Breaking and Provably Restoring Authentication: A Formal Analysis of SPDM 1.2 including Cross-Protocol Attacks

Version 1.0, December 18, 2024*

Cas Cremers, Alexander Dax, and Aurora Naska

CISPA Helmholtz Center for Information Security
{cremers,alexander.dax,aurora.naska}@cispa.de

Abstract

The SPDM (Security Protocol and Data Model) protocol is a standard under development by the DMTF consortium, and supported by major industry players including Broadcom, Cisco, Dell, Google, HP, IBM, Intel, and NVIDIA. SPDM 1.2 is a complex protocol that aims to provide platform security, for example for communicating hardware components or cloud computing scenarios.

In this work, we provide the first holistic, formal analysis of SPDM 1.2: we model the full protocol flow of SPDM considering all of its modes – especially the complex interaction between its different key-exchange modes – in the framework of the Tamarin prover, making our resulting model one of the most complex Tamarin models to date. To our surprise, Tamarin finds a cross-protocol attack that allows a network attacker to completely break authentication of the pre-shared key mode. We implemented our attack on the SPDM reference implementation, and reported the issue to the SPDM developers. DMTF registered our attack as a CVE with CVSS rating 9 (critical).

We propose a fix and develop the first formal symbolic proof using the Tamarin prover for the fixed SPDM 1.2 protocol as a whole. The resulting model of the main modes and their interactions is highly complex, and we develop supporting lemmas to enable proving properties in the Tamarin prover, including the absence of *all cross-protocol attacks*. Our fix has been incorporated into both the reference implementation and the newest version of the standard. Our results highlight the need for a holistic analysis of other internet standards and the importance of providing generalized security guarantees across entire protocols.

1 Introduction

Despite decades of research in security protocol analysis (notably in the computational and symbolic traditions), the vast majority of formal analyses still only consider protocol *components*, i.e., a single key exchange using a specific cryptographic primitive (2-6 messages), a ratcheting step in messaging (1-4 messages), a transmission layer step (1-2 messages), or a key renegotiation (2 messages). However, many real-world protocols such as WhatsApp or Signal use all these modes during the same communication session, and protocols like TLS additionally support multiple types of bootstrapping primitives (e.g., pre-shared symmetric keys or certificates). Thus, real-world protocols exhibit behaviors where multiple protocol components share data, which are not covered by the proofs of the components.

It is well known that the composition of protocol components can lead to attacks, even when the individual components were proven secure. Notable examples of attacks that were found at the protocol composition level include the attack on delayed authentication in a draft of the TLS 1.3 protocol [15], the cross-stack attack on the Bluetooth Classic and Low Energy protocols [37], and cross-protocol attacks on Threema [35] and Matrix [1]. While there are protocol composition results in the literature, such as [10, 30, 9, 31, 29, 32], these typically do not apply to real-world protocols because they do not follow certain ideal design principles, or require abstractions that restrict the considered attack classes.

At the same time, increasingly complex real-world protocols are developed and deployed globally. Recently, major industry players such as Broadcom, Dell, Intel, Cisco, NVIDIA, Google, IBM, HP, and many others have prioritized the need for security and integrity of their platforms, in the so-called

*We provide a list of main changes in Appendix F.

platform root of trust, and to this end supported the standardization of the Security Protocol and Data Model (SPDM) protocol [21]. The main goals of SPDM, as stated in the technical note [28], are: a) *cryptographically verifying the identity and firmware integrity of each platform component* and b) *private and secure data communications over platform interfaces.* SPDM is being implemented in hardware components (e.g., Intel, Dell, NVIDIA, Broadcom), cloud platforms (e.g., Google, Cisco, IBM), and operating systems (e.g., Linux). Moreover, SPDM is the core security mechanism for the CXL 2.0 (Compute Express Link) standard, which in turn is part of PCIe 5.0, set to be implemented in all Intel and AMD server CPUs in the near future, and will serve as the cornerstone for hardware-based Confidential Computing in Intel TDX and NVIDIA Hopper.

The SPDM standard is under active development by the Distributed Management Task Force (DMTF). Version 1.2 includes a range of components: several key exchanges (from pre-shared symmetric keys, certificates, or pre-shared public keys), state-reset mechanisms, and a key update mechanism. Much like versions of TLS prior to 1.3, SPDM 1.2 does not explicitly specify state machines or their composition. Instead, the standard explains the structure of messages, and under which conditions certain steps can be taken or responses are sent.

