 1–12.

[BLR00] P. J. Broadfoot, G. Lowe, and A. W. Roscoe. "Automating Data Independence". In: *Computer Security - ESORICS 2000*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 175–190. ISBN: 978-3-540-45299-7.

[BM94] Colin Boyd and Wenbo Mao. "On a Limitation of BAN Logic". In: *Advances in Cryptology — EUROCRYPT '93*. Ed. by Tor Helleseth. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 240–247. ISBN: 978-3-540-48285-7.

[BMV05] David Basin, Sebastian Mödersheim, and Luca Viganò. "OFMC: A symbolic model checker for security protocols". In: *International Journal of Information Security* 4.3 (June 2005), pp. 181–208. ISSN: 1615-5270.

[BR93] Mihir Bellare and Phillip Rogaway. "Entity authentication and key distribution". In: *Annual International Cryptology Conference*. Springer. 1993, pp. 232–249.

[BST17] Mike Bishop, Nick Sullivan, and Martin Thomson. *Secondary Certificate Authentication in HTTP/2*. Internet-Draft draft-bishop-httpbis-http2-additional-certs-05. `http://www.ietf.org/internet-drafts/draft-bishop-httpbis-http2-additional-certs-05.txt`. IETF Secretariat, Oct. 2017.

[Can+03] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. "Password Interception in a SSL/TLS Channel". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 2003, pp. 583–599.

[CCG16] K. Cohn-Gordon, C. Cremers, and L. Garratt. "On Post-compromise Security". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. June 2016, pp. 164–178.

[CDM17] Cas Cremers, Martin Dehnel-Wild, and Kevin Milner. "Secure Authentication in the Grid: A Formal Analysis of DNP3: SAv5". In: *Computer Security – ESORICS 2017*. Springer International Publishing, 2017, pp. 389–407. ISBN: 978-3-319-66402-6.

[CJM98] E. M. Clarke, S. Jha, and W. Marrero. "Using State Space Exploration and a Natural Deduction Style Message Derivation Engine to Verify Security Protocols". In: *Programming Concepts and Methods PROCOMET '98*. Boston, MA: Springer US, 1998, pp. 87–106. ISBN: 978-0-387-35358-6.

[CK01] Ran Canetti and Hugo Krawczyk. "Analysis of key-exchange protocols and their use for building secure channels". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2001, pp. 453–474.

[Coh+17] Katriel Cohn-Gordon, Cas J. F. Cremers, Luke Garratt, Jon Millican, and Kevin Milner. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: *IACR Cryptology ePrint Archive* 2017 (2017). `https://eprint.iacr.org/2017/666`, p. 666.

[Cre+16] 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: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. May 2016, pp. 470–485.

[Cre+17a] 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, Dallas, USA*. 2017.

[Cre+17b] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. *Archive with TLS 1.3 Rev 10 and Rev 20 models and property specifications for the Tamarin prover*. `http://tls13tamarin.github.io/TLS13Tamarin/`. 2017.

[Cre08] Cas J.F. Cremers. "Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, pp. 119–128. ISBN: 978-1-59593-810-7.

[CYBER-27-2] *Middlebox Security Protocol; Part 2: Transport layer MSP, profile for fine grained access control*. Draft Standard DTS/CYBER-0027-2. European Telecommunications Standards Institute, Apr. 2018. 15 pp.

[CYBER-27-3] *Middlebox Security Protocol; Part 3: Profile for enterprise network and data centre access control*. Draft Standard DTS/CYBER-0027-3. European Telecommunications Standards Institute, July 2018. 15 pp.

[Dow+15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015, pp. 1197–1210.

[Dow+16] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. *A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol*. Cryptology ePrint Archive, Report 2016/081. `http://eprint.iacr.org/`. 2016.

[DR11] Thai Duong and Juliano Rizzo. *Here Come the ⊕ Ninjas*. Unpublished manuscript. May 2011.

[DR12] Thai Duong and Juliano Rizzo. *The CRIME Attack*. Ekoparty Security Conference presentation. 2012.

[DY83] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[Far18] Stephen Farrell. *TLS Is Not For Obligatory (Or Ostensibly Optional) Interception, Luckily*. `https://github.com/sftcd/tinfoil`. Mar. 2018.

[Fen18] Steve Fenter. *Why Enterprises Need Out-of-Band TLS Decryption*. Internet-Draft draft-fenter-tls-decryption-00. Work in Progress. Internet Engineering Task Force, Mar. 2018. 21 pp.

[FHG98] F. J. T. Fabrega, J. C. Herzog, and J. D. Guttman. "Strand spaces: why is a security protocol correct?" In: *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*. May 1998, pp. 160–171.

[Fis+16] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3". In: *2016 IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 23-25, 2016*. 2016.

