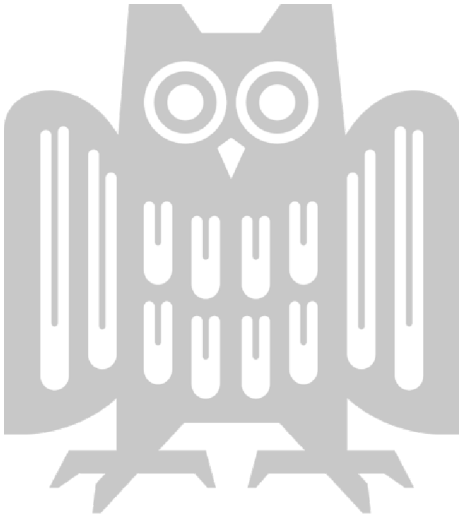
Advancing Security Protocol Verification:
A Journey Along The Boundaries Of The Symbolic Model

Alexander Dax

A dissertation submitted towards the degree
Doctor of Natural Sciences (Dr. rer. nat.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, 2025.



Date of Colloquium:	10.07.2025
Dean of the Faculty:	Prof. Dr. Roland Speicher
Chair of the Committee:	Prof. Dr. Jan Reineke
Reviewers:	Prof. Dr. Cas Cremers
	Prof. Dr. Ioana Boureanu
	Prof. Dr. Jannik Dreier
Academic Assistant:	Dr. Matthias Fassl

Abstract

Cryptographic protocols underpin modern digital security, yet their formal verification remains a significant challenge. The symbolic model of cryptography addresses this, representing bitstrings as terms and cryptographic operations as semantics of function symbol applications. This approach allows a more abstract representation of protocols, improving scalability and automation during analysis. However, this high level of abstraction can overlook attacks that exploit the subtleties of underlying cryptographic operations. This thesis advances the accuracy of symbolic analysis by refining the abstraction of cryptographic primitives and by assessing the limits of leading verification tools.

First, we propose more detailed symbolic models for cryptographic hash functions, authenticated encryption with associated data (AEAD), and key encapsulation mechanisms (KEMs). We integrate these models into Tamarin, a leading symbolic verification tool, and demonstrate their ability to automatically detect both known and novel attacks in real-world security protocols.

Second, we present the first formal analysis of the Security Protocol and Data Model (SPDM), a widely deployed industry security standard. Our work results in one of the largest Tamarin models to date, pushing the boundaries of symbolic analysis and revealing a severe authentication vulnerability. This discovery led to our proposed fixes being included in both the specification and the reference implementation.

Zusammenfassung

Kryptographische Protokolle sind die Grundlage moderner digitaler Sicherheit, doch ihre formale Verifikation bleibt eine große Herausforderung. Aus diesem Grund wurde das symbolische Modell entwickelt, welches eine abstraktere Darstellung von Protokollen ermöglicht und dadurch höhere Skalierbarkeit und Automatisierung erlaubt. Diese hohe Abstraktionsebene kann jedoch Angriffe übersehen, die die Feinheiten der zugrunde liegenden kryptografischen Operationen ausnutzen. Diese Dissertation verbessert die Genauigkeit der symbolischen Analyse durch die Verfeinerung der Abstraktion von kryptographischen Primitiven und wertet währenddessen die Grenzen von aktuellen Verifikationstools aus.

Zunächst erstellen wir detailliertere symbolische Modelle für kryptografische Hash Funktionen, authentifizierte Verschlüsselung mit zugehörigen Daten, und Schlüsselkapselungsmechanismen. Wir integrieren diese Modelle in Tamarin, ein führendes symbolisches Verifikationstool, und demonstrieren ihre Fähigkeit, automatisch bekannte und neue Angriffe in realen Sicherheitsprotokollen zu erkennen.

Zweitens präsentieren wir die erste formale Analyse des Security Protocol and Data Model (SPDM), einem weit verbreiteten Industriesicherheitsstandard. Diese Analyse resultiert in einem der bisher umfangreichsten Tamarin-Modelle, das die Grenzen der symbolischen Analyse erweitert und eine schwerwiegende Schwachstelle in der Authentifizierung aufgedeckt.

Acknowledgements

"Feeling gratitude and not expressing it is like wrapping a gift and not giving it."
– William Arthur Ward

From my first day as a student assistant in 2016, I spent an incredible nine years at CISPA. During this journey, I met far too many amazing people to thank them all properly.

My deepest gratitude goes to Cas, my supervisor, who had just moved to Saarbrücken to join CISPA as faculty when I met him during my master's studies. After getting to know him, his group, and his research vision and ethics, I knew that if I were to pursue a PhD, it would have to be with him. Cas, thank you for accepting me as your student in 2020 and for the incredible five years that followed. Your mentorship went far beyond academic guidance – you gave me the freedom to grow while providing direction when I needed it most, and perhaps most importantly, you gave me the confidence to make mistakes and learn from them. *Thank you!*

I want to thank my wonderful group members Benjamin, Mang, Jacqueline, Esra, Erik, Aleksi, and Maïwenn, who made this journey not just bearable, but genuinely enjoyable. My special thanks go to Aurora, Charlie, and Niklas for helping me navigate the strange and stressful COVID times. During those weird pandemic months when we couldn't see each other in person, our evening gaming sessions and the occasional drink became lifelines – whether we were battling through imaginary worlds or just talking through the challenges of research during a pandemic, you turned what could have been the loneliest time into something I look back on fondly.

Robert, you took me under your wing and showed me how this whole academic *thing* actually works. You didn't just teach me about research – you taught me about being part of a community. Every conversation we had helped me understand not just what I was doing, but why it mattered. I'm still learning from you.

To my uncle Werner, who first introduced me to computers and has been encouraging my curiosity ever since – this journey started with your patient explanations and enthusiasm, and I never forgot that and will always appreciate it.

To my friends Yannik, Sam, Urs, and Alex – you guys are family. Our regular gaming nights, watching football together, just hanging out and talking about anything except research – you kept me sane. When people ask how I survived a PhD, it's because I had friends who reminded me there's a world outside research, and that world is pretty great when you're in it.

To my parents and my family – you always believed in me and always pushed me to do what I enjoy. Even when I called home stressed about work that wasn't working or papers that wouldn't write themselves, you never doubted I'd figure it out. *Mom*, especially during those times when being a grown-up felt overwhelming, you were always there with exactly the right words and all the support anyone could ever wish for.

And Kate – you've been with me through absolutely everything. The late nights when nothing would work, the days when I was stressed and sad, or the random Tuesday afternoons when I wasn't sure if I was still doing the right thing. You celebrated every small win with me and helped me through every setback. This thesis exists because you believed in it – and in me – even when I didn't. *I love you.*

Statement of Originality

I hereby declare that this dissertation is my own original work, supervised by Prof. Cas Cremers, except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree. Detailed statements for each part of the thesis are included before the respective parts.

Contents

1 Introduction

In our modern digital world, computers and networked systems have become deeply embedded in our daily lives, controlling everything from financial transactions to personal communication. While these technologies provide convenience, they also introduce significant risks – sensitive data, if improperly secured, can be exposed, leading to severe privacy violations and security breaches. The protection of personal and financial information has become a fundamental human right, necessitating robust security measures to protect bank accounts, private messages, and confidential data. However, as we continue to integrate digital technologies into every aspect of life, the challenge of ensuring security and privacy only grows more complex.

Since the 1980s, cryptography and security protocols have evolved as essential tools for protecting digital communication. Starting with Goldwasser and Micali [118], classical cryptography laid the foundation for security proofs, ensuring that cryptographic mechanisms could be mathematically verified for their correctness. However, despite advancements in cryptographic techniques, security proofs remain predominantly manual, requiring significant effort and expertise. The process is painstakingly slow, and errors in these proofs can go unnoticed for years. While machine-checked proofs existed, they were still not widely adopted, largely due to the steep learning curve and technical barriers in formal verification tools in that time.

To address these challenges, an alternative approach to security verification emerged as early as the late 1980s. The seminal work of Dolev and Yao [103], followed by significant contributions in the 1990s (e.g., [142, 164]), introduced the symbolic model of cryptography. This model differs from the traditional computational model by abstracting cryptographic primitives and focusing on high-level logical reasoning. Symbolic verification allows for efficient reasoning about security protocols without delving into the mathematical complexities of underlying cryptographic algorithms. Throughout this thesis, we refer to the traditional approach as the computational model and the alternative as the symbolic model. The rise of symbolic verification provided a new way to analyze security properties at a higher level, reducing the effort required for manual proofs and offering a structured approach to protocol verification.

The early 2000s saw the development of automated verification tools within the symbolic model, leading to increased adoption in protocol verification. These developments have been driven by the emergence of powerful formal analysis tools such as Tamarin [170], ProVerif [43], DeepSec [62], and SAPIC [150] which are part of today's state-of-the-art. Advances in formal verification now allow us to analyze complex security protocols like TLS1.3 [80], WPA2 [83], and EMV [23], which have high impact on modern digital security.

Despite the successes of the recent decades, significant limitations remain within the symbolic model and its automated verification capabilities. Research by Jackson, Cremers, Cohn-Gordon, and Sasse [134] has shown that current abstractions of cryptographic primitives may not accurately reflect real-world behaviors, raising concerns about the reliability of symbolic proofs. Additionally, automated tools struggle with protocols that involve large, branching state machines, making certain security properties difficult to express and verify. This results in a gap between theoretical security guarantees and practical implementations, where subtle design choices can introduce vulnerabilities that are not accounted for in abstract models.

Another major challenge lies in the human factor – protocol modeling remains a manual process, prone to errors due to incorrect abstractions or ambiguities in the use of verification tools. The process requires significant expertise and domain knowledge, making it inaccessible to non-experts. Additionally, while automated tools can analyze simple protocols effectively, they often struggle with large-scale, industry-grade protocols.

In this thesis, we explore some of these limitations by **(I)** proposing methods to improve the symbolic verification landscape by increasing the expressiveness of the symbolic model, and **(II)** investigating the

limitations of large-scale analysis and its future perspectives.

Part I: Advancing Symbolic Models The first part of this thesis focuses on extending the symbolic model to better represent cryptographic primitives that have remained unchanged for decades despite advancements in cryptographic research. By improving these representations, we aim to bridge the gap between theoretical models and real-world implementations, ensuring that security proofs remain relevant as cryptographic techniques evolve.

Cryptographic Hash Function: Security proofs in both symbolic and computational models often treat hash functions as perfectly random one-way functions. However, real-world instantiations exhibit weaknesses that can be exploited. We introduce a hierarchy of hash function properties to address these weaknesses, integrating them into the symbolic model. Our automated methodology extends the Tamarin prover, enabling the rediscovery of known attacks and the identification of new vulnerabilities. This refined approach allows for a more accurate representation of hash functions in symbolic analysis, preventing misleading security claims that do not hold in practice.

Authenticated Encryption with Associated Data (AEAD): The landscape of AEAD security is chaotic, with inconsistent definitions and unclear security guarantees. We start by compiling known AEAD properties and attacks on AEAD instantiations from the literature, and group attacks to the specific violated property. Building on the work of Zhao [223], who has proven relations between all the collected properties, we derive major categories of AEAD weaknesses and develop symbolic models to capture them. Additionally, we develop an automated methodology to detect both known attacks and novel subtleties that can be used to exploit real-world protocols.

Key Encapsulation Mechanism (KEM): The transition to post-quantum cryptography has led to the adoption of KEMs as one potential replacement for classical key exchange mechanisms. However, the literature lacks a comprehensive formalization of certain KEM properties, which are often assumed without explicit definitions. We introduce a new class of binding properties for KEMs and propose symbolic models that encompass both existing and our newly defined properties. From the symbolic models, we develop a methodology using the Tamarin prover to automatically analyze security properties that rely on KEMs. We tested our methodology on real-world examples, helping us to automatically discern which binding properties a KEM scheme needs to make sure the targeted protocol stays secure.

Part II: Exploring the Limits The second part of this thesis investigates the limitations of current symbolic analysis methods when applied to large-scale, complex security protocols.

We conduct an in-depth analysis of the *Security Protocol and Data Model (SPDM)* [96], developed by the Distributed Management Task Force (DMTF) [93] with backing from major industry players. SPDM aims to ensure cryptographic verification of platform component identities, firmware integrity, and secure communications over platform interfaces. However, the protocol's extensive specification lacks detailed state machine diagrams and information flow descriptions, requiring significant manual effort for modeling.

After manually reconstructing SPDM's state machines, we formally model the protocol in Tamarin, creating one of the most complex verification models to date. Our analysis uncovered a critical attack on SPDM's goal of mutual authentication, which we successfully demonstrated in the reference implementation. This discovery led to the registration of a Common Vulnerabilities and Exposures (CVE) entry with a critical severity score (CVSS 9.0) [84]. We subsequently proposed a fix, which was adopted in an updated version of the SPDM specification [94].

This case study illustrates the necessity of improving automated verification techniques to better handle large-scale security protocols and prevent vulnerabilities before deployment. Aside from the numerous design flaws of SPDM we could uncover, this case study highlights the importance of domain separation, a concept supported by the research community, in managing large-scale security protocols. Our findings confirm that tools like Tamarin struggle without domain separation, especially when looking at large-scale protocols like SPDM. We discuss potential improvements to symbolic analysis methods, SPDM, and other complex protocols. Our observations underscore the need for continuous improvements in both the protocols and the tools used to verify them.

1.1 Publications

This thesis is based on the following five publications. Joint work is presented with the permission of all co-authors and my contributions to the publications are listed explicitly in the beginning of each respective part of the thesis.

[61] *Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. "Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses." In:* USENIX Security Symposium. *2023*
Distinguished Paper Award 🏆

[69] *Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. "Automated Analysis of Protocols that use Authenticated Encryption: Analysing the Impact of the Subtle Differences between AEADs on Protocol Security." In:* USENIX Security Symposium. *2023*
Distinguished Paper Award 🏆

[72] *Cas Cremers, Alexander Dax, and Niklas Medinger. "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols." In:* Conference on Computer and Communications Security (CCS). *2024*
Distinguished Artifact Award 🏆

[75] *Cas Cremers, Alexander Dax, and Aurora Naska.* Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 including Cross-Protocol Attacks. *Under Submission. 2025*

[76] *Cas Cremers, Alexander Dax, and Aurora Naska. "Formal analysis of SPDM: Security protocol and data model version 1.2." In:* USENIX Security Symposium. *2023*

1.2 Outline

Chapter 1 motivates the work of this thesis and provides a clear outline for the remainder of thesis.

Chapter 2 introduces the necessary background on the analysis of security protocols and gives a clear description of the symbolic model of cryptography. The chapter puts a special focus on the Tamarin prover – a state-of-the-art security verification tool which is used as part of the research conducted throughout this thesis.

Part I: Advancing Symbolic Models

Chapter 3 motivates the general idea underlying the research of creating better representations of cryptographic primitives, providing an introduction to the following chapters.

Chapter 4 addresses the gap between the theoretical assumptions of cryptographic hash functions and their practical implementations. By setting up a hierarchy of hash function properties, we integrate these more accurate models into our symbolic analysis. With the help of these models, we are able to automatically uncover known vulnerabilities from the literature and discover new variants. This chapter is based on [61].

Chapter 5 systematizes the fragmented landscape of AEAD security by compiling and categorizing the known properties and corresponding attacks on AEAD schemes. We develop symbolic models that accurately capture the collected AEAD weaknesses and introduce an automated methodology to detect both known and novel exploits. This chapter is based on [69].

Chapter 6 explores Key Encapsulation Mechanisms (KEMs), a cryptographic primitive gaining popularity in the transition to post-quantum cryptography. Due to the lack of formalized properties of KEMs in the existing literature, we introduce a new class of binding properties for KEMs. We create new symbolic models of KEMs and their properties and, using the Tamarin prover, we develop a methodology to automatically analyze real-world security protocols that use KEMs. This chapter is based on [72].

Chapter 7 presents the limitations of our new approaches from chapters 4 to 6 and discusses related work.

Part II: Exploring the Limits

Chapter 8 introduces the second goal of this thesis: to explore the limits of symbolic analysis when it comes to size and complexity. We introduce DMTF's Security Protocol and Data Model (SPDM), one of the most complex security protocols currently under specification.

Chapter 9 presents the first formal analysis of SPDM, starting with details on the protocol itself. After manually constructing state machines from SPDM's specification we provide details on our modeling approaches in Tamarin, focusing on our modeling choices. In our final model of SPDM, Tamarin automatically finds a critical authentication attack based on the complexity of the protocol. This chapter is based on both [76] and [75].

Chapter 10 discusses the limitations of our analysis approach and presents learned lessons and insights for future analysis of complex protocols.

Conclusion and Future Work

Chapter 11 concludes this thesis by summarizing our contributions before discussing potential future work.

Background 2

Contents

Statement of Originality

The following background content is drawn directly from my previous work [61, 69, 72, 75, 76] and similar theses (Jacomme [135] and Meier [169]) and follows their presentation closely.

2.1 Security Protocols

A protocol is a set of rules that define how two or more entities communicate and exchange information. It ensures that messages are structured correctly and that both parties understand how to send and receive data. Protocols are essential in various fields, including everyday interactions, business and especially technology.

For example, a simple greeting protocol could work as follows: One party sends a message starting with "Hello, I am [Name 1]. What is your name?", the other responds with "Hello [Name 1], my name is [Name 2].", and the conversation ends with the first party replying "Nice to meet you [Name 2]." This structured exchange ensures that both sides follow the same communication pattern. If a message is not understood (dropped message), the parties can either stop talking (aborting the protocol) or repeat what they said previously (retrying/resending messages). While human communication allows flexibility, digital communication requires strict adherence to predefined rules—without structured protocols, reliable data exchange over networks would not be possible.

In the digital world, protocols such as UDP, TCP, and HTTP allow for communication between systems over the internet, ensuring that devices can exchange information in an orderly manner. However, beyond enabling communication, some protocols also have security objectives – they help establish shared secrets, exchange confidential data, protect private information, and maintain anonymity. To achieve security goals, protocols must be designed to meet specific security properties. These properties define what a protocol should guarantee under potential attacks.

2.1.1 Security Properties

Security requirements vary depending on the use case. Some situations demand strong authentication, ensuring that only the right individuals gain access. Others prioritize confidentiality, preventing unauthorized entities from intercepting sensitive data. In more complex scenarios, privacy measures are also essential to protect user identities and interactions.

Consider a remote employee accessing their company's internal services. The system must first verify that this is indeed an authorized employee before granting access. This process, known as *authentication*, ensures that no outsider can enter the system. At the same time, the exchanged data must remain *confidential*, meaning that even if a third party intercepts the communication, they cannot extract any meaningful information. These two aspects – authentication and confidentiality – are fundamental security properties to secure communications.

Beyond access control and confidentiality, privacy adds another layer of complexity. In some cases, organizations may want to prevent external observers from learning who is logging into the system or which resources are being accessed. Privacy mechanisms ensure that even if someone monitors network traffic, they cannot determine whether a specific employee is using the internal services.

Security properties can generally be categorized into two types: reachability and indistinguishability. Reachability properties ensure that certain undesirable events never occur, such as unauthorized users accessing sensitive systems. Indistinguishability properties, on the other hand, focus on making different scenarios appear identical to outsiders, ensuring, for instance, anonymity.

2.1.2 Cryptographic Primitives

Security protocols are designed to guarantee various security properties, such as authentication and data confidentiality. However, constructing these protocols requires fundamental tools – this is where cryptographic primitives come into play. These primitives serve as the essential building blocks of cryptographic systems, enabling secure communication and data protection. They are functions that allow us to encrypt, authenticate, or otherwise process messages securely.
To illustrate, consider the following examples of cryptographic primitives:

Symmetric Encryption This primitive usually consists of three functions, KeyGen, senc and sdec. KeyGen is used to randomly draw a secret key k that is used to encrypt data. Given a message m and

the secret key k, encryption produces a ciphertext $c = \mathsf{senc}(m, k)$, which conceals any information about m. Only someone possessing k can decrypt it, ensuring that the decryption of the ciphertext $\mathsf{sdec}(\mathsf{senc}(m, k), k)$ returns the original message m.

Authenticated Encryption with Associated Data (AEAD) AEAD extends symmetric encryption by providing both confidentiality and integrity. It ensures that, in addition to encrypting a message, any unauthorized modification of the ciphertext will be detected. AEAD schemes also support associated data, meaning that certain parts of the message (such as public headers) can be authenticated but not encrypted.

Hash Functions These are functions that take an input m and return a fixed-size output, called the hash. A hash function is designed to be non-invertible, meaning it is computationally infeasible to recover m from its hash value. Hash functions are commonly used for data integrity checks, password storage, or as a part of more complex cryptographic primitives.

Digital Signatures Digital signatures are designed to provide authentication. A signature scheme consists of three algorithms: KeyGen, sign and verify. KeyGen is used to randomly draw a secret private key sk to sign data, and a corresponding public key pk used to verify signatures constructed using sk. Using a secret key sk, one can generate a signature $\mathsf{sign}(m, sk)$. Anyone with access to the corresponding public key can verify that a given signature x is valid using $\mathsf{verify}(x, m, pk)$. This ensures that messages cannot be forged and that the sender cannot later deny signing them. Note that, for digital signatures to be applicable, a mechanism has to be set in place that allows to distribute the public keys of identities.

Asymmetric Encryption Asymmetric encryption, also known as public-key encryption, uses a pair of keys as well: a public key pk for encryption and a private key sk for decryption. This primitive consists of three functions: KeyGen, Enc, and Dec. The KeyGen algorithm generates a key pair (pk, sk), where pk is publicly shared while sk remains secret. Given a message m, encryption produces a ciphertext $c = \mathsf{Enc}(m, pk)$ using the public key. Only the corresponding private key sk can decrypt the ciphertext, ensuring that $\mathsf{Dec}(\mathsf{Enc}(m, pk), sk)$ recovers the original message m.

Key Encapsulation Mechanisms (KEM) Similarly to asymmetric encryption, KEMs also rely on a pair of keys (pk, sk). The primitive also consists of three functions: KeyGen, Encaps, and Decaps. KeyGen is analogous to key generation of asymmetric encryption. Unlike traditional encryption, however, KEMs are used to securely establish shared keys between parties that share public keys. A sender generates a random key k and a ciphertext c using a recipient's public key pk by running the randomized Encaps algorithm: $k, c = \mathsf{Encaps}(pk)$. The recipient can then use their private key sk to recover the shared key ($k = \mathsf{Decaps}(c, sk)$), which can be used for further encrypted communication, using e.g., symmetric encryption or AEADs.

These cryptographic primitives – and many more – each serve a specific purpose, but they are often used together to provide even stronger security guarantees. By combining them, security protocols can ensure data confidentiality, restrict access to authorized users, and detect any tampering. They form the backbone of modern digital security, enabling secure communication, online transactions, and data protection.

2.1.3 Formal Proofs

When designing security protocols to achieve specific security properties, it is crucial to ensure that the protocol actually meets its objectives. To address this challenge, the concept of provable security was introduced in the 1980s, providing a foundation for defining security protocols and its desired properties precisely and proving them mathematically. This approach allows protocol designers to demonstrate that their constructions are secure based on well-defined models and assumptions.
Two key paradigms have emerged to prove the security of protocols:

Computational Model Introduced by Goldwasser and Micali in their seminal paper [118], this model focuses on security against attackers with limited computational power. A construction is considered secure if a computationally bounded attacker can only break the security with a negligible probability.

Security proofs in this model often rely on the assumption that solving a specific problem, such as factoring large integers or computing discrete logarithms, is computationally difficult.

Symbolic Model In this model, cryptographic primitives are typically assumed to be perfect, meaning they cannot be broken or manipulated beyond their intended purpose. It is based on ideas of Dolev and Yao in [103] where an attacker is assumed to have full control over the network, including the ability to intercept, modify, and inject messages. However, the attacker's capabilities are limited to a fixed set of symbolic operations on the messages, such as signing or decryption using known keys.

While the computational model provides a more realistic view of security by accounting for computational limits, the symbolic model offers a more abstract and simplified analysis of security properties, allowing for analysis of larger objects. Both paradigms are valuable and can be used in complementary ways to gain confidence in the security of a protocol, depending on the goals and complexity of the system being analyzed.

2.2 The Computational Model of Cryptography

The computational model of cryptography provides a formal framework for reasoning about the security of cryptographic constructions using concepts from computational complexity theory. In this model, attackers are modeled as probabilistic polynomial-time (PPT) Turing machines [118], – meaning they have bounded computational resources and can only execute a feasible number of operations relative to input size. Messages, keys, and ciphertexts are represented as bitstrings. Security is framed in terms of an attacker's ability to distinguish different cryptographic outputs with a high probability or perform an attack within realistic computational limits.

Security in the classical computational model is typically proven using *reduction arguments*. These proofs show that if an attacker were able to break a security protocol or cryptographic construction, then they could also solve a well-established computationally hard problem, such as factoring large integers [189] or computing discrete logarithms [92]. Since these problems are considered difficult to solve, the security protocol or cryptographic construction is assumed to be secure. This approach ensures that cryptographic constructions are secure as long as the underlying hardness assumptions hold.

A key example of a security definition in this model is *indistinguishability under chosen plaintext attack (IND-CPA)*, which is a standard for asymmetric encryption [27]. Here, an attacker is given access to an encryption oracle[1] and must attempt to distinguish between encryptions of two messages of their choice. If they cannot do so with a non-negligible probability higher than random guessing, the scheme is considered secure. Similarly, in digital signatures, security may be defined in terms of *existential unforgeability under chosen message attack (EUF-CMA)*, meaning no attacker should be able to forge a valid signature without access to the secret signing key [119].

To make security proofs more manageable, multiple techniques have been proposed over the years, e.g., the *game-playing technique* introduced by Shoup [204]. This method structures security proofs as a sequence of small probabilistic games, each differing slightly from the previous one. The idea is to show that an attacker's success probability changes negligibly at each step, leading to an overall proof that the system is secure. Variants of this technique have enabled researchers to explore automation and tool-assisted verification, resulting in tools like CryptoVerif [42]. Building on this research, other tools enabling proofs in the computational model were developed in recent years, for instance, Squirrel [15] or EasyCrypt [18]. While being promising approaches, these tools do not yet scale well enough to tackle the complexity of real-world secure communication protocols.

2.3 The Symbolic Model of Cryptography

The symbolic model uses function symbols to denote algorithms, and capture their properties through equations. For instance, an encryption is modeled by two binary function symbols senc and sdec, with the

[1] An oracle is an API that the attacker can access. It performs a specific task like encryption under a fixed key and only leaks the result to the attacker.

equation:

$$\mathsf{sdec}(\mathsf{senc}(m,k),k) = m$$

Note that the randomness or nonce usually included into symmetric encryption schemes is not explicit in this classical modeling. And crucially, in the symbolic model, only the equations that are explicitly specified imply equalities. This results in the so-called perfect cryptography assumption: in the previous example, the encryption is perfect, in the sense that given $\mathsf{senc}(k,m)$ and not k, the attacker learns absolutely nothing about m or k as it cannot apply the decryption equation. The attacker cannot change the content of the message, and no collisions will exist.

While the previous assumption may seem too restrictive, it allows for highly automated tools which are one of the strengths of the symbolic model. These tools were already successfully used to automatically find attacks on protocols like WPA2 [83] or Bluetooth [217] and aid standardization processes to avoid design-level flaws [80, 87, 180].

In the following we will first present the used term algebra before taking a deeper glance at one of the most prominent automation tools in symbolic analysis – the Tamarin prover.

2.3.1 Symbolic Model: Term Algebra

Formally, we assume a set of *operators* with their arities as signature Σ and a countably infinite set of variables $\mathcal{V}$. Operators model computations over messages such as symmetric encryption (senc). We similarly treat *atoms* (usually called *names* in symbolic models), i.e. atomic data such as nonces or keys, with a countable set $\mathcal{A}$. The set of *terms* given by the closure of using operators from the signature Σ containing variables in $\mathcal{V}$ and atoms in $\mathcal{A}$ is denoted $T := T_\Sigma(\mathcal{V},\mathcal{A})$. A term t is *ground* if it contains no variables, and we write $T_\Sigma(\mathcal{A})$ for the set of all ground terms, or simply T_Σ. We also call ground terms *messages*. A *substitution* σ is a function from variables to messages. We homomorphically lift substitutions to terms.

Algebraic properties over operators, such as decrypting a ciphertext with the right key yields the plaintext, are expressed through an equational theory. Given a signature Σ, an *equation* is an unordered pair of terms s and t, written $s = t$, for $s,t \in T_\Sigma(\mathcal{V})$. To a set of equations E, we associate an *equational theory* that is the smallest congruence relation over terms $=_E$ that contains E and is closed under substitution of terms for variables and atoms. Two messages s and t are equal modulo E if and only if $s =_E t$.

Example 1. *For a basic model of symmetric encryption, let* Σ *contain the operators* $\mathsf{senc}(\cdot,\cdot)$ *and* $\mathsf{sdec}(\cdot,\cdot)$ *together with the equation* $\mathsf{sdec}(\mathsf{senc}(x,y),y) = x$*. This does model decryption of a ciphertext that was encrypted under the same key* y *as used for the decryption operation.*

Example 2. *We assume a concatenation operator* $\cdot\|\cdot \in \Sigma$ *equipped with an equation* $(x\|y)\|z = x\|(y\|z)$*, that is the concatenation is associative. We shall use* $\|$ *to concatenate (suffix as second argument) different messages prior to hashing.*

To simplify presentation, we interpret all relations to be closed under the equational theory. For example, we simply write $t \in T_\Sigma(\mathcal{V})$ if $\exists t' :\ t =_E t' \wedge t' \in T_\Sigma(\mathcal{V})$ and assume that for any binary relation $\sim$, $t_1 \sim t_2$ implies $t_1' \sim t_2'$ whenever $t_1' =_E t_1$ and $t_2' =_E t_2$.

2.3.2 Symbolic Model: Tools

Our methodologies we introduce throughout this thesis are generalized enough to not be bound to a specific tool. The tool of choice needs to support custom equational theories and explicit means to express attacker knowledge. These are criteria fulfilled by various state-of-the-art symbolic tools like [45, 108, 150]. We choose the Tamarin prover [170], as it offers a straightforward way to add custom equational theories and oracle-like processes.

The Tamarin Prover

The Tamarin Prover [170] is a state-of-the-art protocol verification tool that is widely used in academia and industry to analyze complex real-world protocols. In the following, we provide a brief overview of the tool, and refer the reader to the official documentation for more information [20].

To verify a protocol in Tamarin, the user needs to provide three inputs: (i) the *protocol model*, (ii) the *security property*, and (iii) the *attacker model* against which the property should hold. Given these inputs, the tool will either verify the property and output a proof, or find a counter example and provide the attack steps. However, Tamarin might also not terminate or run out of space and memory. In this case, the user can manually explore the proof search in the user interface and provide additional input to help its reasoning, e.g., in the form of invariants.

Protocol Models Tamarin takes a protocol description in a custom modeling language and security properties specified in a fragment of first-order logic as input. The modeling language allows a user to specify the protocol rules and the adversary's capabilities via *multiset rewriting rules*. These rules induce a labeled transition system. Tamarin then tries to verify whether the given security properties hold for all traces of the transition system.

A multiset of *facts* serves as the state of the labeled transition system. The rewriting rules manipulate this state by adding and removing facts. Facts are special user-defined symbols that contain terms and represent the state of the protocol. The state of the adversary (i.e., their knowledge) is modeled by a distinct set of facts. An example of a fact would be $\mathsf{Alice}(\mathsf{pk}, \mathsf{sk})$, which models Alice who is in possession of some key pair $(\mathsf{pk}, \mathsf{sk})$.

Rules are constructed by a left-hand side (LHS or premise), an action, and a right-hand side (RHS or conclusion). These correspond to the input and conditions to trigger this transition in the protocol, the action label or event marking the transition, and the output state of the protocol, respectively. In the example in Figure 2.1, the Tamarin rule shows the creation of the root authority in a PKI.

$$
\begin{array}{l}
[\ \mathsf{Fr}(\mathit{ltk})\ ] \\
\text{—}[\ \mathtt{CreateRootAuth}(\mathit{ltk})\ ]\!\rightarrow \\
[\ \mathsf{!RootAuth}(\mathit{ltk}), \mathsf{Out}(\mathsf{pk}(\mathit{ltk}))\ ]
\end{array}
$$

Figure 2.1: The LHS shows the generation of a new unique long-term key *ltk* unknown by the attacker, expressed by the built-in Fr fact. In the RHS, the rule outputs the root authority with their private key, modeled by the !RootAuth fact and reveals to the network the public key. The transition is labeled by the action fact CreateRootAuth, which will be used in properties referencing to this transition.

Facts annotated with a *!* are called *persistent* and are not removed from the multiset when a rule is executed. A rule can be executed in a given state if the premises are a subset of the current state. To execute the rule, Tamarin removes the premises from the state and adds the conclusions to it.

The execution of the protocol starts with the empty multiset as state and uses the rules to transition from one state to another. Rules can be used any number of times. The resulting sequence of actions is called a *trace*.

Security Properties To express the security properties Tamarin uses first-order logic notation, where the user can quantify over messages and time-points. In the example below, the property defines secrecy of the long-term key of the root authority that was created by the previous transition.

$$
\begin{aligned}
\forall\ \mathit{ltk}\ \sharp i\ .\mathsf{CreateRootAuth}(\ \mathit{ltk}\ )\ @\ \sharp i \\
\Rightarrow\ \neg(\ \exists \sharp j\ .\mathsf{K}(\ \mathit{ltk}\ )\ @\ \sharp j)
\end{aligned}
$$

The property states that for all traces that created root authorities with long-term key *ltk* at time-point $\sharp i$, the attacker does not know the authority's private key, where the K fact models the attacker knowledge.

We also make use of Tamarin's *restrictions*. Restrictions are formulas like security properties, but they are used to constrain the execution of the protocol: if a trace violates any restriction, Tamarin does not consider it.

Attacker Model The tool has a built-in network attacker, that can read, drop, and inject messages in the protocol. Additionally, the user has the flexibility to specify different attacker models. This can be achieved, for example, by including rewriting rules that simulate the leakage of a party's secret keys, effectively modeling a compromise of that party.

I

Part I: Advancing Symbolic Models

"But nothing's ever perfect, haven't you realized that yet? Earth turns on a tilted axis just doing the best it can."

– Hohenheim, *FMA 2003*

Statement of Originality

This part of the thesis consists of five chapters.

Chapter 4 is based on:

Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. "Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses." In: USENIX Security Symposium. *2023* [61]

This paper is joined work with Vincent Cheval, Cas Cremers, Lucca Hirschi, Charlie Jacomme, and Steve Kremer which won one of the *distinguished paper* awards 🏆 at Usenix 2023. I was the lead author in the project and the substantial contributions in this chapter are my own. My co-authors principally contributed to the initial conception of the work and the final write-up of the paper. This chapter does not contain the parts on the extension of the tool ProVerif, as this part of the research project was mainly conducted by my co-authors.

Chapter 5 is based on:

Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. "Automated Analysis of Protocols that use Authenticated Encryption: Analysing the Impact of the Subtle Differences between AEADs on Protocol Security." In: USENIX Security Symposium. *2023* [69]

This paper is joined work with Cas Cremers, Charlie Jacomme, and Mang Zhao which won one of the *distinguished paper* awards 🏆 at Usenix 2023. I was the lead author in the project. While the substantial contributions within Chapter 5 are my own, Section 5.3.1 presents a condensed version of my co-authors work on generalizing AEAD collision resistance and its relations. The full version and their comprehensive computational proofs can be found in [69, 223].

Chapter 6 is based on:

Cas Cremers, Alexander Dax, and Niklas Medinger. "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols." In: Conference on Computer and Communications Security (CCS). *2024* [72]

This paper is joined work with Cas Cremers and Niklas Medinger which won one of the *distinguished artifact* awards 🏆 at CCS 2024. While I contributed to all parts of the project, I lead the research in:

1. the creation of new security notions (Section 6.3),
2. the initial draft and development of the current symbolic models (Section 6.4), and
3. the creation of the methodology used for the conduction of case studies (Section 6.5).

Sections 6.1 and 6.2 are based on [72] and follow their presentation closely.

Chapters 3 and 7 are all loosely based on my previous work [61, 69, 72].

3 Advanced Cryptographic Primitives

Cryptographic primitives are the basic building blocks used to construct security protocols with various objectives. Primitives such as *symmetric encryption* and *asymmetric encryption* enable protocol designers to conceal data on the network, thereby providing confidentiality when applied correctly. Other primitives, like *message authentication codes* or *digital signatures*, can be used to authenticate parties or ensure data integrity. Without these established and intensely studied primitives, there would be no standardized method for constructing security protocols, potentially resulting in systems that are less reliable and more vulnerable to attacks.

As described in Chapter 2, in the symbolic model we traditionally encode cryptographic primitives using equational theories. For example, consider digital signatures – a primitive often used to provide identity authentication and data integrity. We encode this symbolically by defining three operators in Σ: sign, verify, and true. The binary operator sign models the action of signing a message m with a private key sk, written as $\mathsf{sign}(m, sk)$. To enable another party to verify the signature's validity, we include the equation $\mathsf{verify}(\mathsf{sign}(m, sk), m, pk) = \mathsf{true}$ in the system, where pk is the public key corresponding to sk and verify represents the signature verification process, which succeeds (i.e., evaluates to true) only when the keys and the signed message match.

Some established cryptographic primitives are traditionally modeled even more simply. For example, *cryptographic hash functions* take an input m and return a fixed-size output called the hash. A hash function is designed to make it computationally infeasible to recover m from its hash value. Hash functions are commonly used for data integrity checks and also serve as building blocks for other cryptographic primitives. Symbolically, hash functions are modeled as a single unary operator $H \in \Sigma$ with no accompanying equational theory – providing no way to retrieve m from $H(m)$. As with all other primitives modeled in the symbolic model, these representations are strong over-approximations of what their real-world instantiations would look like.

This gap between symbolic representations and real-world instantiations poses a significant problem. Although symbolic models of protocols may be proven secure, an implementation that closely follows the protocol design might still be vulnerable to attacks. For instance, these vulnerabilities can arise from weaknesses in hash function instantiations [37, 207, 208] that are not accounted for in the symbolic models, as they assume that hash functions are perfect.

Our main goal in this part of the thesis is to narrow the gap between current symbolic representations of cryptographic primitives and their concrete instantiations and computational definitions. Building on the work of [82, 134], who improved the symbolic representations of digital signatures and Diffie-Hellman groups, we will address multiple primitives in the following chapters. In particular, we will examine three common cryptographic primitives: cryptographic hash functions, authenticated encryption schemes with associated data (AEADs), and key encapsulation mechanisms (KEMs). We will not only develop improved symbolic models for each of these, but also test their applicability through numerous case studies.

4 Cryptographic Hash Functions

Contents

4.1 Introduction

Cryptographic hash functions are a fundamental and highly efficient building block in nearly all cryptographic protocols. They are traditionally required to meet several security properties, such as collision resistance and first/second preimage resistance. Ideally, they are "perfect" and do not suffer from phenomena like *length-extension attacks*. Modern hash functions like SHA3 (Keccak) are believed to satisfy all these properties and behave like a perfect hash function.

In many modern protocol security analyses, both in the computational and symbolic setting, hash functions are assumed to be "perfect" in the following sense: the modeled hash function meets all desired cryptographic properties and every input/output combination is completely independent of all others. Such a hash function corresponds to the *Random Oracle Model (ROM)* often used in cryptographic proofs.

In practice though, real hash functions are unfortunately far from perfect. In Table 4.1 we show some widely deployed hash functions and their currently known imperfections. Several of these hash functions, such as SHA1, are considered to be weak or broken with respect to collision or preimage resistance; and nearly all widely deployed hash functions allow so-called *length extension attacks*, which can enable someone to compute $hash(x\|y)$ even if they do not know x – which is not possible with a perfect hash function.

There are several reasons for the gap between reality and the perfect hash function. First, the security of hash functions is often based on a heuristic argument, since we cannot reduce them to a known hard problem. History has shown that many hash functions that initially seemed secure turned out to be broken some years later [13, 216]. Second, it was long believed that potential hash weaknesses would not weaken protocols that use (second) preimage resistant [37, 207] hash functions. Third, even if a hash function satisfies all standard requirements for cryptographic hash functions (resistance to collisions and preimages), it may still not be perfect. For example, many popular hash function designs follow the *Merkle-Damgård (MD)* construction, which in its default setup, allows for length extension attacks. We thus have to face the reality: protocols use hash function that are already imperfect, and history has shown that over time, hash functions that appear secure now will become easier to attack in the future.

This raises the natural question: how can we check if a protocol that uses a hash function with a particular weakness, meets its security guarantees? History so far has shown such attacks can be rare but very subtle, e.g. [37, 207, 208], and therefore extremely hard to detect manually. From a cryptographic perspective, the answer would be: provide a computational proof of the security of the entire protocol,

Hash function	Year	Examples of currently deployed applications	Collision resistance	(2nd) Preimage resistance	No Length-extension
MD4	1990	NTLM key derivation for Microsoft Windows	✗ 2^1	⊗* 2^{95}	✗
MD5	1992	File checksums (md5sum)	✗ 2^{18}	⊗* 2^{123}	✗
SHA1	1995	Europay Mastercard Visa (EMV), File checksums, Telegram	✗ 2^{61}	✓ 2^{160}	✗
RIPEMD-160	1996	Bitcoin	⊗** 2^{80}	✓ 2^{160}	✗
SHA2-256	2001	Bitcoin, TLS, SSL, SSH, S/MIME, IPSec, DNSSEC, Linux/Unix password hashing, Telegram	✓ 2^{128}	✓ 2^{256}	✗
SHA2-512	2001	TLS, SSL, SSH, S/MIME, IPSec, DNSSEC, Linux/Unix password hashing	✓ 2^{256}	✓ 2^{512}	✗
SHA3-256	2012	Ethereum	✓ 2^{128}	✓ 2^{256}	✓

✓= currently still secure ⊗=weak, but no full attack yet ✗= known attack

* = Theoretical attacks on (second) preimage resistance were found[224][195], but they are still not feasible.

** = No known attack but the small bit size makes collision attack doable in practice, but not necessarily feasible.

Table 4.1: Examples of widely used hash functions that are currently deployed in security protocols and do not offer perfect (random-oracle like) guarantees. The numbers indicate the complexity of the currently best known attack on the property [159, 195, 196, 218, 224]. For the hash functions currently deemed secure the best known attacks would be a brute-force approach; e.g. the complexity to break collision resistance on SHA2-256 is 2^{128}. Crucially, this situation is not constant, but expected to get worse: history suggests that the numbers for the best attacks are likely to decrease over time for all hashes, see e.g. [13, 207].

and if this proof seems to rely on properties of the hash function that it does not offer, this may indicate an attack. However, for most protocols, this task ranges from daunting to infeasible; and most existing protocol proofs simply assume that the hash function is perfect, by using the ROM.

In contrast, automated protocol analysis tools have shown to be effective for analyzing real-world protocols [21, 23, 33, 66, 80, 143]. However, they model hash functions as being perfect (traditionally as an operator in a free term algebra). Thus, like computational proofs that use the ROM, such analyses miss any attacks that exploit the use of a non-perfect hash function.

In this chapter, we revisit cryptographic hash function definitions, common weaknesses, and the potential attacker capabilities that arise from them. Based on this, we develop a methodology to systematically discover attacks on protocols that exploit their use of "less-than-perfect" hash functions, and show how this can be implemented in protocol-analysis tools, namely the Tamarin prover. To realize this, we both exploit advanced features of the tool (such as equational theories, event-based modeling, and restrictions) but we in fact also extend it (partial support for associative operators). Our methodology can be used in the design phase to avoid the use of hash functions that are too weak, or to find and fix problems in deployed protocols.

Outline We first provide some background about hash functions, their security properties and how weaknesses of deployed hash functions may break protocols in practice (Section 4.2). Then we present a novel hierarchy of threat models related to hash functions, detailing adversarial capabilities that we are going to use for protocol analysis (Section 4.3) and show how this analysis can be automated in (an improved version of) Tamarin (Section 4.4). Finally, we demonstrate the effectiveness and methodology of our approach on a number of case studies discovering both known and novel vulnerabilities (Section 4.5).

4.2 Background

Many of the problems around hash functions arise from the gap between the properties described in the theory and the property that real-world hash functions satisfy; but in this particular case, there is already sufficient tension in how hash functions are handled in the theory of protocol proofs. We first describe the state of the theory, before returning to the situation in practice. Afterwards we give background on the symbolic model that we will be using in the next sections.

4.2.1 Hash Functions in Theory

The main three desirable properties that a cryptographic hash function should satisfy are well-established: first and second preimage resistance, and collision resistance [172]:

- **Preimage resistance**: given h, it is infeasible to find x such that $h = \mathrm{H}(x)$;
- **Second preimage resistance**: given x, it is infeasible to find $y \neq x$ such that $\mathrm{H}(y) = \mathrm{H}(x)$;
- **Collision resistance**: it is infeasible to find x, y such that $\mathrm{H}(x) = \mathrm{H}(y)$[1].

Collision-resistance implies second preimage resistance, as finding a second preimage effectively results in a collision.

An undesirable property is so-called *length-extension*: Many deployed hash functions are based on the *Merkle-Damgård (MD)* construction: $\mathrm{H}(m\|m') = f(\mathrm{H}(m), m')$ where f is the underlying compression function and $\|$ expresses concatenation of blocks. The origin of this design choice can be traced back to an implicit design goal of many hash functions: it should be possible to compute a hash incrementally, i.e. to compute $\mathrm{H}(m\|m')$ without having to store both m and m' in memory, for example by computing a compact intermediate product based on m to later compute the full result once m' is available. By default, MD constructions satisfy the *length-extension* property: Given $\mathrm{H}(m)$ and m', one can compute $\mathrm{H}(m\|m')$[2]. As we will see below, this property can be problematic in certain protocol contexts because it enables so-called *length-extension attacks*. In theory, this possibility has been known in the academic literature at least as early as 1992 [213].

[1] By the pigeonhole principle, collisions necessarily exist. Hence, collision-resistance is informally defined as the absence of a *known* algorithm to find a collision faster than generic birthday search-based brute-force.

[2] One can design incremental hashes without the length-extension property, e.g. with an internal state that cannot be reconstructed from the hash.

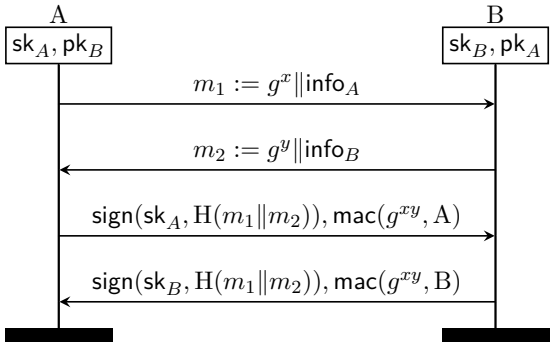


Figure 4.1: The Sigma' protocol [37]

However, when developing proofs of security protocols that use hash functions, it turns out using the three resistance properties as assumptions is very complex and error-prone. Because of this, the *Random Oracle Model (ROM)* was introduced in 1993 by Bellare and Rogaway [28] as a proof methodology to simplify proofs of protocols that use hash functions. We will go into more detail in Section 4.2.2, but intuitively speaking, the ROM models "perfect" hash functions: as functions whose outputs are chosen uniformly at random, independent of the input. We will give a definition of (a symbolic version of the) ROM in Section 4.3.

The ROM satisfies first and second preimage resistance and collision resistance, and does *not* have the length-extension property. The ROM thus has the most desirable cryptographic hash properties, effectively over-approximating the security of real-world hash functions. Therefore, proving that a protocol is secure using the ROM does not guarantee that it is secure when instantiated with a real hash function. However, the vast majority of protocol proofs use the ROM due to its simplicity.

4.2.2 Hash Functions in Practice

In Table 4.1 we review a selection of widely used hash functions, the complexity of the best known collision and preimage attacks against them, and whether length-extension is possible. The conclusion is clear: high-profile cryptographic protocols still use hash functions that suffer from weaknesses that contradict the usual idealized security assumption (ROM) and even (second) preimage and collision resistance. We also see that hash functions get weaker over time as the attacks get more and more efficient (see also the survey [207] and [13]). Moreover, it is likely that hash functions that are deemed secure today will be weakened in the future.

The length extension property of many hash functions can theoretically be used to break a protocol's security, because it enables the following behaviors: (i) collisions can be extended since $\mathrm{H}(x) = \mathrm{H}(y)$ implies $\mathrm{H}(x\|s) = \mathrm{H}(y\|s)$ for any s, and (ii) the adversary can extend the payload under known hash outputs: given $\mathrm{H}(x)$, it can compute $\mathrm{H}(x\|s)$ for any known s. As an example of the latter, if the prefix x contains a shared secret, and the protocol relies on this to authenticate hash values, then the adversary can *forge* hashes by extending any hash values it observes. An early example of a widely deployed protocol that was vulnerable due to such an attack was Flickr in 2009 [106]; we will revisit this attack in Section 4.5.3.

Despite this example, it was thought for a long time that cryptographic protocols are likely to remain secure even though they rely on weakened hash functions as long as the hash functions are (second) preimage resistant [37, 207]. For instance, even if the adversary can compute *some* c, c' such that $\mathrm{H}(c) = \mathrm{H}(c')$, which breaks collision-resistance, it seems unlikely that it can impact honest agents in a protocol who will compute hashes for inputs that are unrelated to c, c'. Unfortunately, this is a false sense of security: it has been shown that cryptographic protocols can be entirely broken when using hash functions that merely suffer from some restricted classes of collisions [207, 208]; see an example in Section 4.2.2. Unfortunately, it is difficult and error-prone to manually assess if a cryptographic protocol can be broken if its hash function is vulnerable to some restricted class of collisions.

Example: Hash Transcript Collisions

Using *hash transcript collisions*, we exemplify how certain collisions can be weaponized against protocols. They have been shown to affect various authentication protocols such as TLS 1.2, SSH, or IKEv2 [37]. As a running example, we use a variant of the sign-and-mac protocol [146]: the Sigma' authentication protocol introduced in [37] and depicted in Figure 4.1. It is essentially a signed *Diffie-Hellman (DH)* protocol with MAC-based key confirmation where additional information info_A and info_B (e.g. for later negotiation) is appended to the exchanged DH shares. $\mathsf{info}_A, \mathsf{info}_B$ are length-varying and therefore prefixed with their lengths. Sigma' aims at guaranteeing *matching conversations*: after a successful execution both parties share the same view of the transcript, even in the presence of an active attacker. As noted in [37], the parties do not directly agree on the transcript but rather on the hash of the transcript. If the hash function were perfect, this would not matter – but it makes a difference for real-world hash functions.

Machine-in-the-Middle (MITM) scenario First, one should note that the protocol is not immediately broken, even if the hash function H is not preimage resistant. Assume a MITM attacker: the attacker can replace messages m_1 and m_2 by messages $m_1' := g^{x'} \| \mathsf{info}_A'$ and $m_2' := g^{y'} \| \mathsf{info}_B'$ of its choice. This results in the message $m_3 = \mathsf{sign}(\mathsf{sk}_A, \mathrm{H}(m_1 \| m_2')), \mathsf{mac}(g^{xy'}, \mathrm{A})$. If the attacker does not know sk_A, it cannot modify m_3. Hence, B will check whether $\mathrm{H}(m_1 \| m_2') = \mathrm{H}(m_1' \| m_2)$. Clearly, (second) preimage attacks do not directly allow such a MITM attack to succeed, as the input for the target hash output $\mathrm{H}(m_1 \| m_2')$ must be of the form $m_1' \| m_2$, where m_2 is fixed and not adversary-chosen. Similarly, the mere existence of collisions, say $c \neq c'$ such that $\mathrm{H}(c) = \mathrm{H}(c')$, cannot be used to break this protocol.

Hash transcript collisions attacks *Chosen-Prefix Collisions (CPC)* [207, 208] are among the least costly collisions to compute and yet can be weaponized against protocols. Given two prefixes, p_1 and p_2, a CPC attack computes two suffixes $s_1 \neq s_2$ such that $\mathrm{H}(p_1 \| s_1) = \mathrm{H}(p_2 \| s_2)$. When additionally $p_1 = p_2$, such a collision is called *Identical-Prefix Collision (IPC)* and is even less costly to compute.

As we shall see, Sigma' is entirely broken as soon as *(i)* the used hash function suffers from CPC attacks, *(ii)* obeys the length-extension property, and *(iii)* the length of m_2 is predictable. Indeed, given m_1 sent by A, the adversary can choose arbitrary x', y' and compute a CPC for prefixes $m_1 \| g^{y'}$ and $g^{x'}$, resulting in suffixes $\mathsf{infoPartial}_B'$ and info_A' such that:

$$\mathrm{H}(m_1 \| g^{y'} \| \mathsf{infoPartial}_B') = \mathrm{H}(g^{x'} \| \mathsf{info}_A'). \tag{\dagger}$$

Moreover, the claimed length field of $\mathsf{infoPartial}_B'$ can be chosen to be $|\mathsf{infoPartial}_B'| + |m_2|$ The MITM adversary then uses $m_1' := g^{x'} \| \mathsf{info}_A'$ and $m_2' = g^{y'} \| \mathsf{info}_B'$ where $\mathsf{info}_B' = \mathsf{infoPartial}_B' \| m_2$. By the length-extension property of H, we obtain by appending m_2 to the above collision (†):

$$\mathrm{H}(m_1 \| m_2') = \mathrm{H}(m_1 \| g^{y'} \| \mathsf{info}_B') = \mathrm{H}(m_1' \| m_2).$$

Therefore, the MITM adversary successfully impersonated A and B and hijacked the session key, i.e. $g^{xy'}$ with A and $g^{x'y}$ with B. To give a sense of the attack cost, finding such a collision costs about 2^{39} for MD5 and $2^{63.4}$ for SHA1 [159, 207]. As we shall see in Section 4.5, other kinds of CPC but no IPC affect Sigma' – findings we formally establish with our automated formal analysis framework (Section 4.4).