The most substantial security analysis that SPDM has received is the symbolic analysis by Cremers, Dax, and Naska [12] using the Tamarin prover [34]. Their analysis considers the main components of the protocol (negotiation phases, key exchange, measurements, loops) and the different key exchange modes (pre-shared symmetric keys, certificates, and pre-shared public keys) and they prove the main security properties for flows of the key exchange modes *in isolation.* The authors noted that a main limitation of their work was that they did not manage to analyze the full protocol, i.e., they performed several separate smaller analyses and did not consider the interaction between modes.

In this work, we set out to perform the first formal analysis of the full SPDM 1.2 protocol, i.e., corresponding to implementations that support all key exchanges. In reality, implementations are likely to use or follow the official SPDM reference implementation (libspdm [23]), and notably generic implementations in CPUs and the Linux kernel will support multiple key exchange modes.

Modeling all modes in a single model leads to one of the largest Tamarin models to date. To give a coarse indication of the complexity: our model has 71 transition rules and 42 analyzed lemmas, with the largest proof search requiring 261 proof steps. The previous SPDM models had between 22 and 41 transition rules each, with 11–23 lemmas per models, and the largest proof being 53 proof steps. We provide more details of this comparison in Appendix D. Comparing our model very coarsely with other large Tamarin case-studies: the PQ3 model has 22 rules [33], the WPA2 model has 69 rules [16], and TLS 1.3 has 63 rules [14].[1] In practice, the complexity of the analysis problem is often exponential with respect to the model size, similar to state space explosion. However, since the general problem is undecidable, there is no a priori guarantee a counterexample or proof can be constructed.

We expected the composition to be secure, but challenging to prove. Much to our surprise, Tamarin finds a critical attack, showing that previous results do not hold for the entire protocol. We implemented the attack from the analysis on the official DMTF reference implementation written in C [27] (and Rust implementation [24]), for which DMTF registered a CVE with critical severity. We proposed and formally proved the security of a fixed version of the standard, and as a result both standard and reference implementation have now been updated.

Our main contributions are:

- We construct the first formal model of the SPDM 1.2 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 specification and its complex state machine interactions.
- Using our formal model, Tamarin finds a critical cross-protocol attack on SPDM 1.2. 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 [17] with CVSS score 9.0 (critical).
- We propose a fix for the vulnerability and formally prove that the fixed version of SPDM 1.2 provides authentication for all modes and other basic security guarantees, *even in the presence of cross-protocol attacks and all mode interactions.* 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,

[1]There are other Tamarin case studies that involve a large number of rules, such as for Noise and Bluetooth, but these perform multiple analyses where each analysis only considers a small subset of the rules, which leads to more linear scaling.

and transcripts. Both the SPDM standard and its reference implementation have been updated based on our work.

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. One possible way to handle complexity is for implementers and standard maintainers to adopt domain separation of their keys. Lastly, our work shows that modeling decisions, e.g., processes vs. transitions, matter for the class of attacks that is covered by the analysis. We discuss lessons learned and how to handle complexity in Sections 3.3 and 6.

We provide all models and additional materials needed to reproduce and study our results at [13].

Roadmap

We first provide background information on Tamarin and SPDM in Section 2. Then, we present our model of SPDM as a whole in Section 3, followed by the attack we discovered in Section 4. To prevent the attack, we propose a formally proven fix in Section 5. We address further issues in Section 6. In the end, we discuss the disclosing process and follow-up in Section 7 and present our conclusions in Section 8. In Appendix A, we give details on protocol transcripts and secrets, in Appendix B we show a lemma to quickly reproduce the attack trace in Tamarin, and in Appendix C we show our mode switch attack. In Appendix D, we compare our analysis to previous ones and in Appendix E we show helper lemma examples.

Related Work

While DMTF's Security Protocol and Data Model (SPDM) is backed by a large number of important industry players, it has only seen limited academic attention.

Recently, [2, 3] conducted benchmark studies of SPDM implementations. While these benchmarks are valuable for evaluating the performance of SPDM protocols, they do not delve into their security aspects.