[Fou95] Michael Foucault. *Discipline and Punish: The Birth of the Prison*. Random House, 1995. ISBN: 0-679-75255-2.

[Gil18] Daniel Kahn Gillmor. *[MLS] key update guidance*. MLS mailing list. `https://mailarchive.ietf.org/arch/msg/mls/ulcn_2pk-N1RMWlWpbQ91mQOlgE`. July 2018.

[GK00] Thomas Genet and Francis Klay. "Rewriting for Cryptographic Protocol Verification". In: *Automated Deduction - CADE-17*. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2000, pp. 271–290. ISBN: 978-3-540-45101-3.

[GL97] François Germeau and Guy Leduc. "Model-based design and verification of security protocols using LOTOS". In: *DIMACS Workshop on Design and Formal Verification of Security Protocols* (1997).

[GNY90] L. Gong, R. Needham, and R. Yahalom. "Reasoning about belief in cryptographic protocols". In: *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*. May 1990, pp. 234–248.

[Gon93] L. Gong. "Variations on the themes of message freshness and replay. or the difficulty in devising formal methods to analyze cryptographic protocols". In: *[1993] Proceedings Computer Security Foundations Workshop VI*. IEEE Comput. Soc. Press, 1993.

[Gou00] Jean Goubault-Larrecq. "A Method for Automatic Cryptographic Protocol Verification". In: *Parallel and Distributed Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, May 2000, pp. 977–984. ISBN: 978-3-540-45591-2.

[GPV15] Christina Garman, Kenneth G Paterson, and Thyla Van der Merwe. "Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS." In: *USENIX Security*. 2015, pp. 113–128.

[Gre+17] Matthew Green, Ralph Droms, Russ Housley, Paul Turner, and Steve Fenter. *Data Center use of Static Diffie-Hellman in TLS 1.3*. Expired Internet-Draft draft-green-tls-static-dh-in-tls13-01. Archived. Internet Engineering Task Force, July 2017. 15 pp.

[Gro18] TLS Working Group. *TLS Working Group Charter*. IETF. `https://datatracker.ietf.org/doc/charter-ietf-tls/`. Jan. 2018.

[Gün90] Christoph G. Günther. "An Identity-Based Key-Exchange Protocol". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1990, pp. 29–37.

[HD18] Russ Housley and Ralph Droms. *TLS 1.3 Option for Negotiation of Visibility in the Datacenter*. Internet-Draft draft-rhrd-tls-tls13-visibility-01. Work in Progress. Internet Engineering Task Force, Mar. 2018. 11 pp.

[HL01] Mei Lin Hui and Gavin Lowe. "Fault-preserving simplifying transformations for security protocols". In: *Journal of Computer Security* 9.1-2 (2001), pp. 3–46.

[Hor16] Marko Horvat. "Formal Analysis of Modern Security Protocols in Current Standards". PhD thesis. University of Oxford, 2016.

[Hoy18a] Jonathan Hoyland. *Exported Authenticators Tamarin Repository*. `https://www.dropbox.com/s/4rgk9m226zex2ed/exported-authenticators.zip`. 2018.

[Hoy18b] Jonathan Hoyland. *Layered Exported Authenticators in TLS*. Internet-Draft draft-hoyland-tls-layered-exported-authenticator-00. Work in Progress. Internet Engineering Task Force, June 2018. 5 pp.

[JSS15a] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015, pp. 1185–1196.

[JSS15b] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "Practical invalid curve attacks on TLS-ECDH". In: *European Symposium on research in computer security*. Springer. 2015, pp. 407–425.

[Kah74] David Kahn. *The Codebreakers*. Purnell, 1974.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[Koh+14] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. "(De-)Constructing TLS". In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 20.

[KPR03] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. "Attacking RSA-Based Sessions in SSL/TLS". In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. 2003, pp. 426–440.

[KW15] Hugo Krawczyk and Hoeteck Wee. "The OPTLS Protocol and TLS 1.3". In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 978.

[Lan12] Li Lanqing. *Works of Art by Li Lanqing: Chinese Seals and Calligraphy*. Trans. by Yuan Ailing and Zeng Yi. Macmillan Publishers, 2012. ISBN: 978-0-2304-5132-2.

[LG92] Kwok-yan Lam and Dieter Gollmann. "Freshness assurance of authentication protocols". In: *Computer Security — ESORICS 92*. Springer Berlin Heidelberg, 1992, pp. 261–271.

[Li+14] Yong Li, Sven Schäge, Zheng Yang, Florian Kohlar, and Jörg Schwenk. "On the Security of the Pre-shared Key Ciphersuites of TLS". In: *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*. 2014, pp. 669–684.

[LLM07] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange". In: *Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings*. 2007, pp. 1–16.