4.3 Generalizing Hash Function (In)Security for Systematic Analysis

In this section, we develop a hierarchy of hash function models. We start from the ROM, which represents an ideal hash function, i.e. for which the adversary has the least possible capabilities to manipulate or learn information from it. We then strengthen the adversary's capabilities in various dimensions, corresponding to possible weaknesses of hash functions.

A core observation is that from the different types of possible weaknesses – here framed as adversarial capabilities – some are independent of others, and some are related. For example, length-extension attacks and CPC are independent: there exist hash functions that have one of these two weaknesses, but not the other. In contrast, CPC and IPC are related: if a hash function is vulnerable to CPC attacks, then the attacker can also choose two identical prefixes: thus, any hash function that is vulnerable to CPC is also vulnerable to IPC.

We identify four main independent dimensions of hash function weaknesses, and thus corresponding adversary capabilities: collision-related weaknesses, length-extension style weaknesses, output-control

weaknesses that can model, e.g. backdoored hash functions, and weaknesses that leak information about the inputs from the output.

Random Oracle Model (ROM) We now take a deeper look into hash functions in the symbolic model (see Section 2.3.) Virtually all prior symbolic analyses model hash functions in the ROM, which corresponds to the weakest possible adversarial capabilities. At the technical level, this means the hash function is symbolically modeled as a free operator, i.e. an operator $\mathrm{H}(\cdot) \in \Sigma$ that does not occur in any equation (in E). Since H does not occur in E, $\mathrm{H} : T \to T$ has the same algebraic property as a random oracle: it associates to an input t, an output value $\mathrm{H}(t)$ that has no other algebraic property than being the hash output of t, modeled as a fresh atom n_t. We informally describe this modeling choice using an abstract hash functionality that is interfaced with a user and an adversary. Here, the adversary has no control over the hash function, as shown in Figure 4.2.

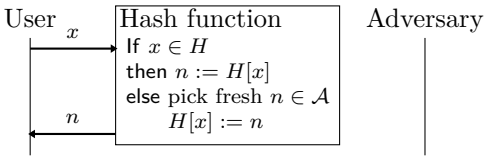


Figure 4.2: Abstract hash function in the ROM. Initially, H is the empty mapping: $H := \emptyset$. Note that the adversary can also act as a user of the hash function, but it cannot *influence* the oracle, unlike in some threat models we will define later.

Modeling dimensions of Hash Weaknesses We identified four main *dimensions* of adversarial capabilities that together can form various *threat models*, i.e. any two capabilities from different dimensions can always be combined.
The overall structured lattice is depicted with its dimensions in Figure 4.3. Capabilities higher up represent stronger capabilities; the lowest capability in each dimension effectively means the attacker does not have a meaningful capability in this dimension. For example, the combination of the capabilities on the bottom row effectively corresponds to an ideal hash function (in the ROM). We use a list notation to represent a specific threat model, by listing the adversarial capabilities in each dimension. For example, we denote the weakest threat model across the bottom row by $\emptyset$. Conversely, {allCol,allExt,anyTarget,inLeak} is the strongest threat model in which the adversary has all capabilities (and corresponds to modeling the weakest hash function).

Given such a threat model, the formal hash model is obtained in the following way: First, determine the so-called collision-relation $\sim_c$ using Table 4.2 (or simplified in Table 4.3). Second, instantiate the generic hash function model in Figure 4.4 (and optionally Figure 4.5) using the capabilities in the threat model and $\sim_c$.

We now introduce the details of each dimension in turn, including various types of collisions, how hash outputs relate to other messages, and modeling hashes that leak their inputs.

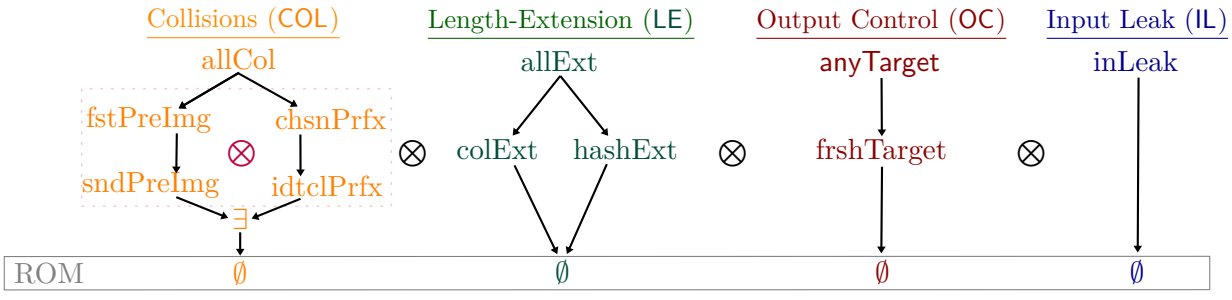


Figure 4.3: Lattice of adversarial capabilities. An edge $x \to y$ expresses that x is a stronger capability than y. Each column is a dimension described at the top. Capabilities from different dimensions can be combined into threat models, e.g.{idtclPrfx,allExt}. The minimal threat model is the empty one, $\emptyset$, which corresponds to the ROM.

Allowed Collisions (COL) Without ROM-like constraints, all kinds of worst-case collisions might be considered, provided that the resulting hash function is indeed a function. For instance, this includes the constant function that maps all inputs to a single value. Such a strong adversarial capability corresponds to an extremely weak hash function requirement, and can be of interest to establish strong security guarantees when possible. In fact, such a constant function is not even the worst-case scenario: if the protocol has authentication properties or inequality checks, modeling a hash function as a constant function might miss attacks. On the other hand, a protocol can be deemed insecure with this strong adversarial capability due to unrealistic attacks. To refine this, we restrict the allowed choices of hash outputs to the relevant classes of collisions one may want to consider.

This is done using a **collision-relation** $\sim_c$: it captures that for all x and y such that $x \sim_c y$, the hash function H allows collisions $H[x] =_E H[y]$. We cover a large spectrum of types of collisions that can be combined as defined in Table 4.2 (the last two rows are explained next). To define those relations, we introduce a number of abstract operators $\mathsf{cp}_1(\cdot,\cdot)$, $\mathsf{cp}_2(\cdot,\cdot)$, $\mathsf{sp}_1(\cdot)$, $\mathsf{sp}_2(\cdot)$, $\mathsf{pi}^1(\cdot)$, $\mathsf{pi}^2(\cdot)$, $\mathsf{c}()$, $\mathsf{c}'() \in \Sigma$ that do not occur in protocols. Those operators correspond to computations performed by the adversary to find a certain collision or preimage, and are motivated by real-world attack strategies. For instance, given two messages p_1, p_2, the messages $\mathsf{cp}_1(p_1,p_2)$ and $\mathsf{cp}_2(p_1,p_2)$ correspond to the message the adversary obtains when computing a CPC for the prefixes p_1 and p_2: the hash of $p_1 \| \mathsf{cp}_1(p_1,p_2)$ equals the hash of $p_2 \| \mathsf{cp}_2(p_1,p_2)$, as expressed by $\sim_{\mathsf{CP}}$. (See an example of CPC in Section 4.2.2.)

Given a relation $\sim_c$ corresponding to the chosen kinds of collisions, the allowed hash outputs determined by the output control dimension OC (explained below) are filtered out. The resulting hash function is depicted in Figure 4.4.

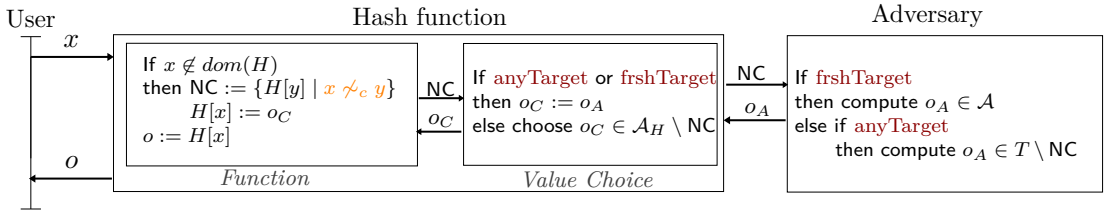


Figure 4.4: Generic hash function model that generalizes the model from Figure 4.2, and can be instantiated with different adversarial capabilities. Initially, $H := \emptyset$. Collisions and Length-Extensions do not appear explicitly since they are captured in $\sim_c$ (see Table 4.2). When the input leak capability is present, the model additionally includes the input leak capability in Figure 4.5.

Capability	$\sim_c$	Intuitions behind the types of allowed collisions	
$\emptyset$	$\sim_\perp$	Ideal model in which hash outputs never collide;	$\forall t \neq t' : t \not\sim_\perp t'$
$\exists$	$\sim_\exists$	There exist two constants c and c' whose hashes collide $H[c] = H[c']$;	$c \sim_\exists c'$
fstPreImg	$\sim_1$	Given $o = H[t]$, the adversary can compute a preimage $t' = \mathsf{pi}^1(o)$ such that $H[t'] = o$; $t \sim_1 \mathsf{pi}^1(H[t])$	
sndPreImg	$\sim_2$	Given t, the adversary can compute a second preimage $t' = \mathsf{pi}^2(t)$ such that $H[t'] = H[t]$; $t \sim_2 \mathsf{pi}^2(t)$	
chsnPrfx	$\sim_{\mathsf{CP}}$	Given t, t', the adversary can compute $u = \mathsf{cp}_1(t,t')$ and $u' = \mathsf{cp}_2(t,t')$ such that $H[t\|u] = H[t'\|u']$; $t\|u \sim_{\mathsf{CP}} t'\|u'$	
idtclPrfx	$\sim_{\mathsf{IP}}$	Given t, the adversary can compute $u = \mathsf{sp}_1(t)$ and $u' = \mathsf{sp}_2(t)$ such that $H[t\|u] = H[t\|u']$; $t\|u \sim_{\mathsf{IP}} t\|u'$	
allCol	$\sim_\top$	All hash outputs can collide, which models the worst possible collision case;	$\forall t, t' : t \sim_\top t'$
hashExt	$\sim_{\mathsf{LEa}}$	Length-extension collision. Given $H[x]$ and s, the adversary can compute $H[x\|s]$; $x\|s \sim_{\mathsf{LEa}} H[x]\|s$	
colExt	$\mathsf{LEc}(\sim_c)$	Length-extension closure. $H[x\|s]$ collides with $H[y\|s]$, if $H[x] = H[y]$ (based on any of the previous capabilities).	

Table 4.2: Intuition behind the basic collision-relations $\sim_c$ depending on the chosen adversarial capabilities. Many of these relations define that there exist collisions that can be computed for very specific, but not all, input patterns. Collision-relations in different dimensions can be combined by taking their union. We give the formal definitions in Table 4.3.

Length-Extension (LE) The two last types of collisions defined in Table 4.2 are specific to the length-extension property and weaknesses of hash functions built with the Merkle-Damgård or similar construc-

Capability	Types of Allowed Collisions	Relation $\sim_c$
$\emptyset$	No collision	$\sim_\perp = \{\}^=$
$\exists$	Existential collisions	$\sim_\exists = \{(\mathsf{c}, \mathsf{c}')\}^=$
fstPreImg	Preimage collisions	$\sim_1 = \{(t, \mathsf{pi}^1(H[t])) : t \in T\}^=$
sndPreImg	Second preimage collisions	$\sim_2 = \{(t, \mathsf{pi}^2(t)) : t \in T\}^=$
chsnPrfx	Chosen-Prefix Collisions (CPC)	$\sim_{\mathsf{CP}} = \{(p_1 \Vert \mathsf{cp}_1(p_1,p_2), p_2 \Vert \mathsf{cp}_2(p_1,p_2)) : p_1, p_2 \in T\}^=$
idtclPrfx	Identical-Prefix Collisions (IPC)	$\sim_{\mathsf{IP}} = \{(p \Vert \mathsf{sp}_1(p), p \Vert \mathsf{sp}_2(p)) : p \in T\}^=$
allCol	All collisions	$\sim_\top = (T \times T)^=$
hashExt	Length-extension collisions	$\sim_{\mathsf{LEa}} = \{(x \Vert s,\ H[x] \Vert s) : x, s \in T\}^=$
colExt	Length-extension closure (closure of $\sim$)	$\mathsf{LEc}(\sim) = \{(x \Vert s,\ y \Vert s) : x, y, s \in T, x \sim y\}^=$

Table 4.3: Basic collision-relations $\sim_c$ depending on the chosen adversarial capabilities. $\sim^=$ denotes the reflexive, symmetric, and transitive closure of the relation $\sim$. For example, by reflexivity, the "no collisions" relation encodes that two hash outputs are the same exactly when their inputs are the same. The "existential collisions" relation encodes that there exists two constants c and c', for which the hash function output collides. Most of the other relations define that there exist collisions that can be computed for very specific, but not all, input patterns. Collision-relations in different dimensions can be combined by taking their union; e.g.sndPreImg,idtclPrfx corresponds to the relations (e.g. $\sim_{\mathsf{IP}} \cup \sim_2$). The only exception is colExt: if this capability is enabled, we first determine the collision relation $\sim$ based on the other capabilities as above, and then compute $\mathsf{LEc}(\sim^=)$ as in the last row.

tions.

The first, hashExt, captures the adversarial capability to extend the payload that is under a hash with some adversary-chosen suffix, and is captured by the collision-relation $\sim_{\mathsf{LEa}}$. Namely, given a hash output $H[x]$, for any suffix s of its choice the adversary can compute $H[x\|s]$ without knowing x. Indeed, the hash output of $x\|s$, which the attacker cannot compute, is allowed to collide with the one of $H[x]\|s$, which the attacker can compute. This is the most classical weakness of length-extension. The second, colExt, corresponds to the fact that collisions may be extended, i.e. as soon as $H[x] = H[y]$, we will also have that $H[x\|s] = H[y\|s]$ for any s. Interestingly, HMAC-SHA2 and HMAC-MD5 constructions have colExt but not hashExt for a given key.[3]

Output Control (OC) In a worst-case scenario, we consider a hash function where the attacker may control the output of the hash function to some extent, provided that the resulting hash is indeed a function. Such scenarios could occur if, e.g. the hash function was badly designed or has a backdoor. It also mirrors attacks similar to the nostradamus attack over MD5[140] where the attacker can by injecting some bytes inside the input make the hash function go to a previously chosen target. The hash outputs can be taken from (i) a set of atoms $\mathcal{A}_H$, that model fresh values unknown to the attacker (unless revealed), as in the ROM, (ii) fresh atoms chosen by the attacker (frshTarget), or (iii) arbitrary messages provided that the adversary knows them (anyTarget). The default capability models one of the behaviour of the ROM, where each hash output is taken from a set of fresh atoms (but still allowing for collisions based on COL). The second capability frshTarget models the case where the attacker can partially control the outputs of the hash function that nonetheless must be names. This captures some type-flaw attacks but the attacker cannot control the actual shape of the hash values which will appear as junk bytes. The third one anyTarget additionally captures arbitrary type-flaws attacks, where the attacker can fully control the hash output to an arbitrary value.

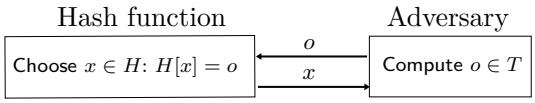


Figure 4.5: inLeak input leak adversarial capability.

Input Leak (IL) Finally, hash functions are sometimes implicitly assumed to guarantee confidentiality of their inputs, and sometimes ill-used for this purpose in practice. In reality, some badly designed hash functions might leak information about their inputs. Previous symbolic analyses did not capture this capability, which can yield practical attacks. We model the worst case scenario in which the adversary

[3]HMAC [147] is message authentication code construction based on hash functions.

can obtain the complete input. We therefore introduce an adversarial capability inLeak that allows the adversary to learn the hash input of a given hash output as defined in Figure 4.5.

4.3.1 Lattice of Threat Models

As explained previously, we use the notation $\{\cdot\}$ to describe a specific threat model made of the explicit hash weakness in each dimension, and omit it when there is none for this dimension. The weakest capabilities occur as $\emptyset$ corresponding to ROM and Figure 4.2. Conversely, in the strongest threat model, {allCol,allExt,anyTarget,inLeak} all collisions are possible (and adversary-chosen), all types of type-flaw attacks are considered, and hash outputs leak their inputs. The purpose of the lattice structure is to structure this spectrum of threat models spanning those two extremes. In the next section we show how we effectively explore the lattice.

Example 3. *The weakest threat model capturing the CPC attack from Section 4.2.2 is {chsnPrfx,colExt}. Indeed, the attack is possible provided that CPC exist (chsnPrfx) and can be extended due to the length-extension property (colExt).*

Remark 1. *In our symbolic model, second preimage resistance (sndPreImg) implies preimage resistance (fstPreImg). This might seems surprising, since in the computational models with the same name [172] this is not the case. This is however not a contradiction, but an* intended *consequence of our modeling choices and our orthogonal inLeak capability, which cause us to use slightly different definitions in this case despite using the same names. inLeak expresses the ability for the adversary to compute* the *preimage of a given hash output. fstPreImg expresses the ability to compute* some *input* $\mathsf{pi}^1(H[t])$ *that when hashed yield a given, target output* $H[t]$ *but that is* always different *from the original input* t *and similarly for sndPreImg. For this modeling choice in our symbolic model, the above implication indeed holds.*

Cross-dimension implications The previous lattice contains some redundant capabilities that are not captured by the joint partial order. For instance, we have that $allCol \Rightarrow hashExt$, $sndPreImg \wedge inLeak \Rightarrow fstPreImg$ and $inLeak \Rightarrow hashExt$. Further, if on a dimension the protocol is secure at some given level, it is secure for all weaker threat models (i.e. levels below), and conversely for attacks.

4.4 Automation Methodology

In this section we present different ways to automate verification for the previously defined threat models. Our goal is to cast these threat models in Tamarin, one of the most widely used tool for symbolic analysis of security protocols.[4]

We first explore a direct way of modeling these capabilities as equational theories (Section 4.4.1), which seems the most idiomatic solution in symbolic tools. This modeling is however limited to a few capabilities due to limited tool support for more advanced equations.

We then explain how we overcome this challenge by extending Tamarin (Section 4.4.2) yielding a fully automated tool that is capable of exploring all of our threat model lattice. We achieve this by extending Tamarin with a associative concatenation operator in order to provide a more precise model for collisions and length extensions. We build on the latter and model hash function computations by a call to an oracle process, in the flavor of Figure 4.4. This process allows fine-grained control over the output values and raises events on each call that allow for a logical specification of the set of possible collisions and can be fully automated in Tamarin. We provide tooling that uses Tamarin as back-end to automatically explore our lattice of threat models and that returns the weakest threat models under which attacks were found (if any).

4.4.1 Equational Theory – Modeling

As explained in Section 2.3.1, the classical way to model cryptographic primitives in a symbolic model is to use an equational theory, that specifies which operations yield equal values.

[4]Note that the threat models were additionally cast to ProVerif [43] by [61]

Capabilities as Operators

Using an equational theory allows us to naturally and efficiently explore the set of scenarios corresponding to the capabilities fstPreImg, sndPreImg, ∃, as well as inLeak. Each threat scenario for a given dimension can be encoded by the following corresponding equation that we explain below.

$$\begin{array}{rlrl} \mathit{fstPreImg} & \mathrm{H}(\mathsf{pi}^1(z)) = z & \exists & \mathrm{H}(c) = \mathrm{H}(c') \\ \mathit{sndPreImg} & \mathrm{H}(\mathsf{pi}^2(z)) = \mathrm{H}(z) & \mathit{inLeak} & \mathsf{i}(\mathrm{H}(z)) = z \end{array}$$

We note that the absence of a weakness for each dimension is the default in symbolic models and do not require any particular modelling. To achieve a particular threat model we simply add the corresponding equations for the collision and input leak dimension.

To express that a hash function is not *preimage resistant* (fstPreImg) we provide the attacker with an explicit function pi^1 that allows the attacker to compute a preimage of a hash z, i.e. a value $\mathsf{pi}^1(z)$ that hashes to z. Similarly, absence of *second preimage resistance* (sndPreImg) is expressed by the function pi^2 that corresponds to the attacker's ability to compute a second preimage.[5] The absence of *collision resistance* (∃) relies on two distinguished constants c and c' on which the hash function collides. These two constants do not have any particular structure and are not attacker chosen, modelling that a collision merely *exists*. Finally, we model that a hash function may leak (part of) its input (inLeak) by giving the attacker the capability to inverse the function using the symbol i, and no leak being the default does not require an equation.

We encoded this part of the hierarchy in Tamarin and used it on multiple case-studies as presented in Section 4.5.1. As we will see, while such an equation based model is easy to deploy using symbolic tools, it is also rather weak (and not a very effective way of finding attacks): the equations are basically obtained by negating the security assumption and model the existence of a collision, or (second) preimage without giving the attacker any additional control. (We exemplified this gap with Sigma' in Section 4.2.2.) On the other hand, when finding an attack with this model generally translates directly to a missing assumption on the security of the used hash function.

Second Preimage Equation

Unfortunately, Tamarin is unable to handle the equation $\mathrm{H}(\mathsf{pi}^2(z)) = \mathrm{H}(z)$: when internally completing this equation it would need to introduce an infinite number of rewrite rules $\mathrm{H}(\mathsf{pi}^2(\ldots \mathsf{pi}^2(z)\ldots) \to \mathrm{H}(z)$ stacking any number of applications of pi^2, because they would all be equal to $\mathrm{H}(z)$. This can be avoided using a trick: additionally providing $\mathrm{H}(z)$ to pi^2 in the equation $\mathrm{H}(\mathsf{pi}^2(\mathrm{H}(z), z)) = \mathrm{H}(z)$ effectively avoids the problem mentioned above. As the attacker is able to compute $\mathrm{H}(z)$ from z this second argument can be added without loss of generality.

Challenges with Modeling Associative ‖ and MD

To go beyond the above *existential* modeling of weaknesses in hash function we will give a more detailed model of the associative ‖ operator and of the MD construction whose properties can be exploited by an adversary. This will allows us to explore IPC, CPC, and length extension attacks, which are not covered using equations from the previous section.

The presence of an associative concatenation operator is required if we want to capture IPC and CPC. Indeed, recall that given prefixes, p_1 and p_2, a CPC attack computes suffixes s_1, s_2 such that $\mathrm{H}(p_1 \| s_1) = \mathrm{H}(p_2 \| s_2)$. If, for example, a protocol participant computes the transcript $\mathrm{H}(m_1 \| m_2 \| m_3)$ and the attacker controlled parts are m_2 and m_3, then the suffix in the previous equation needs to be $m_2 \| m_3$. With a non-associative concatenation operator (that we denote $\langle \cdot, \cdot \rangle$) the CPC attack would fail as $\mathrm{H}(\langle\langle m_1, m_2\rangle, m_3\rangle)$ would not be identified with $\mathrm{H}(\langle m_1, \langle m_2, m_3\rangle\rangle)$. This raises a challenge for Tamarin (and other existing tools,) as it does not allow to model such an *associative operator*. The core difficulty is that the tools rely on unification. Given two messages one needs to be able to compute a finite and complete set of most general unifiers, i.e. a set of substitutions that represents all possible ways of instantiating the messages that make them equal. For instance, $\langle x, 0\rangle =^? \langle 1, y\rangle$ has a unique unifier

[5] The equation for pi^2 does not work out of the box for a technical reason that we describe in Section 4.4.1.

$\{x \mapsto 1, y \mapsto 0\}$. For associative operators, the issue is that such a set is not always finite. For instance, $0\|x =^? x\|0$ has an infinite set of unifiers of the form $\{x \mapsto 0^n \mid n \in \mathbb{N}\}$. A first approach is to model the associativity under a hash function operator for a bounded depth only, for instance specifying that $h(\langle x, \langle y, z\rangle\rangle) = h(\langle\langle x, y\rangle, z\rangle)$, but this does not imply that $h(\langle x, \langle y, \langle z, z'\rangle\rangle\rangle) = h(\langle\langle\langle x, y\rangle, z\rangle, z'\rangle)$. We use such a bounded modeling successfully on some examples in Section 4.5. However, this modeling may miss attacks that require associativity at a deeper depth than modeled. We explain how we can overcome this problem in the next section.

Furthermore, a naive way to encode the MD construction would be to directly consider the equation $\mathrm{H}(\langle x, y\rangle) = f(\mathrm{H}(x), y)$. However, such an equation is out of scope Tamarin (and other popular tools like ProVerif) as it cannot be completed into a convergent rewrite system, which seems to be an inherent difficulty for all tools based on equational reasoning.

4.4.2 Extending Tamarin for a Full and Automatic Lattice Exploration

Extension for the $\|$ operator We extended Tamarin to obtain partial support for associative operators such as $\|$. The key observation is that we do not need the unification problem for an associative operator to yield a finite set in general: it is sufficient that all *particular* unification problems that actually appear in a protocol's verification have a finite set of unifiers.

Tamarin relies on the Maude tool as a backend to perform equational unification. Although unification for an associative operator is infinitary, support has recently been added in Maude [107]: it either returns the complete set of unifiers, or only a subset but with a warning. We integrated this new feature in Tamarin, which now has a built-in associative concatenation operator denoted by $\|$. To ensure correctness, Tamarin stops the analysis as soon as Maude raises a warning. In particular, as we use $\|$ under a hash function only, our case studies (Section 4.5) illustrate that Tamarin encountered this Maude warning in rare occasions only[6].

Proof. We provide here the proof for our extension to Tamarin. To achieve this, we rely on the two main Tamarin thesis [169, 197], from which we reuse notations and definitions without reintroducing them.
From a high-level point of view, the original Tamarin proof is split into three main parts:

1. the validity of exploring possible protocol executions using so-called dependency graphs (Lemma 3.10 [197]);
2. a set of constraint solving rules over dependency graphs that are sound and complete (Theorem 3.33 [197] or Theorem 4 [169]);
3. a set of normal form conditions over dependency graphs, that allows to reduce the set of dependency graphs to consider by removing redundant ones (Lemma 3.19/A.12 [197])).

We note that for the soundness and correctness of Tamarin, only point 1) and 2) are needed. Point 3) should help the constraint solving algorithm terminate.
With the original proof of Tamarin in mind, we can describe our extension as the addition of:

- a builtin concatenation symbol $\|$ with an associative (A) equation;
- the corresponding attacker construction/deconstruction rules:

$$\frac{\mathtt{K}^{\downarrow d}(t_1\|\ldots\|t_n)}{\mathtt{K}^{\downarrow d}(t_1)\ \ldots\ \mathtt{K}^{\downarrow d}(t_n)} \qquad \frac{\mathtt{K}^{\uparrow u}(t_1)\ \ldots\ \mathtt{K}^{\uparrow u}(t_n)}{\mathtt{K}^{\uparrow u}(t_1\|\ldots\|t_n)}$$

- two normal form conditions on dependency graph.

  N7' There is no construction rule for $\|$ that has a premise of the form $K^{\uparrow}(s\|t)$ and all conclusion facts of the form $K^{\uparrow}(s\|t)$ are conclusions of a construction rule for $\|$.

  N8' The conclusion of a deconstruction rule for $\|$ is never of the form $K^{\downarrow d}(s\|t)$.

We thus have the following proof obligation:

1. The proof of Theorem 4 [169] holds for an equational theory containing an A symbol;

[6]The warning only occurred with one of the security properties out of the 21 we verified, and even then only for a particular threat-model.

2. The proof of Lemma A.12 [197] holds with the two added rules N7' and N8'.

Lemma A.12 We note that the (de)construction rules are duplicates from the multiset operator rules. Further, the normal form conditions N7' and N8' are duplicates of N7 and N8 from [197]. As such, the proof for N7 and N8 directly applies to N7' and N8'.

Theorem 4 The original proof holds for an equational theory E for which there is a complete and finite unification, as mentioned in the first sentence of Section 8.2 [169]. We observe that removing the finitary condition does not impact the proof as long as we allow disjunction over constraints to be potentially infinite. The only difference is that the rule $S_{\approx}$ may now create an infinite disjunction when $unify_E^{vars(\Gamma)}(t_1, t_2)$ is infinite. However, this does not change the fact that the completeness and soundness proof for this rule hold, as we do consider all possibilities thanks to the completeness of the unification (albeit there are infinitely many of them). Thus, as (A) does have a complete unification algorithm, Theorem 4 in the infinite interpretation does hold when (A) is integrated inside the equational theory. Hence, Theorem 4 covers the soundness and completeness of the theory behind the constraint solving algorithm with an (A) symbol.

□

Implementation We directly plugged the maude unification algorithm for (A) inside Tamarin, which raises a warning when it encounters a unification case for which the set of unifiers is infinite. The consequence of considering that in theory the constraint solving algorithm may create an infinite disjunction has of course consequences in the practical proof search of Tamarin as we cannot explore this infinite set of cases. As such, whenever in practice Tamarin makes a unification query for which the unification algorithm returns an infinite set, we must abandon the proof. Note that we can still try to find an attack in such a case.

Event based modeling The equational based models presented above have several drawbacks:

- the equations for computing collisions, and (second) preimages are *existential* and do not give the adversary any control over the computed messages, missing most of our threat models and practical attacks (e.g. based on CPC and IPC);
- the associative operator added to Tamarin increases the complexity of the equational theories often leading to non-termination or extensive verification times when modeling CPC and length extension attacks;
- the previous models do not cover the OC dimension.

To overcome these limitations, instead of using an operator to model a hash computation, we define, in parallel to the protocol processes, a dedicated process for computing the hash function, in the spirit of Figure 4.4. Notably this approach allows to either sample the hash value from a fresh set of values ($\emptyset$) or query the output values to the attacker (frshTarget or anyTarget), i.e. let the attacker choose the value. By default, this process is free to create *any* collisions. To restrict this, we will give a *logical specification* when precisely collisions are allowed. More precisely, *(i)* whenever a hash output value o is returned for some input i, we raise an event $\mathsf{Hash}(i, o)$, and *(ii)* we specify that when a trace contains two events $\mathsf{Hash}(i, o)$ and $\mathsf{Hash}(i', o)$, i.e. two inputs i and i' are mapped to the same output o, then we must have $i \sim i'$ for the desired $\sim$ relation. Otherwise the trace is discarded. For example, if $\sim$ is the identity relation we do not allow any collision; if $\sim$ relates all inputs then arbitrary collisions are allowed.

Discarding traces is achieved using Tamarin's *restrictions*. A restriction (or axiom) ρ is a logical formula that is considered part of the specification and discards any trace that does not satisfy ρ: more formally, instead of verifying that all traces satisfy a property φ, we actually check $\rho \Rightarrow \varphi$. Slightly simplifying, we can represent the threat model {chsnPrfx,colExt} with the following restriction:

$$\begin{aligned}\forall i, i', o.\ \mathsf{Hash}(i, o)\ \&\ \mathsf{Hash}(i', o)\ \&\ i \neq i' \\ \Rightarrow \exists p_1, p_2, l.\ (\quad i \ = p_1 \| \mathsf{cp}_1(p_1, p_2) \| l \\ \&\ i' = p_2 \| \mathsf{cp}_2(p_1, p_2) \| l)\end{aligned}$$

This restriction requires that if a collision occurs for i and i', then it must be a length-extended CPC: the $\mathsf{cp}_i(p_1, p_2)$ represent the attacker chosen suffix for prefixes p_1 and p_2 (chsnPrfx) and l is a same length extension on both inputs (colExt).

Plug-and-Play library and tooling Using this approach, we define a modular library for hash functions that allows Tamarin to explore the full lattice of capabilities. We developed a python script that allows to

check a given protocol for all possible scenarios, only exploring non-redundant scenarios and outputting the minimal threat models under which an attack was found. This yields a push-button tool that produces results that can be as compact, yet comprehensive, as those shown in Table 4.6.

4.5 Case Studies

Using the techniques from the previous section, we have analyzed 20 protocols: 15 of them are based on existing Tamarin models from the literature which were made available in the official Tamarin repository [209] The remaining 5 case studies were modeled by us, covering protocols like IKEv2 or SSHv2. We first analyze all of these protocols using the equational theory based hash models in Tamarin (Section 4.5.1), exemplifying the limitations of this approach. We then perform an in-depth analysis of the five new models using the event based modeling, completely automating the exploration of the threat model lattice with Tamarin. We present our analysis methodology in Section 4.5.2, our main insights in Section 4.5.3, and the full results in Section 4.5.4.

4.5.1 Equational Theory – Hash Models

Using the equational theories described in Section 4.4.1, we analyzed all 20 case studies. The results are listed in Table 4.4. Even with the strongest threat model in the hierarchy described in Section 4.4.1 without input leak ({fstPreImg}), only one potential attack is found. The TESLA protocol (v1) instantiates a *pseudorandom function (PRF)* with a weak construction (HMAC-MD5), and we found that this could break the protocol as soon as preimage attacks against it will be found. Adding the equation for input leak (Section 4.4.1) results in the scenario {fstPreImg,inLeak}, and triggered regular non-termination issues in Tamarin. For many protocols the input leak resulted in potential attacks on secrecy. For this particular set of case studies, this is not surprising as the hash function is applied to cryptographic keys.

4.5.2 Fully Automated Analysis Methodology

As described in Section 4.4.2, we developed a Tamarin library that can be imported into a model. It allows verification of a specified threat model in the lattice of hash weaknesses. To automate a systematic exploration of the full lattice of threat models, we developed a Python program that computes the set of all minimal (and maximal) scenarios that invalidate (and, respectively, validate) the security goals. This program allows for parallel verification, and avoids redundant exploration: once a property is falsified for a threat model, we automatically conclude that it is falsified for all stronger threat models (and conversely for verified properties). We also exploit cross-dimension implications discussed in Section 4.3.1, and avoid calling Tamarin systematically for each of the 264 distinct scenarios of the lattice. As a heuristic, we start by verifying the set of strongest and weakest threat model, as they may allow to quickly prune the search space.

For a given protocol, Tamarin may find an attack on some threat model in a very short time, but take much longer to find the same attack in a more complex threat model, and the converse may happen for a security proof. Thus, the optimal order of verifying the scenarios is protocol dependent. We therefore first run each analysis with a short timeout, to ensure that we first find all easy proofs and attacks. We then immediately conclude implied results and prune the corresponding scenarios. We then re-run the remaining scenarios with a longer timeout. We show an example of a fully automatically generated table for Sigma' in Table 4.6 (detailed in Section 4.5.4.1).

After the initial, fully automated analysis, one can perform a more in-depth analysis of the attacks found. Notably, multiple attacks may exist for a given threat model, and by default the tools return the first attack found. This may "hide" some interesting attack variants and can cause Tamarin to initially report different variants. Tamarin's *interactive mode* can be used to semi-automatically find *all* variants of attacks for a given threat model.

Extracted from	Protocols	Modeling variations	Attack found
Schmidt et al. [198] & Meier [169]	DH2 [60]	1	✗
	TS1 [137]	1	✗
	TS2 [137]	1	✗
	KAS1 [17]	1	✗
	KAS2 [17]	1	✗
	KEA+ [156]	5	✗
	STS-MAC [40]	2	✗
	NAXOS [154]	3	✗
	UM [39]	4	✗
	TESLAv1 [185]	1	✓**
Mauw et al. [165][166]	Meadows [168]	3	✗*
	MAD [58]	1	✗*
	Kim & Avoine [141]	1	✗*
	Munilla et al. [177]	1	✗*
	CRCS [188]	2	✗*
original	IKEv2 [139]	2	✗
	Sigma [146]	1	✗
	Telegram KE [210]	1	✗
	SSHv2 [162]	1	✗
	Flickr [106]	1	✗*

* = Attack requires breaking one-wayness of the hash
** = Attack requires to instantiate a PRF with a hash function that is not preimage resistant

Table 4.4: Our initial list of case studies using the equation based threat models from Section 4.4.1. While the results can indicate missing proof assumptions, this methodology is rather weak and not effective at finding attacks. This motivated our decision to develop a more fine-grained methodology.

4.5.3 Results from Automated Analysis

We demonstrate the applicability of our methodology on our original Tamarin case studies and report on the results obtained by running our Tamarin-based automatic tool for exploring our lattice of threat models. We first detail the results for Sigma' in Section 4.5.3 to exemplify how to interpret the results and then discuss a selection of other attacks and insights. In Table 4.5 we summarize the most interesting attacks that our method automatically found and that we describe in the remainder of this section (we refer to attacks with labels such as AT(S1)). Our attacks are at the design level: their severity depends on whether the discovered attack requirements (including the choice of hash primitive) are met given a specific implementation, threat model and use case.

Discovered Attacks and Insights

In this section, we will highlight the results of our analysis and will use our running example Sigma' to showcase how the results should be read and interpreted.

Sigma' We analyzed mutual authentication and key secrecy for Sigma'. As argued in [37], even though Sigma' is not deployed, its protocol logic is similar to many widely deployed authentication protocols such as TLS, SSH, IKEv2, which makes it an interesting and relevant case study. The output of our tool for this protocol model is shown in Table 4.6. Each row contains either one of the strongest threat models under which all of the three properties hold or one of the weakest threat model under which one of the properties is violated. For Sigma', they were actually all violated as soon as one was. This kind of tables (more of them are in Section 4.5) allow to concisely and yet comprehensively describe the security level against any of the threat models in the lattice.

How to read threat model tables. As an example, consider the last row of Table 4.6: the protocol is broken as soon as CPCs are possible (chsnPrfx) and can be extended thanks to colExt, even when hash outputs are fresh values preventing any type-flaw attack. However, without colExt, the protocol is

Protocol	Broken properties	Main attack requirements	New?	In-text ref.	Time (s)	Note
Sigma	Sec,Agr(t)	chsnPrfx,colExt	[37]	AT(S1)	28	
	Sec,Agr(t)	chsnPrfx,colExt	∼[37]	AT(S2)	manual	collisions on shares
	Sec,Agr(t,role)	chsnPrfx	new	AT(S3)	55	role-confusion, no need for colExt
SSH	Agr(nego)	CI(*)	new	AT(SSH1)	3	see Figure 4.6
	Agr(nego)	idtclPrfx,colExt	[37]	AT(SSH2)	28	
	Agr(nego)	CI(I),sndPreImg, colExt	new	AT(SSH3)	41	
IKEv2	Sec(R)	CI(*)	new	AT(IKE1)	6	CI should be on the cookie
	Auth(I)	idtclPrfx,colExt	[37]	AT(IKE2)	20	
	Agr(cookie,t)	∃,colExt	new	AT(IKE3)	9	disagreement on cookies only
Flickr	Auth(I)	hashExt	[106]	AT(F)	9	

Table 4.5: A selection of the most meaningful attacks we found whose details are given in Section 4.5. Those attacks are at the design level and their severity depends on whether the attack requirements are met given a specific implementation and threat model.
Time: number of seconds it takes for our tool and Tamarin to find the attack.
Sec: Secrecy of session data (e.g. session key); only from a given role's perspective if specified (R: responder, I:initiator),
Agr(data): agreement can be either on transcript data (t), negotiation data (nego), protocol role information (role), or cookie data (cookie). Note that disagreement on negotiation data (nego) can lead to downgrade attacks,
Auth(X): authentication of role X (R: responder, I:initiator, *:both),
∼: new variant of an existing attack,
CI(X): role X must suffer from colliding inputs; see Section 4.5.3.

Threat Models				Auth.
COL	LE	OC	IL	
fstPreImg,chsnPrfx	hashExt	anyTarget	inLeak	✓
allCol				✗
fstPreImg,idtclPrfx	allExt	anyTarget	inLeak	✓
chsnPrfx	colExt			✗

Table 4.6: Sigma' analysis. ✓: security holds, ✗: attack found.

deemed secure (otherwise this threat model would not represent a minimal violation). Similarly, a CPC is required. Inspecting the attack trace returned by Tamarin for this threat model, we observe that it corresponds to the CPC attack AT(S1) from [37] described in Section 4.2.2.

The second row shows that when collisions are unconstrained (allCol), then an attack is possible. This was to be expected as such collisions subsume through a cross-dimension implication CPC with colExt. While the automated analysis uses those implications to prune the search space, we chose for clarity to display in the tables all minimal results for the basic partial order of the lattice.

The third row shows that even for the strongest OC, LE, and IL capabilities, the protocol is deemed secure if CPC is impossible (as shown by the COL capability, which is the strongest among the ones that are strictly weaker than chsnPrfx). Similarly, the first row shows that even for the strongest OC and IL capabilities and for {fstPreImg,chsnPrfx}, we find no attacks as long as colExt is impossible.

Colliding input attacks on SSH and IKE Using our methodology, we found *colliding input attacks* on both, SSH and IKEv2. Colliding input attacks allow an adversary to alter exchanged messages between two agents such that they are parsed differently by both agents but yield the same bitstring when they are recomposed into a bitstring *prior* to hashing, hence not relying on hash collisions or other hash weakness. An example of such an attack AT(SSH1) is depicted in Figure 4.6 in the case of SSH, where the attacker can

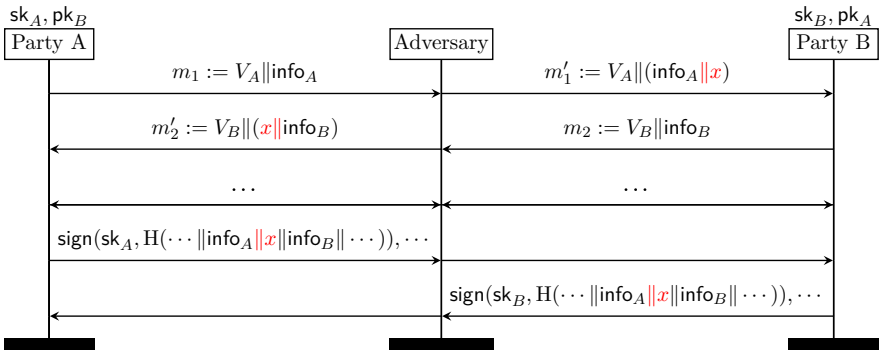


Figure 4.6: Attack AT(SSH1) on SSH: In the set-up phase where A sends the set of its allowed algorithms info_A, the adversary alters info_A by extending it with some message x. The adversary then alters info_B sent by B by putting x before it. Despite the different input messages, the concatenation of $info_A$ and $info_B$ results in the same transcript $\cdots \| \mathsf{info}_A \| x \| \mathsf{info}_B \| \cdots$ on both side.

alter the messages such that both parties agree on different sets of algorithms in the set-up phase. As shown in the figure, the attack relies on the fact that, before hashing, fields of previous messages are concatenated into a bitstring, hence loosing the message structure: $(\mathsf{info}_A \| x) \| \mathsf{info}_B$ is the same as $\mathsf{info}_A \| (x \| \mathsf{info}_B)$. For such attacks to be realistic, the protocol needs to drop some length fields (which initially allowed unambiguous message parsing) before hashing or be very liberal about parsing, e.g. accept any list of length-prefixed fields and process them until the input buffer is empty without requiring knowledge of the list size in advance. Our automated verification framework is able to capture such attacks and did so for both SSH and IKE; it is up to the user to assess if such attacks may occur in actual implementations.

While this attack may not practical on SSH or IKE, in an early version of the *EU Federation Gateway Service*[109] for the EU's contact tracing apps for Covid-19, this kind of colliding input attack was found, which was caused by the hash computation that concatenated multiple fields, including two variable-length fields. The attack allowed circumvention of accountability, e.g. for the gateway server to manipulate country data, or for countries to repudiate data that they uploaded; it was fixed before full deployment.

IKEv2 We already mentioned an input colliding attack AT(IKE1) above for the weakest threat model $\emptyset$. This attack seems impractical since it requires an ambiguous parsing of the cookie. However, it can be simply combined with an IPC to change the length information at the start of the cookie to make it practical as it no longer relies on ambiguous parsing. We automatically found this stronger attack variant AT(IKE2), which corresponds to the one documented in [37], for the threat model {idtclPrfx,colExt}.

Finally, we found a low-severity attack AT(IKE3) for the threat model {∃,colExt} where the adversary breaks transcript agreement as it uses two colliding hash inputs as the cookies for respectively the initiator and responder.

SSH In addition to the colliding input attack previously mentioned, we automatically found the IPC attack AT(SSH2) from [37] for the threat model {idtclPrfx,colExt}. This is similar to the previous attack escalation of IKE and appears to be a recurring pattern.

Finally, our analysis revealed a different attack AT(SSH3) for the threat model {sndPreImg,colExt} that exploits a second preimage attack and the specific way the transcript is reorganized prior to being hashed; the attack would not be possible if the hash input would simply be the transcript. However, it requires the initiator to be liberal in the way it parses B's negotiation information (similarly to input colliding attacks). This attack violates transcript agreement and allows a MITM attacker to completely tamper with the negotiation information sent by the initiator to the responder, potentially enabling downgrade attacks.

Telegram We modeled Telegram's key exchange protocol described in [210] and [6, Figure 57]. The minimal threat model for an attack AT(T) is {frshTarget}, which is not surprising as some output of the hash function is used as a secret key. However, we found no collision attack even under the strong threat model {fstPreImg,chsnPrfx,allExt,inLeak}. This seems to indicate that despite using SHA1 and SHA2-256

as hash functions, Telegram cannot be broken based on the hash weaknesses of our lattice. The attack for {frshTarget} however indicate that PRF like assumptions are needed to prove the security of the protocol, which has been independently identified in [6], and Telegram would benefit from upgrading their hash functions. Note that timing attacks such as the one of [6] are still out of scope for our techniques.

Flickr For the old API of Flickr, we automatically found the length-extension attack AT(F) first documented in [106] under the threat model {hashExt}. We also found variants of these attacks relying either on the inLeak or the fstPreImg capabilities. Our methodology could have then easily spotted the design flaws of this API. After the discovery of the first attack [106], Flickr migrated to the Oauth framework.

4.5.4 Additional Results for the Case Studies

In this section, we showcase some excerpts of the tables that were generated in an automated fashion (with some manual edits for a simplified display). We did explain in section 4.5.3 how to read such tables, which we briefly recall here. Each table corresponds to a protocol analysis, where each row is a threat model and each column after the vertical line is a security property that we verified. We only display the threat models that are a minimal or a maximal one for one of the properties. In each row, the red or green colored tick or cross correspond to these maximal or minimal entries, and each gray tick or cross in a column indicate that the result for this property is implied by one of the colored ones in the same column.

When a result contains a $*$, it means that some simplifications where required to make Tamarin terminate. Those are either a restriction on the number of time the attacker may use a capability, or a restriction on which input values inside the protocol may be used to stuff collisions. Our Python script detects when such simplifications are needed (in case we reach the end of the timeout) but indicates the use of those with $*$.

4.5.4.1 *Sigma*

For the Sigma protocol, we verified the secrecy of both keys and authentication in both directions. We show an excerpt of our results over the sigma protocol in table 4.6, where we only display the results for a single Lemma, as the others strictly have the same set of minimal and maximal threat models. The {chsnPrfx,colExt} scenario corresponds to the attack of [37].

We automatically found the CPC attack from [37], described in Section 4.2.2. We additionally found a variant of this attack where the CPC is computed to replace the DH shares g^x, g^y instead of the $\mathsf{info}_A, \mathsf{info}_B$ fields. This variant AT(S2) seems harder to exploit as it is unlikely that DH shares allow a large enough search space for finding the CPC.

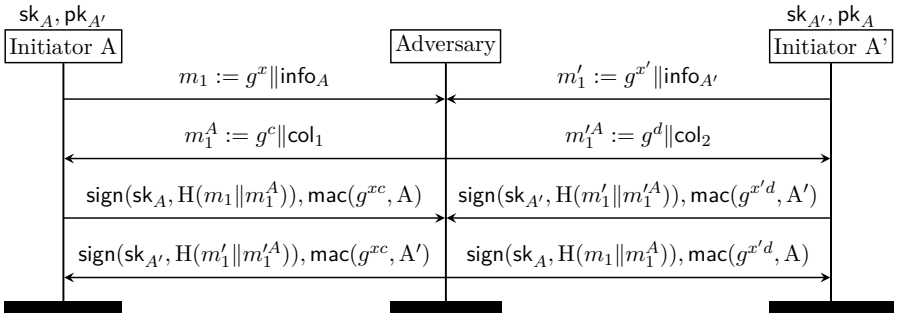


Figure 4.7: The new CPC attack we found on Sigma'. $\mathsf{col}_i = \mathsf{cp}_i(L_1, L_2)$ for $i \in \{1, 2\}$ where $L_1 := g^x \| \mathsf{info}_A \| g^c$ and $L_2 := g^{x'} \| \mathsf{info}_{A'} \| g^d$. Therefore $\mathrm{H}(m_1 \| m_1^A) = \mathrm{H}(m_1' \| m_1'^A)$ so that A's and A''s signatures can be reused.

More interestingly, we also automatically found a different CPC attack AT(S3) that, to the best of our knowledge, was not documented before. The adversary acts as a MITM between two agents A and A' both acting as initiator. He is able to impersonate A towards A', who believes A is acting as a responder, and A' towards A, who believes A' is acting as a responder. The adversary additionally learns both session keys.

This attack thus violates session key secrecy, authentication, and agreement on the agents' role. The attack is depicted in Figure 4.7. A simple fix for this attack is to include the role name (initiator or responder) under the signature to avoid such role confusion. We show automatically that the attack disappears with this fix.

4.5.4.2 *IKE*

We distinguish our analysis between the IKE version where cookies are used and where they are not used. In both scenarios, we allow for weak DH elements.

Cookie and weak DH Due to the type confusion, the attacker can always break the secrecy of the responder by sending him the weak DH element, and using the cookie and the info to create the same transcript on both sides.

The collision scenarios ∃ and idtclPrfx show how the attack can also stuff the transcript with some controlled values, and typically allow to produce the attack from [37] by tampering with the lengths of the cookies (the first attack found by Tamarin in the idtclPrfx scenario does not exactly correspond to the attack, but it can be re-found in the interactive mode by exploring all the attacks).

No cookie and weak DH Without the cookie to allow stuffing elements before the first DH share appear, we see in table 4.8 that the attacker cannot anymore break the secrecy. However, whenever the attacker can compute a second preimage, he can lie about the first element of the transcript.

Threat Scenarios				Lemmas		
OC	COL	LE	IL	trans_auth	secrecy_key_A	secrecy_key_B
	allCol			✗	✗	✗
anyTarget	fstPreImg,chsnPrfx	allExt	inLeak	✗	✓	✗
	∃	colExt		✗	✓	✗
anyTarget	fstPreImg,chsnPrfx	hashExt	inLeak	✓	✓	✗
				✓	✓	✗
anyTarget		allExt	inLeak	✓	✓	✗
	idtclPrfx	colExt		✗	✓	✗

Table 4.7: IKE analysis with Cookie and weak DH

Cookie without weak DH Without the weak DH element, it becomes harder for the attacker to obtain the key of the responder (but the other lemma yield similar results). Notably, we found that, in comparison to table table 4.7, the lemma *secrecy_key_B* is now true (in the simplified setting) for {∃,colExt} and {idtclPrfx,colExt}. However, this is a case where in the most complex threat model, the incompleteness of the associativity operator provoked the failure of the Tamarin proofs.

Threat Scenarios				Lemmas		
OC	COL	LE	IL	trans_auth	secrecy_key_A	secrecy_key_B
	sndPreImg	colExt		✗	✓	✓
anyTarget	chsnPrfx	allExt	inLeak	✓	✓	✓
	allCol			✗	✗	✗
anyTarget	fstPreImg,chsnPrfx	hashExt	inLeak	✓	✓	✓
anyTarget	∃	allExt	inLeak	✓	✓	✓
anyTarget	fstPreImg,chsnPrfx	allExt	inLeak	✗	✓	✓

Table 4.8: IKE analysis with weak DH but no cookie

4.5.4.3 *SSH*

Due to the colliding input attack described in Section 4.5.3, the attacker can always break authentication. The colliding input attack is then evolved into the attack on the transcript from [37], in a similar fashion

to the previous IKE case.

Threat Scenarios				Lemmas			
OC	COL	LE	IL	secrecy_key_A	secrecy_key_B	trans_auth	agree_keys_all
anyTarget	fstPreImg,chsnPrfx	allExt	inLeak	✓*	✓*	✗	✗
				✓*	✓*	✓	✗
anyTarget	fstPreImg,chsnPrfx	hashExt	inLeak	✓*	✓*	✓	✗
	sndPreImg	colExt		✓*	✓*	✗	✗
	allCol			✗	✗	✗	✗
	idtclPrfx	colExt		✓*	✓*	✗	✗
anyTarget	∃	allExt	inLeak	✓*	✓*	✓	✗

Table 4.9: SSH analysis

4.5.4.4 *Telegram*

As illustrated in Table 4.10, Telegram is secure even against very strong threat models, i.e. even with a very weak hash function. We only display the authentication lemma, but the secrecy one has similar results.

Threat Scenarios				Lemmas
OC	COL	LE	IL	trans_auth
frshTarget				✗
	allCol			✗
	fstPreImg,chsnPrfx	allExt	inLeak	✓

Table 4.10: Telegram analysis. trans_auth refers to *transcript agreement*, i.e. do both parties agree on the full transcript at the end of a session?

4.5.4.5 *Flickr*

The results are provided in Table 4.11. The previously known length-extension attack correspond to the threat model {hashExt}. Since the some of the hash function's inputs must remain confidential, we also found an attack under {inLeak}. This reveals a further assumption on the used hash function: the function is required to satisfy some form of confidentiality property such as PRF. (For instance, the identity function is collision and (second) preimage resistant but is not a PRF.) We also found an attack under {frshTarget}, where a hash output must remain confidential for the protocol to provide authentication. This again indicates a KDF assumption may be required. Finally, we found a first preimage attack under {fstPreImg,colExt}, which is a variant of the first attack: instead of providing an extension of the hash, we extend its preimage.