Two recent works [39, 38] aim towards providing post-quantum security for SPDM. [39] focuses on the identifying the changes needed for a post-quantum version of SPDM's device authentication and key establishment. To achieve this, they propose a new key exchange based on Key-Encapsulation Mechanisms (KEMs), because the current the Diffie-Hellman (DH) based key exchanges do not yet have a direct counterpart in the post-quantum setting. They leave the security analysis of their proposed scheme for future work. [38] proposes a slightly different version of a KEM based key exchange with the goal to achieve the same guarantees as [39] without using digital signatures. The authors claim to have proofs submitted, but as of writing this, we could not access them.

The most extensive security analysis of SPDM is the work conducted in [12]. In their research, the authors conducted the first symbolic analysis of four SPDM sub-models, focusing on its core protocol modes, such as device initialization, negotiation phases, device attestation, multiple key exchange modes, session establishment, and key updates. Due to the inherent complexity of the protocol, the authors encountered challenges and "had to split all possible flows into four separate models" [12]. They managed to prove several security properties for each of the four separate models, including key secrecy and multiple authentication properties for the models including the different key exchanges. However, while this approach ensured a detailed analysis of the individual protocol modes in isolation, it left the possibility of potential cross-protocol attacks overlooked, thereby preventing these security properties to be lifted to the entire SPDM protocol.

The attack Tamarin finds in our work, as we will see later, can be regarded as a combination between a cross-protocol attack and a state machine flaw. For the TLS protocol, there has been substantial research on possible state machine attack vectors, and how to infer or fuzz the state machines of an implementation, e.g., [7, 20, 18, 36]. Unfortunately, since the SPDM 1.2 standard does not specify state machines, there is no ground truth. It would nevertheless be interesting to see if state machine inference or dedicated fuzzing on the reference implementation might also reveal the mode switch behavior, in which case a human analyst might detect this as undesired behavior. However, since our attack requires the attacker to form non-standard messages (after the mode switch), this highly depends on the details of the threat models considered in these approaches.

The cross-protocol attack Tamarin found, shows – on a real-world protocol – that security guarantees on individual components do not result in the security of the system as a whole. We examine the state-of-the-art of known composition results from the literature and argue why they do not apply to the SPDM case. The result from Ciobâca and Cortier [11] only applies to protocols for which its sub-protocols do not share cryptographic primitives. SPDM, however, uses, e.g., encryption schemes, hash functions, or MACs in multiple of its sub-protocols. Other composition results [10, 30, 9, 31, 29, 32] only apply to more restricted models, where the results of [29] do not apply to the equational theories we use to accurately model the SPDM sub-protocol using a Diffie-Hellman-style key exchange. Furthermore, SPDM's sub-protocols share complex state that includes cryptographic keys, keying material, and transcripts, which prevents the application of general-purpose composition results.

2 Background

In this section, we provide essential background information on the Tamarin prover in Section 2.1. Then, we explain in-depth details of SPDM's key exchange with certificate in Section 2.3, the key exchange with pre-shared symmetric keys in Section 2.4, and the key exchange with pre-shared public keys in Section 2.5.

2.1 The Tamarin Prover

The Tamarin Prover [34] is a state-of-the-art protocol verification tool that is widely used in academia and industry to analyze complex real-world protocols. Tamarin operates in the symbolic setting were messages are expressed as terms and the cryptographic primitives are assumed to be perfect. In the following, we provide a brief overview of the tool, and refer the reader to the official documentation for more information [34].

To verify a protocol in Tamarin, the user needs to provide three inputs: a) the *protocol model*, b) the *security property*, and c) 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

Protocol models are defined as transition systems, where parties actions are expressed using *multiset rewriting rules.* Rules are constructed by a left-hand side (LHS), an action, and a right-hand side (RHS). 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 below, the Tamarin rule shows the creation of the root authority in a PKI.

$$\begin{aligned}&[\ \mathsf{Fr}(\mathit{ltk})\ ]\\ &\text{--}[\ \mathtt{CreateRootAuth}(\mathit{ltk})\ ]\!\!\rightarrow\\ &[\ !\mathtt{RootAuth}(\mathit{ltk}), \mathsf{Out}(\mathit{pk}(\mathit{ltk}))\ ]\end{aligned}$$

The LHS shows the generation of a new unique long-term key *ltk* unknown by the attacker, expressed by the 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.

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 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.

Attacker Model

The tool has a built-in network attacker, often referred to as the Dolev-Yao 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.