[Low95] Gavin Lowe. "An attack on the Needham-Schroeder public-key authentication protocol". In: *Information Processing Letters* 56.3 (Nov. 1995), pp. 131–133. ISSN: 0020-0190.

[Low96] 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. ISBN: 978-3-540-49874-2.

[Low97] Gavin Lowe. "A Hierarchy of Authentication Specifications". In: *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*. CSFW '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 31–43. ISBN: 0-8186-7990-5.

[Mac17] Colm MacCárthaigh. *Security review of TLS1.3 0-RTT*. TLS mailing list post. Available at `https://www.ietf.org/mail-archive/web/tls/current/msg23051.html`. Internet Engineering Task Force, May 2017.

[Man15] Itsik Mantin. *Attacking SSL when using RC4*. White Paper. Mar. 2015.

[Mar13] Moxie Marlinspike. *Forward Secrecy for Asynchronous Messages*. `https://signal.org/blog/asynchronous-security/`. Aug. 2013.

[Mav+12] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. "A cross-protocol attack on the TLS protocol". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. 2012, pp. 62–72.

[McC17] Kieren McCarthy. "China crams spyware on phones in Muslim-majority province". In: *The Register* (June 2017).

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. *This POODLE bites: Exploiting The SSL 3.0 Fallback*. Security Advisory. Google, Sept. 2014.

[Mea96] Catherine A. Meadows. "Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches". In: *Computer Security — ESORICS 96*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 351–364. ISBN: 978-3-540-70675-5.

[Mei13] Simon Meier. "Advancing automated security protocol verification". PhD thesis. ETH Zurich, 2013.

[Mer88] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[MF18] Kathleen Moriarty and Stephen Farrell. *Deprecating TLSv1.0 and TLSv1.1.* Internet-Draft draft-moriarty-tls-oldversions-diediedie-01. Work in Progress. Internet Engineering Task Force, July 2018. 14 pp.

[Mil95] J. K. Millen. "The Interrogator model". In: *Proceedings 1995 IEEE Symposium on Security and Privacy.* May 1995, pp. 251–260.

[Mit+99] J.C. Mitchell, A. Scedrov, N.A. Durgin, and P.D. Lincoln. "Undecidability of bounded security protocols". In: *Workshop on Formal Methods and Security Protocols.* 1999.

[Mit98] John C. Mitchell. "Finite-state analysis of security protocols". In: *Computer Aided Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 71–76. ISBN: 978-3-540-69339-0.

[Moe04] Bodo Moeller. *Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures.* Unpublished manuscript. `http://www.openssl.org/~bodo/tls-cbc.txt`. May 2004.

[Mon99a] David Monniaux. "Abstracting Cryptographic Protocols with Tree Automata". In: *Static Analysis.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 149–163. ISBN: 978-3-540-48294-9.

[Mon99b] David Monniaux. "Decision procedures for the analysis of cryptographic protocols by logics of belief". In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop.* June 1999, pp. 44–54.

[MSM97] M. Mitchell, U. Stern, and J. Mitchell. "Automated analysis of cryptographic protocols using Murϕ". In: *Proceedings. 1997 IEEE Symposium on Security and Privacy.* Vol. 00. May 1997, p. 0141.

[Nay+15] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.* SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 199–212. ISBN: 978-1-4503-3542-3.

[Nes90] Dan M. Nessett. "A Critique of the Burrows, Abadi and Needham Logic". In: *SIGOPS Oper. Syst. Rev.* 24.2 (Apr. 1990), pp. 35–38. ISSN: 0163-5980.

[NRS-603A] *SECURITY AND PRIVACY OF PERSONAL INFORMATION*. Vol. NRS-603A.215. `https://www.leg.state.nv.us/NRS/NRS-603A.html`. Nevada State Legislature, 2009.

[NS78] Roger M. Needham and Michael D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers". In: *Commun. ACM* 21.12 (Dec. 1978), pp. 993–999. ISSN: 0001-0782.

[NSS18] Yoav Nir, Rich Salz, and Nick Sullivan. *Transport Layer Security (TLS) Extensions*. IANA. `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml`. Aug. 2018.

[OR87] Dave Otway and Owen Rees. "Efficient and timely mutual authentication". In: *ACM SIGOPS Operating Systems Review* 21.1 (Jan. 1987), pp. 8–10.

[PCI DSS] *PCI-DSS 3.2.1*. Industry Standard. `https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf`. May 2018.

[PM16] Kenneth G. Paterson and Thyla van der Merwe. "Reactive and Proactive Standardisation of TLS". In: *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*. 2016, pp. 160–186.

[PS00] A. Perrig and D. Song. "Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols". In: *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*. July 2000, pp. 64–76.

[Res15] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-10. Work in Progress. Internet Engineering Task Force, Oct. 2015. 103 pp.