Threat Scenarios				Lemmas	
OC	COL	LE	IL	authenticate	authenticatePermissions
			inLeak	✓	✗
		hashExt		✓	✗
anyTarget	allCol	allExt	inLeak	✓	✗
	sndPreImg,chsnPrfx	colExt		✓	✓
	fstPreImg,chsnPrfx			✓	✓
	allCol			✓	✗
frshTarget				✓	✗
	fstPreImg	colExt		✓	✗

Table 4.11: Flickr analysis

Protocol	Lemma	ET Secure	EB Secure	ET time (s)	EB time (s)	Speed loss
Flickr	authenticate	✓	✓	0.19	0.19	1
Flickr	authenticatePermissions	✓	✓	0.19	0.29	2
Flickr	KeySecrecy	✓	✓	0.18	0.22	1
SSH	secrecy_key_A	✓	✓	1.76	2.64	2
SSH	secrecy_key_B	✓	✓	2.77	3.12	1
SSH	trans_auth	✓	✓	1.51	1.53	1
SSH	agree_keys_all	✓	✗	1.47	6.89	5
Telegram	t_auth	✓	✓	0.85	1.08	1
Telegram	t_secC	✓	✓	1.50	1.18	1
IKE_NoCookie	trans_auth	✓	✓	1.40	1.53	1
IKE_NoCookie	secrecy_key_A	✓	✓	1.97	2.47	1
IKE_NoCookie	secrecy_key_B	✓	✓	1.87	1.83	1
IKE_Cookie	trans_auth	✓	✓	1.84	1.75	1
IKE_Cookie	secrecy_key_A	✓	✓	2.86	2.15	1
IKE_Cookie	secrecy_key_B	✓	✓	2.76	10.84	4
Sigma	target_secA	✓	✓	0.65	2.90	4
Sigma	target_secB	✓	✓	0.62	0.70	1
Sigma	target_agree_B_to_A	✓	✓	0.56	2.56	5
Sigma	target_agree_A_to_B_or_Bbis	✓	✓	0.58	0.90	2

Table 4.12: Comparing efficiency of ROM modelings. ET is the classical Equational Theory based model (see Section 4.4.1) while EB is the Event Based one (Section 4.4.2). Note that the EB one also contains the associative symbol used for building transcripts, and security properties can become false. All results are in seconds, and correspond to average run time over four runs. ✓: security holds, ✗: attack found. As shown in the Table, the speed loss factor is reasonable given that the ET approach has severe limitations in terms of ability to capture interesting classes of attacks for other threat models than ROM (see Section 4.4.2).

4.5.5 Detailed Timings for Benchmarks

We provide in the following a set of benchmarks for our experiments. Each timing was obtained on a Intel(R) Xeon(R) CPU E5-4650L 0 @ 2.60GHz server with 756GB of RAM, on 8 threads for the Tamarin calls, while ProVerif is single threaded.

We first ran a test set to compare the efficiency of our new event based threat model (Section 4.4.2) to the classical equational based modeling. We executed all protocols Table 4.5 first with a perfect ROM model using a basic function symbol and non-associative concatenation, and then with the perfect ROM model part of our event based approach with associative concatenation. Overall, most protocols verify with almost exactly the same time, and some outliers take up to five times longer. Given the substantially increased expressivity, this is very encouraging. We provide detailed results in Table 4.12.

Overall about 5000 scenarios were verified in 150 minutes through 1600 Tamarin calls. Verification times vary substantially among protocols: most protocols only take a few minutes, while two particular models (SSH and IKE without a neutral Diffie-Hellman (DH) element) take around an hour to complete. Overall, our pruning strategy was very effective: about two thirds of the scenarios were not verified through a dedicated Tamarin call but directly implied by another one. We provide detailed results in Table 4.13.

Protocol	# Lemmas	With Short Timeout of 60 seconds			
		Runtime (s)	# Calls	# Avoided Calls	# timeouts
Flickr (table 4.11)	2	42.0	102	522	0
SSH (table 4.9)	4	497.0	541	707	372
TELEGRAM (table 4.10)	2	107.9	93	531	0
Sigma (table 4.6)	4	301.8	358	890	180
IKE_nocookie (table 4.8)	3	148.0	203	733	0
IKE (table 4.7)	3	115.6	161	775	13

Protocol	# Lemmas	With Long Timeout of 20 minutes		
		Runtime (s)	# Calls	# Avoided Calls
Flickr (table 4.11)	2	-	-	-
SSH (table 4.9)	4	3870.5	58	314
TELEGRAM (table 4.10)	2	-	-	-
Sigma (table 4.6)	4	1739.6	105	75
IKE_nocookie (table 4.8)	3	-	-	-
IKE (table 4.7)	3	-	-	-

Table 4.13: Benchmarking details, where each line corresponds to one of the table of section 4.5, with the given protocols and the corresponding number of lemmas verification. We first run the script with a timeout of 60 seconds for each individual Tamarin call and two threads. If there are timeouts, we then rerun them with a longer timeout of 20 minutes and 8 threads. We proceed this way in order to maximize the number of calls we can prune thanks to other calls with short timeout (60 seconds) and so that we can invest more threads onto the difficult cases, i.e. cases that timed out after 60 seconds and that we could not prune. We give for each of those two phases the total number of time it took to complete, the total number of actual calls to Tamarin, as well as the number of calls avoided thanks to the pruning strategy. The total run time for all those protocols and lemmas (including parallelization) was 220 minutes. Each timing corresponds to an average over four distinct runs.

5 Authenticated Encryption with Associated Data

Contents

5.1 Introduction

Authenticated Encryption (AE) and *Authenticated Encryption with Associated Data (AEAD)* are essential cryptographic tools used in secure communication. AEAD combines symmetric encryption with authentication to protect both the integrity and confidentiality of data. These encryption methods are widely used in securing internet traffic, including in protocols such as TLS, WPA2, WireGuard, Signal and WhatsApp. For instance, in the WireGuard protocol, AEAD encryption ensures that both the message payload and its associated metadata are authenticated, preventing attackers from tampering with or forging messages. However, while AEAD is fundamental to modern security, there is no single agreed-upon definition of what constitutes *strong* AEAD security.

The current security landscape for AEAD is complex, with many different proposed security frameworks and definitions [1, 2, 5, 19, 25, 26, 46, 59, 102, 122, 124, 158, 190, 202]. Some of these definitions have clear relationships with each other, while others are difficult to compare due to technical differences. In practice, weaknesses in AEAD schemes have led to real-world attacks. For example, attackers have exploited the misuse of AEAD mechanisms, such as the unintended reuse of a nonce – a number that should only be used once – to break encryption security [102, 131, 153]. Several such vulnerabilities have been discovered manually by analyzing protocols and understanding the specific AEAD scheme in use. Ideally, we want to formally prove that a given protocol, like WireGuard or WPA2, remains secure when using a specific AEAD algorithm, such as AES-GCM.

Modeling and detecting these vulnerabilities is challenging. A security analysis needs to be precise enough to capture the subtle behaviors of AEAD (such as how nonce reuse impacts security) while also scaling efficiently to analyze complex protocols. To address this, we present the first systematic method for automated security analysis of protocols that use AEAD by constructing fine-grained models of subtle AEAD differences and implementing them in the Tamarin prover (see Section 2.3.2). Our approach can both uncover attacks that exploit specific AEAD behaviors and verify the absence of such attacks.

One of the key challenges in developing our AEAD models is balancing theoretical security definitions, practical attack scenarios, and the need for automation. We identify the most critical aspects of AEAD security and use them to construct generic symbolic models suitable for automated analysis. When testing our methodology, we successfully rediscovered known attack categories, such as failures in *accountability* or *authentication*. Additionally, we uncovered a new attack class in group messaging scenarios, where a dishonest group member could craft a message that different recipients interpret in conflicting ways, which we coin *content agreement*.

Outline First, we provide the necessary background on AEADs in Section 5.2. We revisit the AEAD landscape and real-world attack patterns in Section 5.3. We then develop our symbolic modeling and analysis approach in Section 5.4. Lastly, we evaluate our approach on several real-world case studies and present them in more detail in Section 5.5.

5.2 Background

Authenticated encryption (with associated data) is a fundamental cryptographic primitive that ensures both the confidentiality and authenticity of messages. This extends traditional symmetric encryption by incorporating authentication mechanisms that verify both the encrypted data and any associated plaintext.

While it is theoretically possible to achieve these properties by combining symmetric encryption with message authentication codes (MACs), AEAD schemes offer a more optimized approach. AEADs integrate authentication and encryption into a single, efficient operation, ensuring that authentication tags are tightly coupled with ciphertexts to provide robust security guarantees.

The design of AEAD schemes presents significant challenges. A crucial property in encryption is that if a sender encrypts the same message multiple times, an adversary should not be able to recognize this from the ciphertexts. This is typically achieved by incorporating a unique value, known as a nonce, into each encryption operation. However, in practice, nonce reuse can occur due to implementation errors or protocol design flaws, which can severely compromise security. Even AEAD schemes that are provably secure under correct usage assumptions can be vulnerable when nonce reuse occurs, potentially leading to catastrophic failures.

Another practical consideration is that AEAD schemes, like other symmetric encryption methods, are often used in real-world applications that require incremental encryption and transmission of ciphertexts. Many AEAD APIs support decryption before authentication verification, allowing systems to process ciphertext fragments as they arrive. While this can improve efficiency, it also introduces security risks if not carefully managed.

The concept of AEAD was introduced by Rogaway [190] as a cryptographic mechanism that simultaneously guarantees privacy and authenticity for both message contents and associated metadata, such as headers. AEAD schemes typically follow a nonce-based approach, where the nonce is a ***n**umber used only **once***.

Nonce-based AEADs simplify security requirements compared to fully randomized or counter-based encryption schemes. By ensuring that nonces are never reused, AEADs can maintain both confidentiality and integrity. Additionally, Bellare and Hoang [25] initiated research into binding encryption keys and other auxiliary inputs to ciphertexts, further strengthening the security of AEAD schemes.

The structure of the remainder of this section is as follows: In Section 5.2.1, we formally define AEAD syntax and its standard security guarantees concerning privacy and integrity, before we examine critical attacks on cryptographic protocols that exploit subtle weaknesses in AEAD designs in Section 5.2.2. In Section 5.2.3, we categorize widely known AEAD frameworks from the existing cryptographic literature.

5.2.1 Formal AEAD Syntax and Core Properties

Notations Throughout the remainder of this chapter, all algorithms are implicitly parameterized by the security parameter. We use $s \in S$ that a variable s is part of the set S. Further, we use $s \leftarrow\!\!\$\ S$ to denote the uniform sampling of a variable s from a set S. Similarly, $x \leftarrow\!\!\$\ X$ represents the execution of a probabilistic algorithm X, with its output assigned to x. When the algorithm X is deterministic, we instead write $x \leftarrow X$.

We introduce $\perp$ as a special error symbol that does not belong to any set defined in this chapter. Additionally, we use _ to denote variables that are irrelevant to the discussion.

We focus on defining nonce-based AEAD schemes. This choice primarily simplifies our presentation, as AEAD schemes based on randomness or counters can be naturally expressed as special cases of nonce-based AEADs. All symbolic models developed in the latter parts of this chapter will be applicable to nonce-based AEADs. Consequently, they can be readily extended to capture both randomized and counter-based AEAD schemes.
To formally define a nonce-based AEAD we cite Rogaway [190] as follows:

Definition 1 ([190])**.** *Let* Key*,* Nonce*,* Header*,* $\mathsf{Message}$*,* $\mathsf{Ciphertext}$ *respectively denote the space of keys, nonces, headers (aka. associated data), messages, and ciphertexts. An authenticated encryption with associated data scheme* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a tuple of algorithms where*

- KGen *the key generation algorithm outputs a symmetric key* $k \in \mathsf{Key}$*, i.e.,* $k \leftarrow\!\$\ \mathsf{KGen}()$.
- Enc *the encryption algorithm inputs a key* $k \in \mathsf{Key}$*, a nonce* $N \in \mathsf{Nonce}$*, a header* $H \in \mathsf{Header}$*, and a message* m *and (deterministically) outputs a ciphertext* c*, i.e.,* $c \leftarrow \mathsf{Enc}(k, N, H, m)$.
- Dec *the decryption algorithm inputs a key* $k \in \mathsf{Key}$*, a nonce* $N \in \mathsf{Nonce}$*, a header* $H \in \mathsf{Header}$*, and a ciphertext* $c \in \mathsf{Ciphertext}$ *and deterministically outputs a message* $m \in \mathsf{Message} \cup \{\perp\}$*, i.e.,* $m \leftarrow \mathsf{Dec}(k, N, H, c)$.

In the following, we assume that for any nonce-based AEAD, the N, H and ciphertext c are public as they will be sent over the network. The correctness of the scheme ensures that decrypting a ciphertext using the same parameters N, H, k correctly retrieves the original plaintext; if decryption is performed with inputs outside their defined domains, it must return the special error symbol $\perp$. The two fundamental security properties of these schemes are integrity and privacy, as defined in [190].

Definition 2 (Privacy [190])**.** *We say an* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is* ϵ-$\mathsf{IND\$\text{-}CPA}$ *secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathrm{Expr}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD}}$ *experiment in fig. 5.1 is bounded by:*

$$\mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD}} := |\Pr[\mathrm{Expr}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD}}(\mathcal{A}) = 1] - \frac{1}{2}| \leq \epsilon$$

$\underline{\mathrm{Expr}^{\mathsf{IND\$\text{-}CPA}}_{\mathsf{AEAD}}:}$
1 $\mathsf{b} \leftarrow\!\$\ \{0,1\}$
2 $\mathcal{L}_c \leftarrow \emptyset$
3 $k \leftarrow\!\$\ \mathsf{KGen}()$
4 $\mathsf{b}' \leftarrow\!\$\ \mathcal{A}^{\mathrm{Enc}}()$
5 **return** $[\![\mathsf{b} = \mathsf{b}']\!]$

$\underline{\mathrm{Expr}^{\mathsf{IND\$\text{-}CCA}}_{\mathsf{AEAD}}:}$
1 $\mathsf{b} \leftarrow\!\$\ \{0,1\}$
2 $\mathcal{L}_c \leftarrow \emptyset$
3 $k \leftarrow\!\$\ \mathsf{KGen}()$
4 $\mathsf{b}' \leftarrow\!\$\ \mathcal{A}^{\mathrm{Enc},\mathrm{Dec}}()$
5 **return** $[\![\mathsf{b} = \mathsf{b}']\!]$

$\underline{\mathrm{Enc}(N, H, m):}$
6 **if** $(N, H, m, \_) \in \mathcal{L}_c$
7 **return** $\perp$
8 **if** $\mathsf{b} = 0$
9 $c \leftarrow \mathsf{Enc}(k, N, H, m)$
10 **else** $c \leftarrow\!\$\ \{0,1\}^{\ell(|m|)}$
11 $\mathcal{L}_c \leftarrow \mathcal{L}_c \cup \{(N, H, m, c)\}$
12 **return** c

$\underline{\mathrm{Dec}(N, H, c):}$
13 **if** $(N, H, \_, c) \in \mathcal{L}_c$
14 **return** $\perp$
15 $m \leftarrow \mathsf{Dec}(k, N, H, c)$
16 **if** $m \neq \perp$
17 $\mathcal{L}_c \leftarrow \mathcal{L}_c \cup \{(N, H, m, c)\}$
18 **return** m

Figure 5.1: $\mathsf{IND\$\text{-}CPA}$ and $\mathsf{IND\$\text{-}CCA}$ security for an $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ scheme. Included from [70][Fig. 1]

Definition 3 (Integrity [190])**.** *We say an* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *is* ϵ-$\mathsf{CTI\text{-}CPA}$ *secure, if the below defined advantage of any attacker* $\mathcal{A}$ *against* $\mathrm{Expr}^{\mathsf{CTI\text{-}CPA}}_{\mathsf{AEAD}}$ *experiment in fig. 5.2 is bounded by:*

$$\mathsf{Adv}^{\mathsf{CTI\text{-}CPA}}_{\mathsf{AEAD}} := \Pr[\mathrm{Expr}^{\mathsf{CTI\text{-}CPA}}_{\mathsf{AEAD}}(\mathcal{A}) = 1] \leq \epsilon$$

$\underline{\mathrm{Expr}^{\mathsf{CTI\text{-}CPA}}_{\mathsf{AEAD}}:}$
1 $\mathcal{L}_c \leftarrow \emptyset$
2 $k \leftarrow\!\$\ \mathsf{KGen}()$
3 $(N, H, c) \leftarrow\!\$\ \mathcal{A}^{\mathrm{Enc}}()$
4 **if** $c \in \mathcal{L}_c$
5 **return** 0
6 **return** $[\![\mathsf{Dec}(k, N, H, c) \neq \perp]\!]$

$\underline{\mathrm{Expr}^{\mathsf{CTI\text{-}CCA}}_{\mathsf{AEAD}}:}$
1 $\mathcal{L}_c \leftarrow \emptyset$
2 $k \leftarrow\!\$\ \mathsf{KGen}()$
3 $(N, H, c) \leftarrow\!\$\ \mathcal{A}^{\mathrm{Enc},\mathrm{Dec}}()$
4 **if** $c \in \mathcal{L}_c$
5 **return** 0
6 **return** $[\![\mathsf{Dec}(k, N, H, c) \neq \perp]\!]$

$\underline{\mathrm{Enc}(N, H, m):}$
7 $c \leftarrow \mathsf{Enc}(k, N, H, m)$
8 $\mathcal{L}_c \leftarrow \mathcal{L}_c \cup \{c\}$
9 **return** c

$\underline{\mathrm{Dec}(N, H, c):}$
10 **return** $\mathsf{Dec}(N, H, c)$

Figure 5.2: $\mathsf{CTI\text{-}CPA}$ and $\mathsf{CTI\text{-}CCA}$ security for an $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ scheme. Included from [70][Fig. 2]

Both for privacy and integrity, we can define two security variants, $\mathsf{IND\$\text{-}CCA}$ (see Figure 5.1) and $\mathsf{CTI\text{-}CCA}$ (see Figure 5.2). These variants differ based on whether the attacker has access to a decryption oracle during the experiment. For the fully detailed properties and the relationships between these properties, we refer the reader to [70] and [223].

5.2.2 Historical Real-World Protocol Attacks Exploiting AEADs

Beside the well-studied privacy and integrity, some recent attacks against practical application protocols suggest that the underlying AEADs need an evolution to meet stronger security guarantees. We identify six main classes of these protocol attacks that have occurred in the wild and briefly describe their high-level requirements.

A1 Nonce reuse attacks - While nonces are expected to be used only once, this can fail in practice for three main reasons. First, protocol designs might aim to establish nonces, but their complex state machines may hide edge cases in which they are in fact reused, as in e.g., WPA2 [214]. Second, the generation of nonces might involve external sources, which may be unreliable, e.g., YubiKey [153] or Trustzone [202]. Third, implementations may be flawed. For example, the Zerologon attack[222] notably exploited the fact that the nonce underlying the AES-CBF8 mode in Microsoft Netlogon protocol is a constant string of zero bits. The encryption of a block of zero bits equals to 0^{16} with probability 1/256 for any key k, breaking the authentication of windows servers.

A2 Padding oracle attacks [215] - Many AEADs and symmetric encryption schemes are constructed from block ciphers and require the length of input messages to be multiple of a fixed value. Messages whose length is not a multiple are extended before encryption using a so-called *padding scheme*. These can enable plaintext recovery attacks if the attacker has a way to determine if a ciphertext is correctly padded or not, e.g., through timing leaks or error messages. Padding oracle attacks have found on many protocols, including SSL [57], IPSec [90], and GPG [174].

A3 SSH fragmentation attacks [7] - SSH was designed for securing Internet traffic over the unstable channel, where ciphertext blocks in a packet might get lost. The length of a SSH packet is encrypted in its first block. If the number of delivered blocks is less than the length decrypted from the first ciphertext block, no ciphertext integrity is executed.

If an attacker can inject the first ciphertext block and observe the error message reported by the SSH connection, then the plaintext of the transmitted ciphertext can be recovered.

A4 Partitioning oracle attacks [158] - Some real-world applications do not sample the AEAD symmetric keys randomly but simply pick users' passwords. Thus, attackers might know a set of possible password candidates and perform brute-force attacks. Even worse, if attackers have access to a partitioning oracle, which tells whether the password of a ciphertext belongs to some known sets, then the password can be recovered exponentially faster.

In practice, attackers sometimes can obtain the partitioning oracle by observing the reply messages responding to a selected ciphertext. This causes the vulnerability of applications in the real world, such as Shadowsocks [158].

A5 Salamander attack [102] - The end-to-end secure messaging provides high security against the surveillance of the server but potentially prevents the server from blocking the abusive messages. To mitigate this, Facebook invents a abuse report mechanism that allows each user to report the received abusive messages from a claimed sender.

However, this mechanism turns out to be broken because a malicious sender could send a single encrypted attachment that would decrypt to both an abusive message and an innocent message under two distinct keys.

A6 Sframe attack [131] - An AEAD scheme authenticates the owners of a symmetric key of a ciphertext rather than the sender's identity. This is especially relevant for group communication, where an AEAD cannot use the shared group key to authenticate the specific sender. To provide sender authenticity in groups, while keeping low bandwidth cost, the IETF SFrame protocol v01 [182] requires senders to sign a portion of the AEAD ciphertext using digital signatures.

Unfortunately, the sender identity authenticity of SFrame mechanism turns out to be broken, since the underlying AEAD schemes, AES-CM-HMAC and AES-GCM, do not provide collision resistance for the unsigned portion. This means, a malicious group member holding the symmetric key can forge the unsigned portion of other group members' ciphertexts.

5.2.3 Theoretical AEAD Frameworks

Apart from the previously outlined classes of real-word attacks linked to AEADs, on the theoretical side, many variants of AEADs have been designed in the past twenty years following the seminal work from [190].

Each of those variants come with their own flavors of properties like, e.g., integrity, confidentiality, nonce-misuse resistance, or robustness, leading to dozens of distinct security definitions. Furthermore, these AEAD variants differ in functionality, with some enabling e.g. ciphertext fragmentation or nonce-hiding. We categorize the main differences between distinct AEAD variants as follows:

F1 - does each ciphertext (or a part of it) bind to a set of its encryption inputs? This question motivates the study of a novel (compactly) committing AEAD (ccAEAD, [70, Definition 6]) regime as well as various security properties, such as collision resistance ([70, Definition 9]), commitment ([70, Definition 10]), sender binding ([70, Definition 14]), and receiver binding ([70, Definition 15]) [2, 25, 102, 122]. Roughly speaking, the collision resistance prevents the collisions between AEAD encryption with different inputs. The commitment ensures that each valid AEAD decryption indicates the agreement on a subset of its encryption/decryption inputs. The sender- and receiver binding properties are relevant in the abuse-reporting scenarios. While the sender binding allows every ccAEAD receiver to report abusive messages, the receiver binding prevents malicious receivers from framing honest ccAEAD senders. The full definitions can be found in [70, Appendix B] - motivated by **A5**.

F2 - can we find collisions on valid decryption inputs for the same ciphertext? This question motivates the study of a novel property called robustness ([70, Definition 11])[1, 158]. Briefly speaking, robustness prevents attackers from having a single ciphertext decrypt to multiple distinct valid messages on different inputs. - motivated by **A4**, **A6**.

F3 - is the AEAD supporting fragmentation of the ciphertexts? That is, can we start decrypting chunks of data before having verified the whole ciphertext?

F4 - is the decryption atomic, or split into a decryption and an integrity check? [19] - motivated by **A2**.

F5 - is the AEAD nonce-hiding? That is, is the nonce explicitly needed for the decryption, or is it included and hidden inside the ciphertexts? [26, 59]

F6 - is the AEAD nonce-misuse resistant? [124] Must a nonce be used once strictly, or can repeat? - motivated by **A1**.

5.3 Generalizing Real-World AEAD (In)Security for Systematic Analysis

We identify three main causes for the protocol attacks:

- **A1** comes from a *misuse of nonces*.
- **A2** and **A3** from a *decryption misuse*, where the decryption is not atomic but performed in two steps, in which case we lose the integrity and privacy guarantees.
- **A4**, **A5**, and **A6** actually all stem from a lack of *collision-resistance*.

This leads us to summarizing the concrete security guarantees for AEADs in three categories:

- **privacy** and **integrity** - the core guarantees that we defined previously, and are expected to be met by all AEADs. This is what is lost under decryption misuse.
- **collision-resistance** - this guarantee hinders attackers from coming up with collisions over the output of Enc, i.e. find two distinct sets of inputs $\vec{i}_1$ and $\vec{i}_2$ such that $\mathsf{Enc}(\vec{i}_1) = \mathsf{Enc}(\vec{i}_2)$.
- **nonce-misuse resistance** - this guarantees that using a weak nonce twice or the same nonce for distinct message does not lead to a compromise.

With respect to those core properties, we provide in Table 5.1 the security and weaknesses of many widely deployed AEADs. In addition to the concrete constructions, we also provide in this table the generic constructions of AEAD such as *Encrypt-then-MAC (EtM)*, whose security guarantees depend on the concrete encryption and MAC algorithm instantiations. For the generic construction EtM, we distinguish two cases based on whether the encryption and MAC keys are related, e.g. derived from k with a key derivation function, or unrelated, e.g. simply the first and the second half of the input k.

Notably, while all of AEADs in the table do provide integrity and privacy (otherwise they would not be used), only some of them tolerate that a single nonce is reused twice for different messages. Moreover, we can also observe that the picture for collision-resistance is very disparate and many deployed schemes

Concrete AEAD	Integrity and Privacy	Full Collision Resistance	Nonce Misuse Resistance
XSalsa20-Poly1305	✦	✗ [2]	✗ Xor of plaintexts
AES-GCM	✓ [132, 167]	✗ [102]	✗ Forgeability + xor of plaintexts
ChaCha20-Poly1305	✓ [187]	✗ [2]	✗ Xor of plaintexts
OCB3	✓ [38, 152]	✗ [2]	✗ Forgeability + equality of blocks
EtM (unrelated keys)	✓ [190]	✗ [122]	✗ Encryption dependent
AES-CCM	✓ [115, 138]	✦	✗ Xor of plaintexts
AES-EAX	✓ [29, 173]	✦	✗ Xor of plaintexts
EtM (related keys)	✓ [190]	✓ [122]	✗ Encryption dependent
CAU-C4	✓ [25]	✓ [25]	✗ Forgeability + Xor of plaintexts
AES-GCM-SIV	✓ [124, 133]	✗ [2]	✓ [124]
CAU-SIV-C4	✓ [25]	✓ [25]	✓ [25]

✓ : proven in the cited work(s). ✦ : we conjecture that this holds, but do not know of a proof.
✗ : does not hold, with reference or explanation of counterexample.

Table 5.1: AEADs (in)-security guarantees: *Integrity and Privacy* refers to IND$-CPA and CTI-CPA. *Full Collision Resistance* refers to Definition 4. For *Nonce Misuse Resistance* we indicate the potential impact of reusing nonces if the AEAD scheme does not have this property. Adapted from [223][Table 4.1]

do not meet it. In the remainder of this section, we recall collision resistance and its relation to other properties of AEADs, before having a look at practical attacks on AEADs based on the absence of collision resistance.

5.3.1 Generalizing AEAD Collision Resistance and Relations

This section builds on the results of Zhao [223] and provides a summary of their key findings.

We adopt their formulation of full collision resistance based on the CMT-4 definition from [25], which we recall below. Informally, full collision resistance ensures that each AEAD ciphertext can be generated from a unique set of inputs.

While this may seem like a strong requirement, its absence can lead to unexpected vulnerabilities in certain protocols. As we will see, both known attacks and our case studies highlight that, despite its strictness, full collision resistance remains a meaningful and desirable security property.

Definition 4 (Full Collision Resistance)**.** *We say an* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *has* ϵ*-full collision resistance (or* ϵ-full-CR *), if the below defined advantage of any attacker* $\mathcal{A}$ *against the* $\mathrm{Expr}^{\mathsf{full\text{-}CR}}_{\mathsf{AEAD}}$ *experiment in fig. 5.3 is bounded by*

$$\mathsf{Adv}^{\mathsf{full\text{-}CR}}_{\mathsf{AEAD}} := \Pr[\mathrm{Expr}^{\mathsf{full\text{-}CR}}_{\mathsf{AEAD}}(\mathcal{A}) = 1] \leq \epsilon$$

$\underline{\mathrm{Expr}^{\mathsf{full\text{-}CR}}_{\mathsf{AEAD}}}$:

1 $\Big((k_1, N_1, H_1, m_1), (k_2, N_2, H_2, m_2)\Big) \leftarrow\!\$\ \mathcal{A}()$

2 **if** $\perp \in \{k_1, N_1, H_1, m_1, k_2, N_2, H_2, m_2\}$ **or** $(k_1, N_1, H_1, m_1) = (k_2, N_2, H_2, m_2)$

3 **return** 0

4 $c_1 \leftarrow \mathsf{Enc}(k_1, N_1, H_1, m_1)$, $c_2 \leftarrow \mathsf{Enc}(k_2, N_2, H_2, m_2)$

5 **return** $[\![c_1 = c_2]\!]$

Figure 5.3: full-CR security for an $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$.

Relationship with existing frameworks This notion of collision resistance, though straightforward, is sufficient to encompass various definitions found in the literature, including those from [2, 25, 112, 122, 158]. Informally, these definitions aim to address the following questions:

tidyness - for a fixed key, is the encryption function the inverse of the decryption one?

commitment (CMT-l **and** CMTD-l **for** $l \in \{1, 3, 4\}$ **[25]**) - can we find collisions either over the encryption or the decryption, with different parts of the inputs being allowed to stay fixed based on l? [1]

full robustness (FROB **[112]**) **and** ***even fuller robustness*** (eFROB **[122]**) - is any attacker able to compute a ciphertext that decrypts correctly under two distinct inputs?

key committing KC **security [2]** - is any attacker able to compute a ciphertext that decrypts correctly under different keys but same nonce?

multi-key collision resistance (MKCR) **[158]** - is any attacker able to compute a ciphertext that decrypts correctly under multiple keys but same nonce and header?

receiver binding (r-BIND) **[122]** - is any attacker able to compute a ciphertext that can be verified under the different header and message?

The illustration in Figure 5.4 summarizes the relations of all the properties listed above. For the definitions and the detailed proofs, we refer to [70, 223].

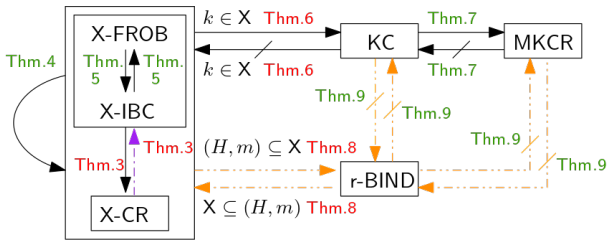


Figure 5.4: The relation between collision related properties for AEAD with key space Key. The black arrow $\rightarrow$ indicates the general implication. The purple dash-dotted arrow -·-▸ indicates the implication for tidy AEAD. The orange dash-dot-dotted arrow -··-▸ indicates the implication for ccAEAD[AEAD]. The X in the figure is a subset of (k, N, H, m), i.e., $\mathsf{X} \subseteq (k, N, H, m)$. The theorems highlighted with red color are claimed or proven in other papers. The theorems highlighted with green color are part of our third contribution. Included from [223][Fig. 4.1]

Collision attacks on deployed AEADs Any form of ciphertext collision can pose a security risk. We argue that general-purpose AEAD schemes should be fully resistant to such collisions. However, as summarized in Table 5.1, many widely deployed AEAD schemes fail to achieve full collision resistance. Below, we summarize known attacks against various notions of collision resistance in different AEAD schemes, as discussed in the literature:

- r-BIND - [122] presents generic attacks against any EtM construction with unrelated keys. In these attacks, an adversary can trivially alter the encryption key. The work also demonstrates attacks against real-world modes using Carter-Wegman MACs, such as GCM and ChaCha20-Poly1305, by crafting a specific final ciphertext block to create collisions. [102] describes a concrete attack against AES-GCM and OCB, leveraging nonce collisions and suggesting an accelerated attack via a birthday-bound approach on keys. Additionally, using a corollary of Theorem 1 from [192], [102] argues that this attack extends to any schemes that make one block cipher call per message block. This may indicate vulnerabilities in AES-GCM-SIV, ChaCha20-Poly1305, and other EtM constructions.
- KC: - [2] extends the attack from [102] on AES-GCM, demonstrating proof-of-concept attacks against several widely used AEAD schemes, including AES-GCM, ChaCha20-Poly1305, AES-GCM-SIV, and OCB3. The attack exploits ciphertext collisions under distinct keys. Moreover, [2] highlights real-world implications, including practical applications in binary polyglot scenarios.
- MKCR: - [158] introduces a novel partitioning oracle attack that effectively compromises MKCR security in widely deployed AEAD schemes, such as AES-GCM, AES-GCM-SIV, ChaCha20-Poly1305, and XSalsa20-Poly1305.
- X-CR and X-IBC: - [25] demonstrates that all the aforementioned attacks also break k-CR and k-IBC security for the corresponding AEAD schemes. Consequently, AES-GCM, AES-GCM-SIV,

[1] In this paper we rename CMT-l to *collision resistance* (X-CR). In particular, the full-CR in Definition 4 is identical to (k, N, H, m)-CR

XSalsa20-Poly1305, ChaCha20-Poly1305, and OCB are all k-CR insecure.

5.4 Symbolic Models for Automated Verification

We next describe how to, using the generalizations we developed in the previous section, develop symbolic models for AEADs that encompass many of the essential weaknesses from Section 5.2.2. For each essential weakness, we will develop a specific model that gives the attacker the capability to use the given weakness. Analyzing a protocol with those attacker capabilities enabled will then allow us to automatically find attacks on protocols that may rely on subtle AEAD weaknesses.
Our models cover:

- *collisions* Coll- covering **A4**, **A5**, **A6**, and definitions from **F1** and **F2**.
- *nonce reuse* NR- covering **A1** and **F6**.
- *decryption misuse* Forge- covering **A2**, **A3**, **F3** and **F4**.

Some modern protocols, like [102] or [182], rely on additional features of AEADs that we cover in a modular fashion:

- *explicit tag* Tag- for most AEADs, one can extract a verification tag from the ciphertext, needed to model protocols like [182] for **A6**.
- *explicit commit* Com- to extract a value from the ciphertext committing to the inputs of the encryption. Needed to model protocols like [102] covering **A5** and **F1**.

Collisions can then be lifted to the tag or the commit in a modular fashion, and are essentially only impacting on the complexity of mounting concrete attacks.

We additionally build a model Leak that provides an explicit capability to reveal the nonce used for encryption to the attacker. Not sending out the nonce by default but using a dedicated functionality allows accounting for nonce hiding AEADs covering **F5**. While we cannot claim completeness of our models w.r.t. to all possible AEAD weaknesses that may arise in the future, we provide a set of models based on our analysis of the real-world security of AEADs Section 5.3 that covers most practical attacks.

We develop and specify the previously enumerated models of AEADs in the *symbolic model* of cryptography, an abstract model used in the formal methods community to express and automate the analysis of cryptographic protocols (see Section 2.3). We then present symbolic models of the before-mentioned AEAD weaknesses in Section 5.4.1.

5.4.1 Symbolic AEAD Models

We first explicitly model all the input parameters, making the senc and sdec having four inputs, $\mathsf{senc}(k, n, h, m)$. Then, we model the multiple weaknesses previously discussed. While we focus on providing models for nonce-based AEADs, as it is the most fine-grained model of AEADs, it is easy to derive from them models for counter-based or randomized AEADs. They can typically be modelled by removing the explicit nonce as $\mathsf{senc}(k, h, m)$, and all equations or capabilites given in the following and that do not directly relate to the nonce can be transposed to this case.

Practical Collision models Coll We start by adding collision capabilities that match the known real-world collision capabilities. When using these models reports an attack on the protocol in one of the automated tools, we can then investigate its feasibility in practice based on the concrete AEAD used and the message encodings by referring to Section 5.3, and in particular Table 5.2.

We start with the capability that can be reasonably computed on many AEADs and was shown to be practical by [102] for Facebook's Message Franking protocol. As an example, consider the scenario where an attacker tries to produce some colliding ciphertexts given two keys. One option would be to brute-force over the nonce for a fixed header, e.g., an empty header. If successful, the attacker would have a ciphertext that could be decrypted to distinct plaintexts under a common nonce and header using the two keys. We model this nonce finding algorithm in the symbolic model as an additional function symbol c_n, modeling the colliding nonce, which will take as input all the given parameters the collision depends

on. We then add the collision model nColl:

$$\begin{aligned}&\mathsf{senc}(k_1, \mathrm{c}_n(k_1, k_2, h, m_1, m_2), h, m_1)\\ &= \mathsf{senc}(k_2, \mathrm{c}_n(k_1, k_2, h, m_1, m_2), h, m_2)\end{aligned} \quad \text{(nColl)}$$

Another widespread collision capability is captured by adding two function symbols c_k^1 and c_k^2 with the collision model KeysColl:

$$\begin{aligned}&\mathsf{senc}(\mathrm{c}_k^1(n, h, m_1, m_2), n, h, m_1)\\ &= \mathsf{senc}(\mathrm{c}_k^2(n, h, m_1, m_2), n, h, m_2)\end{aligned} \quad \text{(KeysColl)}$$

To check whether a potential attack found using KeysColl might be feasible, refer to Table 5.2. Whereas for KeysColl the attacker needs to produce a collision on the AEAD for a fixed nonce and header, the same kind of collision appears also to be feasible in the case where one of the keys is already fixed. We model this slight variation of KeysColl as well (KeyColl).

AEAD	nColl	KeysColl
AES-GCM	[102]	[2, 102, 122, 158]
AES-GCM-SIV	[102]	[2, 102, 122, 158]
ChaCha20-Poly1305	[102]	[2, 102, 122, 158]
Encrypt-then-MAC (EtM)	[102]	[102, 122]

Table 5.2: Effective attacks against collision resistance of several AEADs in the literature. nColl describe collisions where, for given keys and a header, the attacker uses brute-force over the nonce to produce colliding ciphertexts. In KeysColl, the attacker brute-forces, given a nonce and header, over the keys.
For the generic Encrypt-then-MAC paradigm we refer to the concrete attacks for CTR, OFB, CBC, and CFB modes to [70, Appendix A].

Notice that we, for instance, set that the two colliding encryptions may necessarily use the same nonce and header in those equations. This is caused by many existing protocols implementing that nonces and associated data can be computed independently by the parties, or that they cannot be sent out twice with distinct values.

Generic Collision models FullColl The previous collision models allow to efficiently check for the collisions that are most likely to be practical for existing AEADs and given their use in protocols. While obtaining an attack in those models is instantly interesting, we may miss some future practical attacks. Indeed, as illustrated by Table 5.1, most AEADs do not meet the full collision resistance property. As we in fact do not know if they meet any kind of collision resistance properties as no proofs exists for many of them, it is possible that in the coming years, new practical ways of building collisions on the existing AEADs are discovered. As such, from a security point of view, for any non collision resistant AEADs, it is prudent to consider that many more collisions are possible than currently practical.

Our approach makes it easy to define such a prudent model, and capture all attacks that may be possible on a protocol if the AEADs is not fully collision resistant. We do this by allowing more collisions and changing which part of the encryption input is fixed on both sides, and which part the attacker is brute-forcing over.

$$\begin{aligned}&\mathsf{senc}(k, n, h, m)\\ &= \mathsf{senc}(\mathsf{gen\text{-}c}_k(n, n_2, h, h_2, m, m_2), n_2, h_2, m_2)\end{aligned} \quad \text{(FullKeyColl)}$$

$$\begin{aligned}&\mathsf{senc}(k, n, h, m)\\ &= \mathsf{senc}(k_2, n_2, h_2, \mathsf{gen\text{-}c}_m(k, k_2, n, n_2, h, h_2))\end{aligned} \quad \text{(Full-mColl)}$$

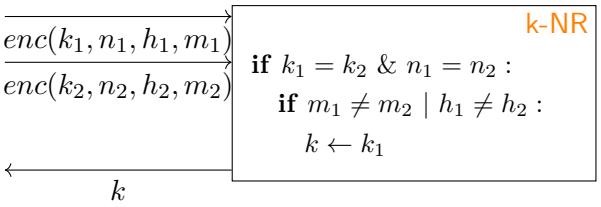
$$\begin{aligned}&\mathsf{senc}(k, n, h, m)\\ &= \mathsf{senc}(k_2, \mathsf{gen\text{-}c}_n(k, k_2, h, h_2, m, m_2), h_2, m_2)\end{aligned} \quad \text{(Full-nColl)}$$

$$\begin{aligned}&\mathsf{senc}(k, n, h, m)\\ &= \mathsf{senc}(k_2, n_2, \mathsf{gen\text{-}c}_h(k, k_2 n, n_2, m, m_2), m_2)\end{aligned} \quad \text{(Full-adColl)}$$

With FullKeyColl, Full-mColl, Full-nColl, and Full-adColl, we capture the capability of an attacker to find collisions by just finding one distinct k, n, h, or m, respectively. These models may cover collisions

that are impractical as their main purpose is to check whether the analyzed protocol relies on collision resistance of AEAD schemes. Indeed, if we get an attack in such a model, it implies that a strong collision resistance notion is needed to prove the security of the protocol in the computational model. Further, and as we see later in Section 5.5, some of these attack may even be practical and could not have been easily discovered in another way.

Nonce-reuse NR The nonce-reuse issue **A1** is slightly more complex to model, as we cannot capture it using an equation. We thus have to use a less classical way to model primitives: we model the attacker capability by providing access to an additional process, or oracle, that does the following:



In this oracle, the attacker can provide two ciphertexts. If those ciphertexts are encrypted under the same key and nonce but differ in either header of message, the attacker learns the secret encryption key. Similar to the collision model, Coll, we included a model of this process into our set of AEAD models and call it k-NR. This process models the strongest possible leak, namely leakage of the secret key. We can also make it more fine-grained by leaking, e.g., $m_1 \oplus m_2$ instead of k. As not all tools in the symbolic model provide support for exclusive-or like equations, we modeled an over-approximation m-NR, which leaks both m_1 and m_2 as an example. Note that with these kinds of oracle-like models, the concrete leaked values can be decided by the capabilities of the chosen tool and the actual weaknesses listed in Table 5.1.

Decryption Misuses Forge Some protocols, notably SSH, that allow for ciphertext fragmentation, also use AEADs in a non-recommended way by splitting the atomic sdec operation into verification and decryption. This may also be the case for protocols building their own AEAD based on the EtM construction. In such a case, instead of sdec that checks integrity, we use a weak decryption function w-dec and a verification function verify, with the equations:

$$\text{w-dec}(k, n, h, \mathsf{senc}(k, n, h, m)) = m$$
$$\text{verify}(\mathsf{senc}(k, n, h, m), k, n, h, m) = \text{true}$$

We model the fact that the decryption is weak by making decryption succeed on messages forged by the attacker using the forge algorithm:

$$\text{w-dec}(k_2, n_2, h_2, \text{forge}(\mathsf{senc}(k, n, h, m), m_2)) = m_2$$

Remark that a limitation of this Forge model is that the attacker cannot compute a valid ciphertext for some function of the message m, which is sometimes possible. Assume that for a given protocol we know that plaintexts are pairs of elements, denoted by $< x, y >$, we can also add dedicated forgery rules:

$$\text{w-dec}(k_2, n_2, h_2, \text{forge}_1(\mathsf{senc}(k, n, h, < m_1, m_2 >))) = m_1$$
$$\text{w-dec}(k_2, n_2, h_2, \text{forge}_2(\mathsf{senc}(k, n, h, < m_1, m_2 >))) = m_2$$

If the encryption is XOR based, the attacker should also be able to encrypt at this stage any XOR of a value to the ciphertext. This limitation notably implies that we cannot cover in general premature release of ciphertexts or the SSH fragmentation attacks. While we can do this for particular cases as illustrated, lifting this limitation generically in the symbolic model requires advances in the existing tools and symbolic techniques that we consider out of scope for this work.

Explicit Tag Tag Despite the recommendations, some protocols do not use AEADs only through a decryption and encryption API, but actually rely on some more low-level detail. For instance, schemes rely on the fact that the ciphertext is often a pair (encryption, tag), where the encryption is a basic symmetric encryption of the message and the tag is what provides the integrity. Instead of going to such a low-level, which would be AEAD dependent, we capture this possibility modularly by adding a new

function symbol get_tag, that inputs ciphertext $\mathsf{senc}(k, n, h, m)$. We can then model collisions over the tags, by adding a variant of each of the previous collision equations over the tag, with, e.g., nTag being:

$$\begin{aligned}&\text{get\_tag}(\mathsf{senc}(k_1, \mathrm{c}_n(k_1, k_2, h, m_1, m_2), h, m_1))\\ =\ &\text{get\_tag}(\mathsf{senc}(k_2, \mathrm{c}_n(k_1, k_2, h, m_1, m_2), h, m_2))\end{aligned}$$

Reasoning about explicit tags allows for a top-down approach rather than bottom up. For example, it allows us to ignore implementation details, such as to which side of the encryption the tag is concatenated.

Explicit commitment Com We model compactly committing AEADs by adding a get-commit function symbol similar to the get_tag. Once again, this allows for a modular model of compactly committing AEADs, where we only specify that a commitment can be extracted, but do not specify how. This extraction can be combined with the collision capabilities to model non-committing AEADs, as a collision on the ciphertexts directly translates to a collision on the commitment. Modeling the fact that the collisions are only on the commitment in a more fine grained way would be possible, but would not yield better attack finding capabilities as they are covered by the ciphertexts collisions. Further, it appears that collisions on the ciphertext or the commitment only differ in the ease of mounting attacks, the commitment being smaller and easier to manipulate than the whole cipher.

Nonce-Leaking Leak Following **F5**, we capture that an AEAD may not hide the nonce by adding a nonce extraction function symbol get_nonce along with the needed equation:

$$\text{get\_nonce}(\mathsf{senc}(k, n, h, m)) = n$$

This equation can now be also used instead of sending the nonce to the network explicitly.

Concrete encodings for tools Previously, we presented general equations that can be used to capture multiple AEAD weaknesses in symbolic models. However, symbolic analysis tools often have restrictions on the type of equations that are supported. The most efficient class of equations that are supported by tools satisfy the so called *subterm convergence* property: the right hand side of the equation only contains terms that occur as subterms in the left-hand side. This is not the case for all equations we introduced previously. However, all of them can be expressed in an equivalent fashion using only subterm convergent equations. For instance, for the nColl equation the same attacker capability can be captured with the following equations, where we introduce a new function symbol ct as an encoding artifact:

$$\begin{aligned}&\mathsf{senc}(k_1, \mathrm{c}_n(\mathrm{ct}(k_1, k_2, h, m_1, m_2)), h, m_1)\\ &\qquad = \mathrm{ct}(k_1, k_2, h, m_1, m_2)\\ &\mathsf{senc}(k_2, \mathrm{c}_n(\mathrm{ct}(k_1, k_2, h, m_1, m_2)), h, m_2)\\ &\qquad = \mathrm{ct}(k_1, k_2, h, m_1, m_2)\end{aligned} \qquad \text{(n-subtermColl)}$$

5.5 Case Studies

We demonstrate our symbolic models for automated verification on a set of eight protocols, classified into four categories depending on the analyzed security property:

- **Key Secrecy** – rediscovering the attack on *YubiHSM* [221]
- **Authentication** – rediscovering the attack on *SFrame* [182]
- **Accountability** – rediscovering the attacks on the accountability of the Facebook Message Franking mechanism [110] and finding that the Web Push [211] standard does not provide server accountability.
- **Content Agreement** – analysis of multiple group messaging and content delivery protocols, namely SaltPack [194], WhatsApp Groups [130], Scuttlebutt [201], and GPG [144].

We tested our methodology on a computing cluster with Intel® Xeon® Gold 6244 CPUs and 1TB RAM against all possible combinations of the threat models from Section 5.4. We automate this process using a Python program as described in Section 5.5.1. For each Tamarin call within our script we limit Tamarin to use 4 threads and set the timeout to 60 seconds per Tamarin call.

For the 8 case studies (plus 3 variants) we had a total evaluation time of 17 hours and 29 minutes with a total of 1404 Tamarin calls. We give an overview of the results in Table 5.3. We show an excerpt of the detailed results in Table 5.4, while all results are reproducible and can be found in GitHub [71].

Protocol	AEAD Scheme	Model	Analysis Results	Time (s)	Novel?	Status
YubiHSM [221]	AES-CCM	NR	Key secrecy attack	2	[153]	Fixed
SFrame [182]	AES-GCM, EtM CTR	Tag	Authentication attack	<1	[131]	Fixed
FB Message Franking [110]	AES-GCM	Coll	Content Agreement attack	8	[102]	Fixed
FB Message Franking [110]	AES-GCM	Coll	Framing attack	3	[102, 122]	Fixed
GPG SED [144]	PGP-CFB	Coll	No Content Agreement	<1	✓	Deprecated
GPG SEIPDv2 [144]	AES-OCB	Coll	No Content Agreement	<1	✓	Infeasible
Saltpack [194]	XSalsa20-Poly1305	Coll	No Content Agreement	8	✓	Infeasible
WebPush [211]	AES-GCM	Coll	Server Accountability	8	✓	Reported
WhatsApp [130]	EtM CBC	Coll	No Content Agreement	3	✓	Reported ‡
Scuttlebutt [201]	XSalsa20-Poly1305	Coll	No Content Agreement	3	✓	Reported *

* = Feasibility depends on the collision resistance of XSalsa20-Poly1305. See discussion in Section 5.5.6.
‡ = Reported to WhatsApp. Feasibility heavily relies on implementation details, which are not open source.

Table 5.3: Summary of the main analysis results from our case-studies, illustrating the generality of our models by rediscovering previous attacks and finding new subtleties. In each case, we give the threat model, the used AEAD scheme, the analysis result, as well as the time it took Tamarin to find it. We also give some additional notes on the status of the observation.

After presenting our automated analysis methodology in Section 5.5.1, we additionally explain how to choose the correct threat model for targeted analysis in Section 5.5.2. We then show our case studies from Section 5.5.3 to Section 5.5.6 and highlight our findings.

5.5.1 Automated Analysis Methodology

We now have a set of models to capture multiple weaknesses of AEADs. To analyze a protocol, the following steps should be followed with the symbolic tool of choice:

1. Verify the protocol in all possible threat models (malicious participants, AEADs weaknesses)
2. If there is an attack based on collisions or nonce misuse, check which AEAD the protocol is using and whether it has the corresponding weakness (Table 5.1);
3. If an attack is from KeyColl, KeysColl, or nColl, use Section 5.3.1 to check whether the use AEAD is non collision resistant. If it is not, check Table 5.2 to evaluate if the attack is practical or not.
4. If an attack is from one of the over-approximated capabilities FullKeyColl,Full-mColl,Full-nColl, Full-adColl, there are two consequences:
   - Collision Resistance is probably needed to prove the protocol computationally secure.
   - The attack may however be impractical, and one needs to check the trace to see if the attacker can have enough control over the ciphertext inputs to create a collision.

False attacks In the previously outlined methodology, we say that an attack found in a symbolic model may not be a true attack. To provide more details: during modeling, we sometimes on purpose overapproximate the possible AEAD weaknesses, both to completely rule out classes of attacks or detect subtle attacks. This may indeed lead to us finding "false attacks" that are possible on the design of the protocol but not on all possible implementations for all concrete primitives. In such cases, when by following the methodology we get an attack trace, we have to carefully inspect it, and identify precisely whether it could lead to an attack on the concrete protocol. This typically depends on details of the plaintext encoding used by the protocol (json, concatenation, lengths, etc).

Automated analysis setup We split the models from Section 5.4 into two general classes:

1. collisions and nonce misuse (Coll, NR, Leak)
2. explicit functionalities (Forge, Com, Tag)

Class 1) corresponds to a set of weaknesses that we can check on any protocol using an AEAD scheme. We build a library of those models and a script that verifies the security of a given protocol against those models. Class 2) only makes sense on protocols that do rely on some explicit functionality, like an explicit

commitment. Hence, we only model them in the relevant cases where the protocol relies on these explicit functionalities.

For our set of case studies, we want to explore their security guarantees against all our AEAD models. To do so, we implemented a Python script that for a given protocol, automatically executes Tamarin for all possible combinations of threat models, and provides a summary of the secure or insecure scenarios.

When doing this exhaustively, it would mean running Tamarin 2^{10} times for each case study of class 1) and up to 2^{19} times for class 2). We optimize the script by re-using strict implications of some of our models, e.g. FullKeyColl makes the attacker strictly more powerful than KeyColl. Additionally, some models can be restricted to not be used at the same time with others. This reduces the possible model combinations to 2^9 (and up to 2^{13} for class 2).) As this is still a huge number of calls, we can use the same implications mentioned before to do some dynamic pruning. The number of prunable model combinations can vary a lot depending on the case study. The total Tamarin calls that our script automatically made for our case studies can be found in Table 5.3. Using these implications is useful, both for dynamic pruning and to optimize the search, but also to provide a more compact view of the final results, only displaying non-redundant secure or insecure scenarios. Our script provides us with a summary of the security of a scheme, that can then be formatted in a table as illustrated in Table 5.4.

A limiting factor in our analysis is the run-time of the protocol models. As the problem of automatically analyzing protocol models is in general undecidable, running Tamarin could lead to non-termination. We deal with this possibility by introducing timeouts into our experiments such that for each Tamarin call, we either find a proof, a potential attack trace, or we have a timeout.

Using our technique, which automatically modifies the model for each of the AEAD model combinations, can lead to non-termination more easily, especially on fragile models that were manually tailored toward termination. We selected the value of the timeout depending on the run-time of the protocol model with the classic AEAD model in use.

As an exhaustive search might not be feasible for every future case study, in Section 5.5.2 we describe how one can correctly choose the right AEAD model for a certain protocol model.