2.2 SPDM: Session Setup and Device Attestation

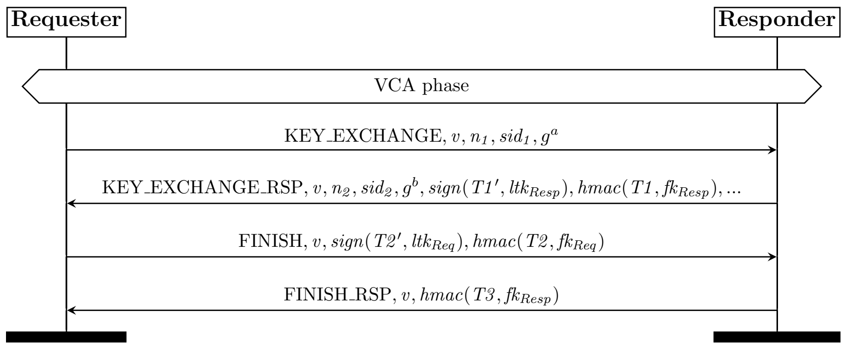
SPDM enables devices to exchange secure and authenticated messages between devices that have been provisioned with identities in three ways: a) certificates signed by a trusted certificate authority, b) pre-shared public keys, and c) pre-shared symmetric keys. Devices supporting SPDM can be provisioned with any combination of these options and are allowed to hold multiple different identities.

Additionally, devices have different capabilities such as the ability to symmetrically encrypt, or sign messages, and they support several algorithms for each of the cryptographic primitives. To bootstrap the protocol, the devices execute the *Version-Capabilities-Algorithms* (VCA) phase of the protocol which allows the parties to agree on the SPDM protocol version and its common set of supported capabilities and algorithms. The device that sends the first request takes on the role of the Requester and maintains the role until the end of the protocol, respectively the responding partner becomes the Responder.

After the VCA phase, the Requester can now either start to establish a secure session (see Sections 2.3 to 2.5) or can first issue any messages to get information from the Responder such as requesting their certificate, issuing a challenge to authenticate, or requesting *measurements* of the devices. The so-called measurements can be any kind of updatable device information like firmware version, firmware policy, device state, or config data. Getting authenticated measurements is one of SPDMs major goals as stated in DMTFs official technical note [28]. Since issuing these requests is optional and does not affect the setup of a secure session later on, this phase is called the *Options* phase.

2.3 SPDM: Key Exchange with Certificates

In the key exchange with certificates, parties establish trust in the identity of their partner using device certificates signed by a root authority. Certificates are then verified using the certificate chain of trust, with the root certificate of a certificate authority (CA) at the top level. The key exchange enables the parties to verify the identity of their partner by signing a challenge using the private key of the certificate, while it uses a Diffie-Hellman computation to derive the base session key, in SPDM called the handshake secret. In the following, we will see how the Requester and Responder mutually authenticate and set up the secure session.



Requester
Responder
VCA phase
KEY_EXCHANGE, v, n_1, sid_1, g^a
KEY_EXCHANGE_RSP, $v, n_2, sid_2, g^b, sign(T1', ltk_{Resp}), hmac(T1, fk_{Resp}), \ldots$
FINISH, $v, sign(T2', ltk_{Req}), hmac(T2, fk_{Req})$
FINISH_RSP, $v, hmac(T3, fk_{Resp})$

Figure 1: Key Exchange with Mutual Authentication in the certificate mode.

The protocol starts in the VCA phase where the parties negotiate the SPDM version, their capabilities and algorithms to be used. Depending on which party sent the first request, they take on the role of the Requester, while their partner will be Responder. These roles will remain unchanged throughout the entire SPDM connection. To start a session, the Requester sends a *KEY_EXCHANGE* request. In this request, they send an ephemeral Diffie-Hellman public key, a 32-byte nonce, 2 bytes of their contribution to the session id and the negotiated version. Upon receiving the request, the Responder generates their own ephemeral Diffie-Hellman pair and computes the handshake secret. Now, the Responder prepares their response by including the a 32-byte nonce, 2-bytes for the session id, a signature of the transcript up until now using their private key of device certificate, and the *ResponderVerifyData*, an HMAC over the transcript using the *finished_key* or the Responder. Details on the transcript can be found in Appendix A.

Once the Requester receives the response, it verifies the signature of the transcript and computes the secrets (role-oriented handshake secrets and finished keys) to setup the session. Then, it verifies the HMAC of the transcript using the responder *finished_key* and starts composing the *FINISH* request to confirm the established keys and optionally authenticate to the Responder.