[Res16] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-18. Work in Progress. Internet Engineering Task Force, Oct. 2016. 118 pp.

[Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-28. Work in Progress. Internet Engineering Task Force, Mar. 2018. 156 pp.

[RFC2804] Fred Baker and Brian E. Carpenter. *IETF Policy on Wiretapping*. RFC 2804. May 2000.

[RFC3552] Eric Rescorla and Brian Korver. *Guidelines for writing RFC text on security considerations*. RFC 3552 (Informational). Internet Engineering Task Force, July 2003.

[RFC3935] Harald T. Alvestrand. *A Mission Statement for the IETF*. RFC 3935. Oct. 2004.

[RFC5056] Nicolas Williams. *On the Use of Channel Bindings to Secure Channels*. RFC 5056. Nov. 2007.

[RFC5869] P. Eronen and H. Krawczyk. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869 (Informational). Internet Engineering Task Force, May 2010.

[RFC7258] Stephen Farrell and Hannes Tschofenig. *Pervasive Monitoring Is an Attack*. RFC 7258. May 2014.

[RFC7540] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015.

[RFC7627] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Alfredo Pironti, Adam Langley, and Marsh Ray. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. RFC 7627. Sept. 2015.

[RFC8446] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018.

[RS03] R. Ramanujam and S.P. Suresh. "Information based reasoning about security protocols". In: *Electronic Notes in Theoretical Computer Science* 55.1 (Jan. 2003), pp. 85–100.

[RS04] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption*. Springer Berlin Heidelberg, 2004, pp. 371–388.

[Sch+12] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE, 2012, pp. 78–94.

[Sch+14] Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David Basin. "Automated Verification of Group Key Agreement Protocols". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 179–194. ISBN: 978-1-4799-4686-0.

[SM84] Goldwasser Shafi and Silvio Micali. “Probabilistic encryption”. In: *Journal of computer and system sciences* 28.2 (1984), pp. 270–299.

[Son99] Dawn Xiaodong Song. “Athena: a new efficient automatic checker for security protocol analysis”. In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. June 1999, pp. 192–202.

[SR09] Joe Salowey and Eric Rescorla. “TLS Renegotiation Vulnerability”. In: *IETF Proceedings*. Nov. 2009.

[Sul17] Nick Sullivan. *Exported Authenticators in TLS*. Internet-Draft draft-ietf-tls-exported-authenticator-05. Work in Progress. Internet Engineering Task Force, Dec. 2017. 9 pp.

[Sul18a] Nick Sullivan. *Exported Authenticators in TLS*. Internet-Draft draft-ietf-tls-exported-authenticator-07. Work in Progress. Internet Engineering Task Force, June 2018. 12 pp.

[Sul18b] Nick Sullivan. *Exported Authenticators in TLS*. Internet-Draft draft-ietf-tls-exported-authenticator-06. Work in Progress. Internet Engineering Task Force, Mar. 2018. 11 pp.

[Syv94] Paul Syverson. “A taxonomy of replay attacks [cryptographic protocols]”. In: *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*. IEEE. 1994, pp. 187–191.

[TAMARIN] *Tamarin prover GitHub repository (develop branch)*. `https://github.com/tamarin-prover/tamarin-prover`. 2015.

[Tsu92] Gene Tsudik. “Message authentication with one-way hash functions”. In: *ACM SIGCOMM Computer Communication Review* 22.5 (Oct. 1992), pp. 29–38.

[Tur17] Sean Turner. *Minutes IETF 99: TLS*. `https://datatracker.ietf.org/meeting/99/materials/minutes-99-tls-00.pdf`. Video available at `https://www.youtube.com/watch?v=ms-0PlY1R-8`. Aug. 2017.

[Tur18] Sean Turner. *Minutes IETF 101: TLS*. `https://datatracker.ietf.org/doc/minutes-101-tls-201803191740/`. Video available at `https://www.youtube.com/watch?v=7hclQbuCBws`. Mar. 2018.

[Vau02] Serge Vaudenay. “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...” In: *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*. 2002, pp. 534–546.

[VK83] Victor L Voydock and Stephen T Kent. "Security mechanisms in high-level network protocols". In: *ACM Computing Surveys (CSUR)* 15.2 (1983), pp. 135–171.