5.5.2 Choosing the Correct AEAD Model

Whereas using the fully automated methodology from the previous section covers all AEAD models, it can be out-of-scope for complex and detailed protocol models. As complex protocol models often need manual work to aid automation, it might be more feasible to a priori choose the correct AEAD model for the instantiations actually used in the protocol. We demonstrate a way to choose the right combinations of AEAD models on the example of a toy protocol using AES-GCM. Assume that the protocol explicitly adds the functionality that compares the tag instead of using authenticated decryption of the ciphertext:

1. As a first step check whether your protocol specification forbids sending the nonce used for AES-GCM. If no, add Leak to you AEAD model combination.
2. Check Table 5.1 and see if the the AEAD is resistant to nonce-reuse attacks. For AES-GCM we see that an XOR of plaintexts can be leaked and there is the possibility to forge ciphertexts. Here, add k-NR to the AEAD models. As this is an over-approximation of the before-mentioned weakness, you can also decide to instead of leaking the encryption key, to leak the XOR of plaintext (if your tool of choice allows modeling of XOR) or to output a forged ciphertext under the given key.
3. When checking Table 5.1 again, AES-GCM is not collision resistant. Then we check Table 5.2 and see that AES-GCM is also vulnerable to collisions of type KeysColl (,KeyColl), and nColl. As KeyColl is strictly stronger than KeysColl, we only need to add KeyColl and nColl to the set of combinations. However, if we would like to future proof the protocol (and we know that AES-GCM is not collision-resistant) we could also decide to add the strongest collision models, e.g. FullKeyColl, instead. With this, we could see if the protocol relies on collision resistant AEADs.
4. As the described protocol explicitly uses AES-GCM tags we would also add the Tag models. As collisions on tags are as hard or even easier than finding collisions on the AEAD scheme itself, we would recommend to use at least the same kind of collision types for tags as well, for instance FullKeyTag.

Protocol	Threat Model	Content Agreement
GPG SED	Full-mColl ∧ Full-nColl ∧ Full-adColl ∧ Forge ∧ k-NR ∧ m-NR ∧ Leak	✓
	KeyColl	✗
GPG SEIPDv2	FullKeyColl ∧ Full-nColl ∧ Full-adColl ∧ Forge ∧ k-NR ∧ m-NR ∧ Leak	✓
	Full-mColl	✗
Scuttlebutt	Full-adColl ∧ Forge ∧ k-NR ∧ m-NR ∧ Leak	✓
	KeysColl ∨ Full-mColl ∨ nColl	✗

Table 5.4: Example of how our methodology can, given a protocol and a security property, automatically establish the minimal requirements on the AEAD guarantees for the property to hold. We achieve this by analyzing all possible AEAD models, here applied to content agreement for GPG SED, GPG SEIPDv2, and Scuttlebutt. For each analyzed protocol, we obtain the strongest combination of AEAD models under which content agreement holds (✓), which directly yields minimal requirements on the AEAD. The weakest combinations of AEAD models under which a potential violation of the target property (here content agreement) is found is marked with (✗).

5.5.3 Key Secrecy

YubiHSM The YubiHSM [221] is hardware security module by Yubico to generate, store and manage cryptographic key material. It implements an API to strictly separate key usage from its applications, to mainly prevent full or partial leakage of secure key material.

In earlier versions of the YubiHSM, [153] found an attack on the YubiHSM API to leak secret keys by exploiting the ability of the user to specify nonces. With the underlying AEAD not being nonce-reuse resistant, they were able to leak secret keys.

By now, this very nonce-reuse issue is known and well-studied throughout the community. However, just recently, Samsungs Trustzone [202] was found to have the same kind of attack, demonstrating that nonce misuse is still worth paying attention towards.

When instantiating the original Tamarin model by [153] with our AEAD library, we efficiently rediscover the attack using k-NR.

5.5.4 Authentication

SFrame SFrame is a communication protocol developed by CoSMo Software and Google [182] with the goal to be used for online audio and video meeting protocols. It uses end-to-end encryption and is made to support groups of multiple users.

[131] found forgery attacks on the authentication of the SFrame protocol. For a malicious user of the protocol, who is part of a group, it is enough to find collisions on the authentication tags of the used AEAD to break authentication of the messages. This is mainly possible by only explicitly authenticating on the tags of AEADs instead of the full ciphertexts. They reported the attack to be practical on:

1. schemes with short tag length, or
2. schemes that allow to easily find collisions with key knowledge.

We modeled the SFrame protocol with its groups and the sending and receiving of frames. We modeled it against an attacker with the power to join groups and the power to act as a group participant. Using our extended AEAD models, which allows adding explicit tags, Tamarin could quickly find the reported attack by using collisions under the tags.

When exploring all possible scenarios of our AEAD models, Tamarin also found a potential attack on the same authentication property as in the original attack. Executing it would require to produce a full AEAD ciphertext collision (Full-mColl & Full-nColl) instead of a collision on the tags. However, this attack does not appear to be practical and directly implies collision of tags as a collision on the whole ciphertext is computationally harder.

5.5.5 Accountability

Facebook Message Franking In the setting of End-to-End encryption, reporting the abusive behavior of users seems hard to achieve without weakening security guarantees. In 2016, Facebook introduced *Message Franking* [110] to allow reporting of offensive message attachments. The idea is for a recipient of a malicious message attachment to use a cryptographically sound way to prove that it was sent by a specific sender.

[102] found an attack against Facebook's message franking mechanism in 2018. The practical attack they demonstrated involved finding a collision on the used AEAD's ciphertext. As the sender in this scenario was able to choose the cryptographic keys, messages, and the nonce, they showed how to compute two keys k_1 and k_2, two message attachments (for which one is the malicious one) m_1 and m_2, and a nonce n, such that the encryption of m_1 under n and k_1 leads to the same ciphertext as the encryption of m_2 under k_2 and n.

After reporting this attack to Facebook, Facebook immediately patched it. That attack demonstrates the practicality and the impact of collision attacks on real-world schemes.

To show that such attacks can be found on the design level by our analysis, we modeled Facebook's Message Franking mechanism in Tamarin. In the initial setting, with the attacker being a malicious sender, we could automatically find the reported attack in a few seconds using the KeysColl model.

In addition to analyzing the property violated by the initial attack – can a malicious sender avoid detection? – we also studied the converse property – can a malicious receiver create a fake report? The converse property got first reported as a concern by [122].

We thus additionally model a malicious receiver that tries to report an honest user. In this threat model, we, therefore, look at frameability properties. Being able to frame another party can be severe in practice, for instance, by falsely accusing another person of having sent illegal material. After testing our AEAD models against it, we could re-discover a potential attack [122] on the previously property. However, this attack would require finding a collision on the ciphertext for which one key, the nonce, and the ciphertext itself need to be fixed. Unless further weaknesses of AEADs are found, in this particular case over AES-GCM, this attack is, as of now, impractical.

Web Push The Web Push protocol provides means for a server to push notifications to clients by depositing an encrypted notification to the push service that will be fetched by the client when they go online. Web Push is standardized at IETF [211], and, for instance, Apple is planning to integrate it into its ecosystem.

Web Push aims to provide confidential push notifications from a server to its users and to ensure certain privacy properties, like the the unlinkability of unique identifiers through the push notification content. Given the wide array of possible applications and concrete use cases, we consider it interesting to check whether the server is accountable or not: can a client prove to a third party that it received a particular push notification from a given server? In contexts where push notifications trigger important actions from a user, protecting users from malicious servers that would try to make the user act and then be able to claim never having done so is crticial. The importance of this guarantee would depend on the actual deployment and usage of Web Push; we are currently in an ongoing discussion with IETF on this point. To include this case in our threat model, we thus consider a malicious server controlled by the attacker and verify if it is possible to upload one notification that could be interpreted in two different ways, for instance, offensively or benignly.

Our analysis reports that this guarantee does not hold w.r.t. Full-mColl: a single notification can be decrypted validly to two different plaintexts, depending on whether we use the current or deprecated public key of the user. As Full-mColl is a strong over-approximation an attack first seemed impractical. After manual inspection of the counterexample trace given by Tamarin, we could see that this theoretical attack carries over to the real world: WebPush relies on AES-GCM, and we can then reuse similar techniques as for Facebook Message Franking attack: concretely, an attacker can brute-force over the salt used to produce a nonce/key pair to encrypt the message to find a collision over the unauthenticated part of the ciphertext, and then inject at the end of the ciphertext the needed block to create a collision over the tag. The practicality of the attack depends on the encoding of the plaintexts, and the severity depends on whether the server being not accountable is critical given the use case.

5.5.6 Content Agreement

We focus now on analyzing the design of multiple messaging mechanisms. We study them in the multiple-recipient setting, trying to answer the following question: *Can a dishonest member of the group send a single message that will be read differently by some recipients?* This question leads us to analyze Content Agreement in the following contexts:

- end-to-end encrypted group messaging applications, like WhatsApp or Signal, or
- dedicated encrypted message mechanism, like GPG, Saltpack, or Scuttlebutt.

Our study reveals that there is a discrepancy between existing guarantees, which we summarize in Table 5.5.

Protocol	Content Agreement with CR	Content Agreement without CR	Notes
WhatsApp	✓	✗	Practicality depends on plaintext encodings
Scuttlebutt	✓	✗	Practical
GPG SED (to be deprecated)	✓	✗	Practical
GPG SEIPD v1/v2	✓	✓	Only theoretical attacks
SaltPack	✓	✓	Only theoretical attacks
Signal	✗	✗	Pairwise channels, hence no content agreement

Table 5.5: Content Agreement summary, with and without Collision Resistance (CR): for a set of group messaging applications and multiple recipients message sending mechanism, we summarize whether a given message can yield to different message for multiple users. In this table, we mention that the Signal application does not meet consistency as a side-remark: as Signal uses pairwise channels to send messages in groups, a different message can be sent to each member of the group.

WhatsApp groups We model the design of WhatsApp's group messaging. Because the code of WhatsApp is not available, we constructed our model based on the available information provided in its whitepaper [130, p. 10]. While it relies on the Signal X3DH protocol to establish pairwise channels between the members of the groups, sending a message is slightly different:

- The sender generates a so called *sender key* (also part of the Signal library), and sends this key to each participant over the corresponding pairwise channel;
- To send a message, there is then a single encrypted payload which is uploaded to the server.

While content agreement is trivially broken in Signal itself because of the pairwise channels, it could intuitively be expected within the setting of group messaging. It is however not guaranteed, as reported by our model. Our model captures a group of three people, where one of them is the attacker. We then aim to verify that a given message uploaded to the server will yield the same plaintext for all group members. Our automated analysis reports an attack on this property when enabling ciphertext collisions under KeyColl.

The group messaging mechanism relies on an AES-CBC encryption which is then signed with an independent key. This is similar to the Encrypt-then-Mac with unrelated keys scenario. It means that the complexity of mounting an attack in practice is equivalent to the complexity of finding meaningful collisions over AES-CBC. We have seen that with the current capabilities, this strongly depends on the concrete encoding of plaintexts, and whether we can find so-called polyglot plaintexts [2]. As WhatsApp is closed source, verifying the practicality of the attack would require to reverse engineer the full message encoding, which we consider out of scope for this paper. However, given the variety of possible message contents (notification, GIFs, media, React, ...) it is likely that the encoding would be loose enough to carry out the attack.

GPG GPG is the golden standard for file and mail encryption and signing. We review its ongoing cryptographic update [144]. It contains three different encryption formats:

- *Symmetrically Encrypted Data* (SED) – the legacy encryption with the dedicated GPG-CFB encryption mode. It is now marked as deprecated, and accepting such messages must raise a warning.

- *Version 1 Symmetric Encrypted Integrity Protected Data* (SEIPD v1) – the legacy authenticated encryption format, not deprecated.
- SEIPD v2 – the current proposal relying on AEADs.

SED is not integrity protected, and is simply a symmetric encryption with a dedicated mode. SEIPD v1 is a variant, still relying on a dedicated encryption mode, and given the plaintext p, returns the encryption of $p \| SHA1(p)$ (abstracting away some message formatting). SEIPD v2 relies on AEADs, with the specificity that the plaintext can be split into chunks, each chunk corresponding to a call with the same AEAD and same key but different nonces. A final specific tag is always appended to the ciphertext by computing a final AEAD over a *null* plaintext with the same key and a nonce depending on the number of chunks.

We modeled all three encryption modes, checking if an attacker can send the same message to different recipients. In all cases, our automated analysis reported attacks, but under different collision capabilities. That gives us the following practical consequences:

- SED is trivially broken, and even more so as we showed that collisions over the dedicated mode can be found in constant time ([70, Appendix A]). The severity is however low as it is to be deprecated.
- We find that for a non collision resistant encryption, SEIPD v1 is theoretically broken, but appears to be secure in practice. While the attacker can brute force the keys to try to come up with collisions, the SHA1 value appended to the plaintext puts too many constraints over the collision. This indicates however that if e.g. weak keys were found for AES, this could be attacked.
- The results are similar for SEIPD v2. By appending an additional call to the AEAD (either GCM, EAX or OCB) with the same key, it implies that in practice, one would need to find not one collision, but a pair of collisions, which greatly increases the complexity and makes the attack impractical. Going back to the variants of known collision that are practical (Table 5.2), finding two collisions at the same time has not been explored, and is for the moment an open question. As such, we consider this to not be possible in practice as of now.

Scuttlebutt Scuttlebutt is a protocol that provides an authenticated append-only feed. The private box feature [201] allows publishing encrypted messages that are uploaded in public and meant for multiple recipients. In this case, Content Agreement appears to be valuable and intuitively expected.

We model the mechanism with a malicious sender, and Tamarin does report an attack under the collision models KeysColl and nColl. Scuttlebutt uses the XSalsa20-Poly1305 AEAD, a variant of ChaCha20-Poly1305 (which is in Table 5.2). Hence, this attack appears to be feasible under the condition that the known attacks against ChaCha20-Poly1305 can be translated to XSalsa20-Poly1305.

SaltPack SaltPack is a proposed alternative to GPG. We modeled the surprisingly involved version 2 format [194].

Nonces and integrity packet checks are derived with multiple iterations of MACs and hashes. Notably, after a fresh payload key k has been asymmetrically encrypted for each of the recipients public key, a MAC is computed for each recipient. The payload key k is used both to encrypt the desired plaintext $\mathsf{Enc}(k, N_1, \emptyset, m)$, but also the sender public key spk with $\mathsf{Enc}(k, N_2, \emptyset, spk)$. We use $\emptyset$ to denote empty headers.

On this protocol, Tamarin does report an attack with Full-mColl. Intuitively, it seems that the scheme ensures consistency, as we need once again to come up with a pair of collisions for the desired plaintext m_1, m_2:

$$\mathsf{Enc}(k_1, N_1, \emptyset, m_2) = \mathsf{Enc}(k_2, N_1, \emptyset, m_1)$$
$$\mathsf{Enc}(k_1, N_2, \emptyset, spk) = \mathsf{Enc}(k_2, N_2, \emptyset, spk)$$

This is a similar kind of collisions that is required for the GPG case, and is once again not possible in practice with respect to the current techniques.

Key Encapsulation Mechanisms

6

Contents

6.1 Introduction

A Key Encapsulation Mechanism (KEM) [68] is a common building block in security protocols and cryptographic primitives such as hybrid encryption. Intuitively, a KEM can be seen as a specialized version of Public Key Encryption (PKE) that, instead of encrypting a payload message, specifically serves to generate and share a symmetric key between sender and recipient. During the last decade, many post-quantum secure KEMs have been proposed, see e.g., [3, 10, 24, 31, 47, 48, 85, 128, 171, 179]. This has made KEMs a prime candidate to replace Diffie-Hellman constructions, for which no practical post-quantum secure scheme is currently available.

The traditional security notion for a KEM is a version of IND-CCA that is directly inherited from its related Public Key Encryption (PKE) notion. Intuitively, a KEM is (IND-CCA) secure if, given a ciphertext, an adversary that does not have the corresponding private key cannot tell the difference between a random key and the encapsulated key. Additionally, robustness-like properties have been proposed for KEMs in [123], which similarly inherit from their PKE counterparts. Initially, "robustness" [1] was defined in the PKE setting as the difficulty of finding a ciphertext valid under two different encryption keys. Phrased differently, a PKE is robust if a ciphertext "binds to" (only decrypts under) one key.

In this part of the thesis, we set out to enable automated analysis of KEM-based security protocols that can take the differences between concrete KEMs into account. We first systematically explore the possible binding properties of KEMs. Our work is similar in spirit to explorations in the space of digital signatures [49, 79, 134, 186] and authenticated encryption [69, 102, 122, 157], where recent works have identified many desirable binding properties for these primitives that could have prevented real-world attacks.

Our systematic analysis leads to the formulation of several core binding properties for KEMs, with multiple variants. Whereas traditional KEM robustness properties only considered binding values to a specific ciphertext, we propose variants that bind values to a specific output key. We argue this is much more in line with viewing a KEM as a one-pass key exchange. Similarly, implicitly rejecting KEMs resemble implicitly authenticated key exchanges, where correct binding properties of the established key prevent classes of unknown key share attacks [40]. We relate our properties to properties previously reported in the literature, as well as related notions such as contributory behavior.

We provide a new hierarchy of our binding properties and use it to develop novel symbolic analysis models that reflect the binding differences between concrete KEMs. We implement our models in the framework of the Tamarin prover and apply the methodology to several case studies.

Notably, our automated analysis uncovers an attack on an example key exchange protocol in the Kyber documentation when instantiated with another KEM, which proves that the protocol design in fact relies on properties of the used KEM beyond just IND-CCA. We coin this type of attack a "re-encapsulation attack", as it relies on the adversary encapsulating keying material that it previously obtained from decapsulation, causing two ciphertexts to decapsulate to the same key. We also show how our novel properties can prove the absence of such attacks.

Outline We provide background and motivate our binding properties by the example of the re-encapsulation attack in Section 6.2, before developing our family of new security notions in Section 6.3. We then turn to developing our automated symbolic analysis in Section 6.4 and report on case studies in Section 6.5.

6.2 Background

A key encapsulation scheme [68] KEM consists of three algorithms (KeyGen, Encaps, Decaps). It is associated with a key space $\mathcal{K}$ and a ciphertext space $\mathcal{C}$. The probabilistic key-generation algorithm KeyGen creates a key pair $(\mathsf{pk}, \mathsf{sk})$ where pk is the public key and sk is the secret key. Given a public key pk as input, the probabilistic encapsulation algorithm Encaps returns a ciphertext $c \in \mathcal{C}$ and a key $k \in \mathcal{K}$. In this paper, we sometimes want to view Encaps as a deterministic algorithm with explicit randomness r, in which case we write $\mathsf{Encaps}(pk; r)$. To avoid ambiguity, we refer to k as the *output key* or the *shared secret*. The deterministic decapsulation algorithm Decaps uses a public key pk, a secret key sk, and a ciphertext $c \in \mathcal{C}$ to compute an output key $k \in \mathcal{K}$ or the error symbol $\perp$ that represents rejection. If decapsulation never returns $\perp$, we call KEM an *implicitly rejecting* KEM. Otherwise, we call it an *explicitly rejecting* KEM. We say that a KEM is ϵ-correct if for all $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{KeyGen}()$ and $(c, k) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk})$, it holds that $\mathbf{Pr}[\mathsf{Decaps}(\mathsf{sk}, c) \neq k] \leq \epsilon$.

The security of a KEM is defined through indistinguishability of the output key $k \in \mathcal{K}$ computed by Encaps against different adversaries. The standard security notion is resistance against a chosen-ciphertext attack (IND-CCA) [48, 203]. We recall the formal definition of the IND-CCA experiment shown in Figure 6.1.

$\text{IND-CCA}^{\mathsf{KEM}}_{\mathcal{A}}$:	$D(sk, pk, c)$:
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{KeyGen}()$	**if** $c \neq c_0$ **then**
$(c_0, k_0) \leftarrow\!\$\ \mathsf{Encaps}(\mathsf{pk})$	$k \leftarrow \mathsf{Decaps}(ct, sk)$
$k_1 \leftarrow\!\$\ \mathcal{K}$	**return** k
$b \leftarrow\!\$\ \{0, 1\}$	
$b' \leftarrow\!\$\ \mathcal{A}^{D(\mathsf{sk},\mathsf{pk},\cdot)}(c_0, k_b, pk)$	
return $b = b'$	

Figure 6.1: IND-CCA experiment for KEMs. Originally introduced in [68], we re-use syntax from [48].

First, the experiment creates a key pair $(\mathsf{sk}, \mathsf{pk})$ and encapsulates against the public key, returning (c_0, k_0). Next, it samples a random key k_1 from the key space and a random bit b. Then, the adversary $\mathcal{A}$ is given c_0, the key corresponding to the bit b, and pk, and outputs its guess b'. Finally, the adversary wins if they correctly guessed b, i.e., $b = b'$. During the experiment, the adversary has access to the decapsulation oracle $\mathsf{Decaps}(sk, \cdot)$, which returns the decapsulation of any ciphertext c except for the challenge ciphertext c_0.

6.2.1 Fujisaki-Okamoto (FO) Transform

A common construction for KEMs is the FO transform [116]. The FO transform can be used to turn any weakly secure (i.e., IND-CPA) public-key encryption scheme into a strongly (i.e., IND-CCA) secure KEM scheme by hashing a random message (and optionally other values) into an output key. Since the FO transform gives cryptographers a straightforward way to create a post-quantum secure KEM from a post-quantum secure PKE, these KEMs have surged in popularity and are now the de-facto standard post-quantum secure KEMs. All the finalists of the KEM NIST PQC [179] process are FO-KEMs.

6.2.2 Re-encapsulation Attacks

Our initial motivation for this work was to uncover the subtle difference in guarantees offered by different KEM designs and to analyze their impact on protocols. As we will see later, this leads to a hierarchy of new binding properties, which we used to build an automated analysis that discovered new attacks. Notably, Tamarin found instances of a class of attacks that we call *re-encapsulation attacks*. While re-encapsulation

attacks were not our original motivation, they clearly illustrate important binding properties that were not captured by previous security notions.

Intuitively, re-encapsulation attacks exploit the fact that for some KEMs, it is possible to decapsulate a ciphertext to an output key k and then produce a second ciphertext for a different public key that decapsulates to the same k. This can be possible even for robust IND-CCA KEMs since neither IND-CCA nor robustness prescribe that the output key binds a unique public key. At the protocol level, a re-encapsulation attack can typically manifest as an unknown-key-share attack, where two parties compute the same key despite disagreeing on their respective partners.

Unfortunately, strong robustness notions that already exist for KEMs do not prevent this class of attacks. Robustness properties reason about *a single ciphertext* c that should not decapsulate to the same key under different key pairs. However, a re-encapsulation attack revolves around two *different* ciphertexts: based on one party's ciphertext, the attacker creates a different ciphertext that decapsulates to the same key as the initial one by reusing the randomness. We refer to Section 6.5 for concrete examples of re-encapsulation attacks we discovered on, e.g., Kyber.

6.3 Generalizing New Security Notions for KEMs

We now turn to our first main objective: to establish a generic family of binding properties of KEMs. We first identify the elements that may be candidates for binding. The syntax of a KEM includes a long-term key pair, a ciphertext, and an (output) key. In some formalizations, the randomness of the KEM is made explicit, but we are looking for universal black-box notions that do not require us to know the internals of a KEM. With respect to the long-term key pair, we note that we want the guarantees to be relevant for both sender and recipient, which means we only consider the public key as the identifying aspect of the key pair. This leaves us with pk, ct, and k: we expect that for each invocation of the KEM's encapsulation with the same pk, the outputs ct and k would be unique.

We can thus wonder: if we have a specific instance of one of these, does that mean the others are uniquely determined? If we have a ciphertext, can it only be decapsulated by one key?

6.3.1 Design Choices

To define our notions, we make the following design decisions:

1. We consider the set of potential binding elements $BE = \{\mathsf{pk}, \mathsf{ct}, \mathsf{k}\}$.
2. We will consider if an instance of a set $P \subset BE$ "binds" some instance of another set of elements $Q \subset BE$ with respect to decapsulation with the KEM. Thus, "P binds Q" if for fixed instances of P there are no collisions in the instances of Q.
3. When using a KEM, pk is re-used in multiple encapsulations by design. Thus, pk does not bind any values on its own, and we hence exclude it from occurring in P alone. However, ciphertexts or keys might bind a public key pk, so it may occur in Q alone.
4. Adding multiple elements in the set Q corresponds to a logical "and" of the singleton versions, i.e., we have that P binds $\{q_1, \ldots, q_n\}$ iff for all $i \in [n]$. P binds $\{q_i\}$. We therefore choose to focus on the core properties, i.e., with $|Q| = 1$.
5. We require P and Q to be disjoint: elements that would occur on both sides are trivially bound. Additionally, we require both P and Q to be non-empty.
6. For all of our properties, we will consider honest variants (i.e., the involved key pairs are output by the key generation algorithm of the KEM), leakage variants (i.e., the involved key pairs are output by the key generation algorithm of the KEM and then leaked to the adversary), and malicious variants (i.e., the adversary can create the key pairs any way they like in addition to the key generation).

Based on the above choices, we have five choices for P. We refer to this set of choices as $\mathbb{P} = \big\{\{\mathsf{k}\}, \{\mathsf{ct}\}, \{\mathsf{k}, \mathsf{ct}\}, \{\mathsf{pk}, \mathsf{k}\}, \{\mathsf{pk}, \mathsf{ct}\}\big\}$. For Q, we can choose from the set $\mathbb{Q} = \big\{\{\mathsf{pk}\}, \{\mathsf{k}\}, \{\mathsf{ct}\}\big\}$. Without disjointness this would yield 5×3 options, but since we require the sets to be disjoint, this yields seven combinations.

ID	P	Q	Property	Explanation	Relation to existing notions
1	$\{\mathsf{k}\}$	$\{\mathsf{ct}\}$	*X-BIND-K-CT*	Output key binds the ciphertext.	
2	$\{\mathsf{k}\}$	$\{\mathsf{pk}\}$	*X-BIND-K-PK*	Output key binds the public key.	
3	$\{\mathsf{ct}\}$	$\{\mathsf{k}\}$	*X-BIND-CT-K*	Ciphertext binds the output key.	
4	$\{\mathsf{ct}\}$	$\{\mathsf{pk}\}$	*X-BIND-CT-PK*	Ciphertext binds the public key.	*HON-BIND-CT-PK* is equivalent to SROB [123].
5	$\{\mathsf{k},\mathsf{ct}\}$	$\{\mathsf{pk}\}$	*X-BIND-K,CT-PK*	Together, the output key and ciphertext bind the public key.	*HON-BIND-K,CT-PK* is equivalent to SCFR [123].
6	$\{\mathsf{k},\mathsf{pk}\}$	$\{\mathsf{ct}\}$	*X-BIND-K,PK-CT*	Together, the public key and the output key bind the ciphertext.	*LEAK-BIND-K,PK-CT* is equivalent to CCR [16].

Table 6.1: The six core instantiations of our generic binding property *X-BIND-P-Q* before choosing $X \in \{HON, LEAK, MAL\}$.

One of these seven cases is the case where $P = \{\mathsf{pk},\mathsf{ct}\}$ and $Q = \{\mathsf{k}\}$. This property holds when the public key pair and the ciphertext, which are the inputs to Decaps, bind the output key. If Decaps is deterministic, this is trivially true. We will therefore not consider this case in the remainder of the paper, leaving us with six combinations that we will investigate further, which we show in Table 6.1.

6.3.2 Naming Conventions

Naming security notions is hard; once names are fixed, they tend to stick around for (too) long. We opt here for clarity and being descriptive at the cost of some verbosity. In the literature, it is more common to collapse all of these properties into "robustness" or "collision-freeness", but this becomes very ambiguous because one can imagine many subtle variants, depending on the exact robust/collision-free element in the construction. This has lead to a long list of non-descriptive names in the literature, including: Robustness, Fuller Robustness (FROB), even Fuller Robustness (eFROB), CROB, KROB, SROB, USROB, WROB, XROB, SCFR, WCFR, CCR, etc. In contrast, we illustrate our descriptive naming scheme for our binding properties in Figure 6.2.

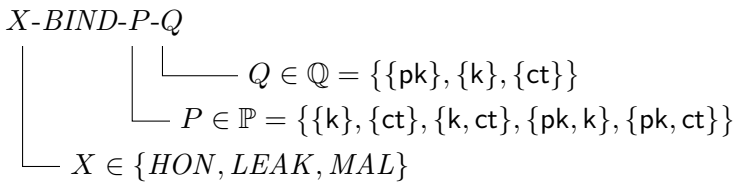


Figure 6.2: Design space and naming conventions for our security properties: For a KEM scheme that is *X-BIND-P-Q* secure, we say that "P [honestly|leak|maliciously] binds Q", using "honestly" when $X = HON$, "leak" when $X = LEAK$, and "maliciously" when $X = MAL$. We commonly omit set brackets in the notation when clear from the context, and we use uppercase for all characters. For example, *HON-BIND-CT-PK* corresponds to "ct honestly binds pk."

6.3.3 Generic Binding Notions of KEMs

We now introduce the generic security notion for our class of binding properties. In Figure 6.3 we show the generic game for $X \in \{HON, LEAK\}$, and in Figure 6.4 we show the game when $X = MAL$.

Definition 5. *Let* KEM *be a key encapsulation mechanism. Let* $X \in \{HON, LEAK, MAL\}$*, let* $P \in \mathbb{P}$ *and* $Q \in \mathbb{Q}$ *such that* $P \cap Q = \emptyset$*. We say that* KEM *is X-BIND-P-Q-secure iff for any PPT adversary* $\mathcal{A}$*, the probability that X-BIND-P-Q*$^{\mathsf{KEM}}_{\mathcal{A}}$ *returns 1 (true) is negligible.*

In our definitions, $X \in \{HON, LEAK, MAL\}$ indicates the adversary's control over the considered

$X\text{-}BIND\text{-}P\text{-}Q^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$
if $\mathsf{pk} \in Q$: $b \leftarrow 1$
else if $\mathsf{pk} \in P$: $b \leftarrow 0$
else : $b \in \{0,1\}, st \leftarrow \mathcal{A}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{sk}_b, \mathsf{pk}_b$
if $X = HON$: $\mathsf{ct}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}^{D_{b'}(\mathsf{sk}_{b'}, \cdot)}(\mathsf{pk}_0, \mathsf{pk}_1, st)$
if $X = LEAK$: $\mathsf{ct}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}(\mathsf{pk}_0, \mathsf{sk}_0, \mathsf{pk}_1, \mathsf{sk}_1, st)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct}_0)$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct}_1)$
if $k_0 = \bot \vee k_1 = \bot$: **return** 0
// $\mathcal{A}$ wins if $\neg\big((\forall x \in P \,.\, x_0 = x_1) \implies (\forall y \in Q \,.\, y_0 = y_1)\big)$
return $\forall x \in P \,.\, x_0 = x_1 \wedge \exists y \in Q \,.\, y_0 \neq y_1$

Figure 6.3: Generic game for our new binding notions $X\text{-}BIND\text{-}P\text{-}Q$ for $X \in \{HON, LEAK\}$.

$MAL\text{-}BIND\text{-}P\text{-}Q^{\mathsf{KEM}}_{\mathcal{A}}$:

$g, st \leftarrow \mathcal{A}(st)$
if $g = 1$:
  $(\mathsf{sk}_0, \mathsf{pk}_0), (\mathsf{sk}_1, \mathsf{pk}_1), \mathsf{ct}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}(st)$
  $\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct}_0)$
  $\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct}_1)$
if $g = 2$:
  $(\mathsf{sk}_0, \mathsf{pk}_0), (\mathsf{sk}_1, \mathsf{pk}_1), \mathsf{r}_0, \mathsf{ct}_1 \leftarrow \mathcal{A}(st)$
  $\mathsf{k}_0, \mathsf{ct}_0 \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk}_0; \mathsf{r}_0)$
  $\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct}_1)$
if $g \notin \{1,2\}$:
  $(\mathsf{sk}_0, \mathsf{pk}_0), (\mathsf{sk}_1, \mathsf{pk}_1), \mathsf{r}_0, \mathsf{r}_1 \leftarrow \mathcal{A}(st)$
  $\mathsf{k}_0, \mathsf{ct}_0 \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk}_0; \mathsf{r}_0)$
  $\mathsf{k}_1, \mathsf{ct}_1 \leftarrow \mathsf{KEM.Encaps}(\mathsf{pk}_1; \mathsf{r}_1)$
if $k_0 = \bot \vee k_1 = \bot$: **return** 0
// $\mathcal{A}$ wins if $\neg\big((\forall x \in P \,.\, x_0 = x_1) \implies (\forall y \in Q \,.\, y_0 = y_1)\big)$
return $\forall x \in P \,.\, x_0 = x_1 \wedge \exists y \in Q \,.\, y_0 \neq y_1$

Figure 6.4: Generic game for our new binding notions $MAL\text{-}BIND\text{-}P\text{-}Q$. The adversary can use g to choose whether they want to find a collision between two calls to Encaps, Decaps or a single call to both, in line with [111].

key pairs. In the honest case $X = HON$, two honestly generated key pairs are considered, and we give the adversary access to a decapsulation oracle $D_{b'}(\mathsf{sk}_{b'}, \cdot)$, where $b' \in \mathcal{P}(0,1)$, that they can use to decapsulate ciphertexts with either secret key. For the leak case $X = LEAK$, we also give the adversary access to both secret keys. In the malicious case $X = MAL$, the adversary can choose or construct the key pairs in any way they want. For $X \neq HON$ we do not need a decapsulation oracle since the adversary already has

the secret keys.

If $X \in \{HON, LEAK\}$, we check whether $\mathsf{pk} \in P$ or $\mathsf{pk} \in Q$ and choose the key pairs for the second call to Decaps accordingly. For $X = MAL$, the adversary chooses the key pairs.

The difference between $X = LEAK$ and $X = HON$ is whether the adversary only has access to a decapsulation oracle or has access to the secret keys. Given the secret keys, the adversary can decapsulate ciphertexts and learn intermediate values of the decapsulation. If they only have the oracle, they only learn the output of decapsulation but no intermediate values.

6.3.4 Relating Binding to Contributive Behavior

In the context of other cryptographic primitives, the notion of contributive (or contributory) behavior exists. Intuitively, in a two-party protocol that yields some randomized output, a protocol is contributive if the output is not only determined by one of the parties, but both contribute to the results.

For example, in a standard FO-KEM such as $\mathsf{KEM}_m^{\perp}$ from [127], the randomness sampled for encapsulation is the direct (and only) input for the key derivation function (KDF). Thus, for $\mathsf{KEM}_m^{\perp}$, the only party that contributes to the output key is the sender. We say that such KEMs are non-contributory and can enable re-encapsulation attacks, as described in Section 6.2.2.

In contrast, if the KEM's key binds the public key (e.g., by including the public key in the KDF), then the KEM satisfies *LEAK-BIND-K-PK*, and we say that the KEM is contributory because the recipients' key contributes to the output key.

If the KEM's key binds the ciphertext (*LEAK-BIND-K-CT*), it is not immediately clear whether this is enough to make the KEM contributory, and it depends on the collision freeness [123] (SCFR) of the underlying PKE. If the underlying PKE is not SCFR, i.e., it is possible to decrypt a single ciphertext to the same message with different secret keys, then the KEM is not contributory. The reason for that is that a single ciphertext is valid for multiple public keys, and thus the identity of the receiver is not bound by including the ciphertext in the output key of the KEM. On the other hand, if the PKE is strongly collision free (or even robust) then including the ciphertext makes the KEM contributory.

6.3.5 Relationship to Other Properties

Our generic security notions cover a wide array of different properties and generalize existing security properties in the literature. In this section, we give a short overview of other properties that can be expressed using our generic notions.

When $P = \{\mathsf{ct}\}$ and $Q = \{\mathsf{pk}\}$, our generic games resemble different robustness notions. For example, *HON-BIND-CT-PK* corresponds to strong robustness (SROB) from [123] and *HON-BIND-K, CT-PK* corresponds to strong collision freeness (SCFR) from [123]. Interestingly, the strong robustness notion from [1], which coined the term in the context of PKEs, is weaker than both our *HON-BIND-CT-PK* notion and the strong robustness notion from [123], since they both allow the adversary to query an oracle; this was not possible in the original definition. As an example, we compare our notions to the SROB and SCFR notions from [123] in Figure 6.5.

The properties introduced for PKEs in [111] are formulated analogously to ours: complete robustness (CROB) resembles *MAL-BIND-CT-PK*, and their intermediate notion unrestricted strong robustness (USROB) resembles our *HON-BIND-CT-PK* notion.

Our *HON-BIND-K, CT-PK* is equivalent to the strong collision freeness property from [123]; it is a weaker version of SROB, where an adversary has to decapsulate a single ciphertext to the same output key for distinct public keys.

LEAK-BIND-K, PK-CT matches the ciphertext collision resistance (CCR) property for KEMs from [16].

6.3.6 Relations and Implications

In this section, we summarize the results by [73] on the relations between our various binding notions. The proofs and separating examples can all be found in [73], and we only show the main results here.

$\text{SROB-CCA}^{\mathsf{KEM}}_{\mathcal{A}}$:

$(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}^{D(\cdot,\cdot)}(\mathsf{pk}_0, \mathsf{pk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
return $\mathsf{k}_0 \neq \perp \wedge \mathsf{k}_1 \neq \perp$

$\text{SCFR-CCA}^{\mathsf{KEM}}_{\mathcal{A}}$:

$(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}^{D(\cdot,\cdot)}(\mathsf{pk}_0, \mathsf{pk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
return $\mathsf{k}_0 = \mathsf{k}_1 \neq \perp$

$\textit{HON-BIND-CT-PK}^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}^{D_{b'}(\mathsf{sk}_{b'},\cdot)}(\mathsf{pk}_0, \mathsf{pk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
if $k_0 = \perp \vee k_1 = \perp$
  return 0
return $pk_0 \neq pk_1$

$\textit{HON-BIND-K, CT-PK}^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}^{D_{b'}(\mathsf{sk}_{b'},\cdot)}(\mathsf{pk}_0, \mathsf{pk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
if $k_0 = \perp \vee k_1 = \perp$
  return 0
return $k_0 = k_1 \wedge pk_0 \neq pk_1$

$\textit{LEAK-BIND-CT-PK}^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}(\mathsf{pk}_0, \mathsf{sk}_0, \mathsf{pk}_1, \mathsf{sk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
if $k_0 = \perp \vee k_1 = \perp$
  return 0
return $pk_0 \neq pk_1$

$\textit{LEAK-BIND-K, CT-PK}^{\mathsf{KEM}}_{\mathcal{A}}$:

$\mathsf{sk}_0, \mathsf{pk}_0 \leftarrow \mathsf{KeyGen}()$
$\mathsf{sk}_1, \mathsf{pk}_1 \leftarrow \mathsf{KeyGen}()$
$\mathsf{ct} \leftarrow \mathcal{A}(\mathsf{pk}_0, \mathsf{sk}_0, \mathsf{pk}_1, \mathsf{sk}_1)$
$\mathsf{k}_0 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_0, \mathsf{pk}_0, \mathsf{ct})$
$\mathsf{k}_1 \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ct})$
if $k_0 = \perp \vee k_1 = \perp$
  return 0
return $k_0 = k_1 \wedge pk_0 \neq pk_1$

Figure 6.5: At the top, the strong robustness and strong collision freeness definitions from [123]. In the middle, our *HON-BIND-CT-PK* and *HON-BIND-K, CT-PK* definitions which correspond to SROB and SCFR respectively. At the bottom, our *LEAK-BIND-CT-PK* and *LEAK-BIND-K, CT-PK* definitions which give the adversary access to the secret keys.

We first formalize the ordering of our threat models.

Lemma 1. *Let* KEM = (KeyGen,Encaps,Decaps) *be a key encapsulation mechanism. If* KEM *is MAL-BIND-P-Q-secure, then* KEM *is also LEAK-BIND-P-Q-secure.*

Lemma 2. *Let* KEM = (KeyGen,Encaps,Decaps) *be a key encapsulation mechanism. If* KEM *is LEAK-BIND-P-Q-secure, then* KEM *is also HON-BIND-P-Q-secure.*

The next lemma states that adding elements to P or removing elements from Q weakens a property. Intuitively, if, e.g., k binds pk, then k *and* ct also bind pk (since it is already bound by k).

Lemma 3. *Let* KEM = (KeyGen,Encaps,Decaps) *be a key encapsulation mechanism. For* $X \in \mathcal{P}(MAL, LEAK, HON)$*, if* KEM *is* X*-BIND-P-*Q'*-secure and* $P \subseteq P' \wedge Q \subseteq Q'$*, then* KEM *is also* X*-BIND-*P'*-Q-secure.*

Theorem 1. *Let* KEM = (KeyGen,Encaps,Decaps) *be a key encapsulation mechanism. For* $X \in \mathcal{P}(MAL, LEAK, HON)$, *if* KEM *is* X*-BIND-*P*-*Q'*-secure,* X*-BIND-*Q*-*R'*-secure and* $P \subseteq P'$, $Q \subseteq Q' \cup P'$, *and* $R \subseteq R'$, *then* KEM *is also* X*-BIND-*P'*-*R*-secure.*

Lemma 4. *Let* KEM *be a KEM that is HON-BIND-CT-PK secure. Then* KEM *is also HON-BIND-CT-K secure.*

Lemma 5. *Let* KEM *be a KEM that is LEAK-BIND-CT-PK secure. Then* KEM *is also LEAK-BIND-CT-K secure.*

Proposition 1. *There exists a KEM scheme* KEM *that is MAL-BIND-CT-PK but not MAL-BIND-CT-K.*

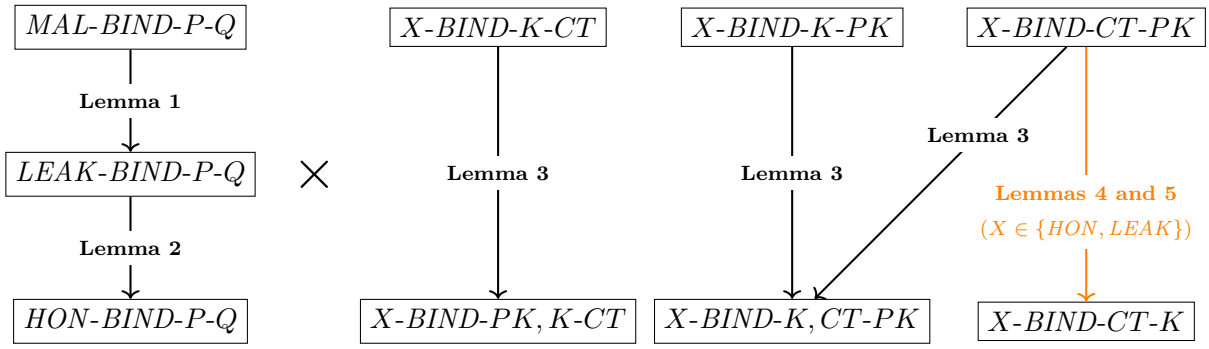


Figure 6.6: General hierarchy of binding properties for KEMs. An edge from A to B indicates that any KEM that is A-binding is also B-binding. Missing edges represent the existence of separating examples, which can be found in [73][Table 4]. The hierarchy left of the $\times$ denotes the implications between the different attacker capabilities {MAL, LEAK, HON}. The hierarchy on the right of the $\times$ represents the implications between our binding properties, independent of the attacker capabilities. We can combine both hierarchies by choosing a node from the left and instantiating P and Q according to a node from the right, resulting in, for instance, an implication between *MAL-BIND-CT-PK* and *HON-BIND-K, CT-PK*. For X = MAL, *X-BIND-CT-PK* and *X-BIND-CT-K* are incomparable. The orange edge indicates that for $X \in$ {HON, LEAK}, *X-BIND-CT-PK* implies *X-BIND-CT-K*.

These results give rise to a hierarchy for our properties, which we visualize in Figure 6.6. We want to highlight that for so-called *implicitly* rejecting KEMs, i.e., KEMs whose Decaps algorithm never returns $\perp$, we establish a reduced hierarchy in Section 6.3.7 by showing that for these KEMs the ciphertext alone cannot bind other values. As a result, implicitly rejecting KEMs cannot meet properties like *HON-BIND-CT-PK*, i.e., be robust.

6.3.7 Implicitly Rejecting KEMs

Many real-world KEMs are so-called *implicitly rejecting*: Their decapsulation algorithm never returns $\perp$ for a valid ciphertext and any valid private key. However, only when decapsulating with the correct private key (corresponding to the public key used for encapsulation), the correct key is output.

Intuitively, an implicitly rejecting KEM is similar to an implicitly authenticated key exchange: Successfully completing the protocol does not imply that someone else has the same key or sent any message; instead, the guarantee is that only the correct party can possibly compute the same secret key.

Rejection Keys

When decrypting a ciphertext using implicitly rejecting KEMs, one of two outcomes can occur: if the ciphertext is valid for the KEM key pair, decryption proceeds successfully. However, if the ciphertext is invalid, the algorithm still yields an output key, referred to as the *rejection key*. In Lemma 6, we state a useful, informal lemma that establishes how an implicitly rejecting KEM has to compute its rejection keys.

Lemma 6. *The rejection key computation of an implicitly rejecting KEM has to at least contain a secret random value and the rejected ciphertext.*

We do not give a formal proof of this statement. Instead, we argue informally why for any KEM that does not compute its rejection keys in this manner, an adversary can actually distinguish the rejection keys from a random key, turning them into an error flag. Note that this does not indicate a problem with IND-CCA, as IND-CCA only requires "accepting" keys to be indistinguishable from a random key.

Recall that $\mathsf{Decaps}(sk, pk, ct)$ is a deterministic algorithm. Therefore, sk, pk, and ct are the only possible inputs to the rejection key computation, as Decaps cannot sample random values. Notice that if the computation only contains pk or ct, the adversary can easily compute the same rejection key since only public values were used for the computation. Thus, a secret random value z has to be part of the computation.

Due to Decaps's deterministic nature, KEM designers are now left with two choices: Modify the decapsulation API to include z directly or let the secret key sk contain z. Depending on the choice made here and the origin of z (is it randomly sampled independent or dependent of sk?), it has implications on the achievable binding properties which we will discuss in Section 7.2.

Lastly, we point out that the ciphertext needs to be part of the rejection key computation, as otherwise there will be collisions: The rejection key would be the same for every ciphertext since sk and pk are static. One might wonder whether the statement is no longer true if we allow for a probabilistic Decaps. We argue that the statement still holds. If Decaps were to sample random, instead of using *static* randomness contained in the secret key, different queries with the same secret key, public key, and ciphertext would result in different rejection keys.

Modifying the Hierarchy

A trivial side effect of implicitly rejecting KEMs is that the ciphertext alone cannot bind any other value: Any ciphertext will be accepted, and these will (with overwhelming probability) decapsulate to different keys. A special case of this is the observation in [123] that an implicitly-rejecting KEM cannot satisfy SROB, i.e., *HON-BIND-CT-PK*. We recall [73][Theorem 4.10]:

Theorem 2. *An implicitly rejecting key encapsulation scheme* KEM *cannot satisfy X-BIND-CT-PK or X-BIND-CT-K for* $X \in \{HON, LEAK, MAL\}$.

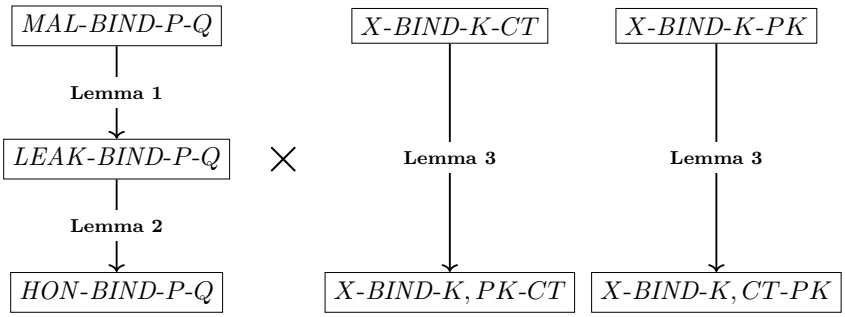


Figure 6.7: Restricted hierarchy of binding properties for implicitly rejecting KEMs, as *X-BIND-CT-K* and *X-BIND-CT-PK* cannot be met by any implicitly rejecting KEM

Thus, for implicitly rejecting KEMs, we have a reduced hierarchy of relevant properties. The separation between honest and malicious variants persists, but only four core properties are relevant and distinct, which are the key-binding properties. This leaves us with a simple hierarchy with only eight relevant binding properties overall for implicitly rejecting KEMs, which we visualize in Figure 6.7.

6.4 Symbolic Analysis of KEMs

We now turn to our second main objective: to develop a formal analysis framework for automatically analyzing security protocols that use KEMs. Our framework is rooted in the *symbolic model* of cryptography.

We will use Tamarin to implement our framework. Lastly, we create new fine-grained symbolic models for KEMs that allow us to configure which cryptographic properties they have, allowing us to precisely model real KEM schemes in the symbolic model.

6.4.1 Symbolic Models for KEMs

In this section, we develop a new symbolic model for KEMs that allows the user to specify exactly which binding properties the model gives. Like the *Symbolic Verification of Signatures* model from [134], our model achieves this by only relying on the implications that the standard computational security definitions, e.g, IND-CCA, and our binding properties give, which makes it perfectly suited for verification.

Specification We observe that the definitions of correctness and IND-CCA only hold when the key pair that is used is honestly generated. When this is not the case, no guarantees are given. Correctness requires that the decapsulation of a ciphertext created by encapsulating against an honest public key returns the same output key for both algorithms. IND-CCA requires that the output key, created by encapsulating against an honest public key, is indistinguishable from true randomness (even when the adversary has access to a decryption oracle). Additionally, we note that Encaps can be a probabilistic algorithm, while Decaps is deterministic.

We now build a symbolic model that follows these constraints but allows for any other behavior. To do so, we model the key- and ciphertext space of a KEM and allow the adversary to choose arbitrary values from them as the result of Encaps and Decaps, as long as they respect the following constraints:

1. If the public key was honestly generated, an Encaps computation must return a fresh key different from any other Encaps computation.
2. If the public key was honestly generated, Encaps and Decaps computations using the same ciphertext and public key pair must return the same output key.
3. Given *X-BIND-P-Q*, any pair of calls to Decaps (and/or Encaps if $X = MAL$) must agree on Q if the parameters in P are equal.
4. Multiple computations of Decaps with the same inputs give the same result.
5. Any Encaps computation by the adversary only results in fresh keys or known values from the key space.

Constraints 1) and 2) model IND-CCA and correctness respectively, 3) ensures that the relevant binding properties are met, 4) makes Decaps deterministic, and 5) models the adversary computing a *derandomized* Encaps as seen in Section 6.2.2. We over-approximate this by allowing the adversary to let Encaps result in any element of the key space if they were already aware of this value, as letting the adversary choose any value would break IND-CCA.

Additionally, we add an option that, when enabled, specifies that Encaps and Decaps only work on honestly generated key pairs, rejecting any other values. This allows us to prevent the adversary from breaking IND-CCA and correctness by feeding *bogus* values into Encaps and Decaps. To achieve this, we require that a user of our library annotates rules in which the protocol honestly generates a public key pk with an action $\mathsf{GoodKey}(pk)$.

Due to space limitations, we only give a brief example of how our KEM model works. Recall the previous multiset rewriting rule. It shows how Alice, who possesses a public key pk, a secret key sk, and a ciphertext ct, decapsulates with these values to obtain key k. As long as the semantic constraints of our KEM model are fulfilled, the key can be an arbitrary value from the key space, represented by !KeyValues. Note that the key space only contains atomic values, which allows our KEM model to avoid achieving certain binding properties, e.g., *MAL-BIND-CT-PK*, by construction, which was not possible for previous symbolic models (see Section 7.2.3).

Definition 6. *HON-BIND-K-PK restriction*

$$\begin{aligned}&\forall\ k\ ct1\ \ ct2\ \ pk1\ \ pk2\ \ sk1\ \ sk2\ \sharp i\ \sharp j\ \sharp l\ \sharp m\ .\\&\mathsf{Decaps}(k, ct1, pk1, sk1)@\sharp i \wedge\ \mathsf{Decaps}(k, ct2, pk2, sk2)@\sharp j\\&\wedge\ \mathsf{GoodKey}(pk1)@\sharp l \wedge \mathsf{GoodKey}(pk2)@\sharp m\\&\Rightarrow\ (pk1 = pk2)\end{aligned}$$

6.4.2 Tamarin Implementation

To model the key generation of the KEM, the event $GoodKey(\mathsf{pk})$ is introduced, indicating that pk was generated honestly. Properties such as *Correctness* are later specified to apply only to keys that were produced via honest key generation.

Users of the library can employ arbitrary values as secret keys, but they must use the function symbol $kem\_pk(\cdots)$ to generate the corresponding public key.

Encapsulation and decapsulation operations of KEMs are modeled with events $\mathsf{Encaps}(\mathsf{k}, \mathsf{ct}, \mathsf{pk})$ and $\mathsf{Decaps}(\mathsf{pk}, \mathsf{ct}, \mathsf{pk}, \mathsf{sk})$. The input is provided via the premises of the protocol rule annotated with the event, while the output is delivered through persistent facts $\mathsf{!KeyValues}(\mathsf{k})$ and $\mathsf{!CTValues}(\mathsf{ct})$ in the rule's premises. These persistent facts come from our symbolic model of KEMs and represent the key- and ciphertext space, respectively. For instance, when modeling an encapsulation:

$$[\mathsf{Alice}(\mathsf{pk}, \mathsf{sk}), \mathsf{!KeyValues}(\mathsf{k}), \mathsf{!CTValues}(\mathsf{ct})]$$
$$-[\mathsf{Encaps}(\mathsf{k}, \mathsf{ct}, \mathsf{pk})]\rightarrow$$
$$[\mathsf{Out}(\mathsf{ct})]$$

$\mathsf{!KeyValues}(\mathsf{k})$ and $\mathsf{!CTValues}(\mathsf{ct})$ are needed to bind the output of the Encaps call on pk. By default, no restriction is placed on these values, and arbitrary collisions in the key and ciphertext are possible.