In the finish step, the Requester sends a signature of the transcript using their private key and an HMAC of the transcript using the requester *finished_key*. Upon verifying the incoming request, the Responder sends the *FINISH_RSP* with an HMAC of the transcript, derives the session encryption keys from the handshake secret, enters the application data exchange phase. Symmetrically, the Requester verifies the HMAC, derives the session keys and enter the application data phase.

Note that the SPDM standard allows for an arbitrary number of parallel sessions which can be executed independently of one another. This means that while the parties are still negotiating the handshake secret in one session, they can update their keys of another session.

Mutual Authentication In the certificate key exchange, parties establish trust by verifying: 1) the validity of their partner's certificate, and b) a signature over the transcript and digests of the certificates. The Responder is always authenticated at the end of the handshake, however, to authenticate the Requester, the Responder needs to explicitly require it the response. To establish a mutually authenticated session, the *FINISH* request needs to include the Requester's signature, since the *RequestorVerifyData* serves only to verify transcript consistency and confirm the correct computation of the finished key.

2.4 SPDM: Key Exchange with Pre-shared Symmetric Keys

In the key exchange with pre-shared symmetric keys, the parties have been provisioned symmetric keys (PSKs) during the manufacturing process in a secure environment. The pair of Requester and Responder can share one or multiple keys, therefore they can setup multiple connections by using different PSKs per role direction. For simplicity, we will assume that the parties share a single key between them. To setup a session, they use the pre-shared key both as a mutual authentication credential and to derive the encryption keys for the application data phase.

Similarly, to the key exchange based on Diffie-Hellman, the Requester starts the handshake phase by sending a *PSK_EXCHANGE* request containing the protocol version, a 32-byte nonce and 2-byte of session identifier. Upon processing the request, the Responder retrieves the PSK of the Requester and calculates the handshake secret: $handshake\_secret = HKDF(Salt_0, PSK)$. Then, it derives from it the role-directed handshake secrets and finished keys as shown in Appendix A. Now, the Responder calculates the *ResponderVerifyData* using the finished key as the HMAC key and sends the *PSK_EXCHANGE_RSP* response along with 2-bytes of session id, and a 32-byte nonce.

The Requester computes the same keys as the Responder, composes the transcript of the session and verifies the *ResponderVerifyData*. Upon successful verification, the Requester calculates the *RequestorVerifyData* and sends the *PSK_FINISH* request to authenticate to the Responder and confirm the established secrets. The Responder verifies the HMAC over the transcript and implicitly authenticates the Requester since they proved knowledge of the shared PSK. Then, they send the *PSK_FINISH_RSP* response, compute the encryption secrets as shown in Appendix A, and proceed to the application data phase.

Lastly, the Requester confirms upon receiving the response that the partner has authenticated them and enters the application data phase with the computed keys. Note that in PSK mode, the finish step message exchange between the parties is encrypted using the role directed handshake secrets, namely the $handshake_{Req}$ for the request and $handshake_{Resp}$ for the response.

Mutual Authentication Parties are always mutually authenticated at the end of the *PSK_FINISH_RSP* in the PSK mode. All secrets in this mode are derived from the preshared key, therefore successful verification of the transcripts (*ResponderVerifyData*/*RequestorVerifyData*) and computation of same keys, implicitly proves the identity of the peer, as shown in the excerpt from the standard.

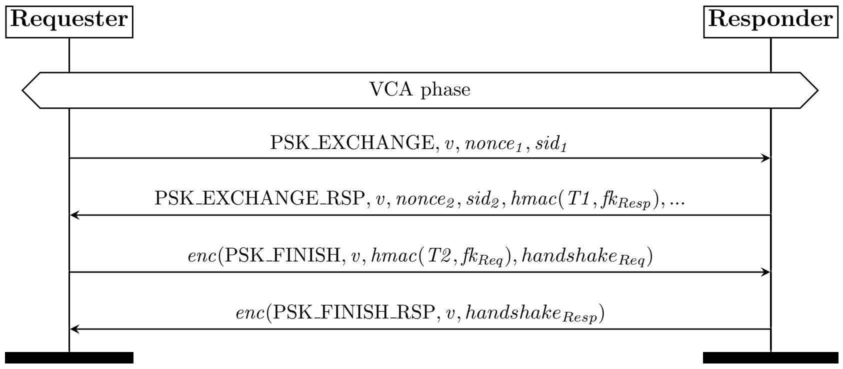

Requester
Responder
VCA phase
PSK_EXCHANGE, v, $nonce_1$, sid_1
PSK_EXCHANGE_RSP, v, $nonce_2$, sid_2, $hmac(T1, fk_{Resp})$, ...
enc(PSK_FINISH, v, $hmac(T2, fk_{Req})$, $handshake_{Req}$)
enc(PSK_FINISH_RSP, v, $handshake_{Resp}$)

Figure 2: Key Exchange with Mutual Authentication in Preshared Symmetric Keys mode.