Appendix A

TLS Model State Diagrams

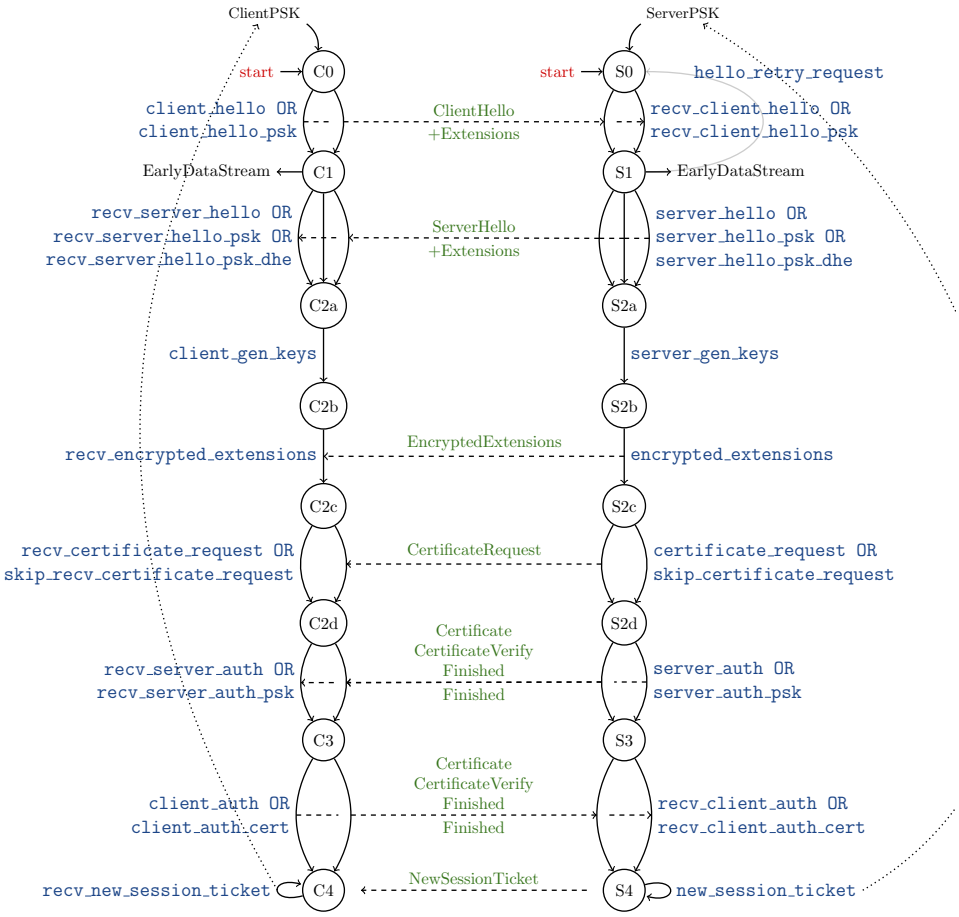


Figure A.1: Part 1 of the full state diagram for Tamarin model, showing all rules covered in the initial handshake (excluding rules dealing with record layer).