To restrict collisions in the key and ciphertext spaces, the computational properties such as *Correctness*, *HON-BIND-K-CT*, and *MAL-BIND-K-CT* are encoded as logical formulas to ensure desired behavior, including key equality and binding properties. Tamarin traces violating these properties are discarded through imposed *restrictions*. For instance, we can encode the *Correctness* property using the following formula:

Definition 7. *Correctness*

$$\forall\ k1\ \ k2\ \ ct\ \ sk\ \sharp i\ \sharp j\ \sharp k\ .$$
$$\mathsf{Encaps}(k1, ct, kem\_pk(sk))@\sharp i \wedge\ \mathsf{Decaps}(k2, ct, kem\_pk(sk), sk)@\sharp j$$
$$\wedge\, \mathsf{GoodKey}(kem\_pk(sk))@\sharp k$$
$$\Rightarrow\ (k1 = k2)$$

The restriction encodes that an Encaps call and a Decaps call that use the same public key pair–produced by the honest key generation algorithm–and the same ciphertext must also produce the same key.
In the same manner, we can encode our binding properties. For instance, *HON-BIND-K-CT*:

Definition 8. *HON-BIND-K-CT*

$$\forall\ k\ \ ct1\ \ ct2\ \ pk1\ \ pk2\ \ sk1\ \ sk2\ \sharp i\ \sharp j\ \sharp l\ \sharp m\ .$$
$$\mathsf{Decaps}(k, ct1, pk1, sk1)@\sharp i \wedge\ \mathsf{Decaps}(k, ct2, pk2, sk2)@\sharp j$$
$$\wedge\, \mathsf{GoodKey}(pk1)@\sharp l \wedge \mathsf{GoodKey}(pk2)@\sharp m$$
$$\Rightarrow\ (ct1 = ct2)$$

To encode *MAL-BIND-K-CT*, we drop the requirement that the public keys have to be good keys, and we add additional restrictions that enforce the same behavior for pairs of Encaps calls as well as pairs of Encaps and Decaps.

For IND-CCA, it is required that the output key k of an Encaps call with a good key is distinct from any other output key produced by Encaps. We model this with a restriction as well.

Additionally, in a classical symbolic model with function symbols, the adversary can freely use these symbols as well. To also allow this behavior for our event-based computations, we provide protocol rules in our model that allow the adversary to perform these computations. While this would encode the adversary to emulate honest parties, we also give the adversary access to a modified Encaps computation where they can force the output key to be any value they already know. This models the case where the adversary does not use *fresh* randomness but reuses randomness to compute an output key of their choice. Note that this behavior is also why re-encapsulation attacks exist

Optional restrictions are provided to enforce the use of *good keys* in *Encaps* and *Decaps* computations, allowing users to exclude traces where bad keys are used.

Finally, we want to explain why we only implement our properties for $X = HON$ and $X = MAL$ in the symbolic setting. In Section 6.3.3, we note that *LEAK* and *HON* are indeed different in the computational model since an adversary can learn intermediate values when computing Dec themself, as is the case in the *LEAK* setting. This is not the case in the *HON* setting, where they can only observe the result of *honest* Dec computations done by an oracle. Thus, *LEAK* and *HON* are not equivalent in the computational model. In our KEM model, on the other hand, the output key is not computed from intermediate values but from an atomic fresh value. Thus, an adversary cannot learn any additional information by computing Dec themself. As a result, *HON* and *LEAK* are equivalent for our KEM model. We leave building a symbolic KEM model that allows for arbitrary combinations of our binding properties, where *LEAK* is not equal to *HON*, as future work. This is challenging because modeling arbitrary partial information leakage in the symbolic model is still an open problem, and modeling the output key as a compound term built from other terms (which could be leaked) inherently satisfies some of our binding properties (see Section 7.2.3).

6.5 Case Studies

This section showcases the practicality of our approach through case studies. We begin with a brief overview of the evaluation methodology applied to evaluate the various Authenticated Key Exchange (AKE) protocols we modelled as case studies. In Section 6.5.2, we summarize the outcome of the chosen case studies. As case studies, we cover diverse post-quantum AKE protocols from the literature like the Kyber-AKE [48] or PQ-SPDM [219, 220], detailed in Sections 6.5.3 to 6.5.6.

6.5.1 Methodology

Our novel KEM model allows us to reason about both (i) an adversary that is restricted to use honestly generated, valid public key pairs as required by the KEM scheme and (ii) arbitrary, potentially malicious keys. Together with the option to use any combination of our specified binding properties from Section 6.3, this leads to a high number of configurations. Thus, it is infeasible to analyze each security property of every case study with every configuration of our KEM model. To analyze the influence of our binding notions on a protocol's security properties, we develop a methodology that allows us to discover the minimal requirements on a KEM that are needed to prove the property, while pruning the search space.

Initial Configuration Testing First, we analyze each statement of a protocol with the following initial configurations:

1. Only keys from KeyGen and no binding properties
2. Only keys from KeyGen and **all** leak binding properties
3. Any keys and no binding properties
4. Any keys and **all** malicious binding properties

If, for a specific property of a protocol, Tamarin terminates with the same result in each of these configurations, we conclude that the protocol gives this property independent of any binding properties of the KEM or maliciously generated key pairs.

Key-Based Inference If Tamarin falsifies a property when we allow malicious keys, we infer that the protocol indeed relies on honestly generated key pairs to give the property.

Binding Property Analysis In the event that Tamarin gives different results when we change the binding property, we proceed with a more in-depth analysis:

1. For both the *LEAK* and *MAL* setting, we construct a directed acyclic graph whose nodes correspond to the possible combinations of our binding properties. We add an edge from node u to node v iff the properties that correspond to u imply the properties that correspond to v.
2. To efficiently compute for which combinations of properties a given statement is valid, we explore the graph as follows: We pick an unexplored node randomly and try to verify the statement using

the corresponding binding properties. If Tamarin proves the statement, we mark the node as proven and recursively mark all of its parent nodes as proven too. We know that Tamarin will also prove the statement for the parent nodes since their corresponding properties imply the properties that were sufficient for a proof. If Tamarin falsifies the statement we mark the node as falsified, and recursively mark all of its child nodes as falsified too. This result is also valid for the child nodes because the corresponding binding properties of these nodes are weaker, removing fewer traces from the model and thus allowing for the same counterexample. In the event of a timeout, we mark the node as timed out and continue.

3. Once all nodes are marked, we extract the nodes for which Tamarin could still verify the statement but for whose direct child nodes Tamarin cannot verify the statement. The properties corresponding to these nodes are the minimal binding properties Tamarin needs to verify the statement.

6.5.2 Discussion of Results

We ran our models on an Intel(R) Xeon(R) CPU E5-4650L 2.60GHz machine with 1TB of RAM and 4 threads per Tamarin call. The execution time of our full methodology was approximately ∼16h30m.

Model	#Lemmas Secrecy	Auth.	Auxiliary	#Tamarin calls	KEM-Binding Dependent Protocol Properties	Runtime for Initial Configurations
Onepass AKE	5	4	3	48	Implicit Key Authentication (Init.)	∼ 1m
Σ_0'-protocol	3	6	0	36	Implicit Key Authentication (Init.), SK-security	∼ 6m
PQ-SPDM	6	12	8	104		∼ 38m
Kyber-AKE	4	2	7	52	Implicit Key Authentication (Init., Resp.)	∼ 4h10m

Table 6.2: Summary of the analysis for our initial configurations. For the listed protocol properties, Tamarin returns different verification results in our initial configurations. For the minimal binding properties required to prove them, we refer to Table 6.3.

We summarize the results of our initial configuration (see Section 6.5.1) in Table 6.2. We can observe that the specified security properties of the one-pass AKE, Kyber-AKE, and Σ_0'-protocol do not solely rely on IND-CCA but also on other binding properties of the KEM. Details regarding this can be found in the corresponding sections 6.5.3, 6.5.4, and 6.5.6.

In Table 6.3 we show the concrete binding properties the aforementioned protocols require of their KEMs to achieve the desired security properties.

Challenges The newly introduced symbolic definitions of Encaps and Decaps can now result in arbitrary values from the key- and ciphertext space instead of only compound terms built from their inputs–vastly increasing the size of the search space. As the default heuristics of Tamarin do not prioritize solving, e.g., Encaps goals, when exploring the search space, we additionally develop *proof tactics*, tailored towards our KEM model. These tactics are prioritizing goals related to the output key, the KEM secret keys, and key-derivation functions, as well as deprioritizing goals related to ciphertexts.

Model	Protocol Property	Minimal Binding Properties	
One-pass AKE	Implicit Key Authentication (Init.)	*X-BIND-K-PK*	{ *X-BIND-K-CT*, *X-BIND-K*,*CT-PK* }
Σ_0'-protocol	Implicit Key Authentication (Init.)	*X-BIND-K-PK*	{ *X-BIND-K-CT*, *X-BIND-K*,*CT-PK* }
	SK-Security	*HON-BIND-CT-K*	
		MAL-BIND-CT-K	*MAL-BIND-PK-K*
Kyber-AKE	Implicit Key Authentication (Init.)	*X-BIND-K-PK*	{ *X-BIND-K-CT*, *X-BIND-K*,*CT-PK* }
	Implicit Key Authentication (Resp.)	*X-BIND-K-PK*	{ *X-BIND-K-CT*, *X-BIND-K*,*CT-PK* }

Table 6.3: Minimal binding properties required by Tamarin to prove the property of an AKE model. We omit the set brackets for singleton sets like *HON-BIND-CT-K*. We write *X-BIND-P-Q* without specifying *X* to indicate that this set of properties is a solution for $X = MAL$ when all keys are allowed and for $X = HON$ when only honestly generated keys are allowed.

6.5.3 One-Pass AKE

As a starting point, we model a one-pass AKE based on the SIKE protocol from [127], which we show in Figure 6.8. Note that in this protocol the Recipient does not receive standard mutual authentication guarantees since it cannot verify any information from the Initiator. Thus, we focus on the properties the Initiator can achieve.

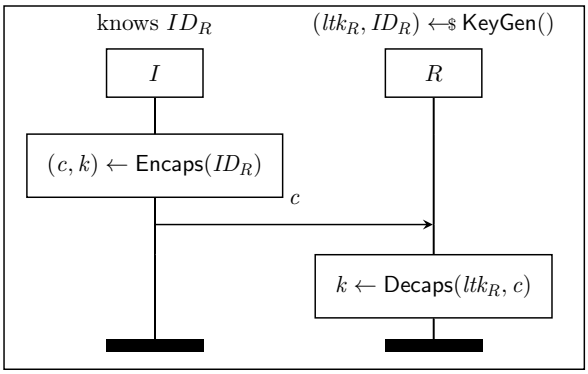


Figure 6.8: Simplified one-pass AKE.

In particular, we are interested in authentication properties like *Implicit Key Authentication* (as defined in Definition 9). However, in this protocol, neither the Initiator nor the Recipient can achieve it, since only one party contributes to the final key. Thus, we focus on a weaker, unilateral version, where only the identity of the Recipient has to match in both sessions, i.e., both Initiator and Recipient agree on the Recipient's identity when they derive the same shared key.

We find that the Initiator can achieve this weaker property when the protocol uses a KEM that is at least *X-BIND-K-PK*-secure. When the KEM does not have this binding property, the adversary can mount a re-encapsulation attack against the Initiator by leaking ltk_R of the Recipient and then re-encapsulating towards another Recipient, resulting in two Recipient sessions with the same key.

Definition 9. *Implicit Key Authentication Initiator*

$$\begin{aligned}&\forall\ id1\ \ id2\ \ pkI1\ \ pkI2\ \ pkR1\ \ pkR2\ \ k\ \ ct1\ \ ct2\ \ \sharp i\ \sharp j\ .\\&\mathsf{FinishInitiator}(id1, pkI1, pkR1, k, ct1\ )@\sharp i\\&\wedge \mathsf{FinishResponder}(id2, pkI2, pkR2, k, ct2\ )@\sharp j\\&\Rightarrow\ (pkI1 = pkI2 \wedge pkR1 = pkR2)\end{aligned}$$

6.5.4 Σ_0'-Protocol

The Σ_0'-protocol is introduced by [184] as a KEM-based variation of the Σ_0-protocol [56]. The original Σ_0-protocol is a component of the Internet Key Exchange (IKE) protocols [146], and Σ_0' was suggested as a post-quantum replacement. We provide a description of the Σ_0'-protocol in Figure 6.9. [184] claims that the Σ_0'-protocol is SK-secure in the post-specified peer model [56] for any IND-CCA KEM. We analyze whether Σ_0' achieves SK-security (Definition 11) and Implicit Key Authentication and Final Key Secrecy (Definition 10).

Definition 10. *Final Key Secrecy Initiator*

$$\begin{aligned}&\forall\ id\ \ pkI\ \ pkR\ \ k\ \ ct\ \sharp i\ \sharp j\ .\\&\mathsf{FinishInitiator}(id, pkI, pkR, k, ct\ )@\sharp i \wedge \mathsf{GoodKey}(pkR)@\sharp j\\&\Rightarrow\ (\exists\ \sharp x\ .\mathsf{K}(k)@\sharp x) \vee (\exists\ \sharp x\ .\mathsf{RevealLTK}(pkR)@\sharp x)\end{aligned}$$

Definition 11. *SK-Security*

$$\begin{aligned}
&\forall\ sid\ pkI\ pkR\ k\ k2\ \sharp i\ \sharp j\ .\\
&\mathsf{FinishInit}(sid, pkI, pkR, k)@\sharp i \wedge \mathsf{FinishResp}(sid, pkI, pkR, k2)@\sharp j\\
&\wedge not(\exists\ \sharp y\ .\mathsf{RevealLTK}(pkR)@\sharp y) \wedge not(\exists\ \sharp x\ .\mathsf{RevealLTK}(pkI)@\sharp x)\\
&\Rightarrow\ (k\ = k2) \wedge not(\exists\ \sharp z\ .\mathsf{K}(k)@\sharp z)
\end{aligned}$$

When modeling Σ_0', we noticed two issues with the protocol description in [184]. First, the authors assume that the Responder can store an unlimited number of session identifiers it receives from the Initiator and that it only accepts sessions that use a new, unused identifier, which is notoriously hard to achieve. Second, after replying to the Initiator, the Responder should erase pk_I from its state. However, when the Responder receives the final message, it has to verify σ_I–which contains pk_I. It is unclear how the Responder can verify this signature after erasing pk_I.

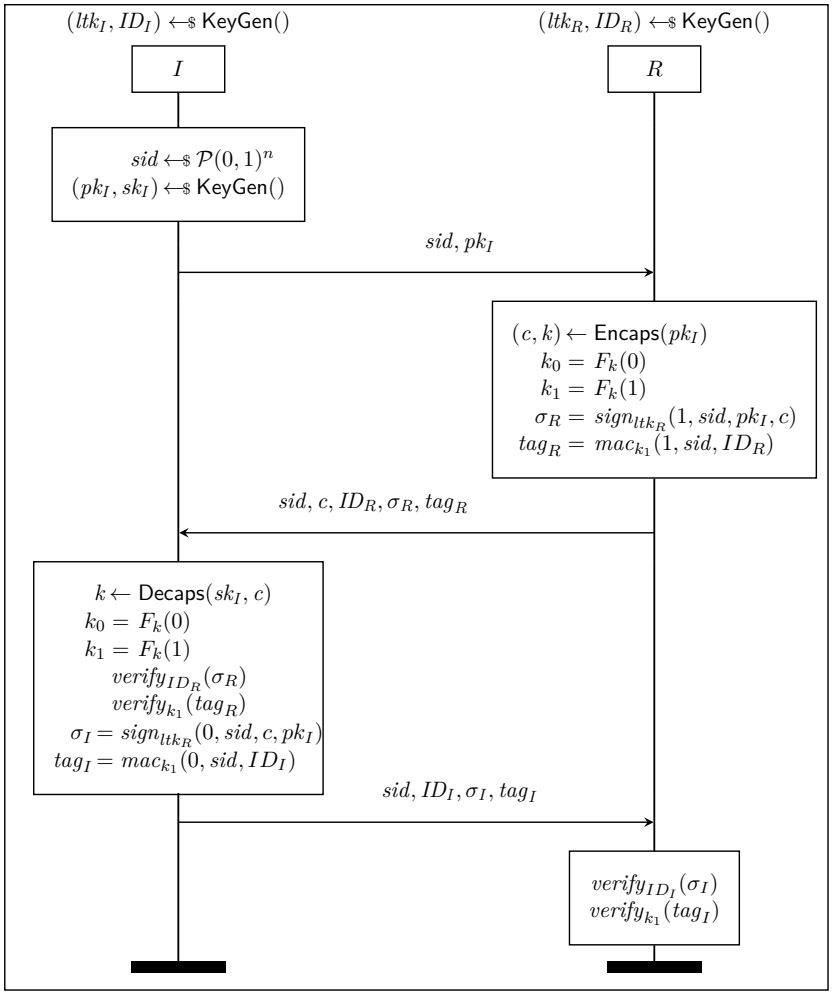


Figure 6.9: The Σ_0'-protocol introduced by [184].

To address these issues, we create two Σ_0' models. In the first model, Σ_0'-perfect, the Responder keeps pk_I in its state and verifies σ_I correctly, as well as only accepting the first message when it sees a new, fresh session identifier. The second model, Σ_0', does *not* keep pk_I in the Responder's state, and the verification of $\sigma_I = (0, sid, c, pk)$ succeeds for any KEM public key pk as long as 0, sid, and c are correctly signed. In this model, the Responder always replies to the first message, even if session identifiers repeat.

We find that Σ_0' and Σ_0'-perfect achieve Implicit Key Authentication and Final Key Secrecy for the Responder in all of our initial configurations. Additionally, we prove *Full Key Confirmation* for the Responder, which we define in Definition 12.

Definition 12. *Full Key Confirmation Responder*

$$\forall\ sid\ pkI\ pkR\ k\ epkI\ \sharp i\ \sharp j\ .$$
$$\mathsf{FinishResponder}(sid, pkI, pkR, k, epkI\ )@\sharp i \wedge \mathsf{GoodKey}(epkI)@\sharp j$$
$$\Rightarrow\ (\exists\ pkI2\ pkR2\ epkI2\ \sharp x\ .\mathsf{FinishInitiator}(sid, pkI2, pkR2, k, epkI2\ )@\sharp x)$$

As is the case for Kyber-AKE (Section 6.5.6), both Σ_0' and Σ_0'-perfect do not achieve Implicit Key Authentication for the Initiator for any IND-CCA-secure KEM. This is because the adversary can switch pk_I for their own ephemeral key and reveal ltk_R of the Responder. Let A and B be honest agents. The attack then proceeds as follows: the adversary waits until A initiates a session as initiator with peer B and starts another session impersonating as C towards B in the Responder role. After replacing pk_A with their own ephemeral KEM key pk_C and revealing ltk_B, the adversary forwards sid, pk_C to B, who acts according to the protocol. Then, the adversary decapsulates c to learn k, k_0, and k_1. Next, the adversary mounts a re-encapsulation attack against A's actual ephemeral key pk_A, resulting in a ciphertext c' that also decapsulates to key k. Since the adversary knows both lkt_B and k_1, they can forge B's signature on c' and create a valid tag_B, which they both forward to A, completing A's run. Finally, the adversary creates a valid signature and tag for B, who thinks they are communicating with the adversary. At the end of their respective runs, A and B agree on key k but not on their peers' identities.

We find that the protocol gives Implicit Key Authentication for the Initiator when the KEM satisfies at least *HON-BIND-K-PK* or both *HON-BIND-K-CT* and *HON-BIND-K,CT-PK*. Note that the later two, together, imply *HON-BIND-K-PK* by [73][Corollary 4.11]. Thus, *HON-BIND-K-PK* really is the property that prevents the attack: I and R deriving the same key k implies that they agree on pk_I. This stops the adversary from switching out pk_I for their own ephemeral key, which prevents them from learning the key k. However, knowledge of k is necessary to create a valid tag_I for R in the last message. Thus, the above attack is prevented.

We observe that Σ_0' does not achieve SK-security for any IND-CCA-secure KEM when the Responder erases pk_I from its state and accepts duplicate session identifiers.

To understand this attack, we want to highlight that the signatures σ_I and σ_R only include c; they *do not* include the actual output key k. Thus, the adversary can choose c such that it decapsulates to different keys for the Initiator and the Responder. This behavior is not excluded by correctness of the KEM, since SK security does not enforce that Initiator and Responder agree on the ephemeral key pk_I; they only need to agree on their respective identities. Thus, the adversary can force a session between A and B where they agree on their identities but disagree on the ephemeral key pk_B. Together with the deficiencies of Σ_0', this allows for an attack on SK security, which can be prevented by using, for instance, a *MAL-BIND-CT-PK*-secure KEM.

The adversary starts a session between Initiator A and Responder B where they agree on their identities and pk_A. After receiving the first message, B computes σ_B and tag_B, which the adversary intercepts. Then, impersonating A, the adversary starts another session with B, where they switch out pk_A for another KEM public key but reuse the session identifier sid. Recall that the Responder only accepts this session in the Σ_0' protocol where we do not assume infinite, persistent storage at the Responder side. The Responder B again acts according to the protocol and computes $(c, k) \leftarrow \mathsf{Encaps}(pk_A)$. The adversary then forwards σ_B and tag_B to A, who computes σ_A, tag_A, and finishes their session with B decapsulating k' from c because of the lack of binding properties of KEM. The adversary then forwards these values to B in the session where they disagree on pk_A. Since B accepts any σ_B as long as c and sid match, they also finish their session with A computing k. As these sessions agree on the identities but disagree on the final shared key, SK security does not hold.

When honestly generated keys are used, the KEM must at least satisfy *HON-BIND-CT-K*, and, in the presence of maliciously generated keys, the KEM must at least satisfy *MAL-BIND-CT-PK* or *MAL-BIND-CT-K* to prevent this attack. These properties prevent Initiator and Responder from opening the same ciphertext to different output keys.

To summarize, we find that Σ_0' does not achieve Implicit Key Authentication for the Initiator when used with any IND-CCA-secure KEM due to a re-encapsulation attack, and that SK security does not hold when the Responder *misbehaves* as described previously. A KEM with additional binding properties, e.g., *HON-BIND-K-PK* and *HON-BIND-CT-K*, could have prevented these attacks.

6.5.5 PQ-SPDM

The Security Protocol and Data Model (SPDM) [94] is an emerging industry standard aimed at ensuring end-to-end trust in infrastructure, focusing on hardware and chip-to-chip communication. Although the standard is being developed by major industry players, there has been limited cryptographic analysis.

In this dissertation, we provide the first formal model of SPDM in Part II. In addition to the normal version, a post-quantum version of SPDM's session establishment has been proposed [219, 220].

We model this proposed post-quantum key exchange of SPDM and analyze for both Initiator and Responder whether the protocol achieves Final Key Secrecy, Implicit Key Authentication, and Full Key Confirmation. A detailed description of the post-quantum SPDM protocol can be found in Part II and [76, 219, 220].

We find that, for the Initiator, Final Key Secrecy holds as long as neither the ephemeral KEM key pair nor the long-term key of the Responder is revealed. For the Responder, we find that Final Key Secrecy holds even when the ephemeral KEM key pair is revealed. The Initiator obtains Implicit Key Authentication only if the ephemeral KEM key pair is not revealed, and the Responder as long as either the long-term key of the Initiator or the ephemeral KEM key pair is not revealed.

Full Key Confirmation does not hold for the Initiator if either key pair is revealed. For the Responder, we find that the property holds as long as at least one key pair is not revealed.

We find these results across all initial configurations and conclude that PQ-SPDM provides these guarantees independently of any KEM binding properties or maliciously generated keys.

6.5.6 Kyber-AKE

We model the Kyber-AKE (see Figure 6.10) in Tamarin and analyze the protocol, both in terms of secrecy and authentication properties.

The secrecy of the final key of the Kyber-AKE is guaranteed for both Initiator and Responder as long as the long-term secrets are not revealed. The property is defined analogously to Definition 10.

We also analyze Implicit Key Authentication for both the Initiator and Responder analogously to Definition 9. Without any additional binding properties, even when we restrict the adversary to only use honest keys out of KeyGen, Tamarin is able to produce a counterexample violating the defined property. We call this attack *re-encapsulation* attack, as described in Section 6.2.2.

We show that, in the setting where we do not allow keys outside KeyGen, the used KEM in the Kyber-AKE additionally needs to provide *HON-BIND-K-PK* or both *HON-BIND-K-CT* and *HON-BIND-K, CT-PK* to guarantee implicit authentication for both parties. Analogously, in the stronger adversary model, the used KEM needs to provide *MAL-BIND-K-PK* or both *MAL-BIND-K-CT* and *MAL-BIND-K, CT-PK* to guarantee implicit authentication.

This observation also confirms why we do not see this kind of attack in the original Kyber-AKE, as the Kyber KEM is conjectured to fulfill these properties [73]

Note that [48] claims the Kyber-AKE is secure in the Canetti-Krawczyk (CK) model with weak forward secrecy [55]. However, they state no explicit proof and claim that this "*follows directly from the generic security bounds of [12, 64]*". Our results show that this statement is incorrect if the KEM is only IND-CCA secure. In particular, in the CK model with weak PFS, the re-encapsulation attack would imply that the sessions are not matching, and this would allow the adversary to reveal the session key at A's session.

We illustrate this on a concrete example, which was automatically found by Tamarin, on an authenticated key exchange protocol from the Kyber paper [48] shown in Figure 6.10.

We stress that when the key exchange protocol is instantiated with Kyber as intended by the paper, the protocol seems secure. However, can Kyber be replaced by any other KEM? In the paper, Kyber is only proven to be IND-CCA secure. Is IND-CCA sufficient for the protocol's security? It turns out this is not the case.

To show this, we consider the same key exchange protocol, but instantiated with $\mathsf{KEM}_m^{\perp}$ from [127]. $\mathsf{KEM}_m^{\perp}$ is an FO-KEM (cf. Section 6.2.1). In the context of our work, FO-KEMs are interesting because they have a property that is not captured by the current syntax of KEMs: when a party A decapsulates a ciphertext to learn k, they can also learn the message m that was used by the underlying PKE. This

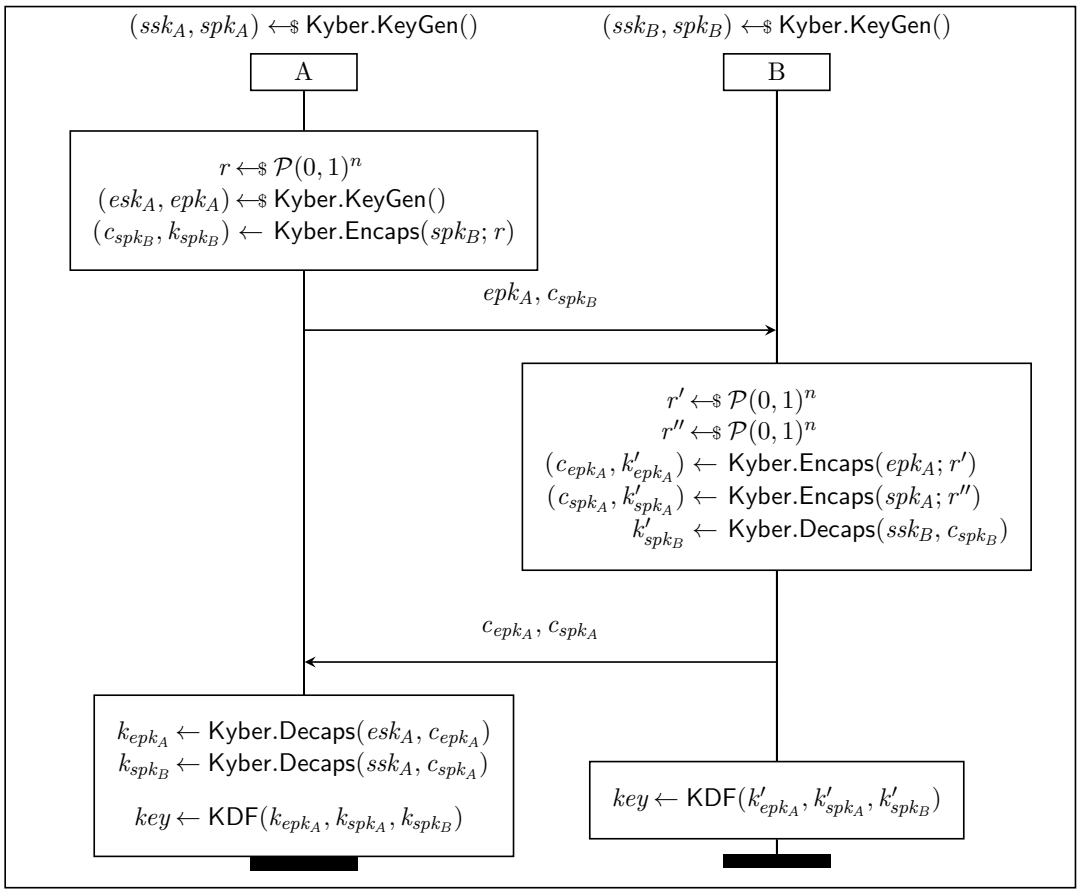


Figure 6.10: The authenticated key exchange described in the original Kyber paper [48].

is unavoidable for any PKE-based KEM because the ciphertext that contains m needs to be decrypted before deriving k from m.

To simplify notation in this example, we assume that we can infer the randomness r from the message m. This allows for a slightly more abstract description of the attack, but we can instantiate the attack for any concrete FO-KEM without this assumption. We capture this by writing $(k, r) \leftarrow \mathsf{KEM.Decaps}(\mathsf{sk}, c)$ in this example.

We now explain the re-encapsulation attack in Figure 6.11. In the attack, A and B are honest. The adversary C wants to coerce B into establishing a key shared with A, where B mistakenly assumes that A thinks they share the key with B; instead, A will think they share it with C. This is a so-called unknown-key-share attack [40], which violates B's implicit key agreement.

The attack proceeds as follows: A initiates communication with C, after which C decapsulates the ciphertext c_{spk_B} to obtain k_{r_0} and, more importantly, r_0, which was used by A to create c_{spk_B}. Now, C impersonates A towards B by encapsulating against B's static public key with r_0 and forwarding the resulting ciphertext and epk_A. B responds with the expected values to A, as B thinks A is communicating with them. Finally, A decapsulates the ciphertexts received from B, and both A and B derive the final key. Since we instantiated the protocol with $\mathsf{KEM}^{\perp}_m$, the keys obtained via Decaps only depend on the randomness supplied by the encapsulating party. As a result, A and B derive the same *key*; this is a violation of implicit authentication since A thinks they now share a key with C, which does not match B's expectations.

One might wonder whether a KEM with strong robustness properties would prevent this attack. Unfortunately, this is not the case: robustness properties reason about *a single ciphertext* c that should not decapsulate to the same key under different key pairs. However, our re-encapsulation attack revolves around two *different* ciphertexts: based on A's ciphertext, the malicious C creates a different ciphertext c_{spk_C} that decapsulates to the same key as c_{spk_B} by reusing the randomness r_0.

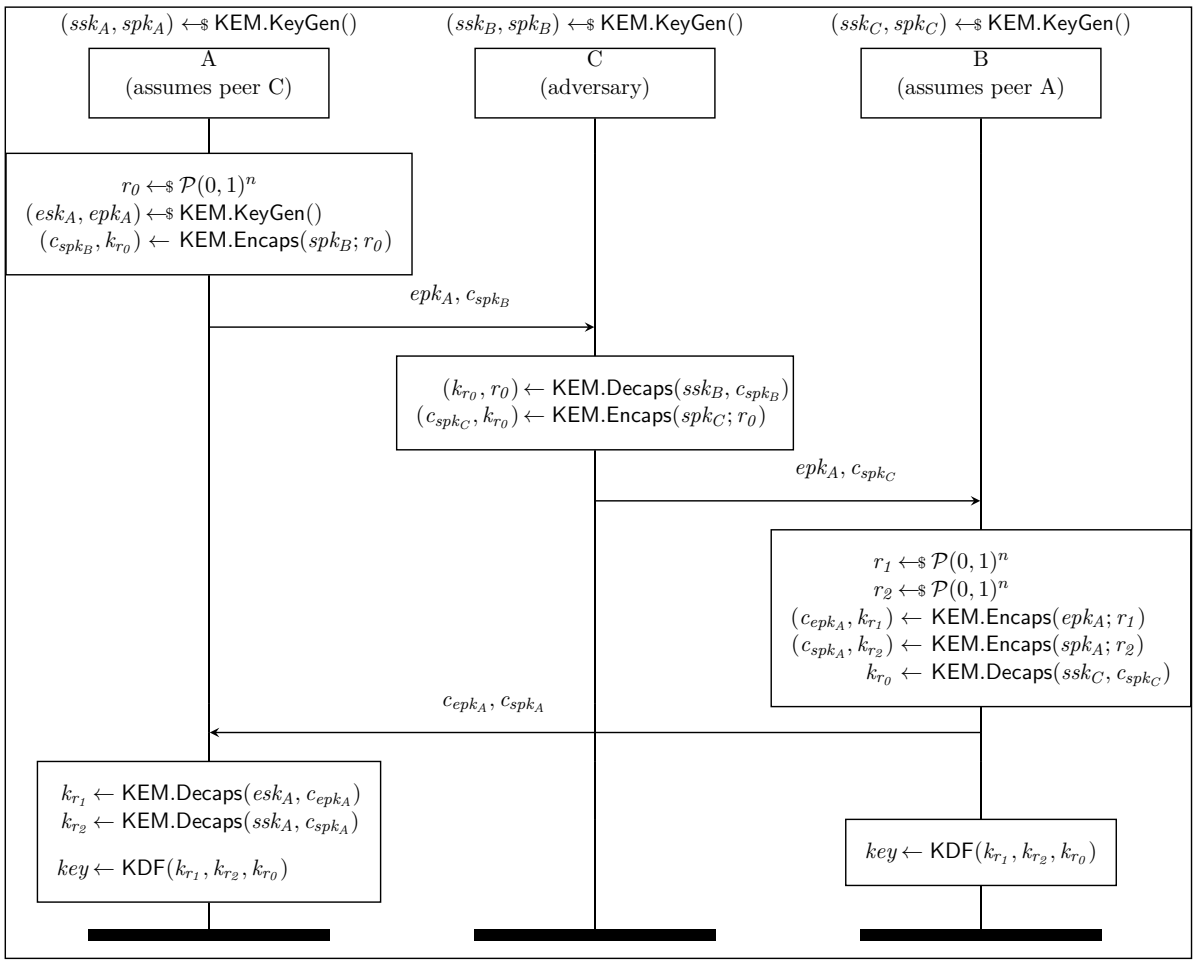


Figure 6.11: Re-encapsulation attack against the Authenticated Key Exchange (AKE) suggested for the Kyber KEM [48] where the adversary C coerces honest A into unknowingly sharing the *key* with honest B, who correctly thinks they are being contacted by honest A. This violates the implicit key agreement guarantee for B, who expects to share a key with someone that assumes B is the peer. Note that this attack is only possible when the AKE is instantiated with a KEM that does not bind the output key to the public key, and is not possible when instantiated with Kyber.

7 Limitations and Related Work

We are the first to develop methodologies to automatically analyze security protocols with fine-grained models of cryptographic hash functions, authenticated encryption schemes with associated data (AEAD), and key encapsulation mechanisms (KEM). Our methodologies for all primitives allow both: (i) Analyzing a protocol for a concrete primitive instantiation, or (ii) analyzing a protocol with the goal to discover the weakest properties the primitive needs to provide in order to achieve the desired security guarantees. To construct our fine-grained primitive models, we build on the results of several cryptographic definitions and known attacks from the literature.

With all the methodologies outlined in Chapters 4 to 6 we discovered attacks and undesired behavior in several real-world protocol designs. Whereas most of these behaviors are of theoretical interest (e.g., the discovered flaws found using the hash methodology are rediscoveries or variants of already known attacks), our methodology from Section 5.5.1 discovered undesired behavior in deployed protocols. We detail the disclosure process in Section 7.1.
We will discuss the literature Section 7.2, before we discuss the limitations of our approach and methodologies in Section 7.3.
Source Code and Reproducibility All of our technical contributions from Chapters 4 to 6 including the implementation of our symbolic representation of the primitives in Tamarin and all models of the case studies are open-source. We provide all models, instructions, and software to reproduce our results for hashes at [101], for AEADs at [71], and for KEMs at [74].

7.1 Disclosure

Using our methodology from Section 5.5.1, we detected several undesirable behaviors in the protocol design of Scuttlebutt, Web Push, and WhatsApp Groups [130, 201, 211] related to their use of AEADs. While the behaviors are possible on the protocol design level, their implementation-level feasibility depends on low-level encoding choices.

The behaviors we found did not violate the main specified goals of the respective protocols, and hence we did not mark them as "attacks". Nevertheless, we contacted the developers of the affected protocols and explained our observations, such that they can assess their implementation-level feasibility, with distinct feedback:

- The developers of the WebPush standard acknowledged the issue, and a discussion is ongoing to determine how to best document the possible behaviors in the standard;
- WhatsApp considered this outside their threat model, and noted that using a different AEAD would still allow a variant of the behavior with the same effect; and
- The Scuttlebutt developers did not respond.

7.2 Related Work

Improving the symbolic models of primitives to enable automated attack finding has recently been explored for several types of basic primitives, like Diffie-Hellman groups [82] or digital signatures [134].

ProVerif [44] is, besides Tamarin, the other major tool for automated analysis of protocols in the symbolic model. While we developed our models and case-studies in Tamarin, they could also be used in the ProVerif framework (see [61] for a first attempt to include our hash models).

As we are focused on automated attack finding with our analyses, our work is orthogonal to tools from the computational model [15, 18, 42] that are all focused on proving security, and cannot find attacks. Our automated models can be useful to establish the assumptions on the primitives before attempting a computational proof.

7.2.1 Cryptographic Hash Functions

Besides the works already mentioned in Section 4.2, there are many works on hash functions, their design, and their cryptographic properties (see e.g. [172, 175, 191]). A further related work is [117], which analyzes the impact of breaking cryptographic primitives on Bitcoin, including its hash functions RIPEMD-160 and SHA2-256.

The ROM shares features with the classical symbolic (Dolev-Yao) model of hash functions used in nearly all previous symbolic analyses. One exception is the automated TLS 1.3 analysis in [33], which explicitly considers the possibility of a very weak hash function where all inputs collide. However, as we have seen, such a model does not necessarily give the adversary strictly more powers: for authentication properties, or protocols with inequalities, such a model may inadvertently miss possible protocol behaviors.

The use of the ROM in symbolic models also connects to the so-called computational soundness question: can we obtain computational guarantees from a symbolic proof? In this field, the only two works that consider hash functions model them as a random oracle [14, 67]. Additionally, [14] shows an impossibility result in the standard model for a hash function that is symbolically represented as a free symbol.

7.2.2 Authenticated Encryption with Associated Data

AEADs do not have a designated way to model them symbolically. Ad-hoc approaches include a specific form of nonce-reuse in the Tamarin analysis of WPA [83] and the analysis of Yubikey [153]. In a different approach, [151] modeled the fine-grained block based encryption in the tool ProVerif, but this approach did not scale to protocols of the complexity considered here. Overall, our work is the first to systematically explore weaknesses of concrete algorithms or formal definitions for AEADs and provide models amenable to automation.

With respect to our case studies, based on the absence of collision resistance of AEADs, also referred to as robustness or key-commitment, [102] already reported an attack on the Facebook abuse reporting mechanism, which violates accountability. We are the first to report on behaviors in which content agreement is not satisfied due to AEAD weaknesses. Other undesirable behaviors have arisen due to collisions, which are linked to oracle partitioning attacks [158], where an attacker can obtain a better than brute force advantage against the OPAQUE protocol. While we can model the relevant AEAD weakness in our framework, modeling the violated security property in the symbolic model is left to future work.

7.2.3 Key Encapsulation Mechanism

We are not the first to model KEMs in the symbolic model. Some KEM-based protocols, e.g., KEMTLS [199, 200, 205], post-quantum Wireguard [129], and PQXDH [36], were recently analyzed in the symbolic model. In this section, we investigate how these case studies model KEMs, which of our binding properties these models achieve, and why a class of symbolic models for KEMs implicitly assumes certain binding properties, highlighting the need for a new symbolic model that can model any combination of our binding

properties.

In [205], the authors create and analyze two Tamarin models of KEMTLS, a variant of TLS that uses KEMs to achieve post-quantum security. Similarly, [129] uses Tamarin to analyze a KEM-based variant of Wireguard. The authors of [36] use another tool, ProVerif, to analyze the PQXDH protocol. In all of these case studies, function symbols and equational theories are used to model the behavior of KEMs. In fact, all but one model from [205] use the built-in equational theories for public key encryption to model KEMs. Concretely, they use the function symbols $\mathsf{aenc}/2$, denoting encryption, and $\mathsf{adec}/2$, denoting decryption. The function symbols are related through the equation $\mathsf{adec}(sk, \mathsf{aenc}(pk(sk), msg)) = msg$ and mapped to the standard KEM API in the following way:

$$\mathsf{Encaps}(pk(sk), r) = \mathsf{aenc}(pk(sk), r)$$
$$\mathsf{Decaps}(sk, ct) = \mathsf{adec}(sk, ct)$$

Note that Encaps does not return a tuple of ciphertext and key but only the ciphertext. Instead, r directly serves as the output key. One model from [205] uses a slightly different approach: instead of using r directly as the output key, they incorporate the receiver's public key and model the output key as a function $\mathsf{kdf}(r, pk)$. These models for KEMs are not surprising, since PKEs and KEMs are also strongly related in the computational model.

However, unlike the computational model, the above symbolic models encode much stronger assumptions on KEMs than just IND-CCA. Because the output key and the ciphertext are deterministic functions of pk and r, they bind these values by construction. That is, given an output key (or a ciphertext), the corresponding public key and randomness are uniquely determined. In fact, the reverse is also true. Thus, any symbolic model that computes, for instance, the ciphertext as $ct = \mathsf{Encaps}(pk, r)$ is *MAL-BIND-CT-PK*, *MAL-BIND-CT-K* (assuming k is also a function of pk and r), and *MAL-BIND-K, PK-CT* by construction.

As a result, the second KEM model from [205] implicitly assumes that the KEM satisfies all of our *MAL* binding properties. This means they cannot detect, e.g., re-encapsulation attacks. On the other hand, the first model from [205] and the models from [36, 129] do not assume *HON-BIND-K-PK* or *HON-BIND-K-CT*, because the output key is independent of the public key. Consequently, this model might detect some of our re-encapsulation attacks, which the findings of [36] confirm.

Further security notions

While IND-CCA is still the main security notion for KEMs, additional security notions have been proposed.

The term "robustness" was initially coined by Abdalla, Bellare, and Neven in [1] in the context of PKE schemes. In a nutshell, robustness means it is hard to produce a ciphertext that is valid for two different key pairs (or users). They introduce both *weak* (WROB) and *strong* (SROB) robustness. In the weak robustness game, an adversary has to find a message m and two distinct public keys pk_0 and pk_1 such that encrypting m with pk_0 results in a valid ciphertext when decrypted with sk_1, pk_1's secret key. In the strong robustness game, the adversary has to find a ciphertext c and two distinct public keys such that c decrypts under both corresponding secret keys. This strengthens the adversary since c does not have to be the result of an *honest* encryption but could have been specifically created by the adversary.

In [111], the authors make a case for new, stronger robustness notions by showing how the notions of [1] fail to prevent attacks in certain applications such as fair auction protocols. First, they observe that the original strong robustness definition does not allow the adversary to query their oracle with the secret keys of the public keys they are challenged with; removing this restriction leads to an intermediate notion that they call *unrestricted* strong robustness (USROB). Then, they go on to remove the restriction that the adversary is challenged with honestly generated public keys. Instead, the adversary is given complete control over the key generation, and it is up to the decryption algorithm to reject invalid key pairs, which leads to their *full* robustness (FROB) notion. The USROB and FROB notions define robustness via the decryption routine of a PKE, implicitly assuming that robustness "carries" over to the encryption algorithm since encryption and decryption are related through correctness. However, in a setting where the adversary can freely choose key pairs and ciphertexts, correctness may no longer hold, since the adversary can feed values from outside the key- and ciphertext-space into the PKE algorithms. Thus, it is necessary that the whole cryptosystem satisfies a robustness notion. To capture this, [111] defines

complete robustness (CROB), which challenges the adversary to find a ciphertext that decrypts under different key pairs for any combination of encryption and decryption calls.

In [123], Grubbs, Maram, and Paterson define anonymity, robustness, and so-called *collision freeness* for KEMs, building upon Mohassel's work [176] that only defined these properties for PKEs. They investigate whether a PKE constructed via the KEM-DEM paradigm inherits anonymity and robustness from the underlying KEM. They show that this is true for explicitly rejecting KEMs. However, for implicitly rejecting KEMs, this is not the case in general. Since all NIST PQC finalist KEMs are implicitly rejecting KEMs constructed via variants of the FO transform [116], they then go on to analyze how the FO transform lifts robustness and anonymity properties from a PKE scheme, first to the KEM built via the FO transform and then to the hybrid PKE scheme obtained via the KEM-DEM paradigm. They apply their generic analysis of the FO transform to the NIST PQC finalists Saber [85], Kyber [48], and Classic McEliece [31] as well as the NIST alternate candidate FrodoKEM [47]. Another finding of [123] regarding the IND-CCA secure Classic McEliece scheme will be relevant for our work: for any plaintext m, they find that it is possible to construct a single ciphertext c that always decrypts to m under *any* Classic McEliece private key.

Our Binding Properties in the Wild

Building on an earlier pre-release of our work on KEMs, Schmieg reports in [206] attacks on ML-KEM's *MAL-BIND-K-PK* and *MAL-BIND-K-CT* security, which we conjectured in an earlier version (Version 1.0.5) of [73].

The attack in [206] on *MAL-BIND-K-CT* exploits the fact that implementations of ML-KEM store a hash of the public key, which is needed to compute the shared secret when the KEM accepts, inside the secret key. This is done to avoid recomputing this hash across decapsulation operations. While this improves the performance of the KEM, it allows strong adversaries that can control the secret key for decapsulation–like the adversary in our *MAL* properties–to manipulate this hash. In a nutshell, an adversary can carefully manipulate the hash which does not trigger the FO rejection flow during Dec and create two different ciphertexts that decapsulate to the same shared secret. This is possible because ML-KEM derives the shared secret only from the stored hash of the public key and the sampled message when it accepts. For further details of the attack, we refer the reader to [206]. We want to highlight that this attack is neither specific to ML-KEM nor to implicitly rejecting KEMs (since it does not trigger the rejection flow) but caused by the serialization format of the secret key.

To mitigate the attack, [206] suggests to not cache the hash of the public key inside the secret key but to recompute it for every decapsulation operation. Another mitigation they suggest is to check whether the stored hash is actually the hash of the public key. In summary, both of these mitigation strategies assert that the secret key is well-formed. Therefore, we believe that KEM implementations can achieve the *MAL-BIND-K-CT* property in practice, albeit at the cost of a minor performance loss, and the attack does not indicate a problem with our property.

The attack on ML-KEM's *MAL-BIND-K-PK* is very similar to the previously described attack on *MAL-BIND-K-CT*. The difference is that the adversary now replaces the rejection value z, which is also stored inside the secret key. The attack then proceeds as follows: The adversary creates two secret keys which share a rejection value z, produces a random ciphertext c, and tries to decapsulate c with both secret keys. With overwhelming probability, both decapsulation calls will reject the ciphertext resulting in the same shared secret because the rejection flow computes the shared secret as a hash of c and z.

Again, the mitigation that Schmieg suggests tries to verify that the secret key is indeed well-formed. The idea is to not store the secret key directly, but to store the seed that the key was derived from and to recompute the key in every decapsulation call. Unfortunately, this mitigation is impossible to implement for the ML-KEM proposal (FIPS 203, August 24, 2023 [178]) without technical changes, since it currently derives the rejection value z from a different seed that is independent of the key generation seed. Because of this, it is impossible to check whether a secret key is well-formed: the rejection value z stored inside the key is random and independent of the key; any value is possible. As a consequence, there seems to be no mitigation for Schmieg's attack without changing the proposal, and implicitly rejecting KEMs with such independent values cannot achieve our *MAL-BIND-K-PK* and *MAL-BIND-K,CT-PK* properties just by performing simple sanity checks. If, instead, the key pair and rejection value are derived from the same seed, a KEM can verify whether a secret is well-formed by recomputing it from the seed.

Lastly, we want to highlight that both attacks can be prevented by including both the ciphertext and the public key in the derivation of the shared secret *as well as* the rejection key, instead of using only one of them in each derivation respectively. This can, for instance, be achieved by the generic wrapper construction of [73].

Recall that ML-KEM derives the shared secret from the sampled message and the stored hash of the public key when it accepts, and from the rejection value and the ciphertext when it rejects. If it were to include both the public key and the ciphertext in both cases, the *MAL-BIND-K-CT* attack by Schmieg, which relies on ML-KEM accepting, would not work because including the different ciphertexts in the key derivation trivially leads to different shared secrets. For Schmieg's *MAL-BIND-K-PK* attack, which relies on ML-KEM rejecting, including the different public keys in the derivation of the rejection key also forces different rejection keys.

Note that for ML-KEM we cannot simply add the ciphertext to the computation of the shared secret, and we have to resort to the wrapper construction. The underlying reason is that ML-KEM computes the shared secret *and* the coins r used for the call to the underlying PKE from the same call to G (see Algorithm 16 in [178]). Hence, it is impossible to include the ciphertext c in the key-derivation of the shared secret at this point, because it cannot be computed yet.

To conclude, we find that the attack on *MAL-BIND-K-CT* is tightly linked to the additional information that is stored inside the secret key, but KEM implementations can achieve the property if they check the secret key for well-formedness. However, we are not aware of any current KEM implementations that do this due to its negative performance implications. Regarding the attack on *MAL-BIND-K-PK*, we find that it is tightly linked to the relationship of the key pair's seed and the rejection value's seed. If they are independent, there appears to be no way of mitigating this attack since any combination of key pair and rejection value is well-formed. If they are dependent, checking for well-formedness becomes possible, however, this would require changes to current proposals as well as their implementations.

7.3 Limitations

Building advanced, fine-grained models of cryptographic hashes, AEADs, and KEMs resulted in significant improvements in symbolic verification. In all three cases, our methodologies allow us to automatically discover protocol flaws based on subtle differences in cryptographic primitives. However, we are far from done, even with the primitives covered in Chapters 4 to 6. In the following, we will talk about the main two limitations for our primitive models: the computational overhead our methodologies introduce and the generality of our results.

Runtime

Our models of cryptographic hashes, AEADs, and KEMs introduce a computational overhead compared to their "perfect" representations previously used in Tamarin.[1]

This increase in needed computational resources is to be expected. The more fine-grained models offer a deeper level of detail, which, while more expressive and accurate, also increase the amount of computations of the tool for the same problem. While the overhead is usually small (see for instance Table 4.12), a small increase can already lead to big challenges in protocol verification. One such problem is the "state space explosion", where the number of different states that the verification process needs to consider becomes very large, very quickly.

At present, with the level of efficiency Tamarin offers, using multiple of our advanced primitive models within one analysis is out-of-scope for any bigger real-world protocol.

Coverage

In an ideal world, we would like to (i) cover all possible definitions and weaknesses of a primitive, and (ii) have the guarantee that if our method reports an attack, the attack is always feasible in practice.

[1] We conjecture that this will be the case for any tool that relies on idealized representations of cryptographic primitive, e.g., ProVerif.

Unfortunately this is not the case yet.

Let us take our models of AEADs as an example. In terms of possible AEAD definitions there are subtle differences that we currently do not capture yet. This includes, for example, properties beyond collisions and nonce-reuse, such as the "s-way committing security property" [25] that generalizes the CMT notions to the multi-user setting.

Our models can also be improved with respect to the Forge capability, as discussed in section 5.4. On the positive side, we define general models and capture for instance collisions that given the current knowledge are not practical, but could become so in the future, e.g., with new developments on AES. While we do not claim to cover all possible AEAD attacks in the future, this allows to future-proof protocols.

The same holds for all our primitive models. Especially, properties that reason about multi-user settings or properties that we cannot encode in tools like Tamarin are still out-of-scope for our approaches.

With respect to practical feasibility of attacks, the fundamental problem is that our analysis method and standard cryptographic analyses in fact consider protocols *designs* and not their implementation details. For example, this includes abstracting away from encoding details, i.e., how values and compound structures are exactly mapped to bitstrings. Yet such details are critical to determine whether certain attacks are possible or not. As a consequence, when we find an attack on the protocol design, this should intuitively be interpreted as: there exists an encoding scheme for which the protocol implementation is insecure. We argue that security of a protocol design should avoid depending on its encoding scheme, and if not, specify the requirements explicitly. The problems we found here using our methodologies are therefore real concerns for the protocol designs. Still, manual inspection of the implementation is still needed to check whether the encoding allows the attack execution.

II

Part II: Exploring the Limits

"If you always put limits on everything you do, physical or anything else, it will spread into your work and into your life. There are no limits. There are only plateaus, and you must not stay there, you must go beyond them."

– Bruce Lee, *Description of personal philosophy, n.d.*

Statement of Originality

This part of the thesis consists of three chapters.

Chapters 8 to 10 are all based on my previous work

Cas Cremers, Alexander Dax, and Aurora Naska. "Formal analysis of SPDM: Security protocol and data model version 1.2." In: USENIX Security Symposium*. 2023* [76]

and

Cas Cremers, Alexander Dax, and Aurora Naska. Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 including Cross-Protocol Attacks*. Under Submission. 2025* [75].

Both papers are joined work with Cas Cremers and Aurora Naska. In both, Aurora Naska and I shared the lead on the project.
While both of us worked on all parts of this project, I lead the research in formalizing the device attestation part of the SPDM protocol. I also took the lead on identifying challenges and limitations of our approach and developed strategies on how to overcome them during the merging and composition of our final monolithic SPDM model.
I will describe my contributions in more detail in the following.

The general description of SPDM in Section 9.1 is drawn directly from [75, 76] and follows their presentation closely. The models described in Section 9.2 were jointly constructed by Aurora Naska and I. My primary focus within this section encompassed the abstraction layer and the modeling of the phases of SPDM leading up to session establishment phases (Section 9.2.1).

The security properties of the phases up to the session establishment phases presented in section 9.3.1 from Section 9.3 as well as the initial formulation of the threat model in Section 9.3.3, were extracted and formulated by myself.

Additionally, I took the lead in the research concerning the merging and composition of protocol components into a monolithic model (Section 9.2.2), with a particular emphasis on exploring scalability and identifying limitations inherent in the monolithic model (Section 9.4).

In Section 9.5, I describe our analysis results for both the initial models and the monolithic model. While the general results are to be attributed to all authors, I additionally describe an attack uncovered in the monolithic Tamarin model and its respective fix. The process of finding, describing, implementing, and fixing the attack was lead by Aurora Naska and is presented here in a shortened version.