Specification [21], Section 10.18, line 534:

```
This option requires the Requester and the Responder to have prior knowledge of a common PSK before the
handshake.  Essentially, the PSK serves as a mutual authentication credential and the base of the session
key establishment.  As such, only the two endpoints and potentially a trusted third party that provisions
the PSK to the two endpoints know the value of the PSK.
```

2.5 SPDM: Key Exchange with Pre-shared Public Keys

In the key exchange with pre-shared public keys, the parties have been equipped during the trusted manufacturing process with a pair of longterm private-public keys, as well as their partners' public keys. This allows them to trust the public keys during a protocol run without the need to request certificates.

The message sequence of this mode is similar to the one using certificates presented in Figure 1. The difference stands in the *KEY_EXCHANGE* where the Requester specifies in a parameter the usage of public keys, respectively for the Responder in the *KEY_EXCHANGE_RSP*. As a result, the parties verify the signatures over the transcript using the provisioned longterm keys instead of a certificate's keys.

3 Formal Model of SPDM 1.2 as a Whole

We develop the first full model of SPDM that encompasses all the main modes of its design and their interactions. In Section 3.1, we revisit prior work and existing models on SPDM's components. In Section 3.2, we expand on our approach to construct a full SPDM model and provide details on our modeling decisions in Section 3.3.

In total our formal analysis is constructed across two models: the composition of SPDM, which contains a mode switch attack we see in Section 4, and fixed version of the protocol we describe in Section 5. In the fixed model, we prove four main security guarantees:

1. Responder authentication
2. Measurement integrity
3. Mutual authentication in all modes
4. Secrecy of handshake key in all modes

We elaborate on the properties in Section 3.4 and give details on the threat model against which they hold in Section 3.5.

3.1 Prior models

We build on the simpler initial models developed by [12]. In their work, they divide the SPDM protocol into five main components: a) initialization of devices with their capabilities and VCA to bootstrap the protocol, b) options to allow for measurement exchange, responder authentication and retrieving the Responder's certificate, c) session handling to model the choice of the key exchange mode, d) the three key exchanges, and finally e) application data exchange and key update.

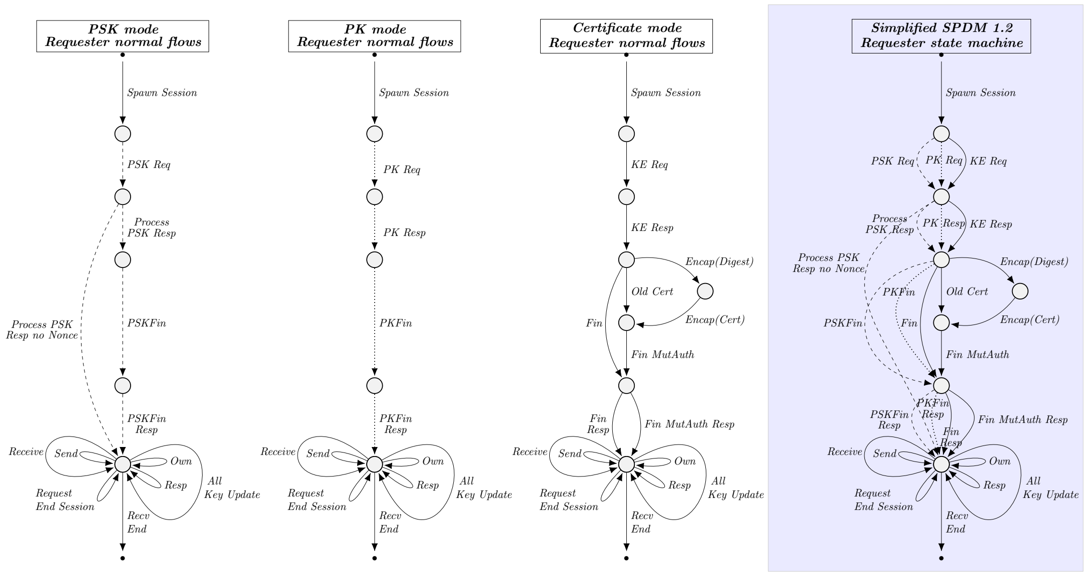


Figure 3: To illustrate a small part of our model, we provide simplified state machines for the Requester for the normal flows of the key exchange modes (the three on the left), and our model of SPDM as a whole (highlighted on the right) for a single session. To simplify the presentation, we omit several 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.