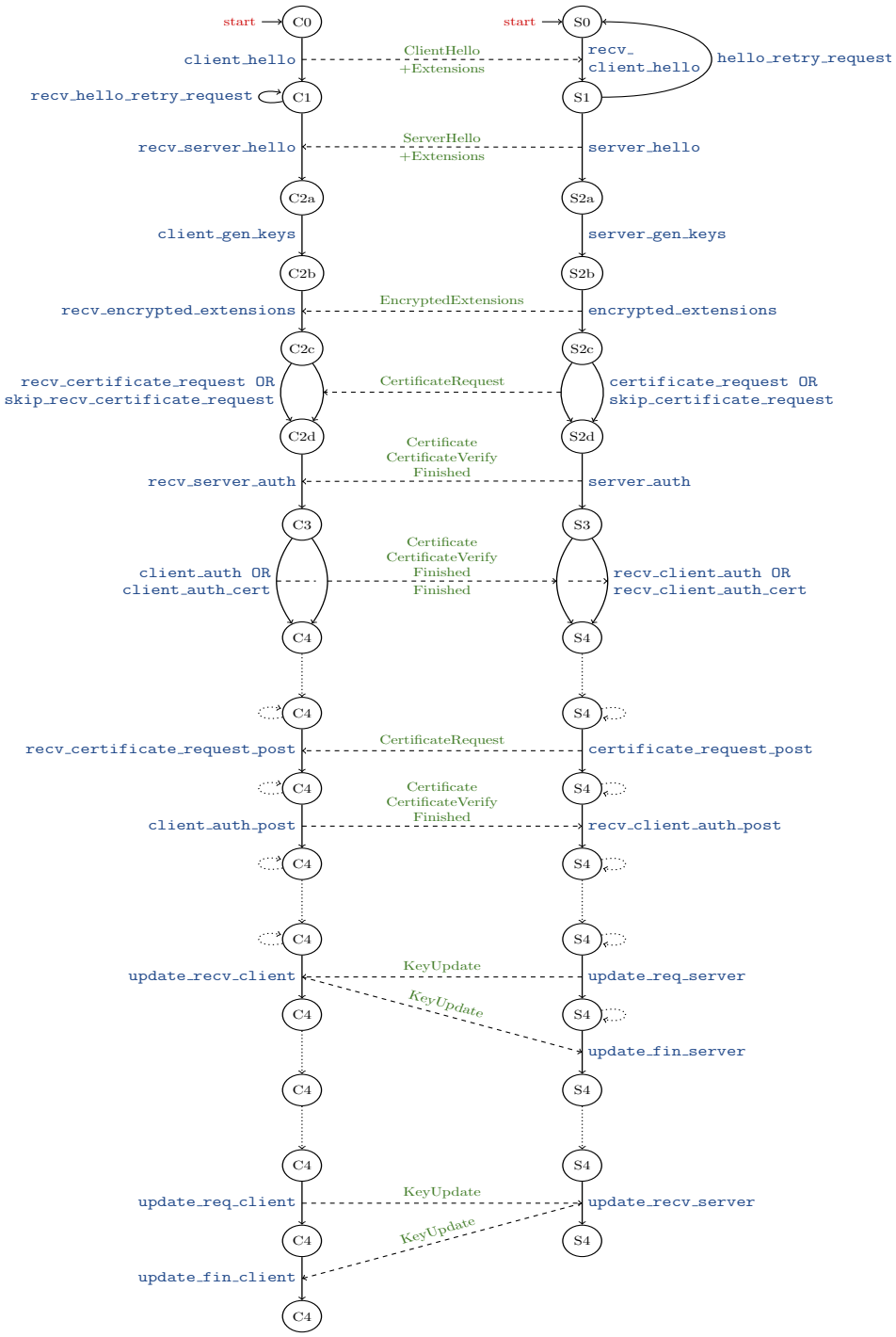


Figure A.2: Part 2 of the full state diagram for Tamarin model, showing all post-handshake rules covered.

Appendix B

draft-hoyland

Internet Engineering Task Force J. Hoyland, Ed.
Internet-Draft Royal Holloway, University of London
Intended status: Standards Track June 25, 2018
Expires: December 27, 2018

Layered Exported Authenticators in TLS
draft-hoyland-tls-layered-exported-authenticator-00

Abstract

This document describes an extension that allows for Exported Authenticators (EAs) to authenticate each other. The extension includes a reference to a previous EA. An EA containing this extension constitues an attestation of the authenticity of the referenced EA.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 27, 2018.

Copyright Notice

Table of Contents

1. Introduction

Exported Authenticators (EAs)[EA] provide a method for authenticating one party of a Transport Layer Security (TLS) communication to the other after the session has been established. EAs are defined for TLS 1.3[TLS13] and TLS 1.2 with extended master secret, RFC 7627 [RFC7627]. Multiple EAs sent on the same channel do not prove joint authentication. They prove that the sender is individually authoritative over each certificate, but not jointly authoritative over all certificates. By including this extension a sender can prove joint authentication. This extension can be included in CertificateRequest messages and Certificate messages.

Joint authentication could be used, for example, to securely update pinned certificates. When a client connects to a server for which it has a pinned certificate, the server could send the new certificate

to be pinned, and then bind the previously pinned certificate to it. This proves to the client that the server is jointly authoritative over both certificates. To defeat this mechanism an attacker is required to both compromise the key of the old certificate and improperly obtain a certificate from the PKI.

Another potential use is to provide proof that a certificate has been accepted. Because EAs do not have a response mechanism, the sender of an EA does not know the receiver's view of its authentication status. By using this extension to reference EAs sent by its peer, a party can prove to its peer that it has accepted a particular certificate.

By constructing a chain of referenced EAs complex joint authentication properties can be achieved.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Extension Format

The "extension_data" field of this extension SHALL contain:

```
struct {
     opaque prev_certificate_request_context<0..2^8-1>;
     opaque binding[Hash.length];
} LayeredEA;
```

where "prev_certificate_request_context" is the certificate request context of the EA you are referencing, and "binding" is the Finished message of that same EA. The hash used is that used in the exported authenticator, which is the hash function used by the TLS connection.

A party who wishes its peer to prove it is jointly authoritative over multiple certificates can request a sequence of certificates, each bound to its predecessor. Receipt of a series of EAs binding these certificates into a chain proves the sender is jointly authoritative over all those certificates.

A party who receives a CertificateRequest with this extension MUST verify that it previously received or sent an EA with the appropriate certificate request context and Finished message. If so then the party MAY respond with a Certificate fulfilling the request, or it MAY choose to not fulfil the request.

A party who receives a request from its peer for which it does not recognise the referenced certificate or does not want to link to the referenced certificate for some other reason, but still wishes to respond with an EA MAY send an EA omitting the extension, or it MAY choose to not fulfil the request. If the peer receives an EA with the extension omitted it proves the sender is authoritative over the

certificate in the EA, but makes no claims about the previous EA referenced in the request.