Furthermore, the limitations and discussion in Chapter 10 are collected and presented by me but include a culmination of the experiences and observations of all authors throughout the projects, and thus, it does not solely reflect my own insights.

The figures presented in the following chapters are taken from [75, 76] with the permission of all authors and are therefore not to be seen as solely my own contributions.

8 Exploring the Limits – Analyzing SPDM

The preceding chapters have examined methods to align symbolic models of basic cryptographic primitives more closely with real-world scenarios. These analyses have advanced our understanding of protocol verification and allowed us to detect subtle flaws previously only detectable by manual inspection.

However, while we were able to allow symbolic analysis to be more fine-grained, it does not assist us in tackling the other prominent issue with automated analysis – the size and complexity limitations of the object to be analyzed.

Most formal analyses of security protocols, even after decades of research, primarily focus on isolated protocol components, such as single key exchanges using specific cryptographic primitives, ratcheting steps in messaging, transmission layer steps, or key renegotiations. However, real-world protocols like WhatsApp, Signal, and TLS combine multiple such components and often involve various bootstrapping methods, such as starting a conversation with either pre-shared symmetric keys or using certificates within an established public key infrastructure (PKI). Combining components in such a way can exhibit complex behaviors when components share data – which is not covered by component-specific proofs.

Consequently, in practice, the composition of these components often leads to vulnerabilities, as demonstrated by attacks on delayed authentication in TLS 1.3 [81] or cross-protocol attacks on Bluetooth [217], Threema [183], and Matrix [4]. Although studies on protocol composition exist, such as those by [52, 54, 120, 121, 125, 126], they often do not fully apply to real-world scenarios because they do not accommodate less idealized design principles or the full range of potential attack vectors encountered in practice.

Although there are analysis results available for complex protocols like that TLS [80], WPA2 [83], and Apple PQ [161], much work remains to make analyzing such protocols easier. With the growing number of complex standards being introduced recently and expected in the future, the challenge is increasing. Our goal here is twofold:

1. First, we aim to analyze a new and complex real-world security protocol, namely DMTF's SPDM, and be the first to provide formal security guarantees for it.
2. Second, by tackling the formal analysis of a large-scale security protocol currently in standardization, we hope to better understand the current limitations and capabilities of state-of-the-art methodologies within the symbolic model.

The Security Protocol and Data Model (SPDM) standard is actively being developed by the Distributed Management Task Force (DMTF) [93], an industry organization focused on creating standards for IT infrastructures, including cloud computing, virtualization, networks, servers, and storage. Its board and members include major companies such as AMD, Google, Huawei, IBM, Intel, Lenovo, NVIDIA, and many more. One of DMTF's core objectives is to establish common solutions that enable products in this space to work seamlessly together. Recently, key members of DMTF have placed a high priority on securing their platforms through what is known as the *platform root of trust*. As part of this effort, they have supported the development and standardization of SPDM protocol [96].

The SPDM protocol has two main goals, as outlined in its technical note [97]: (i) to cryptographically verify the identity and firmware integrity of platform components, and (ii) to ensure private and secure data communication over platform interfaces. SPDM is already being implemented across hardware components (e.g., Intel, Dell, NVIDIA), cloud platforms (e.g., Google, Cisco, IBM), and operating systems (e.g., Linux). It is also a core security mechanism in CXL 2.0 (Compute Express Link), part of PCIe 5.0, which will be adopted in all Intel and AMD server CPUs, supporting Confidential Computing in technologies like Intel TDX and NVIDIA Hopper.

Analyzing SPDM is challenging due to several reasons. SPDM is often described as a six-round message flow, but this abstraction hides its true complexity. The actual state machines involve loops, optional flows, session resets, and delayed authentication. Although it borrows from TLS 1.3, and regularly compares itself to it, its state machines and transcript handling are significantly more complex, leading

the existing analysis of TLS [11, 30, 34, 35, 41, 50, 51, 80, 81, 86, 91, 104, 105, 113, 114, 136, 145, 149, 155, 160] to be non-transferable to SPDM. SPDM also uses filtered message transcripts in a non-standard way, making its state machines even more complex than those of TLS 1.3. Additionally, the specification of SPDM provides only informal security goals and a high-level STRIDE analysis, limiting its precision.

8.1 Outline

In the following chapters, we will address the goals previously outlined.

In Chapter 9, we formally model the SPDM protocol in Tamarin, one of the state-of-the-art tools to perform automated, symbolic analysis of security protocols. Our final model of SPDM results in one of the largest Tamarin models created to date. To provide a rough measure of its complexity: our model includes 71 transition rules and 42 analyzed lemmas, with the most demanding proof search requiring 261 proof steps. For a general comparison with other major Tamarin case studies, the PQ3 model includes 22 rules [161], the WPA2 model has 69 rules [83], and TLS1.3 comprises 63 rules[80].

As a result of our analysis, Tamarin identified a critical attack on the mutual authentication needed to ensure private and secure data communication. We successfully implemented this attack on the official DMTF reference implementation written in C [100] and the Rust implementation [98], leading DMTF to register a CVE [84] with critical severity (CVSS score 9.0). In response, we proposed a fix and formally proved the security of the updated version of the standard. As a result, both the standard and the reference implementations have been revised and updated accordingly.

In Section 9.4, we address the primary challenges encountered while constructing our monolithic SPDM model in Tamarin, working towards our second goal. Continuing in Chapter 10, we elaborate on the substantial effort required to develop our models and discuss our observations along with the limitations of our analyses. Our work highlights the significance of modeling decisions, such as choosing between processes-based modeling and transition-based modeling, as without choosing the latter, we most likely would have missed our discovered attack. We also demonstrate that domain separation is not only a wise design choice but also allows for the analysis of extensive case studies in Tamarin. With the progress we made on our goal to understand the current limitations and capabilities of state-of-the-art methodologies within the symbolic model, we explore potential future directions for SPDM and the broader field of security protocol analysis.

Our results showcase that analysis results of single modes of protocols do not propagate to the entire protocol analysis. In turn, this calls for holistic analysis of protocol standards and development of tools to support their size. We recommend protocol designers to take the state of current formal analysis techniques into account when designing new protocol standards. This can be prudent design measures like adopting domain separation of their keys or proactively setting out explicit security goals.

8.2 Related Work

The Security Protocol and Data Model (SPDM) by DMTF is supported by numerous significant industry stakeholders, and it is already implemented in hardware or cloud components as well as being a part of PCIe 5.0. Yet, the protocol has seen very limited attention within the academic community.

Studies like [8, 9] have conducted benchmark assessments on SPDM implementations, focusing on performance but not exploring their security dimensions. On the other hand, [219, 220] aim towards advancements in post-quantum security for SPDM. [220] discusses necessary adjustments for a post-quantum adaptation of SPDM, proposing a new key exchange mechanism that substitutes traditional Diffie-Hellman (DH) approaches with Key-Encapsulation Mechanisms (KEMs) to meet future cryptographic challenges. Meanwhile, [219] introduces a variant of the KEM-based key exchange that forgoes digital signatures, though their security proofs were unavailable at the time of writing. In Part I of this thesis, we analyzed a minimal version of post-quantum SPDM (see Section 6.5.5.)

In the analysis of our comprehensive SPDM model, Tamarin reveals a hybrid threat combining a cross-protocol attack and a state machine flaw within the SPDM framework. There has been extensive research into potential state machine attack vectors for the TLS protocol, including methods to infer or test the state machines of an implementation through fuzzing, as seen in studies such as [32, 89, 193, 212].

However, the SPDM 1.2.1 standard lacks defined state machines, which means there is not a clear baseline or "ground truth" for comparisons.

Despite this, it would be interesting to apply state machine inference techniques or specialized fuzzing to SPDM's reference implementation to see if these methods could uncover any mode-switch behaviors. However, the effectiveness of these techniques could vary significantly. Our particular attack involves creating non-standard messages after the mode-switch occurs, which complicates matters. The success of detecting such issues heavily depends on the specific threat models used, which influence how state machine behaviors are interpreted and analyzed.

Furthermore, the security community's efforts to apply composition results to ensure holistic protocol security do not currently align with SPDM's architecture. Results such as those from Ciobâca and Cortier [63] do not extend to protocols sharing cryptographic elements, as is the case with SPDM. Other results [52, 54, 120, 121, 125, 126] are similarly inadequate due to their limitations or the complex shared state and cryptographic dependencies within SPDM's sub-protocols, demonstrating the challenge of applying general-purpose composition theories to SPDM.

9 Security Protocol and Data Model

Contents

9.1 Security Protocol and Data Model 1.2.1

The Security Protocol and Data Model (SPDM) is a protocol designed for two parties: an *Requester*, who starts the communication, and a *Responder*. SPDM is focused on achieving two principal security objectives: *device attestation* and *authenticated, secure communication*. Device attestation allows the Requester to verify various aspects of the Responder, such as firmware integrity and device identity. This goal ensures that the device being communicated with is secure and trustworthy. The second objective, similar to that of the TLS protocol, involves establishing a secure and authenticated channel that allows for the safe transmission of data between the two parties over a network. The SPDM protocol is structured into five distinct phases:

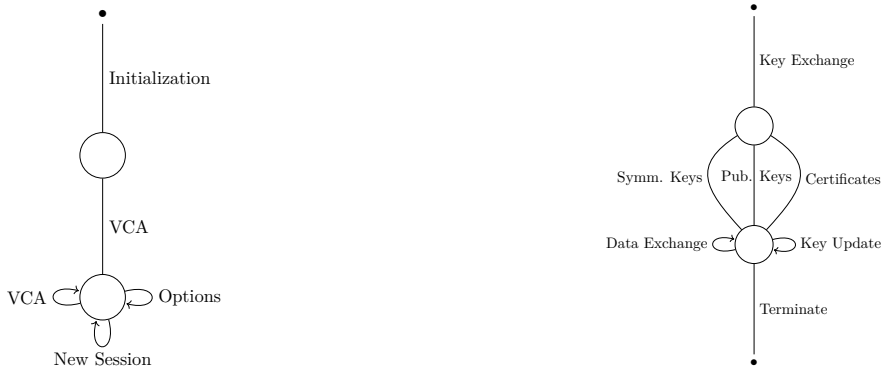
(i) **Device Initialization phase** This phase occurs outside the formal protocol and is focused on setting up devices with their initial cryptographic functions and protocol configurations.

(ii) **VCA phase** The Version-Capabilities-Algorithms phase begins the protocol process, negotiating the ciphersuites and protocol versions to be used by the participants.

(iii) **Options phase** This phase allows for the unilateral authentication of the Responder and the attestation of various device attributes through what are termed *measurements* – the changeable configurations of the Responder's device. These actions depend on prior device initialization and the objectives of the Requester. It is possible for the Requester to bypass this phase entirely and proceed directly to the Session phase by sending a key exchange request immediately following the VCA phase.

(iv) **Key Exchange phase** The key exchange phase (or handshake phase) establishes a secure and authenticated session for exchanging application data later on. In this phase session keys are derived through a Diffie-Hellman exchange or using pre-shared symmetric keys.

(v) **Application Data phase** The final phase involves data exchange under an AEAD protocol as detailed in the SPDM architecture whitepaper [95]. This phase also incorporates a *key update* mechanism to maintain security integrity over time.



(a) High-level view of the phases of SPDM's main process: *Device Initialization*, *VCA* phase, *Options* phase, and creation of new sessions.

(b) High-level view of the sub-phases of each new session: the *Key Exchange* phase and the *Application Data* phase.

Figure 9.1: High-level views of SPDM's protocol flow. (a) gives an overview of the connections of SPDMs different main phases, while (b) depicts an high-level view of the sub-phases within the session phase. Note that multiple sessions can be spawned and executed concurrently for each SPDM protocol run.

In Figure 9.1 we show an overview of the five phases of SPDM and the sub-phases of each new session. We will now give a more in-depth description each SPDM phase.

9.1.1 Device Initialization

Prior to initiating the protocol, parties are equipped with their initial protocol software and cryptographic resources during the *device initialization* phase. This setup is typically carried ouf messages exchanged

during the protocol, are detailed in the SPDM specification [96, p. 33–36]. Furthermore, the initialization process should include at least one of the following elements:

(i) pre-shared symmetric keys with one or more other devices,
(ii) pre-shared public keys with one or more other devices, or
(iii) a public key pair, certificates over the public key, and a root of trust to verify certificates.

Options (i) and (ii) are configured with pre-established communication partners, and interestingly, there is no specified upper limit on the number of shared keys that can be used. For option (iii), a device is capable of storing up to eight certificates. These certificates must be in ASN.1 DER-encoded X.509 v3 format as stipulated in [65]. The first certificate slot, labeled as certificate slot 0, should only be configured or modified in a secure and trusted environment. The remaining seven slots allow for dynamic updates: SPDM provides functionality for retrieving a certificate signing request from a Responder [181], and for setting certificates remotely via *GET_CSR* and *SET_CERTIFICATE* commands. It is specified that *SET_CERTIFICATE* should be executed only within a *secure session*, although the details of what constitutes such a session are not elaborately defined in the standard.

Furthermore, during initialization, vendors have the flexibility to create and implement their own custom request and response codes. This is allowed through the optional "vendor defined functionality," which enables vendors to tailor the protocol to specific needs and scenarios.

9.1.2 VCA Phase

This phase begins with the Requester sending a version request to determine the versions supported by the Responder. Upon receiving this information, the Requester chooses the highest common version supported by both parties, as recommended by the specification.

Each party possesses a set of capabilities that outline the operations they can perform under the SPDM specifications. For example, if a party has pre-shared symmetric keys, it will set and communicate the *PSK_CAP* flag. Following the capabilities exchange, both parties identify and store the common capabilities that they both support.

The exchange of supported cryptographic algorithms follows, where each party presents a list of the cryptographic algorithms they can implement, such as various signature or encryption schemes. While ideally, the strongest available cryptographic algorithms are selected, the standard does not provide a specific method for choosing these algorithms, leaving some flexibility in their selection.

The parties keep a transcript of all messages sent and received. For more specific details see Appendix A.

9.1.3 Options Phase

Runtime Responder Authentication

When establishing a connection using public keys and certificates for the first time, the communicating parties initially lack cryptographic information about each other to perform authentication. To address this, SPDM allows the Requester to obtain the public key and certificate of its partner (option (iii) of device initialization) for use in the protocol run.

After obtaining the certificate, the Requester can challenge the Responder's knowledge of the private key associated with the certificate by requesting a signature over the communication transcript and a randomly chosen Requester nonce. For details on transcript computation, refer to [96], line 355.

If the Responder's certificate was already stored in a previous session, the Requester can request a certificate digest for comparison (*GET_DIGESTS*) instead of retrieving the entire certificate. The following summarizes SPDM's functionality for verifying the responder before session establishment:

GET_DIGESTS Instead of retrieving and checking the complete certificate in every session, the Requester can store and compare the certificate's hash. If the hash of the requested certificate is not yet available, the Requester must execute *GET_CERTIFICATE*.

GET_CERTIFICATE During this phase, the complete certificate is retrieved and verified against the chain of trust. Once verified, the corresponding public key and the hash of the certificate are stored for future use.

CHALLENGE In this phase, the Requester challenges the Responder to prove knowledge of the long-term key associated with the certified public key. The responder must sign the transcript, which includes a challenge nonce, using its long-term key. Details are provided in [96], page 78, v1.2.1.

The specification recommends performing unilateral responder authentication using *CHALLENGE* at least once before proceeding with device attestation through measurements.

Device attestation through Measurements

The Requester can query *measurements* from the Responder. The Requester sends a nonce along with the measurement request the responder responds with a bitstring that represents the measurement or a set of measurements. The response is not specified in SPDM, but envisioned to be, e.g., some hash of the device state, certain software versions or any other user- or manufacturer-defined function. The Requester sends a nonce along with the measurement request. In response, the responder provides a bitstring representing the measurement or a set of measurements. While SPDM does not define the specific format of this response, it is generally envisioned to include elements such as a hash of the device state, specific software versions, or any user-defined or manufacturer-defined function.

The measurements can optionally be authenticated through digital signatures in the public key settings, provided this capability is supported by the communicating parties. However, in scenarios involving pre-shared symmetric keys, explicit requests for measurements cannot be made at this stage. Instead, the measurements may be included as part of the PSK exchange at a later point.

9.1.4 Key Exchange Phase

SPDM allows devices to establish a secure session using key exchanges in three ways (i) using certificates signed by a trusted certificate authority, (ii) using pre-shared public keys, or (iii) using pre-shared symmetric keys. Parties that support digital signatures and public key cryptography can start a Diffie-Hellman based key exchange to derive the session secret with option i) or ii). Otherwise, devices provisioned with pre-shared symmetric keys iii) can perform a key exchange based on those secrets without relying on public key cryptography.

Key Exchange with Certificates

In the key exchange process with certificates, trust in the partner's identity is established using device certificates signed by a root authority. Verification is performed through the certificate chain-of-trust, with the root certificate of a CA at the top level. During the key exchange, identity verification occurs when the parties sign a challenge using the private key associated with the certificate. Concurrently, a Diffie-Hellman computation is used to derive the base session key, known as the handshake secret in SPDM. Note that SPDM supports multiple parallel sessions.

To initiate the session, the Requester sends a *KEY_EXCHANGE* request containing an ephemeral Diffie-Hellman public key, a 32-byte nonce, 2 bytes of its contribution to the session ID, and the negotiated version. Upon receiving the request, the Responder generates its own ephemeral Diffie-Hellman key pair and computes the handshake secret. The Responder then prepares its response, which includes a 32-byte nonce, 2 bytes for the session ID, a signature of the transcript so far (signed using the private key of its device certificate), and the *ResponderVerifyData*– an HMAC over the transcript using the *finished_key*. Details on transcript construction are given in Appendix A.

Upon receiving the response, the Requester verifies the signature and computes the required session secrets, including the role-oriented handshake secrets and finished keys. The Requester then verifies the HMAC using the responder's *finished_key* and composes the *FINISH* request to finalize the key establishment and optionally authenticate to the Responder. The Requester then sends a signature of the transcript using its private key and an HMAC of the transcript with the requester's *finished_key*. The Responder, upon verification, responds with a *FINISH_RSP* message containing an HMAC of the transcript. It then derives the session encryption keys from the handshake secret and enters the application data exchange phase.

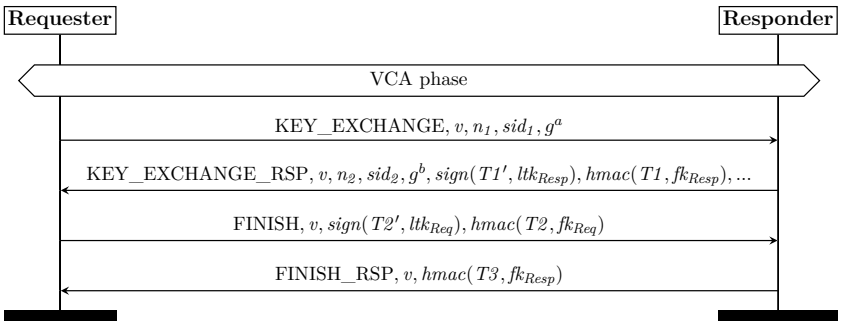


Figure 9.2: Key Exchange with Mutual Authentication in the certificate mode.

Mutual Authentication The certificate-based key exchange always authenticates the Responder, but to explicitly authenticate the Requester, the Responder needs to explicitly require it.

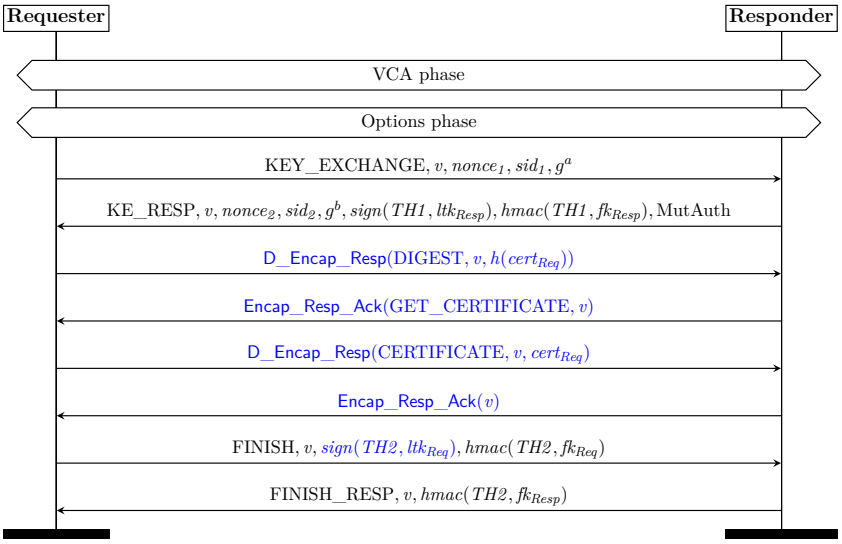


Figure 9.3: Key Exchange with Mutual Authentication in the optimized encapsulation flow. In the *FINISH* message, if Mutual Authentication is requested (MutAuth), the Responder can request digests and certificates for the Requester's public key using encapsulated messages. Additionally, the Requester does sign the transcript TH2 (which is marked in blue.) If the Responder already has the certificate of the Requester from another protocol run, they can skip directly to *FINISH*.

The Responder can signal mutual authentication by setting the *MutAuthRequested* flag in their key exchange reply. Depending on the value of this flag, the protocol can either trigger the mutual authentication mechanism using an encapsulated flow or proceed directly to the finish phase. The latter option is preferred when the certificate has already been obtained from a previous protocol run or when the parties have pre-shared public keys.

For the encapsulated mutual authentication flow, the protocol employs dedicated messages, such as *ENCAPSULATED_REQUEST*, to handle the exchange. An example of this mutual authentication process is illustrated in Figure 9.3.

Key Exchange with Pre-shared Public Keys

The key exchange using pre-shared public keys is similar to the key exchange presented in Section 9.1.4 with a message flow similar to Figure 9.2. The main difference lies in the usage of public keys: In a key exchange with pre-shared public keys, both parties are equipped during the trusted manufacturing process with long-term private-public key pairs and their partners' public keys. This eliminates the need for certificates, as they directly verify signatures over the transcript using the pre-provisioned long-term keys.

Key Exchange with Pre-shared Symmetric Keys

In the key exchange using pre-shared symmetric keys, both the Requester and Responder are provisioned with shared keys during a secure manufacturing process. These keys serve two purposes: mutual authentication and session key derivation for the application data phase. Although they can share multiple PSKs for different connections, we assume a single PSK for simplicity.

The handshake begins when the Requester sends a *PSK_EXCHANGE* request containing the protocol version, a 32-byte nonce, and a 2-byte session identifier. Upon receiving this, the Responder retrieves the corresponding PSK for the Requester and calculates the handshake secret as:

$$\text{handshake\_secret} = \text{HKDF}(\text{Salt}_0, \text{PSK})$$

From the *handshake_secret*, the Responder derives the role-specific handshake secrets and *finished_keys*, as described in Appendix A.9. The Responder then calculates the verification tag *ResponderVerifyData* using the *finished_key* as the HMAC key and responds with a *PSK_EXCHANGE_RSP* message that includes the session ID and a 32-byte nonce.

Upon receiving the *PSK_EXCHANGE_RSP* message, the Requester derives the same handshake secrets and keys as the Responder and composes the session transcript. It verifies *ResponderVerifyData* to ensure the integrity and authenticity of the response. If the verification succeeds, the Requester calculates its verification tag *RequestorVerifyData* and sends the *PSK_FINISH* request to authenticate itself to the Responder and confirm the session secrets. The Responder verifies the HMAC over the session transcript using the *finished_keys*. This step implicitly authenticates the Requester (achieving mutual authentication,) confirming that it possesses the correct shared PSK. The Responder then responds with a *PSK_FINISH_RSP* message and derives the encryption keys for the application data phase, as shown in Figure 9.6.

9.1.5 Application Data Phase

After completing one of the key exchanges, the session reaches the application data phase where Requester and Responder can exchange data and update their keys without starting a new handshake. In this phase, the roles are not bound to Requester and Responder anymore and both parties can act as a sender or receiver.

Key Update Keys can be updated in two ways: the sender can decide to only update their own key, or it can decide to update all session keys.

In the first case, the Requester sends a key update request (*KEY_UPDATE*) and, upon acknowledgment, derives the new session major secret: $msk_{i+1} = HKDF(msk_i, 'upd', '0')$. The new encryption key is then derived as $enckey = HMAC(msk_{i+1}, 'key')$. To verify the update, the Requester encrypts a request using the new key. After the Responder decrypts it, it deletes the old keys and encrypts an acknowledgment response.

In the second case, both parties' major secrets are updated, and the Responder encrypts the verification acknowledgment using its new key. If the Responder initiates the update, the same process applies, but it sends the key update through encapsulated messages, such as *ENCAPSULATED_REQUEST* (*KEY_UPDATE* (...)).

9.2 Formal Model of SPDM v1.2.1

As SPDM has shown to be a highly complex security protocol relying on several sub-protocols, we attempt the first formal analysis in two stages. First, we analyze each sub-protocol in a modular fashion, reducing the initial complexity while still covering all core mechanisms and features. Using the gained experience and the manually extracted state-machines, we construct a monolithic, comprehensive model of SPDM, covering the full protocol. The resulting Tamarin model is in size and complexity one of the biggest models to date.

9.2.1 Modular Approach

In this section we describe our formal, modular modeling approach of the SPDM protocol. Because of the size and complexity of the protocol, we split the analysis into four Tamarin models. This is an inherent limitation of the current scalability of the analysis tools, and as we will see later, our split models already push the boundary of what can be realistically handled by state-of-the-art tools. We tried to reduce the impact of this modularization by identifying naturally distinct cases in the protocol flow, and identified five main components: (i) Device initialization and the VCA phase (Section 9.2.1), (ii) Options phase (Section 9.2.1), (iii) Session setup (Section 9.2.1) used in Tamarin, (iv) Three different key exchanges (Section 9.2.1), and (v) Application data exchange and termination (Section 9.2.1). Whereas the (i,ii,iv,v) naturally correspond to the five phases of SPDM from section 9.1, (iii) is a component needed in Tamarin to connect the different state machines of Figure 9.1. We use these components to create four models:

- **Device Attestation**, includes device initialization, VCA and the options phase (i,ii).
- **Key Exchange Certificate**, includes device initialization, VCA, the key agreement with certificates, and application data (i,iii,iv-1,v).
- **Key Exchange pre-shared Public Keys**, includes device initialization, VCA, the key agreement with public keys, and application data (i,iii,iv-2,v).
- **Key Exchange pre-shared Symmetric Keys**, includes device initialization, VCA, the key agreement with symmetric keys, and application data (i,iii,iv-3,v).

We will now describe how we model the five main components (i-v), need to construct the four final models.

Device Initialization and VCA Phase

Each device gets initialized with a unique device identifier. Additionally, each device gets its supported software versions, device capabilities, and cryptographic algorithms. We decided to model them as fixed after initialization since updating, e.g. the capabilities, is not clearly specified as of now. In addition, following the standard we model three ways to initialize cryptographic key material of the device: pre-shared symmetric keys (PSK), pre-shared public keys, and public keys with an associated certificate; none or any number of them can be used for initialization. After the initialization the parties negotiate these capabilities during the *VCA* phase, as shown in the state machines in Figure 9.4a.

Certificates SPDM specifies using certificates as defined in [65], which implies the need to include a public key infrastructure (PKI) and certificate chains with a root of trust into our analysis. We created an abstract model with a single trusted root *certificate authority* (CA), which issues certificates directly to the devices. We assume that this keeps the trust anchor assumption of certificate chains in place while abstracting from all points of failures during the trust delegation. We claim that this abstraction is reasonable, as formally analyzing PKIs and certificate chains lie outside of the scope of this paper. Further, we restrict devices to only have a single slot to store a certificate for each communication partner.

In Tamarin we created the root CA from a unique public key pair and model it as a persistent fact $!\mathsf{RootCA}(key, \mathsf{pk}(key))$. In the same way a device with identifier *id* is represented by $!\mathtt{Device}(id, \cdots)$, where we use $\cdots$ to omit some details. Now given the initialized device, a fresh long-term key and the root CA, the latter can sign a certificate for the newly generated public key of the device as shown in the

following rule:

$$\left[\ !\texttt{Device}(id, \cdots), \mathsf{Fr}(ltk), !\mathsf{RootCA}(key, \mathsf{pk}(key))\ \right]$$
$$-\!\!\left[\ \texttt{HonestlyGenerated}(id, ltk, \mathsf{pk}(ltk)), \cdots\ \right]\!\!\rightarrow$$
$$\left[\ !\texttt{Cert}(id, \mathsf{pk}(ltk), sign(< id, \mathsf{pk}(ltk) >, key)),\ \mathsf{Out}(< \mathsf{pk}(ltk), sign(< id, \mathsf{pk}(ltk) >, key) >), \cdots\ \right]$$

Notice that to label the honest generation of the public key pair, we use an `HonestlyGenerated` action fact containing the identifier of the device and the keys. As we will see in Section 9.3, this helps us to express our security properties.

VCA phase For the VCA phase, we model the established channel of communication (handled by the underlying network protocol in the implementation) using so called thread identifiers *tid*, which are fresh, unique terms. In the following we can see the *GET_VERSION* request generate such an identifier and bootstrap a protocol run:

$$\left[\ !\texttt{Device}(idReq, \cdots), !\texttt{Device}(idRes, \cdots), \mathsf{Fr}(\ tid), \cdots\ \right]$$
$$-\!\!\left[\ \mathsf{StartThread}(tid\ , idReq\ , idRes)\ ,\ \texttt{Channel}(idReq\ , idRes)\ \right]\!\!\rightarrow$$
$$\left[\ \mathsf{StateReq}(tid\ , idReq\ , idRes\ , \cdots\ , \text{'GETVERSION'}),\ \mathsf{Out}(< \text{'GET\_VERSION'}, \text{'1'} >)\ \right]$$

We restrict the model to only contain a single communication channel (restriction on the `Channel` fact) per pair of devices at the same time. We then restricted that no messages are allowed to be sent using older thread ids (*tid*.) With this we aim to distinguish one run of the VCA phase from another run between the same parties. At any point the Requester sends a new version request a new thread id gets created and a main part of the device state gets deleted.

Further, transcripts of a VCA run need to be stored in each party's state and need to be updated with every message sent and received. For the details on modelling transcripts refer to Appendix A. To make accessing, updating and deleting the transcripts easier in Tamarin, we used the built-in *multiset* feature. While using multisets in state facts makes proving harder for Tamarin, we can circumvent that problem by enforcing the structure of the transcript on the multisets. With this, we do not lose any efficiency while proving, but make modeling easier and more clear for users. Moreover, during a protocol run the Requester should be able to return to the start of the protocol, by issuing a *GET_VERSION* request and restart the entire conversation. From here, all sessions and data related to that conversation are not accessible anymore, i.e., no further transitions are allowed in the old thread. This is modeled as a restriction in Tamarin:

$$\begin{aligned}&\forall\ tid1\ \ tid2\ \ idReq\ idRes\ \sharp i\ \sharp j\ .\ \sharp i\ < \sharp j\ \&\\ &\quad \texttt{StartThread}(tid1\ , idReq\ , idRes\ )@\sharp i\ \&\\ &\quad \texttt{StartThread}(tid2\ , idReq\ , idRes\ )@\sharp j\\ &\quad\ \Rightarrow \neg(\exists\ \sharp t\ .\ \sharp j\ < \sharp t\ \&\ \texttt{CurrentThread}(tid1\ , idReq\ , idRes\ )@\sharp t\ )\end{aligned}$$

As we can see, at the *GET_VERSION* rule we log an action fact called `StartThread` with a fresh thread identifier *tid* for the conversation, and later on every other transition of the protocol we always use the `CurrentThread` action fact. `CurrentThread` keeps record of the current thread being executed. The restriction can be read as: whenever an old thread tid_1 is replaced by a new tid_2, there cannot be any other transition being executed in the old thread tid_1.

Options Phase

To model this phase, we captured the multiple transitions, namely between certificates, digest, challenge and measurements. In total, our model included 17 rewriting rules. Further, we modeled the transcript needed during the responder authentication procedure using multisets as described for the VCA phase in Section 9.2.1. For the details on modelling transcripts refer to Section 9.2.1.

Responder Authentication Runtime Responder authentication encompasses the request codes *GET_DIGESTS*, *GET_CERTIFICATE*, *CHALLENGE*, and their respective response codes (see Appendix B.) With the

restriction of only modelling one certificate slot, we also model that the Requester only stores one certificate of the Responder it communicates with. Hence, when Requester request the digest of the Responders certificate, we model that verification of the digest either succeeds or that the digests do not match. With this, we need to construct a total of 4 rules to exchange a digest: (i) requesting the digest, (ii) responding to the digest, (ii) reaction of Requester if the digest is already stored, and (iv) reaction of Requester if the digest is unknown. In the case that the digests do not match, the honest Requester requests the full certificate of the Responder and verifies it using the root of trust stored in their persistent state.

When modeling *CHALLENGE* and *CHALLENGE_AUTH*, we implicitly require that the Requester issued a *GET_CERTIFICATE* request at some previous point, as it is the only means for the Requester to learn the public key of the Responder in the certificate setting.

Measurements The standard offers both signed and non-signed measurement requests. We modeled only measurement requests, for which the Requester requires the measurement response to be signed using the Responder's private signing key. As non-signed measurement requests cannot guarantee any form of authentication, they are irrelevant to our security analysis. Additionally, we needed to model two versions of measurement requests and responses: one for the shared public key setting and one where certificates are used.

Session Setup

Our formal models includes the two main session sphases: the key exchange handshake, and the application data exchange loop with key update and termination. In Figure 9.1b, we give an overview of the state machines of a session's phases.

While devices only have limited memory, there is no a priori bound on the number of parallel sessions. In our models, we therefore allow for an arbitrary number of parallel sessions, each independent from another and executing different phases of the execution. To capture this, we maintain a main state of the conversation thread and on each session creation we generate a new temporary state for that session's handshake. Specifically, we modeled a rule that given the main state of the Requester, StateReq, generates a fresh session identifier *sid* and outputs the key exchange state KeyExchangeReq. Respectively, the same transition is possible for the state of the Responder. In the agents' state machines, this transition is represented by the *Spawn Session* edge. The core of the rule is the following:

$$
\begin{aligned}
&[\ \mathsf{StateReq}(tid, idReq, idRes, v, \cdots, \text{'IDLE'}),\ \mathsf{Fr}(\ sid)\ ] \\
&-[\ \texttt{CurrentThread}(tid\ , idReq\ , idRes)\ ]\rightarrow \\
&[\ \mathsf{StateReq}(tid, idReq, idRes, v, \cdots, \text{'IDLE'}), \\
&\quad \texttt{KeyExchangeReq}(sid, tid, idReq, idRes, v, \cdots, \text{'START\_KE'})\ ]
\end{aligned}
$$

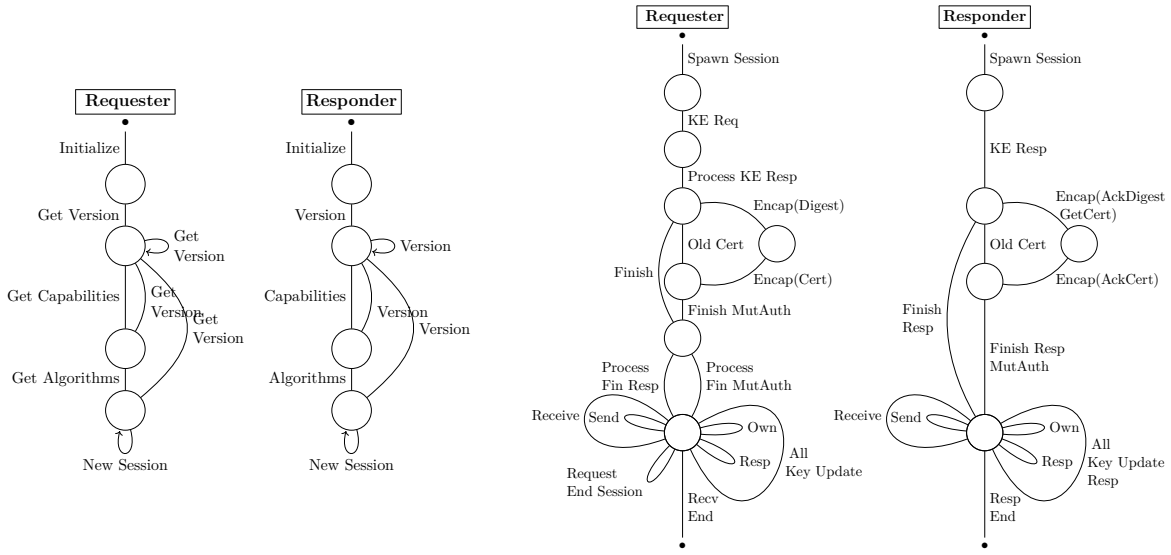
As a result, parties can share data easily between sessions by accessing the main thread's memory state. This is desirable when data needs to be accessible to all other sessions, such as when a certificate obtained in one session, should be available when creating the next.

Once at the end of the key exchange phase, the parties' states transition to the application data phase, AppDataKey. Here they are stripped of their roles as Requester and Responder during the message exchange and can send and receive messages arbitrarily. However, the key update request still adheres to their initial roles, meaning only the Requester can directly send a key update, while the Responder needs to use the encapsulated mechanisms. In the end, the Requester can terminate the specific session by issuing an *END_SESSION*.

Key Exchange Phase

For the key exchange phase we again distinguish between the three main modes to establish a session from Section 9.1.4. In the following, we describe their respective models and describe how to the final sessions keys are computed.

Certificates In this model we capture the protocol flow of the key agreement between parties that have been initialized with certificates signed by a certificate authority CA. From the specification we modeled the transitions of the Requester and the Responder to perform the handshake with unilateral and mutual



(a) Detailed state machines of the *Device Initialization* and *VCA* phase of the Requester and Responder.

(b) Detailed state machines of the *New Session* phases of the Requester and Responder with Certificates model.

Figure 9.4: Detailed state machines of the Requester and Responder in the Key Exchange with Certificates model. Each of the labeled edges corresponds to a Tamarin rule in our model.

authentication. In the second case, the Responder can either use the encapsulated flow to request for the certificate or use a certificate obtained from a previous session.

To initiate the key exchange the Requester sends a *KEY_EXCHANGE* message to the Responder in which it includes a new Diffie-Hellman public key and its random values. While sending the message, it updates the transcript by adding the current message, stores the private key and goes in a state of waiting for a response. To encode the transition, we use a multi-set rewriting rule as follows:

$$
\begin{aligned}
&[\ \mathtt{KeyExchangeReq}(sid, tid, \cdots, \text{'NULL'}, tscript, \text{'START\_KE'}),\\
&\quad \mathsf{Fr}(\ nonce), \mathsf{Fr}(\ privK), \cdots\ ]\\
&-[\ \ \mathtt{CurrentThread}(tid\ , idReq\ , idRes)\ ,\ \mathsf{KETranscriptR}(tscript)\ ]\rightarrow\\
&[\ \mathsf{Out}(< KEY\_EXCHANGE, nonce, \mathrm{g}^{privK}, \cdots >),\\
&\ \mathtt{KeyExchangeReq}(sid, tid, \cdots, privK, new\_tscript, \text{'WAIT\_RESP'})\ ]
\end{aligned}
$$

Notice that we label this transition with the action facts `CurrentThread` and KETranscriptR. The first is used to keep track of the current active thread in the conversation, as we saw previously in Section 9.2.1, while the second serves for message transcript structure checks, as we will see later.

After receiving the response, the state of the Requester can trigger three possible transitions in our model: a) send a finish request with only unilateral authentication, b) send their certificate using the encapsulated flow to mutually authenticate (see Figure 9.3), and c) send a finish request with mutual authentication if the certificate was provided in a previous session. In each of these cases, the parties cannot go back the protocol steps or change their choice within the same session. In the end, we model two ways to finish the key exchange, unilateral and mutual authentication, where in the latter the Requester also signs the protocol transcript and certificate digests.

In Figure 9.4 we give an in-depth description of the Requester and Responder state machines of setting up a session with certificates. More precisely, in Figure 9.4a we show the state machine of the Device Initialization and VCA phase which are shared across all models, and in Figure 9.4b we give the specifics of all possible modes in running a session in this model. Note that the *KEY_EXCHANGE* rule in Tamarin we saw previously, corresponds to the *KE Req* edge on the Requester state machine in Figure 9.4b. Similarly, the *Encap(Digest)* and *Encap(Cert)* represent the encapsulated mechanism for mutual authentication, while *Old Cert* the transition when the certificate has been received in a previous

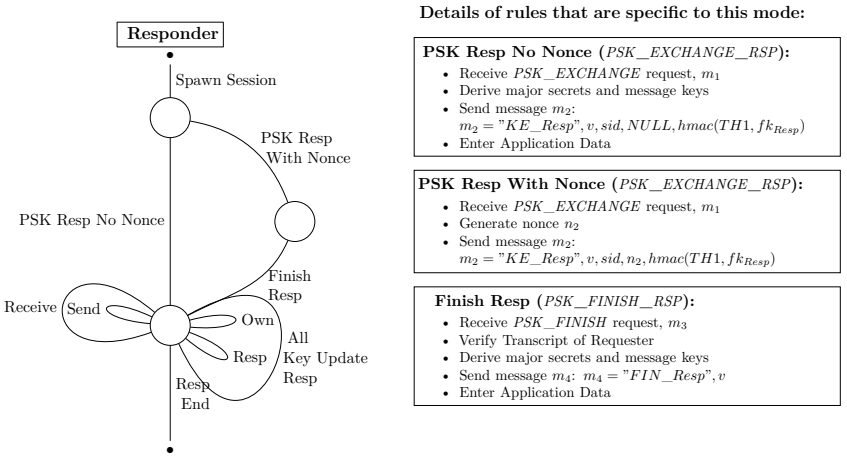


Figure 9.5: Detailed state machine of the Responder in the Key Exchange with pre-shared Symmetric Keys. We give the details of the rules that are specific to this key exchange mode corresponding to the SPDM responses *PSK_EXCHANGE_RSP*, and *PSK_FINISH_RSP*, while omitting the VCA phase, and the details of application data phase, shared across models. The state machines of the Requester are similar with the additional edges that process the responses of the Responder.

iteration. Lastly, after finishing the handshake the parties reach the Application Data state where they send and receive message, and update their keys until the session is terminated.

To represent the handshake with certificates and mutual authentication we modeled 18 rules in total, in addition to the 13 rules needed for device initialization and VCA phase.

Pre-shared public keys The key agreement using pre-shared public keys is a slight variation of our certificate model. Here, the parties do not need to exchange certificates, but rather only verify their partner's knowledge of the keys provisioned to both devices before the start of the protocol. From the certificate model, we made the following two changes: a) removed the encapsulated mutual authentication, and b) used pre-shared public keys instead of certificates to create signatures of the transcript. In total we needed 8 rules to capture the transitions of the handshake.

Pre-shared symmetric keys We modeled the two options to perform the handshake of parties who have been provisioned with pre-shared symmetric keys, namely with both parties contributing on the session secret derivation or only the Requester. The distinction lays on whether the Responder expresses intent to contribute by sending a random nonce in the key exchange response. Concretely in the protocol flow, the Responder either directly enters the application data phase after the key exchange request/response or continue with the finish request/response.

To model the decision, we write two distinct rewriting rules as seen in the state machine of the Responder in Figure 9.5. Notably, after receiving the key exchange request *m1*, the Responder either replies without a nonce and enters the application phase, or sends a nonce and waits for the finish request *m3*. Similarly, the Requester receives *m2* on two different rules, where the one that requires the nonce not to be Null sends out *m3*, and the other enter the application phase. The messages from *m2* until *m4* are encrypted using symmetric cryptography with the role-dependent handshake keys derived at this point in the protocol. In our model, this is expressed using the symmetric encryption theory, where $senc(m4, resp\_hkey)$ models the encryption of the finish response *m4* with responder handshake secret *resp_hkey*. In total we model the pre-shared symmetric key agreement with 9 Tamarin rules.

Transcripts During the protocol run, the parties need to sign and/or authenticate the transcript of the conversation. Transcripts are concatenations of different messages, e.g., in the key exchange it is the concatenation of the following: 1) messages of the VCA phase, 2) hash of Responder's certificate or public key, 3) key exchange request/response 4) hash of Responder's certificate (if mutual authentication) or public key, and 5) finish request/response messages. For the pre-shared private keys case, items 2 and 4 are not included. Note that the transcript only includes the already issued and the current message. For

example, when the Responder signs the transcript in the key exchange response it will only include up to item 3.

In our models, we store the messages of the transcript using multisets. In Tamarin's syntax, + denotes multiset union. We initialize two variables, respectively for the VCA phase and key exchange phase, and update their content on every new message exchange. To help the tool's reasoning, we also prove helper lemmas to show the consistency of the transcript structure in these variables, like in the following:

$$\begin{aligned}&\forall\ ke\_transcript\ \sharp i.\ \mathsf{KETranscriptR}(ke\_transcript)@\sharp i\\ &\Rightarrow (\exists\ m1\ m2\ m3\ m4.\\ &\quad ke\_transcript =< \text{'Get\_Key\_Exchange'}, m1 >\\ &\qquad + < \text{'Key\_Exchange\_Resp'}, m2 >\\ &\qquad + < \text{'Finish'}, m3 > + < \text{'Finish\_Resp'}, m4 >)\end{aligned}$$

The lemma states that all KETranscriptR labels at time $\sharp i$ containing the transcript of the key exchange *ke_transcript*, will have a transcript with the defined structure. Transcripts are heavily used in the protocol, not only to be verified and sent to their partners, but also to derive the session keys and the authentication keys for the transcripts themselves as we will see next.

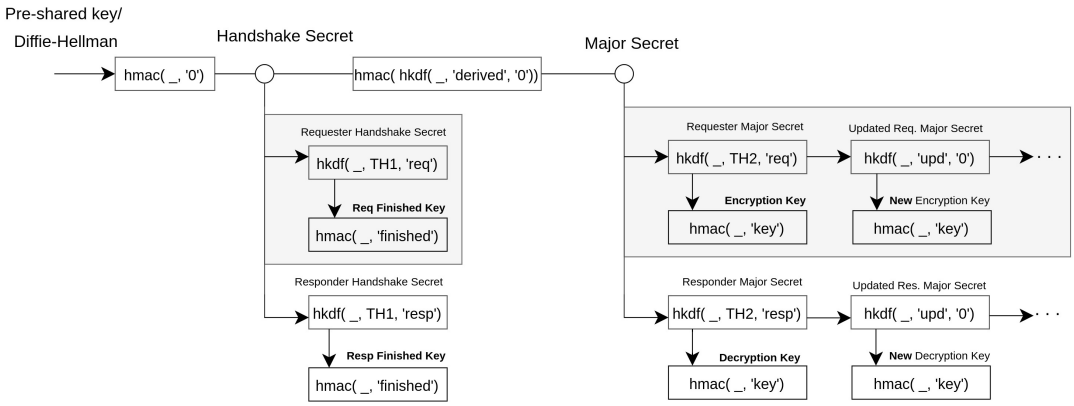


Figure 9.6: Derivation of finished key, session keys and update of message keys for Requester, symmetrically for the Responder. The _ is a placeholder for the input secret in the function, e.g., the handshake secret is calculated as $hmac(key_{pre-shared}, '0')$ in the pre-shared key setting. For the certificate model, transcript TH_1 and TH_2 are defined as follows: $TH_1 = T_{VCA} \parallel h(cert_{Resp}) \parallel T_{KeyExchange}$, and $TH_2 = T_{VCA} \parallel h(cert_{Resp}) \parallel T_{KeyExchange} \parallel h(cert_{Req}) \parallel T_{Finish}$. See Appendix A for more details on transcripts.

Session key derivation During the key exchange the parties need to derive two keys: a) the finished-keys, used for authenticating the transcript, and b) the encryption/decryption keys, used to send encrypted data during the application phase. Starting from the shared secret, which can be a Diffie-Hellman output or a pre-shared symmetric key, both parties derive role-oriented secrets by incorporating the transcripts and pre-determined strings. This mean that the parties derive their own key to encrypt and their partner's key to decrypt the messages. The same is applied for the finished key.

The mechanism uses an *HMAC* and an *HKDF* function to expand and derive keys, as defined respectively in [147] and [148]. However, in the protocol, the parties decide the hash algorithm to instantiate these functions during the VCA phase. In Tamarin, we defined two functions symbols to model the same functionality. In Figure 9.6 we can see how the entire key derivation is performed by the Requester.

Application Data Phase

The application data phase starts at the end of the key agreement, as shown in Figure 9.4b. At this point in the protocol, the parties are no longer restricted to their roles as Requester and Responder. In fact,

either of them can send and receive messages. To capture this, we modeled two rules *Send_Message* and *Receive_Message*. In the first, the sender encrypt a fresh payload using their encryption key and sends it in the network. In the latter, the receiver decrypts the cipher text using their decryption key.

At any point during the session, the parties can update their own keys or all keys of the session. This includes several back and forth between the parties, either in their normal flow or in an encapsulated way (for the Responder). To model this mechanism we had to abstract the request and verify in the same step. This was due to the large state the protocol has accumulated at this point in the protocol, which makes it difficult to reason about the key secrecy. In total we used 6 rules to model the back and forth of the messages and a restriction to deprecate the old session key.

In the end, the Requester can send a *END_SESSION* to finish the application data and remove all secrets from the memory. Once the partner processes the request, they send an acknowledge to end the session and will no longer send or update in this session. On processing the response the Requester performs the same operations.

9.2.2 Monolithic Approach

In a second attempt, using the gained insight from the modular approach, we attempted to model SPDM in a monolithic fashion. Our model of SPDM 1.2.1 as a whole comprehensively captures all main interactions among sub-protocols in the Tamarin prover. This model contains all components described in Section 9.2.1. It supports an unbounded number of devices that can initiate unilateral SPDM connections with any other devices, allowing device A to establish two connections (in Requester and Responder roles) with all available partners. Additionally, within a connection, the parties can spawn an unlimited number of sessions in any of the three modes.

In Figure 9.7, we present the state machines for the Requester in each individual mode's intended flow, as well as a state machine modelling SPDM's session establishment as a whole. These state machines are based on the standard's specified transitions. The rightmost figure highlights the complexity of the key exchange composition of the three modes. However, the complexity shown in the figure is only a subset of the actual state machine model, which includes far more transitions, for example to model VCA and device attestation mode request and responses. Our final model includes 71 Tamarin rules, where we model the device initializations with 5 rules, 1 rule for malicious certificates, 7 for the VCA phase, 16 for the Options phase, 2 rules to spawn sessions, 29 rules for the key-exchange showed in Figure 9.7, and 11 rules for the application data phase.

To analyze the protocol for the desired properties with respect to an adversary that controls the network, we do not just consider one instance of the Requester and Responder state machine. Instead, our final Tamarin model considers that participants can potentially execute an unbounded number of instances of the Requester or Responder state machine concurrently, while communicating over a network that is completely controlled by a Dolev-Yao adversary. I.e., the adversary can decrypt any messages it receives for which it knows the corresponding keys, can generate fresh values, and can construct arbitrarily complex messages from the knowledge derived from messages it observed, and send these to the participants.

We discuss the design choices we made to enable the analysis of such a complex model in Section 9.4.

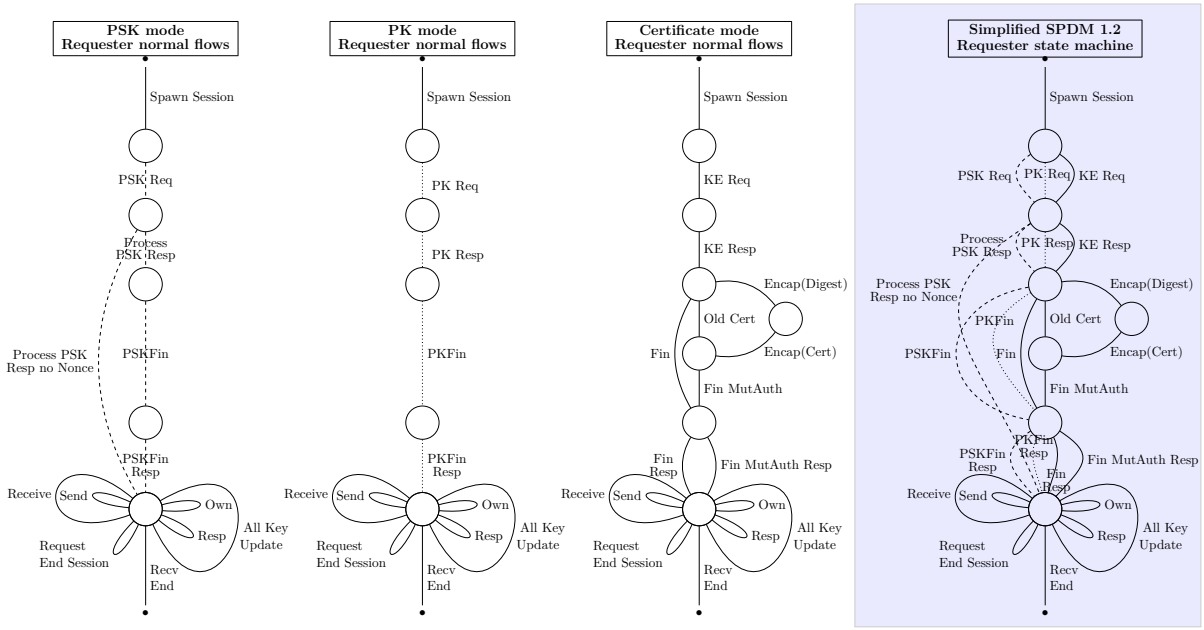


Figure 9.7: To illustrate a small part of our model, we provide simplified state machines for the Requester for the normal flows of the key exchange modes (the three on the left), and our model of SPDM as a whole (highlighted on the right) for a single session. To simplify the presentation, we omit several edges that concern concurrent session handling and session resets, but which are included in our formal model. The dashed lines denote transitions unique to the message flow of key exchange states in PSK mode, dotted lines denote PK mode, and solid lines the certificate mode and shared edges. We constructed these state machines based on the standard's description of (a) transition/event preconditions and (b) flow sequences, because the SPDM 1.2 standard and reference implementation do not specify any explicit state machines. Each edge in such a state machine is modeled by a Tamarin rule in our model.