These components are then used across four individual models to analyze different security guarantees and phases of four sub-protocols in isolation. The four models are:

1. the *device attestation mode* including a) and b)
2. the *certificate-based mode* containing a), c), d), and e) where c) and d) are restricted to only allow the key exchange based on provisioned certificates (Cert).
3. the *pre-shared symmetric key mode* containing a), c), d), and e) where c) and d) are restricted to only allow the key exchange with pre-shared symmetric keys (PSK).
4. the *pre-shared public key mode* containing a), c), d), and e) where c) and d) are restricted to only allow the key exchange with pre-shared public keys (PK).

All these models were analyzed independently from each other and their security properties were proven to be secure using the Tamarin prover. For more details on the models and proven properties, we refer the reader to [12].

3.2 Full model of SPDM 1.2

Our model of SPDM 1.2 as a whole comprehensively captures all main interactions among sub-protocols in the Tamarin prover. Our model includes initialization with pre-shared symmetric keys, pre-shared public keys, and certificates, VCA negotiation phase, device attestation options, session handling layer, three key exchanges, and the application data phase. 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 3, we present the state machines for the Requester in each individual mode's intended flow, as well as a state machine modelling SPDMs 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 3, 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.

Next, we discuss the design choices we made to enable the analysis of such a complex model.

3.3 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. According to [12], the TAMARIN prover already faced challenges with the size of their initial models, which are much smaller than ours. 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 original models being non-transferable and not reusable for proving similar properties in our 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.
3. The unbounded number of devices, sessions, application messages exchanged, multiple keys and transcripts being computed similarly 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 [12], 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. [12] 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 composed 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 [12], 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 prior work, 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.

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: a) *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. b) *message ordering lemmas* that help the prover when reasoning about the order of events, and c) *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 d) *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. In Appendix E we provide more detail about the helper lemmas that we used to prove mutual authentication.

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 specification, we observed that the transcripts of the key exchanges consistently differ between the three modes: a) the PSK mode includes the *psk exchange request code*, while the other two modes include the *key exchange request code*, and b) 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, we conclude that in practice, there is domain separation for the different keys. Since these differences occur deep within the transcript, we aided Tamarin to find this separation earlier during its proof search by tagging the keys.

3.4 Security Properties

In this section, we recap the four main security properties of SPDM initially modelled by [12] and elaborate on the modifications we have made in order to prove them in our model of SPDM as a whole.

Responder Authentication *Unilateral responder authentication* is achieved during the device attestation and in the key exchange with certificates of SPDM. The Requester verifies that they are running the protocol with a specific Responder by challenging them with a fresh nonce that the Responder needs to sign using the private key of their certificate. Additionally, the parties can agree on other data that they have exchange in the protocol such as the transcript.

To model the property in TAMARIN, we adjust the definition of the device attestation proposed by [12] syntactically, matching it to the new model of SPDM as a whole.

Measurement Integrity One of the major security goals stated by SPDM is to ensure the integrity of device measurements. This means that anytime the Requester requests measurements of a device in the Responder role, the data received should indeed be the same as sent by this specific Responder.

SPDM's standard describes two different ways to obtain measurements during the device attestation which depend on the used key material. Measurement requests are defined using either certificates or pre-shared public keys. For instance, we define measurement integrity for pre-shared public keys as follows.

Definition 1. Measurement Integrity PK

$$\begin{aligned}
&\forall\ tid\ id_{Req}\ id_{Rsp}\ pk_{Rsp}\ data\ \sharp i\ .\\
&\mathsf{ReceiveMeasurementPK}(tid, id_{Req}, id_{Rsp}, pk_{Rsp}, data\ )@\sharp i\\
&\Rightarrow\ (\exists\ id\ sk_{Rsp}\ \sharp t\ .\ \sharp t < \sharp i\ \wedge\ (pk_{Rsp} = pk(sk_{Rsp}))\ \wedge\\
&\mathsf{SendMeasurement}(tid, id, id_{Rsp}, sk_{Rsp}, data\ )\ @\sharp t\ )
\end{aligned}$$

Notably, proving measurement integrity with TAMARIN requires the most amount of helper lemmas out of all properties. This is potentially caused by the vast amount of looping behaviors during the device attestation in addition to the introduced complexity of all the different key exchanges and the way we spawn sessions in the composed models.