For spontaneous certificates The server MUST include a unique (within the context of the connection) certificate_request_context for any EA it may wish to bind to. To be able to verify bindings both parties must keep a list of accepted EAs they are willing to bind to, including certificate_request_contexts and Finished messages. A client that receives a spontaneous EA with a

certificate_request_context that it has already seen and for which it is willing to receive a binding MUST ignore it.

3. Acknowledgements

4. IANA Considerations

This document requests IANA to update the TLS ExtensionsType registry, defined in [TLS13], to include the layered_exported_authenticator extension.

5. Security Considerations

For the authentication guarantees to apply, requests, and thus responses, must unambiguously identify previous EAs. Because EAs do not place a restriction on both parties to a connection using the

same certificate_request_context, the certificate_request_context is not sufficient to unambiguously identify previous EAs. Because EAs are unidirectional, and the Finished message is dependent on the labels used to enforce this, the Finished message is sufficient to identify previous EAS unambiguously. In the case of spontaneous EAs a malicious server or an attacker who had compromised the TLS channel could send two identical spontaneous EAs. To militate against this a client receiving such an EA MUST check that it has not already accepted an EA with the same certificate_request_context that it is willing to bind to. If it previously accepted such a certificate but did not add it to the list of certificates which it was willing to bind to, adding it to the list is still secure. The certificate_request_context is included in the request to ease identification of the previous EA, but is not sufficient alone.

Both parties can be sure the Finished messages that are used to reference previous EAs are unique. For requested EAs the inclusion of the certificate_request_context, which is generated by the requestor, guarantees this is the case. For spontaneous certificates the client may only accept EAs after checking it does not have any EAs it is willing to bind to with the same certificate_request_context.

The Finished messages amount to channel bindings as defined in RFC5056 [RFC5056], and thus publication of them should not weaken the security of either the referenced EA or the TLS channel.

This extension only authenticates prior EAs. Thus, an attacker who is able to compromise a TLS connection could append authentications to the connection. Any attempt to bind to these certificates by an honest agent would not be accepted by the peer.

6. References

6.1. Normative References

[EA] Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-07 (work in progress), June 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <https://www.rfc-editor.org/info/rfc7627>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

6.2. Informative References

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure

Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007,
<https://www.rfc-editor.org/info/rfc5056>.

Author's Address

Jonathan Hoyland (editor)
Royal Holloway, University of London
Egham
UK

Email: jonathan.hoyland@gmail.com

Appendix C

Partial Deconstructions

For ease of reference we reproduce the code provided with the Tamarin Prover to highlight the problems with partial deconstructions. The code was authored by Simon Meier[1]. We discuss the issue of partial deconstructions in Chapter 5.

C.1 The Needham-Schroeder-Lowe protocol

```
theory NSLPK3
begin

builtins: asymmetric-encryption

/*
   Protocol:    The classic three message version of the
                Needham-Schroeder-Lowe Public Key Protocol
   Modeler:     Simon Meier
   Date:        June 2012
   Source:      Modeled after the description by Paulson in
                Isabelle/HOL/Auth/NS_Public.thy.

   Status:      working

   Note that we are using explicit global constants for
   ↪  discerning the
   different encryption instead of the implicit sources.
```