9.3 Security Properties and Threat Model

We now turn to formalizing the security properties of the SPDM models which we described in Section 9.2.1 and Section 9.2.2. Recalling previously defined security objectives of SPDM, we first introduce the security properties centered on identity verification and firmware integrity – which we refer to as *device attestation* throughout this thesis – and discuss them in Section 9.3.1. Additionally, to address SPDM's goal of ensuring private and secure data communication, we outline the relevant properties for *secure session establishment* in Section 9.3.2. Lastly, we will define the threat model we consider for our analysis in Section 9.3.3.

9.3.1 Device Attestation

The main security properties during the device attestation phase of the protocol encompass the authentication of the Responder by the Requester and the integrity of measurements.

Unilateral Responder Authentication The primary authentication guarantee of SPDM focuses on the authentication of the Responder by the Requester, namely *unilateral responder authentication* where only the Requester receives an assurance of the Responder's identity. This guarantee can be obtained in two distinct ways: 1. **before** the key exchange when the parties agreed on using public key cryptography and certificates, and 2. **during** the key exchange when the parties agreed on using public key cryptography, certificates, and mutual authentication was disabled.

We derive our formalization of authentication from the hierarchy established by Lowe [163], which outlines that if a party A in a specific role $role_A$ participates in a protocol session with another party B and agrees on certain data ds, there must indeed be a party B assuming the role $role_B$ who is running the

protocol and agrees on the same data *ds*.

In the SPDM context, where public keys serve as the default identifiers for parties, ensuring that an Requester is actually communicating with the correct Responder equates to confirming that the interaction is occurring with the legitimate owner of the public key. As the public key of the Responder can be compromised, we define the authentication property as follows: for every successful authentication attempt by an initiator, we require that 1. the public key challenged belongs to an honest party, and 2. there is a corresponding responder (Responder) who possesses the correct public key and is actively participating in the protocol session.

As an example, we define responder authentication before a key exchange as follows in Tamarin:

Definition 13. *Responder Authentication 1:*

$$\begin{aligned}
&\forall\ tid1\ \ id1\ \ id2\ \ pk2\ \ ltk2\ \ \sharp i\ \sharp j\ .\\
&\mathsf{CommitChallenge}(tid1, id1, id2, pk2)@\sharp i\\
&\wedge\ \mathsf{HonestlyGenerated}(id2, ltk2, pk2)@\sharp j\\
&\Rightarrow\ (\exists\ tid2\ \sharp t\ .\ \sharp t < \sharp i\ \wedge\ \mathsf{RunningChallenge}(tid2, id2, ltk2)\ @\sharp t\ )
\end{aligned}$$

Responder Authentication 2 representing the authentication during the key exchange is formalized analogously but uses different action facts. Note that the difference between formulating responder authentication in both the modular and monolithic approach is only syntactically.

Measurement Integrity Similar to the unilateral responder authentication, integrity of device measurements is another goal of SPDM. Whenever an Requester requests measurements using the public key of an honest Responder, the Requester should be communicating with the intended Responder, and the measurements that the Responder sends should be received by the Requester.

The SPDM standard specifies two methods for requesting measurements during device attestation, based on the type of key material used: certificates or pre-shared public keys. For example, in Tamarin the latter is specified as follows:

Definition 14. *Measurement Integrity PK:*

$$\begin{aligned}
&\forall\ tid\ \ id_{Req}\ \ id_{Rsp}\ \ \mathsf{pk}_{Rsp}\ \ data\ \sharp i\ .\\
&\mathsf{ReceiveMeasurementPK}(tid, id_{Req}, id_{Rsp}, \mathsf{pk}_{Rsp}, data\ )@\sharp i\\
&\Rightarrow\ (\exists\ id\ \ sk_{Rsp}\ \sharp t\ .\ \sharp t < \sharp i\ \wedge\ (\mathsf{pk}_{Rsp} = \mathsf{pk}(sk_{Rsp}))\ \wedge\\
&\mathsf{SendMeasurement}(tid, id, id_{Rsp}, sk_{Rsp}, data\ )\ @\sharp t\ )
\end{aligned}$$

Typically, Tamarin can prove such lemmas quite easily. However, due to the intricate loops within the SPDM protocol, Tamarin's backward search might unroll these loops excessively, potentially leading to non-termination. To assist Tamarin in proving these properties, we had to manually add extra helper lemmas. This requirement is likely due to the complex looping behaviors encountered during device attestation. The complexity increases further in our monolithic approach due to the various key exchanges and how sessions are initiated, which complicates the proof process even more.

9.3.2 Secure Session Establishment

Next to unilateral responder authentication, the main goals of the session establishment is to create a mutually authenticated session while guaranteeing the secrecy of the final keys after the key exchange.

Mutual Authentication We define mutual authentication of the parties at the conclusion of the key exchange in all three modes of the protocol. This property ensures that once an Requester has committed to a specific Responder and an agreed-upon transcript by the end of a mode, then that Responder should also be actively engaged in the protocol, using the same transcript and operating within the same mode. This requirement should hold for both directions. We formally define this property both for public keys and for pre-shared symmetric keys. As an example, we define mutual authentication for public keys in Tamarin like:

Definition 15. *Mutual Authentication PK: We define mutual authentication for models with pre-shared public keys and certificates as follows, where we require agreement over the used public keys:*

$$\begin{aligned}
&\forall\ sid1\ tid1\ pk1\ pk2\ secret\ TH2\ role1\ id2\ ltk2\ \sharp i\ \sharp j\ .\\
&\mathsf{CommitMutAuth}(sid1, tid1, pk1, pk2, secret, TH2, role1\ )@\sharp i\\
&\wedge\ \mathsf{HonestlyGenerated}(id2, ltk2, pk2)@\sharp j\\
&\Rightarrow\ (\exists\ sid2\ tid2\ role2\ \sharp t\ .\ \sharp t < \sharp i\ \wedge\ not(role1 = role2) \wedge\\
&\mathsf{RunningMutAuth}(sid2, tid2, pk2, pk1, secret, TH2, role2\ )\ @\sharp t\ )
\end{aligned}$$

In our monolithic approach, we further refine the property by dividing it into two distinct parts: one for certificate-based handshakes and another for handshakes that use pre-shared public keys. For pre-shared symmetric keys, we model the property analogously. However, there is a possibility that parties might end up executing the protocol with themselves if they are allowed to use the same key for both the Requester and Responder roles. We address this by including a specific check to ensure that pre-shared keys are used only in one direction, preventing such self-directed interactions within the protocol.

Secrecy of Handshake Key We define the secrecy of the final handshake keys for each key exchange mode and for each role, leading to a total of six secrecy lemmas. As an example, we define handshake secrecy of an honest Requester as:

Definition 16. *Handshake Secrecy:*

$$\begin{aligned}
&\forall\ sid\ tid\ id_{Req}\ id_{Rsp}\ pkRes\ secret\ id\ ltk\ \sharp i\ \sharp j\ .\\
&\mathsf{SesssionMajorSecretReq}(sid, tid, id_{Req}, id_{Rsp}, pkRes, secret)@\sharp i\\
&\wedge\ \mathsf{HonestlyGenerated}(id, ltk, pkRes)@\sharp j\\
&\Rightarrow\ \neg(\exists\ \sharp t\ .\ \mathsf{K}(secret)@\sharp t)
\end{aligned}$$

9.3.3 Threat Models

For both the modular approach (see Section 9.2.1) and the monolithic approach (see Section 9.2.2) we analyze the protocol models against the same class of attacker, who can control the network and create malicious certificates.

Attacker-controlled Network The attacker in our models has full control over the network. We use the built-in Dolev-Yao attacker of Tamarin, which can inject, modify and drop any messages in the network.

Malicious Certificates The attacker can register a malicious certificate for any honest device. For example, the attacker can abuse the Certificate Authority to sign for a victim device a private-public key pair that are known to the attacker. We modeled this with a rule that takes as input the Certificate Authority state, !RootCA (key, pk(key)) with private key key, an attacker provided private key ltk_{bad}, and the identifier of the victim's device id_{honest} and outputs a correct certificate $cert$ in the network, $cert = sign(< id_{honest}, pk(ltk_{bad}) >, key)$. We question whether such an attacker can break the authentication guarantees in the certificate mode of the key agreement.

9.4 Addressing Challenges

We faced several challenges when using the Tamarin prover to process and analyze this substantially large model. We identified three major challenges introduced while modeling and analyzing SPDM as a whole:

1. For the models described in Section 9.2.1, the Tamarin prover already faced challenges with the size of the models, which are much smaller than the monolithic one (Section 9.2.2). Constructing a model capturing all modes in a coherent manner is expected to significantly increase these issues. Throughout the modeling process, there were instances where Tamarin encountered difficulties in loading the models or even aborted during proof attempts.

2. The Tamarin prover lacks a modular composition/proof framework, resulting in proofs of modular models being non-transferable and not reusable for proving similar properties in the model of SPDM as a whole. This extends to the technical formulation of proven properties and their helper lemmas, where we needed to redo the proof effort. In essence, the proof process does not benefit from reusability and modularity, making it more challenging to establish proofs for the monolithic model based on those of the modular models.
3. The unbounded number of devices, sessions, application messages exchanged, multiple keys and non-standard transcripts being computed simultaneously from multiple sources led to difficulties when exploring the proof space or finding the needed helper lemmas to guide the tool.

To address these challenges in our model of SPDM as a whole, we incorporated several changes and modeling decisions which we outline in the following.

Modeling the session handling of SPDM as a whole In the models from Section 9.2.1, different modes were verified separately, which implied that the individual models did not have access to the complete information that might be present in a full implementation. We solved this by including a modeling trick in the session handling layer that artificially added information in the state of the key exchange mode from the other phases of the protocol, e.g., receiving certificates. In contrast, our monolithic model explicitly includes all phases, removing the need for this modeling trick. As a result, we could model the session-handling layer in two rules, instead of the six rules of the models from Section 9.2.1, which reduced the complexity of the session-handling component.

Detailed Transcripts We model the key exchange and the device attestation transcripts to explicitly include all the fields of the messages exchanged. In the modular models, the transcripts would only include the message fields needed for the specific sub-protocol. Then, we proved a helper lemma to help guide Tamarin proofs on the structure of the transcript. While we compute transcripts as in the modular models, having this consistent structure throughout the full model improves Tamarin's proof search.

Helper Lemmas To enable analysis of our complex model, we needed to develop some helper lemmas to guide the proofs of the main properties. This was done by manually exploring partial proofs in the user interface and then providing feedback to the tool in an iterative fashion. In general our helper lemmas are of four categories:

1. *Loop breakers* that help reason about sequences of messages happening one after the other in a loop, e.g., the Requester can request a certificate and authenticate the same Responder's certificate on repeat.
2. *Message ordering lemmas* that help the prover when reasoning about the order of events,
3. *certificate and key origin lemmas* that help close the branches with artificially created secrets in the proof search, e.g., the accepted certificates can only be signed by the root Certificate Authority, and
4. *structure lemmas* that verify the format of certain terms, e.g., the format of a certificate's digest.

In general, these classes of lemmas can be useful when verifying future protocols to provide structure to the protocol sequences and the possible sources of secrets.

Example 4. *As an example, to prove mutual authentication for the certificate mode, we needed to guide the proof with some helper lemmas. At a high level, we wrote six helper lemmas where we prove that all stored digests of certificates adhere to the same format across the protocol, that the certificates were created by a root authority, and this authority existed before. Below we give the helper lemmas for the Requester and indicate to which category they belong. The lemmas are written analogously for the Responder.*
Initially we prove that any digest the Requester has received and stored should be signed by a root authority and have the defined structure. This helper lemma falls into category 4. structure lemmas.

$$
\begin{aligned}
&\forall\ tidI\ \ oidI\ \ oidR\ \ pkR\ \ digestR\ \sharp i\ \sharp j\ .\\
&\mathsf{IStoredCert}(tidI, oidI, oidR, pkR, digestR\ )@\sharp i\\
&\wedge \mathsf{CreateRootCert}(rootkey)@\sharp j\ \ \wedge\ \ \sharp j < \sharp i\\
&\wedge \neg(pkR = NULL)\\
&\Rightarrow\ \ digestR = h(sign(< oidR, pkR >, rootkey))
\end{aligned}
$$

Then, we show that root authority should have been created before storing the digest. This helper lemma falls in category 2. message ordering lemma.

$$\begin{aligned}&\forall\ tidI\ \ oidI\ \ oidR\ \ pkR\ \ digestR\ \sharp i\ \sharp j\ .\\&\mathsf{IStoredCert}(tidI, oidI, oidR, pkR, digestR\ )@\sharp i\\&\wedge \mathsf{CreateRootCert}(rootkey)@\sharp j\\&\wedge \neg(pkR = NULL)\\&\Rightarrow\ \sharp j < \sharp i\end{aligned}$$

Lastly, we show that the certificate of the digest stored should have been created before. This helper lemma falls in category 3. certificate and key origin lemmas.

$$\begin{aligned}&\forall\ tidI\ \ oidI\ \ oidR\ \ pkR\ \ digestR\ \sharp i\ .\\&\mathsf{IStoredCert}(tidI, oidI, oidR, pkR, digestR\ )@\sharp i\\&\wedge \neg(pkR = NULL)\\&\Rightarrow\ \exists(someoid\ \sharp j\ .\mathsf{GenDeviceCert}(someoid, pkR\ )@\sharp j\ \wedge\ \sharp j < \sharp i\ )\end{aligned}$$

Custom Proof Search Heuristics We noticed that Tamarin was not able to automatically prove some helper lemmas reasoning about the origin of certificates. While investigating the proving attempts of Tamarin, we observed that the proof search would explore branches that unrolled looping behaviors. To guide the tool, we created a custom heuristic with Tamarin's built-in *tactic* feature, prioritizing sources which, e.g., model attacker knowledge of signatures and certificates.

Domain Separation of Keys The design of SPDM deviates from modern designs like TLS 1.3 in that it does not explicitly use tags in the key derivation functions to provide domain separation between keys. Such tags simplify proof construction. Fortunately, the key derivation functions in SPDM include the transcripts. From the SPDM 1.2 specification, we establish that the transcripts of the key exchanges consistently differ between the three modes: (i) the PSK mode includes the *psk exchange request code*, while the other two modes include the *key exchange request code*, and (ii) the latter modes differ because of the slot_id value, where *0xFF* is used explicitly for preshared mode, and a value between *0x00-0x07* for the certificate mode. Thus, in practice, there is domain separation for the different keys in SPDM 1.2. Since these differences occur deep within the transcript, we aided Tamarin to find this separation earlier during its proof search by tagging the keys.

9.5 SPDM Analysis

We analyzed the previously defined properties for the models of the both the modular and monolithic approaches. For the modular approach, we were able to automatically prove all desired security properties. For the monolithic model, however, we did find an attack on the mutual authentication of SPDM. We will present that attack first in Section 9.5.1, before proposing fixes. We will then present all results for both, the modular and the fixed monolithic models, in Section 9.5.2.

9.5.1 Mode-Switch Attack

During our analysis using Tamarin, we discovered a mode switch attack that compromises mutual authentication in the key exchange protocol when using pre-shared symmetric keys (PSK). This attack allows a network attacker – without knowledge of secret keys or malicious certificates – to establish a secure session with the responder by exploiting missing state separations. Essentially, the attacker initiates the key exchange using certificate mode and then switches to PSK mode mid-handshake. The responder mistakenly believes it has authenticated an honest partner and establishes session keys, which are fully known to the attacker.

Breaking Authentication

As the attack relies on starting a key exchange in certificate mode and ending in the mode of the key exchanges that uses pre-shared keys, we coin this attack *mode-switch attack*. To illustrate the attack, we provide a message sequence chart in Figure 9.8, and we will look into how to execute this attack based on the SPDM 1.2.1 standard [96]. The attacker starts the key exchange in certificate mode, causing

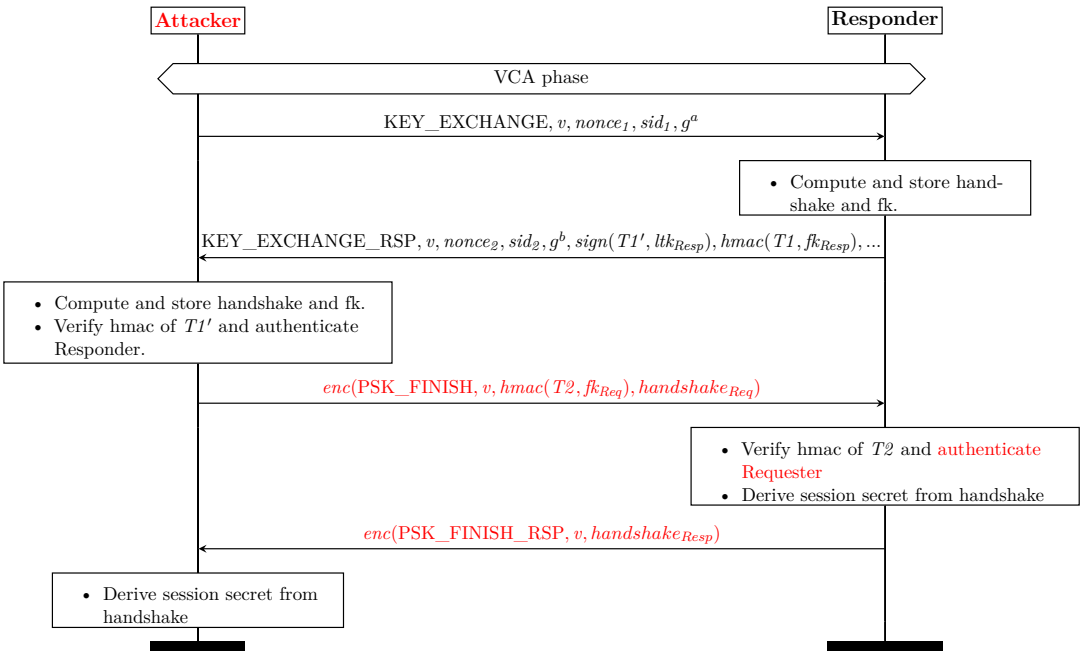


Figure 9.8: Message sequence diagram of the discovered mode switch attack. Messages in red highlight the attack behavior.

the Responder to compute and store keys derived from the Diffie-Hellman. After receiving the answer, the attacker switches to PSK mode by sending a *PSK_FINISH* request. The Responder accepts the request, believing it authenticated an honest initiator using PSK, but the session keys remain based on the Diffie-Hellman output of the attacker.

This attack on mutual authentication occurs because: (i) the responder (Responder) fails to verify the consistency of the protocol mode being executed, and (ii) the keys for PSK mode are precomputed and stored during the earlier certificate mode handshake. The attacker takes advantage of the differences in how authentication is handled between the two modes. By switching the protocol's context mid-handshake, the attacker influences the use of the generated keys. As a result, a session that should only achieve one-sided authenticated is mistakenly interpreted as a mutually authenticated session by the Responder and thereby compromising the security guarantees of SPDM.

Attacking the Implementation In addition to the standard, we inspected SPDM's open-source reference implementation libspdm [99]. We confirmed the attack on libspdm version 2.3.1 (released on January 10, 2023) [100] which propagates back to versions as far back as 1.0.0 (released on December 21, 2021).

Fixing Authentication

To address the mode-switch vulnerability in the SPDM protocol and its reference implementation, we recommend solutions that involve stricter protocol state management and explicit mode verification.

The mode switch attack is enabled by the lack of separation between protocol states for different key exchange modes and missing checks on valid transitions. We propose two key fixes:

1. *Separate Storage of Keys:* Store handshake secrets and finished keys separately for each mode to prevent improper reuse across modes.
2. *Mode Verification:* Explicitly verify the key exchange mode during every handshake transition to ensure consistency.

Proposed Implementation To fix the implementation, we require the protocol to store and verify the key exchange mode throughout the session. The Responder should only accept a finish request if it matches the mode initiated. For this we utilize the existing *use_psk* flag in the SPDM implementation to enforce mode-specific checks before accepting finish requests:

1. Allow *PSK_FINISH* only when *use_psk* is true, and
2. allow *FINISH* only when *use_psk* is false.

To improve long-term robustness, we suggest replacing the simple boolean flag with a detailed state representation of the key exchange mode.

Impact on our Monolithic Tamarin Model We modeled these changes in Tamarin by introducing variables representing the protocol mode and enforcing mode-specific constraints before finish requests are accepted. These constraints ensure that mode consistency is maintained throughout the handshake, avoiding unauthorized transitions.

Responsible Disclosure We shared our findings, proof-of-concept attack on libspdm, and our proposed fixes with the SPDM development team in March 2023. The issue was classified as a critical vulnerability (CVE with CVSS score of 9.0) [84] and fixed in SPDM specification version 1.3.0 [94]. A patch was also released for the reference implementation. For more details on the attack implementation, the implemented fix, and the interaction with DMTF, we refer the reader to Cremers, Dax, and Naska [75].

9.5.2 Final Analysis Results

In Table 9.1 we summarize the main security guarantees that we proved in our modular models and our monolithic model. Tamarin automatically proves all guarantees in this section and their helper lemmas. By applying the proposed fix and modeling it in Tamarin, we successfully verified all main properties related to our monolithic SPDM model, and summarize the results in Table 9.1. Our analysis implies the absence of the previously discovered mode switch attack.

We ran our models on an Intel(R) Xeon(R) CPU E5-4650L 2.60GHz machine with 756GB of RAM, and 4 threads per Tamarin call. For the modular models, the execution time per property proven spans from 3s (Device Attestation, responder authentication) to 4m09s (certificate model, handshake secrecy). It is important to note that due to the substantial size of our monolithic Tamarin SPDM model, each lemma execution required between 3 and 20 minutes. All the main properties with their helper lemmas are proven in under 3 hours.

To ensure transparency and reproducibility, we have made the models, the results, and the necessary resources to reproduce all findings publicly available [77, 78].

Model	Property	#Helper Lemma	Result	Runtime (s)
Device Attestation	Responder Authentication 1	-	✓	3
	Measurement Integrity Cert	7	✓	6
	Measurement Integrity PK	7	✓	6
Certificates	Responder Authentication 2	-	✓	53
	Mutual Authentication PK	-	✓	91
	Handshake Secrecy (Init/Resp)	-	✓	249
Pre-shared Public Keys	Mutual Authentication PK	-	✓	33
	Handshake Secrecy (Init/Resp)	-	✓	18
Pre-shared Symmetric Keys	Mutual Authentication PSK	-	✓	13
	Handshake Secrecy (Init/Resp)	-	✓	10
Monolithic Model	Responder Authentication 1	-	✓	324
	Measurement Integrity Cert	9	✓	251
	Measurement Integrity PK	7	✓	232
	Responder Authentication 2	-	✓	352
	Mutual Authentication Cert	6	✓	516
	Mutual Authentication PK	-	✓	444
	Mutual Authentication PSK	-	✓	267
	Handshake Secrecy Cert (Init/Resp)	-	✓	900
	Handshake Secrecy PK (Init/Resp)	-	✓	931
	Handshake Secrecy PSK (Init/Resp)	-	✓	538

Table 9.1: Summary of formal analysis results of the modular and monolithic approaches in Tamarin. The listed properties and their helper lemmas are proven automatically. The runtime shows the time it takes for the tool to prove the main property.

10 Limitations and Discussion

Previously, we evaluated the SPDM protocol based on our detailed models and state machines developed – using the Tamarin prover. Our work aimed to provide better understanding on the protocol's mechanisms and add trust and insight during the time of specification before the protocol becomes widely deployed. Additionally, we believe that our detailed models as well as our state machines can support future research and aid implementation efforts.

Discovering a critical design flaw in SPDM's implementation and specification did not only help to improve the protocol but also highlighted the importance of formal security analysis. We have invested considerable effort into modeling the protocol, leading to the discovery of even more potential flaws within the SPDM standard. Here, we will discuss these design flaws, reflect on our analytical efforts, and suggest directions for further research to refine and enhance protocol security.

First, we will report the effort that went into our analysis in Section 10.1. Afterwards, we will discuss the specific design flaws these efforts have revealed, and consider how they might affect the protocol in Section 10.2. We will then discuss the current limitations of our approach in Section 10.3. With Section 10.4, we will include suggestions for further research and adjustments needed to improve SPDM, while also discussing the current state of symbolic tools like Tamarin.

10.1 Modeling Effort

We spent about 6-7 person-months modeling SPDM in our modular approach as we carefully double-checked each step to ensure everything matched the specified standards. The detailed models developed through this process consist of between 1000 to 1800 lines of code. Additionally, modeling and analyzing the monolithic model of SPDM took about 3-4 more person-months of work. This contains the time needed to uncover the mode switch attack, which then required us to go back and make substantial revisions to the model. Our final model alone includes roughly 3800 lines of code.

				#Lemmas			
	LoC	#Rules	#Sources	secrecy	attestation	authentication	sanity
Pre-shared Symmetric Keys	1109	33	101	2	/	2	9
Pre-shared Public Keys	1412	32	106	7	/	5	9
Certificates	1798	41	129	2	/	6	14
Device Attestation	981	29	91	/	9	/	12
Monolithic Model	3768	71	371*	6	12	15	15

Table 10.1: SPDM models from prior analysis [76] compared to our holistic model. Tamarin uses a backward search method to identify the origins of all protocol facts. The term "sources" in this context refers to the specific partial executions that result in the generation of a given protocol fact. "Sanity" lemmas refer to lemmas that check that the protocol model can be executed correctly.

* = Loading the sources for our SPDM model in the browser required over one hour, indicative of the greater computational demands associated with our models expanded scope.

Table 10.1 highlights a significant increase in the number of rules and proof steps in the monolithic SPDM model compared to the modular models. Notably, the monolithic SPDM model incorporates 71 rules producing 371 unique sources derived from 28 cases. Sources are precomputations performed by Tamarin to enable its backwards search. This makes it one of the biggest Tamarin models up until now in the literature.

In addition to the increased model size compared to the smaller, modular models, the complexity of proving each desired property also increased significantly. For instance, the monolithic model requires significantly more proof steps to prove mutual authentication – 261 steps spread across 20,733 lines – compared to 53 proof steps in the key_exchange model, which spanned 4,512 lines. This increase highlights the expanded coverage and the increased complexity.

10.2 Potential Design Flaws

In the preceding chapter, we have focused on the positive results of properties that hold in SPDM as shown by our formal analysis. However, our modeling and analysis did yield several other observations. We will elaborate on potential flaws introduced by SPDM's specification in the following. We group these flaws into two major categories: *Underspecified behavior* and *prudent design choices*.

Underspecified Behavior Currently, the SPDM specification leaves many cryptographically relevant aspects underspecified, which can lead to ambiguities and potential security risks. The underspecified behaviors are: (i) inconsistent authentication properties due to unclear SPDM identifiers, (ii) unrestricted vendor-defined request and response mechanisms, and (iii) insecure provisioning of protocol secrets.

(i) (*Inconsistent authentication properties*) The specification references multiple identifiers but does not clearly specify which ones are used to uniquely identify devices. While communication relies on UUIDs, the specification primarily focuses on OIDs. Furthermore, certificates typically bind only the public key, with the OID being optional. Even if adding the OID would be mandatory, which would effectively lift them into transcripts, this would only help in the certificate-based public key setting. This leaves the pre-shared modes unsolved. As a result, the authentication process lacks a clear and consistent definition.

(ii) (*Vendor-defined behavior*) To ensure flexibility, the standard explicitly permits vendor-defined request/response mechanisms with minimal restrictions. None of those restrictions mentions long-term or ephemeral secrets from the protocol. Reusing them could undermine the security guarantees of the protocol. We recommend prohibiting the reuse of protocol secrets in vendor-defined mechanisms. Instead, such requests should be treated as standard data transfers managed by the SPDM core.

(iii) (*Insecure secret provisioning*) The provisioning and management of cryptographic secrets within SPDM highlight several areas of concern for potential misuse. Initially, the SPDM 1.2.1 standard allowed vendors to handle the provisioning of protocol secrets, with mechanisms to update certificates after the initial setup. SPDM 1.3 extended this by allowing an indefinite number of pre-shared public keys to be set after the devices are already deployed. However, without strict procedures, this flexibility could lead to security vulnerabilities. We emphasize that provisioning and replacing cryptographic secrets should be governed by well-defined, thoroughly analyzable protocol procedures. In addition, the ability to remotely set trusted certificates raises another serious issue. Currently, the standard does not include a default deny-all policy for remotely provisioning certificates, leaving this feature susceptible to misuse without proper access controls. The current SPDM design also lacks proper policies for handling old or expired certificates, which are maintained even after a protocol reset, highlighting the need for policies governing deletion or active validity checks. Similarly, the specification does not place sufficient restrictions on which initiators can provision certificates to responders

Another critical aspect involves certificate management across SPDM connections. Discussions with developers revealed that devices, particularly those with limited memory, may store certificates across different connections in shared storage to avoid duplicates. However, this can create security risks, especially since an Requester can choose which certificate to authenticate with, even if that certificate does not strictly bind to a device identifier (i). An attacker could exploit this by impersonating another device using a different OID while reusing certificates stored from prior sessions. Furthermore, pre-shared public keys and certificates share the same identifier, increasing the likelihood of cross-protocol attacks if certificates are accessible outside specific protocol runs.

Prudent Design Choices Protocols like SPDM are tailored to address specific use cases, balancing

potential security guarantees with practical requirements dictated by the infrastructure or stakeholders. However, such design choices can become problematic when they compromise or weaken the intended security goals of the standard. Two notable design decisions that could potentially have impact on the security of SPDM are: (i) the session ID size during the PSK key exchange and (ii) the use of counters instead of random nonces.

(i) In certain scenarios, the replay protection of the protocol heavily relies on the uniqueness of the session ID. However, the ID share assigned to each party is only two bytes in size, which is cryptographically inadequate to reliably prevent replay attacks. This limitation opens the door to practical attacks that exploit the small range of possible session IDs.

(ii) Instead of using random nonces, SPDM allows counters during specific phases of the protocol. The standard emphasizes that (a) counters must not be reused or reset during the lifetime of the device and (b) devices may undergo resets, typically erasing all volatile state. This presents as a conflict, as maintaining counter uniqueness without non-volatile memory storage may be impractical. To mitigate risks, it would be wise to minimize reliance on the uniqueness of counters.

10.3 Limitations

Achieving a cohesive, automated analysis of the full SPDM setup, device attestation and session establishment as whole showed to be challenging, both for Tamarin and in terms of computing power. Due to this, we needed to limit our scope in two ways: (i) Fix the number of certificates and pre-shared keys, and (ii) limit the application data exchange to a single round.

(i) We performed our current analysis on a fixed amount of certificates and pre-shared keys. A first attempt in lifting this to an arbitrary number showed to be extremely difficult for Tamarin to handle. We leave it to future work to lift the current analysis to an arbitrary number of shared secrets by finding better abstractions and/or improving the provers. Another future work would be to lift the current analysis to allow for an arbitrary number of shared secrets.

(ii) We had to limit the application data exchange to a single round of exchange messages. Attempting to increase the limit lead to an exponential growth of pre-computation time while several lemmas started to diverge. Any attempt to inspect Tamarin's sources or to inspect the traces of the respective lemmas lead to Tamarin crashing.

Furthermore, we opted to model only a minimal version of a Public Key Infrastructure (PKI). Typically, a PKI supports multiple root authorities and has several levels of certificate chains, but our analysis was limited to a single root authority and certificate chains of size one. We believe that this simplification is justified for the current scope of our work. However, as future iterations of SPDM evolve to include features like certificate revocation and the issuance of new certificates – which are part of the current SPDM specification that are underspecified – a more comprehensive model of PKIs will become necessary. This future model will need to address the complexities of managing multiple layers of certificate authorities and enhancing the robustness of the security framework. With the current state of tools and methodologies we believe that this is out-of-scope.

Lastly, we included the VCA phase, but abstracted from the actual negotiation of versions, algorithms, and capabilities. A more detailed modeling of this phase could provide insights into potential downgrade attacks. While we tested an over-approximation model that incorporated compromised cryptographic primitives to assess the robustness of security properties, a thorough analysis of downgrade attacks remains a topic for future research. This is primarily because (i) the VCA negotiation is essential to every part of the protocol model, and any modifications to it would complicate all associated lemmas, and (ii) incorporating broken cryptographic primitives into a Tamarin model significantly increases the complexity of proofs, as seen in the first part of the thesis (see Part I). This underscores the need for a novel approach to address these challenges in future work.

10.4 Future Directions

There are two significant directions for future work based on the limitations of our current analysis and the possible design flaws identified in SPDM. Firstly, there is considerable opportunity to enhance SPDM

and strengthen its formal guarantees. Secondly, there is potential to improve the tools and methodologies we employ in our analysis.

10.4.1 SPDM

The development of the SPDM protocol by the DMTF is ongoing, and regular updates to the protocol are expected. It will be necessary to continually revise and improve the analysis and models to keep pace with the protocol's design evolution.
As mentioned in Section 9.5.2, our current modeling does not encompass all functionalities of SPDM. While certain functions like sending messages in segments/chunks do not fit within the scope of our symbolic model, there are other aspects that still require detailed specification. This includes, for instance, provisioning of new certificates during protocol executions. For a complete list of not modeled request and response codes, refer to Appendix B.

Domain separation In our analysis we identified that adding key domain separation in the protocol modes helps the Tamarin prover in automatically finding proofs. This is especially relevant when having to analyze large protocols at the scale of the composition of SPDM. Note that the key domain separation is not part of the standard, however we recommend it to standardization developers. First, it clearly divides the usage of keys and second, it can aid the automatic formal analysis of their protocols, ultimately contributing to stronger security measures.

Security Properties Considering SPDM's role as a foundational solution for trusted low-level communication, the specifics of the security properties it aims to provide are not well-defined. We believe that the trust in DMTF's SPDM will benefit from an open discussion regarding these security properties.

Additionally, existing specifics on the desired security properties are rather sloppily designed. For instance, the specification suggests that it could be interesting to authenticate the Responder before requesting its measurements as stated in:

Specification [96], Section 10.11, line 407:

> Because issuing GET_MEASUREMENTS clears the M1/M2 message transcript, it is recommended that a Requester does not send this message until it has received at least one successful CHALLENGE_AUTH response message from the Responder. This ensures that the information in message pairs GET_DIGESTS / DIGESTS and GET_CERTIFICATES / CERTIFICATES has been authenticated at least once.

However, our threat models indicate that this step may not be strictly necessary, as the integrity and authentication of the measurements do not seem to depend on prior authentication of the Responder. Notably, even with prior authentication, attestation can proceed without effective integrity protection.

10.4.2 Tooling and Methodologies

In modeling SPDM, we utilized Tamarin's rewrite rules to faithfully represent the transitions of state machines as described in the standard. Had we opted for a process-based specification language, such as those used in ProVerif or SAPIC[+], it would have led us to model certain actions (like handling *PSK_EXCHANGE* and *PSK_FINISH*) as a continuous sequential process. This approach would likely have missed our identified attack, highlighting how subtle differences in modeling can influence the detection of potential vulnerabilities.

Our analysis underscores the importance of revisiting earlier formal verifications of protocols such as EMV [22, 23, 88] or TLS [33, 80] to ensure that possible transitions within their state machines were not overlooked due to modeling oversimplifications. The goal should be for models to accurately reflect the true intent and decision-making processes of the parties involved, rather than simplifying these elements to ease the modeling and verification process.

Additionally, we have used standard symbolic models for cryptographic primitives. SPDM utilizes several cryptographic primitives like digital signatures, AEADs, and hashing throughout its phases. Future efforts could aim to refine these models to more closely align with the specific cryptographic primitives employed by SPDM, potentially employing the techniques from Part I of this thesis. Although we tested

an over-approximation model using compromised cryptographic primitives to check the robustness of security properties, a comprehensive analysis remains a task for future research.

Lastly, overcoming many of the limitations mentioned in Section 10.3 can potentially be tackled by optimizing our current tools. Developing such a complex model proved particularly challenging for several reasons. Any minor modification required restarting Tamarin along with all pre-computations, which led to several minutes of loading time for each change in the model. The interactive mode of Tamarin became almost unmanageable due to the size of the computations. Traces and their corresponding image files expanded drastically in size, loading gigabytes of images into RAM. This substantial load overwhelmed the graphical user interface (GUI) to such an extent that it became impossible to effectively work with it.

III

Conclusion and Future Work

"One never notices what has been done; one can only see what remains to be done."
– Marie Curie, *1894*

11 Conclusion

In this thesis, we set out to advance the current state of security protocol verification. Our goals were twofold: (i) to enhance current verification efforts by enabling more nuanced analysis that more closely mirrors the actual behavior of protocols and primitives, and (ii) to explore the limits of size and complexity that objects can possess while still being verifiable using state-of-the-art techniques – and to determine what can be learned from pushing these boundaries.

In greater detail, we developed several symbolic models that enhance the state-of-the-art representations of cryptographic primitives, namely cryptographic hashes, authenticated encryption schemes with associated data, and key encapsulation mechanisms. For each primitive, we examined both the cryptographic definitions and real-world attacks, which enabled us to formulate multiple detailed representations. Furthermore, we established methodologies for each primitive that automate the analysis of protocols utilizing these primitives within the Tamarin prover. Additionally, we modeled and analyzed the Security Protocol and Data Model (SPDM) using the Tamarin prover, resulting in one of the largest Tamarin models and objects ever analyzed through monolithic symbolic verification.

Working on both goals, our approaches led to the discovery of attacks on deployed protocols. Utilizing our hash function methodology, we were able to automatically re-identify known vulnerabilities in IKEv2, SSHv2, and the Sigma protocol. Additionally, we discovered previously unknown attack variants for each of these. In the case of AEAD, our automatic analysis confirmed known attacks on YubiHSM, SFrame, and Facebook's *Message Franking*, and we also uncovered novel security violations in WebPush, WhatsApp, and Scuttlebutt. Furthermore, our comprehensive analysis of the SPDM revealed a critical flaw in its session establishment. Addressing and reporting this flaw resulted in modifications to both the standard and its reference implementation, adding our work to one of the few that got a CVE assigned to an attack identified by the Tamarin prover.

While our research on hash functions and AEADs significantly advanced the state-of-the-art in symbolic analysis by enabling the automatic detection of protocol flaws due to poor choices of primitives or their incorrect usage, our work on KEMs differed slightly. In our study of KEMs, we identified a novel class of attack which we named *re-encapsulation attack*. Intuitively, at the protocol level, a re-encapsulation attack can often emerge as an unknown-key-share attack, where two parties compute the same key despite having different perceptions of the identity of their respective partners. This type of attack was not addressed by traditional KEM properties, prompting us to develop a new class of binding properties. Unlike the models for hashes and AEADs, the symbolic models and methodologies we developed for KEMs are specifically designed to investigate whether protocols employing KEMs depend on more than just the IND-CCA security of the KEMs, but also on these essential binding properties.

Reflecting on our analysis of SPDM, we approached the symbolic analysis traditionally. Our objective was to explore the current state of protocol verification and its limitations, which led us to rely on established modeling techniques while also attempting to exploit many features of Tamarin. Following this approach, we encountered multiple limitations: we frequently faced performance issues that caused the tool to crash, and features like Tamarin's built-in GUI became impractical due to the immense size of the protocol states. Our work should not only be regarded as a success in terms of security improvements to the SPDM protocol but should also serve as a catalyst for directing more research efforts toward developing new methodologies that facilitate large-scale analysis.

We will now summarize the contributions of this thesis in Section 11.1, before concluding with our perspectives on potential future work in Section 11.2.

11.1 Contributions

In the following, we will provide a brief summary of our contributions:

In Chapter 4: Cryptographic Hash Functions

1. We develop the first systematic methodology to find protocol attacks that exploit weaknesses of real-world hash functions. We achieve this by symbolically modeling cryptographic weaknesses (i.e. the lack of desirable cryptographic properties) as well as real-world attack classes that are not captured by classical security definitions for cryptographic hash functions.
2. We automate our methodology in Tamarin, by (i) proposing a dedicated modeling technique, and (ii) by extending Tamarin with new required features that are of independent interest beyond this work.
3. We apply our methodology to over 20 protocols, automatically rediscovering all previously reported attacks on those protocols that exploit weak hash functions, as well as finding several new variants. The main attacks can be seen in Table 11.1.

In Chapter 5: Authenticated Encryption with Associated Data

1. We develop the first systematic methodology for analyzing security protocols that takes the subtle properties of specific AEAD instantiations into account.
2. We automate our methodology in Tamarin and provide details on how to choose the right AEAD models for a given case study.
3. In case studies, we show our methodology effectively rediscovers known attacks on several protocols, including YubiHSM [153], Facebook's Message Franking [102], and SFrame [131]. We also rediscover a theoretical attack variant on Facebook's Message Franking first mentioned in [122]. Moreover, our analysis uncovers unexpected behavior in WebPush [211], Whatsapp Group Messaging [130], and Scuttlebutt [201]. The main results can be seen in Table 11.1.

Protocol	Attacked properties	New?	Status / Notes	Primitive
	Secrecy,Transcript Agreement	[37]		Hash
Sigma [146]	Secrecy,Transcript Agreement	∼[37]	Variant	Hash
	Secrecy,Transcript/Role Agreement	new	Role-confusion	Hash
	Negotiation Data Agreement	new	See Figure 4.6	Hash
SSHv2 [162]	Negotiation Data Agreement	[37]		Hash
	Negotiation Data Agreement	new	Variant	Hash
	Colliding Inputs (CI)	new	CI should be on the cookie	Hash
IKEv2 [139]	Initiator Authentication	[37]		Hash
	Transcript Agreement	new	Disagreement on cookies	Hash
Flickr [106]	Initiator Authentication	[106]		Hash
YubiHSM [221]	Key secrecy	[153]	Fixed	AEAD
SFrame [182]	Authentication	[131]	Fixed	AEAD
FB Message Franking [110]	Content Agreement	[102]	Fixed	AEAD
FB Message Franking [110]	Framing	[102, 122]	Fixed	AEAD
GPG SED [144]	Content Agreement	new	Deprecated	AEAD
GPG SEIPDv2 [144]	Content Agreement	new	Infeasible	AEAD
Saltpack [194]	Content Agreement	new	Infeasible	AEAD
WebPush [211]	Server Accountability	new	Reported	AEAD
WhatsApp [130]	Content Agreement	new	Reported ‡	AEAD
Scuttlebutt [201]	Content Agreement	new	Reported *	AEAD

* = Feasibility depends on the collision resistance of XSalsa20-Poly1305. See discussion in Section 5.5.6.
‡ = Reported to WhatsApp. Feasibility heavily relies on implementation details, which are not open source.
∼= New variant of an existing attack.

Table 11.1: Summary of the main analysis results from our case-studies.

In Chapter 6: Key Encapsulation Mechanisms

1. We introduce a novel hierarchy of computational binding properties for KEMs and position existing notions within it; the remaining properties are new. KEMs that satisfy our key-binding properties will leave fewer pitfalls for protocol designers.

2. We develop a symbolic analysis methodology to automatically analyze the security of KEM-based protocols, using fine-grained models of their KEMs, and implement them in the Tamarin prover. Our methodology can also be used to automatically establish the KEM binding properties that are needed for a protocol to be secure. In case studies, our automated analysis finds new attacks and missed proof obligations.

In Chapter 9: Security Protocol and Data Model

1. We construct the first formal model of the SPDM 1.2.1 standard as a whole, which considers the interaction between its main modes and sub-protocols, by a fine-grained modeling of the SPDM 1.2.1 specification and its complex state machine interactions. As a first step towards this goal, we developed models for the several components. Our models include device initialization, the five phases of the protocol, its three modes of key exchange and session setup, and the optional requests performed outside a secure session. Using the knowledge and experience from constructing the component models, we construct a monolithic model of SPDM 1.2.1, one of the largest and most complex Tamarin models to date.
2. Using our formal model, Tamarin finds a critical cross-protocol attack on SPDM 1.2.1. Our attack completely breaks the mutual authentication guarantees of the pre-shared key mode. We implemented the attack on the official SPDM reference implementation by the DMTF consortium, and reported our findings and suggested fixes to its developers. DMTF reported our attack as a CVE [84] with CVSS score 9.0 (critical). We formally prove that the fixed version of SPDM 1.2.1 provides authentication for all modes and other basic security guarantees, *even in the presence of cross-protocol attacks and all mode interactions*. Both the SPDM standard and its reference implementation were updated based on our work.

In Chapter 10: Limitations and Discussion

1. Our formal modeling and analysis leads to several suggestions for next versions of the standard, as well as potential design pitfalls. We list and discuss these design flaws, highlighting potential future research efforts.
2. The formal analysis effort is very challenging, notably because none of the composition results in the literature can be applied to SPDM, because its sub-protocols share complex state, including long-term keys, short-term keys, keying material, and transcripts. We discuss the current limitations of our approach and include suggestions for further research to guide future research in tackling large-scale security protocol verification.

11.2 Future Work

Whereas Chapters 7 and 10 only hinted at potential future work, this section concretely outlines potential research directions and talks about further primitives, modularity, and the human factor in security.

More Primitives! Throughout this thesis, we studied cryptographic hashes, AEADs, and KEMs, and developed new symbolic representations for them. Together with the works by [134] and [82], who explored digital signatures and Diffie-Hellman groups, we have significantly advanced towards creating a more detailed and robust symbolic model. Yet, this progress represents just the beginning of what is still to be explored in this field.

Other Primitives Many deployed protocols continue to use primitives for which we either assume "perfect" symbolic representations, like public key encryption and commitments, or lack adequate default representations entirely, such as mix networks or homomorphic encryption. Developing nuanced symbolic models for these primitives would be the straightforward next steps towards enhancing the security analysis of complex systems.

Multiple Primitives A natural next step in this line of research would be to explore how analysis using multiple advanced primitive models could become more computational feasible. Real-world security protocols, in general, use more than a single cryptographic primitive. It would be interesting to see, whether we are able to analyze protocols, like for instance SPDM, which uses both hashes and

AEADs (and in the PQ variant even KEMs) with our fine-grained models. We conjecture that this would require both advancements in the tools to handle the specific case explosion introduced by the primitive model, and a novel modeling strategy to easily apply our primitive models to existing case studies.

Why stop at Primitives? Going back to the idea why we even wanted to find better symbolic models of primitives is the larger goal to push automated, formal verification closer to real-world behavior. While our work on primitives brought us closer to this goal, we are far from done. Future directions might start with a comprehensive exploration of vulnerabilities or attack vectors that current symbolic analysis are not able to capture. This could include attacks based on low entropy as symbolic frameworks abstract from probabilities or attacks based of fragmentation like seen in SSH [7]. It could also include following Bursuc and Kremer's [53] work on protocols for zero-knowledge contingent payment. While they showed the first automated verification of symbolic protocols using multi-party computations and zero-knowledge proofs, it would be interesting to generalize their approach and potentially find better symbolic representation for these building blocks.

Modularity One major issue with refining the symbolic model to be more detailed is that it causes the size of the objects we can analyze to decrease. This happens because tools like Tamarin require additional computational power and resources to manage the increased detail, on top of their existing workload. This is problematic because, as discussed in the second part of the thesis, Tamarin already approaches its limits with protocols like SPDM, even without a detailed representation of primitives. Consequently, this implies two things: if we wish to analyze protocols like SPDM more thoroughly, or if we need to handle more complex protocols than SPDM, we must develop a better approach.

While small-scale improvements can be achieved through code optimization or additional computational resources, a long-term solution likely requires the development of a modular analysis framework. This can be pursued by further advancing research in existing composition frameworks, e.g., [52, 54, 63, 120, 121, 125, 126] or by dedicating time and resources to exploring alternative approaches.

Progress in the area of composition has been limited since the early 2010s, with most results being highly specific and not broadly applicable. Notably, the most successful results emerge when composition requirements involve disjoint primitives – where protocols do not share any cryptographic primitives. Further research in this direction could highlight the importance of mechanisms such as tagging and domain separation. Investigating whether these mechanisms can be improved in practice may help bridge the gap between theoretical composition results and their applicability for real-world protocols.

Moreover, as most constraints identified in symbolic composition research are syntactical, it could be worthwhile to explore the implementation of these checks within popular tools. ProVerif could be a good tool to start, as composition results are often defined over applied pi-calculus variants. Making these checks automatic could be an important step to getting usable composition results in the symbolic model for actual deployed protocols.

The Human Factor Let us shift our focus to a different aspect: the human factor in formally verifying security. We are particularly concerned with usability and accessibility, which have posed significant challenges throughout the process of working on this thesis. While the academic research itself is demanding, the tools and methodologies designed to support this work often end up being barriers, even though they were initially developed to facilitate research.

Academic Tools One significant challenge of formal verification tools is their lack of accessibility. Typically, these tools are developed by researchers deeply engaged in solving complex theoretical problems that resonate within their respective academic communities. This intense focus often leaves little room for enhancing usability, developing comprehensive documentation, or gathering broad feedback. Consequently, these tools tend to have a steep learning curve due to their sophisticated nature and minimal user-oriented design, which restricts their user base to a niche group.

For more established tools like Tamarin or ProVerif, which have achieved considerable scientific recognition and thus attract larger user communities, usability still poses a substantial challenge. Developers have put efforts into improving the user experience by offering detailed documentation

and tutorials and continuously updating features. Researchers also started to put more emphasize on making results, models and descriptions more accessible through artifacts published together with their papers. However, these enhancements are generally driven by a small group of developers working within a somewhat closed community, limiting the diversity of feedback and the pace of usability improvements. If we aim to make formal security verification more approachable for new researchers and encourage its adoption by industry professionals, it is essential that these tools become more accessible and user-friendly. A constructive approach would involve conducting targeted user studies with both novice and seasoned users to identify and address the barriers that hinder broader usage.

Generalizing Security Properties Accessibility in the field of formal security verification is not only limited by the tools but also by the methodologies we use to approach analysis of protocols. When dealing with real-world protocols, a significant challenge is that the desired security properties and the corresponding threat models are often not well-defined. Common standards and specifications, such as RFCs, drafts from the IETF, or drafts from NIST, typically provide only a basic outline of the required security measures. This lack of specificity often leaves researchers who are modeling these protocols with the responsibility to choose appropriate threat models and security properties, both in the symbolic and computational model. This decision-making process depends heavily on the researcher's expertise and their communication with the protocol developers.

The complexity of this task is compounded by the fact that over the years, researchers have identified hundreds of security properties, defined over hundreds of variations of threat models. Most of these definitions originate from cryptographers working within the computational model of cryptography. This leads to another challenge: translating these properties and models from the computational context into the symbolic model, which is necessary for applying automated verification tools effectively.

Looking ahead, there is a promising path for future research in this area. By gathering and organizing security property definitions and threat models, we can create clear hierarchies that link these different elements. We conjecture that this first step would on its own be very beneficial to the whole field, but the main goal is to use this organized information to improve how we analyze protocols.

One possible next step could involve building a framework based on these hierarchies. This framework would help researchers choose the right security properties and threat models and show them how to translate these into the symbolic model to be used in verification tools. We believe that moving towards a more organized and clear approach would greatly enhance our ability to handle and adapt to new security challenges, especially with the rise of more complex threat models, protocols, and political requirements in the years ahead of us.

List of Figures

List of Tables

Bibliography

[1] Michel Abdalla, Mihir Bellare, and Gregory Neven. *Robust Encryption*. Cryptology ePrint Archive, Report 2008/440. https://ia.cr/2008/440. 2008 (cit. on pp. 45, 49, 65, 70, 87).

[2] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. "How to Abuse and Fix Authenticated Encryption Without Key Commitment." In: *31st USENIX Security Symposium*. 2020 (cit. on pp. 45, 49–51, 53, 60).

[3] Martin Albrecht, Carlos Cid, Kenneth G Paterson, Cen Jung Tjhai, and Martin Tomlinson. "NTS-KEM." In: *NIST PQC Second Round* 2 (2019). https://nts-kem.io/ (Accessed December 2023) (cit. on p. 65).

[4] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. "Practically-exploitable Cryptographic Vulnerabilities in Matrix." In: *IACR Cryptol. ePrint Arch.* (2023), p. 485 (cit. on p. 95).

[5] Martin R Albrecht, Jean Paul Degabriele, Torben Brandt Hansen, and Kenneth G Paterson. "A surfeit of SSH cipher suites." In: *ACM Conference on Computer and Communications Security (CCS)*. 2016, pp. 1480–1491 (cit. on p. 45).

[6] Martin R Albrecht, Lenka Mareková, Kenneth G Paterson, and Igors Stepanovs. "Four Attacks and a Proof for Telegram. Long version, https://mtpsym.github.io/paper.pdf." In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (cit. on pp. 37, 38).

[7] Martin R Albrecht, Kenneth G Paterson, and Gaven J Watson. "Plaintext recovery attacks against SSH." In: *IEEE Symposium on Security and Privacy (S&P)*. 2009, pp. 16–26 (cit. on pp. 48, 132).

[8] Renan CA Alves, Bruno C Albertini, and Marcos A Simplicio. "Securing hard drives with the Security Protocol and Data Model (SPDM)." In: *Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2022 (cit. on p. 96).

[9] Renan CA Alves, Bruno C Albertini, and Marcos A Simplicio Jr. "Benchmarking the Security Protocol and Data Model (SPDM) for component authentication." In: *arXiv preprint arXiv:2307.06456* (2023) (cit. on p. 96).

[10] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, et al. "BIKE: bit flipping key encapsulation." In: *https://inria.hal.science/hal-04278509* (2017) (cit. on p. 65).

[11] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. "The privacy of the TLS 1.3 protocol." In: *Proc. Priv. Enhancing Technol.* (2019) (cit. on p. 96).

[12] Atsushi Fujioka and Koutarou Suzuki and Keita Xagawa and Kazuki Yoneyama. *Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices*. Cryptology ePrint Archive, Paper 2012/211. https://eprint.iacr.org/2012/211. 2012 (cit. on p. 81).

[13] Valerie Aurora. *Lifetimes of cryptographic hash functions*. https://valerieaurora.org/hash.html (Retrieved Jan 2022). 2017 (cit. on pp. 23, 25).

[14] Michael Backes, Birgit Pfitzmann, and Michael Waidner. "Limits of the BRSIM/UC Soundness of Dolev-Yao Models with Hashes." In: *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2006. DOI: 10.1007/11863908_25. URL: https://doi.org/10.1007/11863908_25 (cit. on p. 86).

[15] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. "An interactive prover for protocol verification in the computational model." In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2021, pp. 537–554 (cit. on pp. 11, 86).

[16] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, and Bas Westerbaan. *X-Wing: The Hybrid KEM You've Been Looking For*. Cryptology ePrint Archive, Paper 2024/039. https://eprint.iacr.org/2024/039. 2024 (cit. on pp. 68, 70).

[17] Elaine Barker, Lidong Chen, Andrew Regenscheid, and Miles Smid. "Recommendation for Pair-Wise Key Establishment Using Integer Factorization Cryptography." en. In: *Special Publication (NIST SP), National Institute of Standards and Technology*. 2009. DOI: https://doi.org/10.6028/NIST.SP.800-56B (cit. on p. 35).

[18] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-aided security proofs for the working cryptographer." In: *Annual Cryptology Conference*. Springer. 2011, pp. 71–90 (cit. on pp. 11, 86).

[19] Guy Barwell, Daniel Page, and Martijn Stam. "Rogue Decryption Failures: Reconciling AE Robustness Notions." In: *Proceedings of the 15th IMA International Conference on Cryptography and Coding*. 2015, 94–111. DOI: 10.1007/978-3-319-27239-9_6 (cit. on pp. 45, 49).

[20] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. *Documentation of the Tamarin Prover*. https://tamarin-prover.github.io/#documentation. 2022 (cit. on p. 13).

[21] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. "A formal analysis of 5G authentication." In: *ACM CCS*. 2018 (cit. on p. 24).

[22] David Basin, Ralf Sasse, and Jorge Toro-Pozo. "Card brand mixup attack: bypassing the {PIN} in {non-Visa} cards by using them for visa transactions." In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 179–194 (cit. on p. 124).

[23] David Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV standard: Break, fix, verify." In: *IEEE Symposium on Security and Privacy (SP)*. 2021 (cit. on pp. 1, 24, 124).

[24] Mihir Bellare, Hannah Davis, and Felix Günther. "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability." In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by Anne Canteaut and Yuval Ishai. Cham: Springer International Publishing, 2020, pp. 3–32. ISBN: 978-3-030-45724-2 (cit. on p. 65).

[25] Mihir Bellare and Viet Tung Hoang. *Efficient Schemes for Committing Authenticated Encryption*. Cryptology ePrint Archive, Report 2022/268. https://ia.cr/2022/268. 2022 (cit. on pp. 45, 46, 49–51, 90).

[26] Mihir Bellare, Ruth Ng, and Björn Tackmann. *Nonces are Noticed: AEAD Revisited*. Cryptology ePrint Archive, Report 2019/624. https://ia.cr/2019/624. 2019 (cit. on pp. 45, 49).

[27] Mihir Bellare and Phillip Rogaway. "Optimal asymmetric encryption." In: *Advances in Cryptology—EUROCRYPT'94: Workshop on the Theory and Application of Cryptographic Techniques Perugia, Italy, May 9–12, 1994 Proceedings 13*. Springer. 1995, pp. 92–111 (cit. on p. 11).

[28] Mihir Bellare and Phillip Rogaway. "Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols." In: *ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 1993. ISBN: 0897916298. DOI: 10.1145/168588.168596. URL: https://doi.org/10.1145/168588.168596 (cit. on p. 25).

[29] Mihir Bellare, Phillip Rogaway, and David Wagner. "The EAX mode of operation." In: *International Workshop on Fast Software Encryption*. 2004, pp. 389–407 (cit. on p. 50).

[30] Mihir Bellare and Björn Tackmann. "The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3." In: *CRYPTO*. 2016 (cit. on p. 96).

[31] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. "Classic McEliece: conservative code-based cryptography." In: *NIST submissions* (2017) (cit. on pp. 65, 88).

[32] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS." In: *IEEE Symposium on Security and Privacy (S&P)*. 2015. DOI: 10.1109/SP.2015.39 (cit. on p. 96).

[33] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017. DOI: 10.1109/SP.2017.26 (cit. on pp. 24, 86, 124).

[34] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella Béguelin. "Downgrade Resilience in Key-Exchange Protocols." In: *IEEE Symposium on Security and Privacy (SP)*. 2016 (cit. on p. 96).

[35] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. "miTLS: Verifying Protocol Implementations against Real-World Attacks." In: *IEEE Secur. Priv.* 14 (2016) (cit. on p. 96).

[36] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. *An Analysis of Signal's PQXDH*. https://cryspen.com/post/pqxdh/ (Accessed Jan 2024). 2023 (cit. on pp. 86, 87).

[37] Karthikeyan Bhargavan and Gaëtan Leurent. "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016 (cit. on pp. 19, 23, 25, 26, 35–39, 130).

[38] Ritam Bhaumik and Mridul Nandi. "Improved security for OCB3." In: *International Conference on the Theory and Application of Cryptology and Information Security*. 2017, pp. 638–666 (cit. on p. 50).

[39] Simon Blake-Wilson and Alfred Menezes. "Authenticated Diffie-Hellman Key Agreement Protocols." In: *Selected Areas in Cryptography (SAC)*. Springer, 1998. DOI: 10.1007/3-540-48892-8_26. URL: https://doi.org/10.1007/3-540-48892-8_26 (cit. on p. 35).

[40] Simon Blake-Wilson and Alfred Menezes. "Unknown key-share attacks on the station-to-station (STS) protocol." In: *Public Key Cryptography: Second International Workshop on Practice and Theory in Public Key Cryptography, PKC'99 Kamakura, Japan, March 1–3, 1999 Proceedings 2*. Springer. 1999, pp. 154–170 (cit. on pp. 35, 65, 82).

[41] Bruno Blanchet. "Composition Theorems for CryptoVerif and Application to TLS 1.3." In: *CSF*. 2018 (cit. on p. 96).

[42] Bruno Blanchet. "CryptoVerif: Computationally sound mechanized prover for cryptographic protocols." In: *Dagstuhl seminar "Formal Protocol Verification Applied*. Vol. 117. 2007, p. 156 (cit. on pp. 11, 86).

[43] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif." In: *Foundations and Trends in Privacy and Security* 1.1–2 (2016) (cit. on pp. 1, 30).

[44] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. "Proverif with lemmas, induction, fast subsumption, and much more." In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (cit. on p. 86).

[45] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. *ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial*. 2018 (cit. on p. 12).

[46] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G Paterson, and Martijn Stam. "Security of symmetric encryption in the presence of ciphertext fragmentation." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 2012, pp. 682–699 (cit. on p. 45).

[47] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. "Frodo: Take off the ring! practical, quantum-secure key exchange from LWE." In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016 (cit. on pp. 65, 88).

[48] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM." In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018. DOI: 10.1109/EuroSP.2018.00032 (cit. on pp. 65, 66, 76, 81–83, 88).

[49] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. "The provable security of ed25519: theory and practice." In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1659–1676 (cit. on p. 65).

[50] Jacqueline Brendel, Marc Fischlin, and Felix Günther. "Breakdown Resilience of Key Exchange Protocols: NewHope, TLS 1.3, and Hybrids." In: *ESORICS*. 2019 (cit. on p. 96).

[51] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. "Key-schedule Security for the TLS 1.3 Standard." In: *IACR Cryptol. ePrint Arch.* (2021) (cit. on p. 96).

[52] Chris Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C Williams. "Composability of Bellare-Rogaway key exchange protocols." In: *ACM Conference on Computer and Communications Security (CCS)*. 2011 (cit. on pp. 95, 97, 132).

[53] Sergiu Bursuc and Steve Kremer. "Contingent payments on a public ledger: models and reductions for automated verification." In: *Computer Security–ESORICS: 24th European Symposium on Research in Computer Security*. Springer. 2019, pp. 361–382 (cit. on p. 132).

[54] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols." In: *Symposium on Foundations of Computer Science*. IEEE. 2001 (cit. on pp. 95, 97, 132).

[55] Ran Canetti and Hugo Krawczyk. *Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels*. Cryptology ePrint Archive, Paper 2001/040. https://eprint.iacr.org/2001/040. 2001 (cit. on p. 81).

[56] Ran Canetti and Hugo Krawczyk. *Security Analysis of IKE's Signature-based Key-Exchange Protocol*. Cryptology ePrint Archive, Paper 2002/120. https://eprint.iacr.org/2002/120. 2002 (cit. on p. 78).

[57] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. "Password interception in a SSL/TLS channel." In: *Annual International Cryptology Conference*. 2003, pp. 583–599 (cit. on p. 48).

[58] Srdjan Capkun, Levente Buttyán, and Jean-Pierre Hubaux. "SECTOR: secure tracking of node encounters in multi-hop wireless networks." In: *Workshop on Security of ad hoc and Sensor Networks (SASN)*. ACM, 2003. DOI: 10.1145/986858.986862. URL: https://doi.org/10.1145/986858.986862 (cit. on p. 35).

[59] John Chan and Phillip Rogaway. *Anonymous AE*. Cryptology ePrint Archive, Report 2019/1033. https://ia.cr/2019/1033. 2019 (cit. on pp. 45, 49).

[60] Sanjit Chatterjee, Alfred Menezes, and Berkant Ustaoglu. "A Generic Variant of NIST's KAS2 Key Agreement Protocol." In: *Australasian Conference - Information Security and Privacy (ACISP)*. Springer, 2011. DOI: 10.1007/978-3-642-22497-3_23. URL: https://doi.org/10.1007/978-3-642-22497-3_23 (cit. on p. 35).

[61] Vincent Cheval, Cas Cremers, Alexander Dax, Lucca Hirschi, Charlie Jacomme, and Steve Kremer. "Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses." In: *USENIX Security Symposium*. 2023 (cit. on pp. 3, 7, 17, 30, 86).

[62] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. "DEEPSEC: Deciding Equivalence Properties in Security Protocols - Theory and Practice." In: *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*. IEEE Computer Society Press, 2018 (cit. on p. 1).

[63] Stefan Ciobâca and Véronique Cortier. "Protocol composition for arbitrary primitives." In: *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE. 2010, pp. 322–336 (cit. on pp. 97, 132).

[64] Colin Boyd and Yvonne Cliff and Juan M. Gonzalez Nieto and Kenneth G. Paterson. *Efficient One-round Key Exchange in the Standard Model*. Cryptology ePrint Archive, Paper 2008/007. https://eprint.iacr.org/2008/007. 2008 (cit. on p. 81).

[65] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Polk. *Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile*. Tech. rep. 2008 (cit. on pp. 102, 106).

[66] Véronique Cortier, David Galindo, and Mathieu Turuani. "A Formal Analysis of the Neuchatel e-Voting Protocol." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018. DOI: 10.1109/EuroSP.2018.00037. URL: https://doi.org/10.1109/EuroSP.2018.00037 (cit. on p. 24).

[67] Véronique Cortier, Steve Kremer, Ralf Küsters, and Bogdan Warinschi. "Computationally Sound Symbolic Secrecy in the Presence of Hash Functions." In: *Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Springer, 2006. DOI: 10.1007/11944836_18 (cit. on p. 86).

[68] Ronald Cramer and Victor Shoup. *Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack*. Cryptology ePrint Archive, Paper 2001/108. https://eprint.iacr.org/2001/108. 2001. URL: https://eprint.iacr.org/2001/108 (cit. on pp. 65, 66).

[69] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. "Automated Analysis of Protocols that use Authenticated Encryption: Analysing the Impact of the Subtle Differences between AEADs on Protocol Security." In: *USENIX Security Symposium*. 2023 (cit. on pp. 3, 7, 17, 65).

[70] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. "Automated Analysis of Protocols that use Authenticated Encryption: Analysing the Impact of the Subtle Differences between AEADs on Protocol Security (Full version)." In: https://inria.hal.science/hal-04126116. Aug. 2023. URL: https://inria.hal.science/hal-04126116 (cit. on pp. 47, 49, 51, 53, 61).

[71] Cas Cremers, Alexander Dax, Charlie Jacomme, and Mang Zhao. *Tamarin models and analysis scripts to reproduce the results in this paper*. https://github.com/AutomatedAnalysisOf/AEADProtocols. 2023 (cit. on pp. 55, 85).

[72] Cas Cremers, Alexander Dax, and Niklas Medinger. "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols." In: *Conference on Computer and Communications Security (CCS)*. 2024 (cit. on pp. 3, 7, 17).

[73] Cas Cremers, Alexander Dax, and Niklas Medinger. "Keeping Up with the KEMs: Stronger Security Notions for KEMs (Full version)." In: *IACR Cryptol. ePrint Arch.* (2023), p. 1933. URL: https://eprint.iacr.org/2023/1933 (cit. on pp. 70, 72, 73, 80, 81, 88, 89).

[74] Cas Cremers, Alexander Dax, and Niklas Medinger. *KEM library and Case Studies*. https://github.com/FormalKEM/Symbolic_KEM_Models. 2024 (cit. on p. 85).

[75] Cas Cremers, Alexander Dax, and Aurora Naska. *Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 including Cross-Protocol Attacks*. Under Submission. 2025 (cit. on pp. 3, 4, 7, 93, 119).

[76] Cas Cremers, Alexander Dax, and Aurora Naska. "Formal analysis of SPDM: Security protocol and data model version 1.2." In: *USENIX Security Symposium*. 2023 (cit. on pp. 3, 4, 7, 81, 93, 121).

[77] Cas Cremers, Alexander Dax, and Aurora Naska. *SPDM Composition Models*. https://github.com/ComprehensiveSPDM/TamarinSPDMAnalysis. 2024 (cit. on p. 119).

[78] Cas Cremers, Alexander Dax, and Aurora Naska. *Tamarin models and analysis scripts to reproduce the results in this paper*. https://github.com/FormalAnalysisOf/SPDM. 2022 (cit. on p. 119).

[79] Cas Cremers, Samed Düzlü, Rune Fiedler, Marc Fischlin, and Christian Janson. "BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures." In: *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021 (cit. on p. 65).

[80] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A comprehensive symbolic analysis of TLS 1.3." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017, pp. 1773–1788 (cit. on pp. 1, 12, 24, 95, 96, 124).

[81] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication." In: *IEEE Symposium on Security and Privacy*. 2016 (cit. on pp. 95, 96).

[82] Cas Cremers and Dennis Jackson. "Prime, order please! Revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman." In: *IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. 2019, pp. 78–7815 (cit. on pp. 19, 86, 131).

[83] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. "A Formal Analysis of IEEE 802.11's WPA2: Countering the Kracks Caused by Cracking the Counters." In: *29th USENIX Security Symposium*. 2020, pp. 1–17 (cit. on pp. 1, 12, 86, 95, 96).

[84] CVE. *CVE of mode switch attack on the SPDM reference implementaton*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-31127. accessed: 2023-06-20. 2023 (cit. on pp. 2, 96, 119, 131).

[85] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM." In: *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*. Springer. 2018, pp. 282–305 (cit. on pp. 65, 88).

[86] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. "On the Concrete Security of TLS 1.3 PSK Mode." In: *EUROCRYPT*. 2022 (cit. on p. 96).

[87] Alexander Dax, Robert Künnemann, Sven Tangermann, and Michael Backes. "How to wrap it up-a formally verified proposal for the use of authenticated wrapping in PKCS# 11." In: *IEEE Computer Security Foundations Symposium (CSF)*. 2019, pp. 62–6215 (cit. on p. 12).

[88] Joeri De Ruiter and Erik Poll. "Formal analysis of the EMV protocol suite." In: *Theory of Security and Applications: Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31-April 1, 2011, Revised Selected Papers*. Springer. 2012, pp. 113–129 (cit. on p. 124).

[89] Joeri De Ruiter and Erik Poll. "Protocol state fuzzing of TLS implementations." In: *USENIX Security Symposium (USENIX Security)*. 2015 (cit. on p. 96).

[90] Jean Paul Degabriele and Kenneth G Paterson. "On the (in) security of IPsec in MAC-then-encrypt configurations." In: *ACM Conference on Computer and Communications Security (CCS)*. 2010, pp. 493–504 (cit. on p. 48).

[91] Denis Diemert and Tibor Jager. "On the Tight Security of TLS 1.3: Theoretically Sound Cryptographic Parameters for Real-World Deployments." In: *J. Cryptol.* 34 (2021) (cit. on p. 96).

[92] Whitfield Diffie and Martin E Hellman. "New directions in cryptography." In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 2022, pp. 365–390 (cit. on p. 11).

[93] DMTF. *DMTF website*. https://www.dmtf.org/. accessed: 2022-10-09 (cit. on pp. 2, 95).

[94] DMTF. *DSP0274: Security Protocol and Data Model (SPDM) Specification, Version 1.3.0*. https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.3.0.pdf. accessed: 2024-01-24. 2023 (cit. on pp. 2, 81, 119).

[95] DMTF. *DSP2058: Security Protocol and Data Model (SPDM) Architecture White Paper, Version 1.2.0*. https://www.dmtf.org/sites/default/files/standards/documents/DSP2058_1.2.0.pdf. accessed: 2022-10-09. 2022 (cit. on p. 101).

[96] DMTF. *DSP2058: Security Protocol and Data Model (SPDM) Specification, Version 1.2.1*. https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.2.1.pdf. accessed: 2022-10-09. 2022 (cit. on pp. 2, 95, 102, 103, 118, 124, 161).

[97] DMTF. *Enabling Platform Integrity in a Common Way by Utilizing DMTF's SPDM Standard*. https://www.dmtf.org/sites/default/files/DMTF_SPDM_Tech_Note.pdf. accessed: 2024-01-24. 2023 (cit. on p. 95).

[98] DMTF. *Open source code of libspdm in Rust*. https://github.com/jyao1/rust-spdm. accessed: 2024-01-24. 2023 (cit. on p. 96).

[99] DMTF. *The SPDM Respository*. https://github.com/DMTF/libspdm. accessed: 2022-10-07. 2022 (cit. on p. 118).

[100] DMTF. *Vulnerable open source code of libspdm v2.3.1*. https://github.com/DMTF/libspdm/releases/tag/2.3.1. accessed: 2024-01-24. 2023 (cit. on pp. 96, 118).

[101] *Docker image and submission files*. https://www.dropbox.com/sh/6ksmcqy6od8vlfy/AAD4zoBV286o35dy6kIapYwJa?dl=0 (cit. on p. 85).

[102] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. *Fast Message Franking: From Invisible Salamanders to Encryptment*. Cryptology ePrint Archive, Report 2019/016. https://ia.cr/2019/016. 2019 (cit. on pp. 45, 48–53, 56, 59, 65, 86, 130).

[103] Danny Dolev and Andrew Yao. "On the security of public key protocols." In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208 (cit. on pp. 1, 11).

[104] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol." In: *J. Cryptol.* 34 (2021) (cit. on p. 96).

[105] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates." In: *ACM CCS*. 2015 (cit. on p. 96).

[106] Thai Duong. *Flickr's API signature forgery vulnerability*. https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html (Retrieved Jan 2022). 2009 (cit. on pp. 25, 35, 36, 38, 130).

[107] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. "Associative Unification and Symbolic Reasoning Modulo Associativity in Maude." In: *International Workshop on Rewriting Logic and Its Applications (WRLA)*. Springer, 2018. DOI: 10.1007/978-3-319-99840-4_6 (cit. on p. 32).

[108] Santiago Escobar, Catherine Meadows, and José Meseguer. "Maude-NPA: Cryptographic protocol analysis modulo equational properties." In: *Foundations of Security Analysis and Design*. Springer, 2009, pp. 1–50 (cit. on p. 12).

[109] *EU Federation Gateway Service (EFGS)*. https://github.com/eu-federation-gateway-service/efgs-federation-gateway (Retrieved Jan 2022). 2020 (cit. on p. 37).

[110] *Facebook - Messenger Secret Conversations Technical Whitepaper*. https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf. accessed: 2022-08-08. 2017 (cit. on pp. 55, 56, 59, 130).

[111] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. *Robust Encryption, Revisited*. Cryptology ePrint Archive, Paper 2012/673. https://eprint.iacr.org/2012/673. 2012 (cit. on pp. 69, 70, 87).

[112] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. *Security of Symmetric Primitives under Incorrect Usage of Keys*. Cryptology ePrint Archive, Report 2017/288. https://ia.cr/2017/288. 2017 (cit. on pp. 50, 51).

[113] Marc Fischlin and Felix Günther. "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates." In: *IEEE EuroS&P*. 2017 (cit. on p. 96).

[114] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3." In: *IEEE Symposium on Security and Privacy (SP)*. 2016 (cit. on p. 96).

[115] Pierre-Alain Fouque, Gwenaëlle Martinet, Frédéric Valette, and Sébastien Zimmer. "On the Security of the CCM Encryption Mode and of a Slight Variant." In: *International Conference on Applied Cryptography and Network Security*. 2008, pp. 411–428 (cit. on p. 50).

[116] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure integration of asymmetric and symmetric encryption schemes." In: *Annual international cryptology conference*. 1999 (cit. on pp. 66, 88).

[117] Ilias Giechaskiel, Cas Cremers, and Kasper B. Rasmussen. "When the Crypto in Cryptocurrencies Breaks: Bitcoin Security under Broken Primitives." In: *IEEE Security Privacy* 16.4 (2018), pp. 46–56. DOI: 10.1109/MSP.2018.3111253 (cit. on p. 86).

[118] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption." In: *J. Comput. Syst. Sci.* 28.2 (1984), pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9. URL: https://doi.org/10.1016/0022-0000(84)90070-9 (cit. on pp. 1, 10, 11).

[119] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. "A digital signature scheme secure against adaptive chosen-message attacks." In: *SIAM Journal on computing* 17.2 (1988), pp. 281–308 (cit. on p. 11).

[120] Sébastien Gondron and Sebastian Mödersheim. "Vertical Composition and Sound Payload Abstraction for Stateful Protocols." In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021. DOI: 10.1109/CSF51468.2021.00038 (cit. on pp. 95, 97, 132).

[121] Thomas Groß and Sebastian Modersheim. "Vertical protocol composition." In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2011 (cit. on pp. 95, 97, 132).

[122] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. *Message Franking via Committing Authenticated Encryption*. Cryptology ePrint Archive, Report 2017/664. https://ia.cr/2017/664. 2017 (cit. on pp. 45, 49–51, 53, 56, 59, 65, 130).

[123] Paul Grubbs, Varun Maram, and Kenneth G. Paterson. *Anonymous, Robust Post-Quantum Public Key Encryption*. Cryptology ePrint Archive, Paper 2021/708. https://eprint.iacr.org/2021/708. 2021. URL: https://eprint.iacr.org/2021/708 (cit. on pp. 65, 68, 70, 71, 73, 88).

[124] Shay Gueron and Yehuda Lindell. "GCM-SIV: full nonce misuse-resistant authenticated encryption at under one cycle per byte." In: *ACM Conference on Computer and Communications Security (CCS)*. 2015, pp. 109–119 (cit. on pp. 45, 49, 50).

[125] Joshua D Guttman. "Cryptographic protocol composition via the authentication tests." In: *International Conference on Foundations of Software Science and Computational Structures*. Springer. 2009 (cit. on pp. 95, 97, 132).

[126] Andreas V. Hess, Sebastian Alexander Mödersheim, and Achim D. Brucker. "Stateful Protocol Composition." In: *ESORICS (1)*. Vol. 11098. Lecture Notes in Computer Science. Springer, 2018, pp. 427–446 (cit. on pp. 95, 97, 132).

[127] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation." In: *Theory of Cryptography*. Ed. by Yael Kalai and Leonid Reyzin. Cham: Springer International Publishing, 2017, pp. 341–371 (cit. on pp. 70, 78, 81).

[128] Andreas Hülsing, Joost Rijneveld, John Schanck, and Peter Schwabe. "High-speed key encapsulation from NTRU." In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 232–252 (cit. on p. 65).

[129] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. *Post-quantum WireGuard*. Cryptology ePrint Archive, Paper 2020/379. https://eprint.iacr.org/2020/379. 2020 (cit. on pp. 86, 87).

[130] WhatsApp Inc. *WhatsApp encryption overwiew—Technical white paper*. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Accessed: 2019-11-14. 2017 (cit. on pp. 55, 56, 60, 85, 130).

[131] Takanori Isobe, Ryoma Ito, and Kazuhiko Minematsu. "Security Analysis of SFrame." In: *European Symposium on Research in Computer Security*. 2021, pp. 127–146 (cit. on pp. 45, 48, 56, 58, 130).

[132] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. "Breaking and repairing GCM security proofs." In: *Annual Cryptology Conference*. 2012, pp. 31–49 (cit. on p. 50).

[133] Tetsu Iwata and Yannick Seurin. "Reconsidering the Security Bound of AES-GCM-SIV." In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 240–267 (cit. on p. 50).

[134] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. "Seems legit: Automated analysis of subtle attacks on protocols that use signatures." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2165–2180 (cit. on pp. 1, 19, 65, 74, 86, 131).

[135] Charlie Jacomme. "Preuves de protocoles cryptographiques : méthodes symboliques et attaquants puissants. (Proofs of security protocols : symbolic methods and powerful attackers)." PhD thesis. University of Paris-Saclay, France, 2020. URL: https://tel.archives-ouvertes.fr/tel-02972373 (cit. on p. 7).

[136] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption." In: *ACM CCS*. 2015 (cit. on p. 96).

[137] Ik Rae Jeong, Jonathan Katz, and Dong Hoon Lee. *One-round protocols for two-party authenticated key exchange*. 2008 (cit. on p. 35).

[138] Jakob Jonsson. "On the security of CTR+ CBC-MAC." In: *International Workshop on Selected Areas in Cryptography*. 2002, pp. 76–93 (cit. on p. 50).

[139] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. 2014. DOI: 10.17487/RFC7296. URL: https://www.rfc-editor.org/info/rfc7296 (cit. on pp. 35, 130).

[140] John Kelsey and Tadayoshi Kohno. "Herding hash functions and the Nostradamus attack." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2006, pp. 183–200 (cit. on p. 29).

[141] Chong Hee Kim and Gildas Avoine. "RFID Distance Bounding Protocol with Mixed Challenges to Prevent Relay Attacks." In: *Cryptology and Network Security (CANS)*. Springer, 2009. DOI: 10.1007/978-3-642-10433-6_9. URL: https://doi.org/10.1007/978-3-642-10433-6_9 (cit. on p. 35).

[142] Darrell Kindred and Jeannette Wing. "Fast, automatic checking of security protocols." In: *USENIX 2nd Workshop on Electronic Commerce*. 1996 (cit. on p. 1).

[143] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach." In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017 (cit. on p. 24).

[144] Werner Koch, Paul Wouters, Daniel Huigens, and Justus Winter. *OpenPGP Message Format*. Internet-Draft draft-ietf-openpgp-crypto-refresh-06. Work in Progress. Internet Engineering Task Force, June 2022. 163 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-openpgp-crypto-refresh/06/ (cit. on pp. 55, 56, 60, 130).

[145] Hugo Krawczyk. "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)." In: *ACM CCS*. 2016 (cit. on p. 96).

[146] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols." In: *Advances in Cryptology - CRYPTO 2003*. Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-45146-4 (cit. on pp. 26, 35, 78, 130).

[147] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. Tech. rep. 1997 (cit. on pp. 29, 111).

[148] Hugo Krawczyk and Pasi Eronen. *HMAC-based extract-and-expand key derivation function (HKDF)*. Tech. rep. 2010 (cit. on p. 111).

[149] Hugo Krawczyk and Hoeteck Wee. "The OPTLS Protocol and TLS 1.3." In: *IACR Cryptol. ePrint Arch.* (2015) (cit. on p. 96).

[150] Steve Kremer and Robert Künnemann. "Automated analysis of security protocols with global state." In: *Journal of Computer Security* 24.5 (2016), pp. 583–616 (cit. on pp. 1, 12).

[151] Steve Kremer and Mark D. Ryan. "Analysing the Vulnerability of Protocols to Produce Known-pair and Chosen-text Attacks." In: *Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages and Systems (SecCo'04)*. Ed. by Riccardo Focardi and Gianluigi Zavattaro. May 2005, pp. 84–107. DOI: 10.1016/j.entcs.2004.11.043 (cit. on p. 86).

[152] Ted Krovetz and Phillip Rogaway. "The software performance of authenticated-encryption modes." In: *International Workshop on Fast Software Encryption*. 2011, pp. 306–327 (cit. on p. 50).

[153] Robert Künnemann and Graham Steel. "YubiSecure? Formal security analysis results for the Yubikey and YubiHSM." In: *International Workshop on Security and Trust Management*. Springer. 2012, pp. 257–272 (cit. on pp. 45, 48, 56, 58, 86, 130).

[154] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange." In: *Provable Security, First International Conference, ProvSec*. Springer, 2007. DOI: 10.1007/978-3-540-75670-5_1. URL: https://doi.org/10.1007/978-3-540-75670-5_1 (cit. on p. 35).

[155] Xiao Lan, Jing Xu, Zhen-Feng Zhang, and Wen-Tao Zhu. "Investigating the Multi-Ciphersuite and Backwards-Compatibility Security of the Upcoming TLS 1.3." In: *IEEE Trans. Dependable Secur. Comput.* 16 (2019) (cit. on p. 96).

[156] Kristin E. Lauter and Anton Mityagin. "Security Analysis of KEA Authenticated Key Exchange Protocol." In: *International Conference on Theory and Practice of Public-Key Cryptography*. Springer, 2006. DOI: 10.1007/11745853_25. URL: https://doi.org/10.1007/11745853_25 (cit. on p. 35).

[157] Julia Len, Paul Grubbs, and Thomas Ristenpart. *Authenticated Encryption with Key Identification*. Cryptology ePrint Archive, Paper 2022/1680. https://eprint.iacr.org/2022/1680. 2022 (cit. on p. 65).

[158] Julia Len, Paul Grubbs, and Thomas Ristenpart. "Partitioning Oracle Attacks." In: *30th USENIX Security Symposium*. 2021 (cit. on pp. 45, 48–51, 53, 86).

[159] Gaëtan Leurent and Thomas Peyrin. "Sha-1 is a shambles: First chosen-prefix collision on sha-1 and application to the PGP web of trust." In: *USENIX Security Symposium*. USENIX Association, 2020 (cit. on pp. 23, 26).

[160] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. "Multiple Handshakes Security of TLS 1.3 Candidates." In: *IEEE Symposium on Security and Privacy (SP)*. 2016 (cit. on p. 96).

[161] Felix Linker, Ralf Sasse, and David A. Basin. "A Formal Analysis of Apple's iMessage PQ3 Protocol." In: *IACR Cryptol. ePrint Arch.* (2024), p. 1395 (cit. on pp. 95, 96).

[162] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. 2006. DOI: 10.17487/RFC4253. URL: https://www.rfc-editor.org/info/rfc4253 (cit. on pp. 35, 130).

[163] Gavin Lowe. "A hierarchy of authentication specifications." In: *CSFW*. 1997 (cit. on p. 113).

[164] Gavin Lowe. "Breaking and fixing the Needham-Schroeder public-key protocol using FDR." In: *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1996 (cit. on p. 1).

[165] Sjouke Mauw, Zach Smith, Jorge Toro-Pozo, and Rolando Trujillo-Rasua. "Distance-Bounding Protocols: Verification without Time and Location." In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2018. DOI: 10.1109/SP.2018.00001. URL: https://doi.org/10.1109/SP.2018.00001 (cit. on p. 35).

[166] Sjouke Mauw, Zach Smith, Jorge Toro-Pozo, and Rolando Trujillo-Rasua. "Post-Collusion Security and Distance Bounding." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2019. DOI: 10.1145/3319535.3345651. URL: https://doi.org/10.1145/3319535.3345651 (cit. on p. 35).

[167] David A McGrew and John Viega. "The security and performance of the Galois/Counter Mode (GCM) of operation." In: *International Conference on Cryptology in India*. 2004, pp. 343–355 (cit. on p. 50).

[168] Catherine A. Meadows, Radha Poovendran, Dusko Pavlovic, LiWu Chang, and Paul F. Syverson. "Distance Bounding Protocols: Authentication Logic Analysis and Collusion Attacks." In: *Secure Localization and Time Synchronization for Wireless Sensor and Ad Hoc Networks*. Springer, 2007. DOI: 10.1007/978-0-387-46276-9_12. URL: https://doi.org/10.1007/978-0-387-46276-9_12 (cit. on p. 35).

[169] Simon Meier. "Advancing automated security protocol verification." PhD thesis. ETH Zurich, 2013 (cit. on pp. 7, 32, 33, 35).

[170] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. "The TAMARIN prover for the symbolic analysis of security protocols." In: *International conference on computer aided verification*. 2013, pp. 696–701 (cit. on pp. 1, 12, 13).

[171] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. "Hamming quasi-cyclic (HQC)." In: *NIST PQC Round* (2018) (cit. on p. 65).

[172] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997 (cit. on pp. 24, 30, 86).

[173] Kazuhiko Minematsu, Stefan Lucks, and Tetsu Iwata. "Improved authenticity bound of EAX, and refinements." In: *International Conference on Provable Security*. 2013, pp. 184–201 (cit. on p. 50).

[174] Serge Mister and Robert Zuccherato. "An attack on CFB mode encryption as used by OpenPGP." In: *International Workshop on Selected Areas in Cryptography*. 2005, pp. 82–94 (cit. on p. 48).

[175] Arno Mittelbach and Marc Fischlin. *The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography*. Springer, 2021 (cit. on p. 86).

[176] Payman Mohassel. "A Closer Look at Anonymity and Robustness in Encryption Schemes." In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*. Vol. 6477. Lecture Notes in Computer Science. Springer, 2010, pp. 501–518. DOI: 10.1007/978-3-642-17373-8_29. URL: https://www.iacr.org/archive/asiacrypt2010/6477505/6477505.pdf (cit. on p. 88).

[177] Jorge Munilla and Alberto Peinado. "Distance bounding protocols for RFID enhanced by using void-challenges and analysis in noisy channels." In: *Wirel. Commun. Mob. Comput.* (2008). DOI: 10.1002/wcm.590. URL: https://doi.org/10.1002/wcm.590 (cit. on p. 35).

[178] NIST. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. https://csrc.nist.gov/pubs/fips/203/ipd. Accessed: 2024-02-23 (cit. on pp. 88, 89).

[179] NIST. *NIST Post-Quantum Cryptography*. https://csrc.nist.gov/projects/post-quantum-cryptography. Accessed: 2024-01-16 (cit. on pp. 65, 66).

[180] Karl Norrman, Vaishnavi Sundararajan, and Alessandro Bruni. "Formal Analysis of EDHOC Key Establishment for Constrained IoT Devices." In: *CoRR* abs/2007.11427 (2020). arXiv: 2007.11427 (cit. on p. 12).

[181] M. Nystrom and B. Kaliski. *RFC 2986: PKCS #10: Certification Request Syntax Specification Version 1.7*. https://www.rfc-editor.org/rfc/rfc2986 (accessed: 2022-10-09). 2000 (cit. on p. 102).

[182] E. Omara, J. Uberti, A. GOUAILLARD, and S. Murillo. *Secure Frame (SFrame) v01*. https://datatracker.ietf.org/doc/html/draft-omara-sframe-01. accessed: 2022-08-08. 2020 (cit. on pp. 48, 52, 55, 56, 58, 130).

[183] Kenneth G. Paterson, Matteo Scarlata, and Kien T. Truong. "Three Lessons From Threema: Analysis of a Secure Messenger." In: *USENIX Security Symposium*. USENIX Association, 2023 (cit. on p. 95).

[184] Chris Peikert. *Lattice Cryptography for the Internet*. Cryptology ePrint Archive, Paper 2014/070. https://eprint.iacr.org/2014/070. 2014 (cit. on pp. 78, 79).

[185] A. Perrig, R. Canetti, J.D. Tygar, and Dawn Song. "Efficient authentication and signing of multicast streams over lossy channels." In: *IEEE Symposium on Security and Privacy. (S&P)*. 2000. DOI: 10.1109/SECPRI.2000.848446 (cit. on p. 35).

[186] Thomas Pornin and Julien P Stern. "Digital signatures do not guarantee exclusive ownership." In: *Applied Cryptography and Network Security: Third International Conference, ACNS 2005*. 2005 (cit. on p. 65).

[187] Gordon Procter. "A Security Analysis of the Composition of ChaCha20 and Poly1305." In: *Cryptology ePrint Archive, Paper 2014/613*. https://eprint.iacr.org/2014/613. 2014 (cit. on p. 50).

[188] Kasper Bonne Rasmussen and Srdjan Capkun. "Realization of RF Distance Bounding." In: *USENIX Security Symposium*. USENIX Association, 2010. URL: http://www.usenix.org/events/sec10/tech/full_papers/Rasmussen.pdf (cit. on p. 35).

[189] R Rivest. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." In: *Communications of the ACM* (1978) (cit. on p. 11).

[190] Phillip Rogaway. "Authenticated-Encryption with Associated-Data." In: *ACM Conference on Computer and Communications Security (CCS)*. CCS '02. 2002, 98–107. ISBN: 1581136129. DOI: 10.1145/586110.586125 (cit. on pp. 45–47, 49, 50).

[191] Phillip Rogaway and Thomas Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance." In: *IACR Cryptol. ePrint Arch.* (2004) (cit. on p. 86).

[192] Phillip Rogaway and John Steinberger. "Security/Efficiency Tradeoffs for Permutation-Based Hashing." In: *EUROCRYPT 2008*. 2008, pp. 220–236. ISBN: 978-3-540-78967-3 (cit. on p. 51).

[193] Joeri de Ruiter. "A tale of the OpenSSL state machine: A large-scale black-box analysis." In: *Secure IT Systems: 21st Nordic Conference, NordSec*. Springer. 2016 (cit. on p. 96).

[194] *Saltpack v2*. https://saltpack.org/encryption-format-v2. accessed: 2022-08-08. 2017 (cit. on pp. 55, 56, 61, 130).

[195] Yu Sasaki and Kazumaro Aoki. "Finding preimages in full MD5 faster than exhaustive search." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2009 (cit. on p. 23).

[196] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. "New Message Difference for MD4." In: *Fast Software Encryption - International Workshop FSE*. Springer, 2007. DOI: 10.1007/978-3-540-74619-5_21. URL: https://doi.org/10.1007/978-3-540-74619-5_21 (cit. on p. 23).

[197] Benedikt Schmidt. "Formal analysis of key exchange protocols and physical protocols." eng. PhD thesis. ETH, 2012. DOI: 10.3929/ethz-a-009898924 (cit. on pp. 32, 33).

[198] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. "Automated analysis of Diffie-Hellman protocols and advanced security properties." In: *Computer Security Foundations Symposium (CSF)*. IEEE. 2012 (cit. on p. 35).

[199] Peter Schwabe, Douglas Stebila, and Thom Wiggers. *More efficient post-quantum KEMTLS with pre-distributed public keys*. Cryptology ePrint Archive, Paper 2021/779. https://eprint.iacr.org/2021/779. 2021 (cit. on p. 86).

[200] Peter Schwabe, Douglas Stebila, and Thom Wiggers. *Post-quantum TLS without handshake signatures*. Cryptology ePrint Archive, Paper 2020/534. https://eprint.iacr.org/2020/534. 2020. DOI: 10.1145/3372297.3423350 (cit. on p. 86).

[201] *Scuttlebot Private Box v0.3.1*. https://scuttlebot.io/more/protocols/private-box.html. accessed: 2022-08-08. 2019 (cit. on pp. 55, 56, 61, 85, 130).

[202] Alon Shakevsky, Eyal Ronen, and Avishai Wool. "Trust Dies in Darkness: Shedding Light on Samsung's TrustZone Keymaster Design." In: *Cryptology ePrint Archive, Paper 2022/208*. https://eprint.iacr.org/2022/208. 2022 (cit. on pp. 45, 48, 58).

[203] Victor Shoup. "A Proposal for an ISO Standard for Public Key Encryption." In: *IACR Cryptology ePrint Archive* 2001 (2001), p. 112. URL: http://dblp.uni-trier.de/db/journals/iacr/iacr2001.html\#Shoup01 (cit. on p. 66).

[204] Victor Shoup. "Sequences of games: a tool for taming complexity in security proofs." In: *cryptology eprint archive* (2004) (cit. on p. 11).

[205] Sofía Celi and Jonathan Hoyland and Douglas Stebila and Thom Wiggers. *A tale of two models: formal verification of KEMTLS via Tamarin*. Cryptology ePrint Archive, Paper 2022/1111. https://eprint.iacr.org/2022/1111. 2022. URL: https://eprint.iacr.org/2022/1111 (cit. on pp. 86, 87).

[206] Sophie Schmieg. *Unbindable Kemmy Schmidt: ML-KEM is neither MAL-BIND-K-CT nor MAL-BIND-K-PK*. Cryptology ePrint Archive, Paper 2024/523. https://eprint.iacr.org/2024/523. 2024 (cit. on p. 88).

[207] Marc Stevens. "A Survey of Chosen-Prefix Collision Attacks." In: *Computational Cryptography: Algorithmic Aspects of Cryptology*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2021, 182–220. DOI: 10.1017/9781108854207.009 (cit. on pp. 19, 23, 25, 26).

[208] Marc Stevens, Arjen Lenstra, and Benne De Weger. "Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities." In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2007 (cit. on pp. 19, 23, 25, 26).

[209] The Tamarin Team. *Tamarin-Prover Manual*. URL: https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf (cit. on p. 34).

[210] Telegram. *Mobile protocol: Detailed description*. http://web.archive.org/web/20210126200309/https://core.telegram.org/mtproto/description. 2021 (cit. on pp. 35, 37).

[211] Martin Thomson. *Message Encryption for Web Push*. RFC 8291. Nov. 2017. DOI: 10.17487/RFC8291. URL: https://www.rfc-editor.org/info/rfc8291 (cit. on pp. 55, 56, 59, 85, 130).

[212] Jules van Thoor, Joeri de Ruiter, and Erik Poll. "Learning state machines of TLS 1.3 implementations." In: *Bachelor thesis. Radboud University* (2018) (cit. on p. 96).

[213] Gene Tsudik. "Message authentication with one-way hash functions." In: *Comput. Commun. Rev.* 22.5 (1992) (cit. on p. 24).

[214] Mathy Vanhoef and Frank Piessens. "Key reinstallation attacks: Forcing nonce reuse in WPA2." In: *ACM Conference on Computer and Communications Security (CCS)*. 2017, pp. 1313–1328 (cit. on p. 48).

[215] Serge Vaudenay. "Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS..." In: *International Conference on the Theory and Applications of Cryptographic Techniques*. 2002, pp. 534–545 (cit. on p. 48).

[216] Zooko Wilcox. *Lessons From The History Of Attacks On Secure Hash Functions*. https://electriccoin.co/blog/lessons-from-the-history-of-attacks-on-secure-hash-functions/ (Retrieved Jan 2022). 2017 (cit. on p. 23).

[217] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. "Formal Model-Driven Discovery of Bluetooth Protocol Design Vulnerabilities." In: *IEEE Symposium on Security and Privacy (S&P)*. 2022, pp. 2285–2303 (cit. on pp. 12, 95).

[218] Tao Xie, Fanbao Liu, and Dengguo Feng. "Fast Collision Attack on MD5." In: *IACR Cryptol. ePrint Arch.* (2013). URL: http://eprint.iacr.org/2013/170 (cit. on p. 23).

[219] Jiewen Yao, Anas Hlayhel, and Krystian Matusiewicz. "Post Quantum KEM authentication in SPDM for secure session establishment." In: *Design & Test* (2023) (cit. on pp. 76, 81, 96).

[220] Jiewen Yao, Krystian Matusiewicz, and Vincent Zimmer. "Post Quantum Design in SPDM for Device Authentication and Key Establishment." In: *Cryptography* 6.4 (2022), p. 48 (cit. on pp. 76, 81, 96).

[221] *YubiHSM*. https://www.yubico.com/resource/hsm-security-for-manufacturing/. accessed: 2022-08-08. 2021 (cit. on pp. 55, 56, 58, 130).

[222] *Zerologon – hacking Windows servers with a bunch of zeros*. https://nakedsecurity.sophos.com/2020/09/17/zerologon-hacking-windows-servers-with-a-bunch-of-zeros/. accessed: 2023-02-01. 2020 (cit. on p. 48).

[223] Mang Zhao. "Provable security and real-world protocols: theory and practice." In: *Saarländische Universitäts-und Landesbibliothek* (2023) (cit. on pp. 2, 17, 47, 50, 51).

[224] Jinmin Zhong and Xuejia Lai. "Improved preimage attack on one-block MD4." In: *Journal of Systems and Software* (2012) (cit. on p. 23).

IV

Appendices

SPDM Transcripts

A.1 Transcripts for Challenge

For the challenge transcript the following traces are possible:

1. *GET_VERSION*, *GET_CAPABILITIES*, *NEGOTIATE_ALGORITHMS*, *GET_DIGESTS*, *GET_CERTIFICATE*, *CHALLENGE*
2. *GET_VERSION*, *GET_CAPABILITIES*, *NEGOTIATE_ALGORITHMS*, *GET_DIGESTS*, *CHALLENGE*
3. *GET_VERSION*, *GET_CAPABILITIES*, *NEGOTIATE_ALGORITHMS*, *GET_CERTIFICATE*, *CHALLENGE*
4. *GET_VERSION*, *GET_CAPABILITIES*, *NEGOTIATE_ALGORITHMS*, *CHALLENGE*
5. *GET_VERSION*, *GET_CAPABILITIES*, *NEGOTIATE_ALGORITHMS*, *CHALLENGE*
6. *GET_DIGESTS*, *GET_CERTIFICATE*, *CHALLENGE* (if stored VCA)
7. *GET_DIGESTS*, *CHALLENGE* (if stored VCA and cached previous certificate)
8. *GET_CERTIFICATE*, *CHALLENGE* (if stored VCA and cached previous certificate)
9. *CHALLENGE* (if stored VCA and cached previous certificate)

A.2 Transcripts for Measurement

The transcript for measurements is as follows:

$$\text{VCA}, \mathit{GET\_MEASUREMENTS}.*, \mathit{MEASUREMENTS}.* \tag{A.1}$$

A.3 Transcripts during Key Agreement

Transcript for HMAC in Pre-shared Symmetric Keys In the pre-shared symmetric keys, the parties do not include the digest of the certificates or public keys. In addition, some requests in the VCA phase may also not be issued. The transcript can be read as the sequence from the start until the current message to be sent, e.g. to authenticate the transcript in the *PSK_EXCHANGE_RSP*, the Responder will include all the values of the request to be issued itself, except the HMAC field called *ResponderVerifyData*. From the specifications, the transcript is defined as:

- *GET_VERSION*.*
- *VERSION*.*
- *GET_CAPABILITIES*.* (if issued)
- *CAPABILITIES*.* (if issued)
- *NEGOTIATE_ALGORITHMS*.* (if issued)
- *ALGORITHMS*.* (if issued)
- *PSK_EXCHANGE*.*
- *PSK_EXCHANGE_RSP*.* (- *ResponderVerifyData*)
- *PSK_FINISH*.* (- *RequestorVerifyData*)

A.4 Transcript for HMAC in Key Exchange

The parties send the HMAC of the transcript during all messages except the *KEY_EXCHANGE* request during key exchanges. In all cases, the transcript includes VCA, the available certificates, and the session handshake messages up to and including the current one. In the following we show the message sequences:

- VCA
- Hash of the Responder certificate or provisioned public key
- *KEY_EXCHANGE*.*
- *KEY_EXCHANGE_RSP*.* (transcript for Key Exchange Response)
- (Hash of the Requester certificate or provisioned public keys) (if mutual authentication)
- *FINISH*.* (transcript for Finish Request)
- *FINISH_RSP*.Headers (transcript for Finish Response)

A.5 Transcript for Signature in Key Exchange

The signature appended in the *KEY_EXCHANGE_RSP* and *FINISH* messages, is computed by signing a pre-defined transcript with the private key of the device's certificate. The transcript to be signed is the concatenation of the message sequence:

- VCA
- Hash of the Responder certificate/public key
- *KEY_EXCHANGE*.*
- *KEY_EXCHANGE_RSP*.* (transcript for Key Exchange Response, except the Signature and HMAC field)
- (Hash of the Requester certificate/public key) (if mutual authentication)
- *FINISH*.Headers (transcript for Finish Request)

A.6 Transcript for Key Derivation

To compute session secrets, the parties also include the key agreement transcript in the key derivation function. In the protocol we need to define two transcripts: 1. TH1-to derive role-directed secrets in the handshake phase, and 2. TH2-to derive session secrets.

Transcript TH1 for Key-Exchange (and pre-shared keys):

- VCA
- Hash of the Responder certificate
- *KEY_EXCHANGE*.*
- *KEY_EXCHANGE_RSP*.* (- *ResponderVerifyData*)

Transcript TH1 for pre-shared Symmetric Keys:

- VCA
- *PSK_EXCHANGE*.*
- *PSK_EXCHANGE_RSP*.* (- *ResponderVerifyData*)

Transcript for TH2 for Key-Exchange:

- VCA
- Hash of the Responder certificate/public key
- *KEY_EXCHANGE*.*
- *KEY_EXCHANGE_RSP*.* (- *ResponderVerifyData*)
- (Hash of the Requester certificate/public key) (if mutual authentication)
- *FINISH*.*
- *FINISH_RSP*.*

Transcript for TH2 for pre-shared Symmetric Keys:

- VCA
- *PSK_EXCHANGE*.*
- *PSK_EXCHANGE_RSP*.* (- *ResponderVerifyData*)
- *PSK_FINISH*.* (if issued)
- *PSK_FINISH_RSP*.* (if issued)

A.7 Transcripts for Verifying Data

During the key exchange *ResponderVerifyData* and *RequestorVerifyData* are computed by building the HMAC of the transcript with the finished key. The transcript is composed of the VCA phase message exchange and the key exchange messages sent and received up until that point in the protocol. In the certificate mode, the transcript also includes the hash of the parties' certificates, as shown below:

1. VCA
2. Hash of Requester certificate
3. *KEY_EXCHANGE*
4. *KEY_EXCHANGE_RSP* **(T1)**
5. Hash of Responder certificate
6. *FINISH* **(T2)**
7. *FINISH_RSP* only SPDM header fields **(T3)**

Note that the transcript for the signatures in certificates mode is computed similarly. The main difference between the two is that HMAC includes signatures in its calculation and transcript, while *T1′* in the HMAC does not take into account the signature and *ResponderVerifyData* fields. However, *T1* in the HMAC includes the signature field in its computation. The transcripts in the PSK mode is constructed as the concatenation of the following messages:

1. VCA (CA only if issued)
2. *PSK_EXCHANGE*
3. *PSK_EXCHANGE_RSP* **(T1)**
4. *PSK_FINISH* except *ResponderVerifyData* **(T2)**

A.8 Transcripts for Key Derivation

Parties compute two transcripts that are included in the key derivation functions, namely *TH1* for role-directed handshake secrets and *TH2* for the application data secrets. *TH1* is computed from the concatenation from VCA up to and including the (key/psk) exchange response except for the *ResponderVerifyData* field. *TH2* is computed as the concatenation of all the listed messages with all of their fields (including *PSK_FINISH_RSP* for the PSK mode).

A.9 Handshake Secrets

The handshake secret is computed from the initial shared secret between the parties key_{init} and a zero-filled array ($Salt_0$): $handshake$ = HMAC($Salt_0$, key_{init}). Note, that key_{init} is the Diffie-Hellman output for the certificate mode, and the provisioned PSK for the pre-shared symmetric key mode. From the handshake secret the parties derive role-oriented handshake secrets by including in the parameters of the key derivation a fixed string of their role, and the transcript of the VCA phase and key exchange until that point, e.g., for the Requester, the requester handshake secret is $handshake_{Req}$ = HKDF($handshake$, "req", *TH1*, $\cdots$). The role oriented secrets serve to compute the HMAC keys, so-called finished keys fk, of the *ResponderVerifyData* and *RequestorVerifyData*: fk_{Req} = HKDF($handshake_{Req}$, "finished", $\cdots$).

SPDM Request and Response Codes

B

Request/Response Codes	Included	Notes
GET_VERSION & VERSION	✓	
GET_CAPABILITIES & CAPABILITIES	✓	
NEGOTIATE_ALGORITHMS & ALGORITHMS	✓	
GET_DIGESTS & DIGESTS	✓	
GET_CERTIFICATE & CERTIFICATE	✓	
CHALLENGE & CHALLENGE_AUTH	✓	
GET_MEASUREMENTS & MEASUREMENTS	✓	
ERROR		out of scope for Tamarin
RESPOND_IF_READY		out of scope for Tamarin
VENDOR_DEFINED_REQUEST & VENDOR_DEFINED_RESPONSE		See Discussion in Section 9.5.2
KEY_EXCHANGE & KEY_EXCHANGE_RSP	✓	
FINISH & FINISH_RSP	✓	
PSK_EXCHANGE & PSK_EXCHANGE_RSP	✓	
PSK_FINISH & PSK_FINISH_RSP	✓	
HEARTBEAT & HEARTBEAT_ACK		out of scope for Tamarin
KEY_UPDATE & KEY_UPDATE_ACK	✓	
GET_ENCAPSULATED_REQUEST & ENCAPSULATED_REQUEST	✓	
DELIVER_ENCAPSULATED_RESPONSE & ENCAPSULATED_RESPONSE_ACK	✓	
END_SESSION & END_SESSION_ACK	✓	
GET_CSR & CSR		See Discussion in Section 9.5.2
SET_CERTIFICATE & SET_CERTIFICATE_RSP		See Discussion in Section 9.5.2
CHUNK_SEND & CHUNK_SEND_ACK		out of scope for Tamarin
CHUNK_GET & CHUNK_RESPONSE		out of scope for Tamarin

Table B.1: List of all Request and Response codes in SPDM. Details can be found in [96]