[1]The code is sourced from the Tamarin prover repository and can be accessed at `https://github.com/tamarin-prover/tamarin-prover/blob/master/examples/classic/NSLPK3.spthy`

```
 */


// Public key infrastructure
rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule Reveal_ltk:
  [ !Ltk(A, ltkA) ] --[ RevLtk(A)    ]-> [ Out(ltkA) ]


/* We formalize the following protocol

  protocol NSLPK3 {
    1. I -> R: {'1',ni,I}pk(R)
    2. I <- R: {'2',ni,nr,R}pk(I)
    3. I -> R: {'3',nr}pk(R)
  }
*/

rule I_1:
  let m1 = aenc{'1', ~ni, $I}pkR
  in
    [ Fr(~ni)
    , !Pk($R, pkR)
    ]
  --[ OUT_I_1(m1)
    ]->
    [ Out( m1 )
    , St_I_1($I, $R, ~ni)
    ]

rule R_1:
  let m1 = aenc{'1', ni, I}pk(ltkR)
      m2 = aenc{'2', ni, ~nr, $R}pkI
  in
    [ !Ltk($R, ltkR)
    , In( m1 )
    , !Pk(I, pkI)
    , Fr(~nr)
    ]
  --[ IN_R_1_ni( ni, m1 )
```

```
62      , OUT_R_1( m2 )
63      , Running(I, $R, <'init',ni,~nr>)
64      ]->
65      [ Out( m2 )
66      , St_R_1($R, I, ni, ~nr)
67      ]
68
69  rule I_2:
70    let m2 = aenc{'2', ni, nr, R}pk(ltkI)
71        m3 = aenc{'3', nr}pkR
72    in
73      [ St_I_1(I, R, ni)
74      , !Ltk(I, ltkI)
75      , In( m2 )
76      , !Pk(R, pkR)
77      ]
78    --[ IN_I_2_nr( nr, m2)
79      , Commit (I, R, <'init',ni,nr>)  // need to log identities
        ↪  explicitely to
80      , Running(R, I, <'resp',ni,nr>)  // specify that they must
        ↪  not be
81                                       // compromised in the
                                          ↪  property.
82      ]->
83      [ Out( m3 )
84      , Secret(I,R,nr)
85      , Secret(I,R,ni)
86      ]
87
88  rule R_2:
89      [ St_R_1(R, I, ni, nr)
90      , !Ltk(R, ltkR)
91      , In( aenc{'3', nr}pk(ltkR) )
92      ]
93    --[ Commit (R, I, <'resp',ni,nr>)
94      ]->
95      [ Secret(R,I,nr)
96      , Secret(R,I,ni)
97      ]
98
99  /* TODO: Also model session-key reveals and adapt security
    ↪  properties. */
100 rule Secrecy_claim:
101   [ Secret(A, B, m) ] --[ Secret(A, B, m) ]-> []
```

```
102
103
104
105 /* Note that we are using an untyped protocol model.
106 The contents of the 'ni' variable in rule R_1 may therefore in
    ↪ general be any
107 message. This leads to unsolved chain constraints when
    ↪ checking what message
108 can be extracted from the message sent by rule R_1. In order
    ↪ to get rid of
109 these constraints, we require the following sources invariant
    ↪ that relates the
110 point of instantiation to the point of sending by either the
    ↪ adversary or the
111 initiator.
112
113 In order to understand the use of this sources invariant you
    ↪ might try the
114 follwing experiment. Comment out this sources invariant and
    ↪ then check the
115 precomputed case distinctions in the GUI. Try to complete the
    ↪ proof of the
116 'nonce_secrecy' lemma.
117 */
118 lemma types [sources]:
119   " (All ni m1 #i.
120         IN_R_1_ni( ni, m1) @ i
121         ==>
122         ( (Ex #j. KU(ni) @ j & j < i)
123         | (Ex #j. OUT_I_1( m1 ) @ j)
124         )
125     )
126   & (All nr m2 #i.
127         IN_I_2_nr( nr, m2) @ i
128         ==>
129         ( (Ex #j. KU(nr) @ j & j < i)
130         | (Ex #j. OUT_R_1( m2 ) @ j)
131         )
132     )
133   "
134
135 // Nonce secrecy from the perspective of both the initiator
    ↪ and the responder.
136 lemma nonce_secrecy:
```

```
137   " /* It cannot be that */
138     not(
139         Ex A B s #i.
140           /* somebody claims to have setup a shared secret, */
141           Secret(A, B, s) @ i
142           /* but the adversary knows it */
143         & (Ex #j. K(s) @ j)
144           /* without having performed a long-term key reveal.
                ↪  */
145         & not (Ex #r. RevLtk(A) @ r)
146         & not (Ex #r. RevLtk(B) @ r)
147        )"
148
149 // Injective agreement from the perspective of both the
    ↪  initiator and the responder.
150 lemma injective_agree:
151   " /* Whenever somebody commits to running a session, then*/
152     All actor peer params #i.
153         Commit(actor, peer, params) @ i
154       ==>
155         /* there is somebody running a session with the same
            ↪  parameters */
156           (Ex #j. Running(actor, peer, params) @ j & j < i
157             /* and there is no other commit on the same
                  ↪  parameters */
158             & not(Ex actor2 peer2 #i2.
159                     Commit(actor2, peer2, params) @ i2 &
                          ↪  not(#i = #i2)
160                   )
161           )
162         /* or the adversary perform a long-term key reveal on
            ↪  actor or peer */
163         | (Ex #r. RevLtk(actor) @ r)
164         | (Ex #r. RevLtk(peer)  @ r)
165   "
166
167 // Consistency check: ensure that secrets can be shared
    ↪  between honest agents.
168 lemma session_key_setup_possible:
169   exists-trace
170   " /* It is possible that */
171     Ex A B s #i.
172       /* somebody claims to have setup a shared secret, */
173       Secret(A, B, s) @ i
```

```
174         /* without the adversary having performed a long-term
            ↪  key reveal. */
175       & not (Ex #r. RevLtk(A) @ r)
176       & not (Ex #r. RevLtk(B) @ r)
177     "
178
179 end
```