Mutual Authentication We prove mutual authentication of the parties at the end of the key exchange across all three modes of the protocol. This property expresses that once a Requester has committed to a specific partner Responder and an agreed transcript by the end of one of the modes, then that Responder should be running the protocol with the same transcript on the same mode. The property should simultaneously also hold for the reversed roles. Mutual authentication for PSK mode is defined as follows:

Definition 2. Mutual Authentication PSK

$$\begin{aligned}
&\forall\ \mathit{sid_1}\ \mathit{tid_1}\ \mathit{secret}\ \mathit{TH2}\ \mathit{role_1}\ \sharp i\ .\\
&\mathsf{CommitMutAuthPSK}(\mathit{sid_1}, \mathit{tid_1}, \mathit{secret}, \mathit{TH2}, \mathit{role_1}\ )@\sharp i\\
&\Rightarrow\ (\exists\ \mathit{sid_2}\ \mathit{tid_2}\ \mathit{role_2}\ \sharp t\ .\ \sharp t < \sharp i\ \wedge\ \neg(\mathit{role_1} = \mathit{role_2})\ \wedge\\
&\mathsf{RunningMutAuthPSK}(\mathit{sid_2}, \mathit{tid_2}, \mathit{secret}, \mathit{TH2}, \mathit{role_2}\ )\ @\sharp t\ )
\end{aligned}$$

The property is similarly defined for the other modes, where in PK mode the parties agree on each others public keys, while in certificate mode the public keys need to be honestly generated. We defined different action facts for each of the modes to express that the property should hold uniquely for each of them.

Handshake Secrecy We prove that at the end of the key exchange, the established handshake secret is not known by the attacker across all different modes of the protocol. This property is defined for both the Requester and Responder side, and for each of the three modes. In the following, we show handshake secrecy for the Requester in PSK mode:

Definition 3. Handshake Secrecy PSK

$$\begin{aligned}
&\forall\ \mathit{sid}\ \mathit{tid}\ \mathit{id_{Req}}\ \mathit{id_{Rsp}}\ \mathit{secret}\ \sharp i\ .\\
&\mathsf{SesssionMajorSecretReqPSK}(\mathit{sid}, \mathit{tid}, \mathit{id_{Req}}, \mathit{id_{Rsp}}, \mathit{secret})@\sharp i\\
&\Rightarrow\ \neg(\exists\ \sharp t\ .\ \mathsf{K}(\mathit{secret})@\sharp t)
\end{aligned}$$

The property is defined similarly for the PK and certificate modes. However, in the certificate mode the property also states that the certificates of the parties involved in the handshake are honestly generated. This is due to the threat model of maliciously generated certificates, where the attacker can trivially break the property and learn the secret as one of the parties executing the key exchange.

3.5 Threat Model

In our analysis, the attacker has full control over the network where they can inject, modify, or drop messages. Additionally, the attacker may register malicious certificates for honest devices. I.e., they may exploit the Certificate Authority to sign victim devices with an attacker-provided private key.

4 Breaking Authentication

During our analysis of our SPDM model, Tamarin discovered an attack on the mutual authentication property of the key exchange when using pre-shared symmetric keys. At a high level, this attack involves a network attacker without any knowledge of keys or malicious certificates, that can establish a secure session with a Responder by manipulating the modes of the protocol. This causes the Responder to believe they have mutually authenticated an honest partner and set up a secure session, when in reality all session keys are known by the attacker.

In the remainder of this section, we first show the attack trace found by the Tamarin prover, then illustrate the attack steps in the SPDM standard and how to exploit its reference implementation.

4.1 Automatically Discovered Attack

During our attempt to prove mutual authentication in our full model for the PSK key exchange mode in Tamarin's interactive mode, the tool produced a counterexample in one of its search branches. In particular, Tamarin produces a counterexample in which the Responder finishes a mutually authenticated handshake without the Requester being present. In this trace, the network attacker starts a key exchange using certificates before switching to finish the handshake in the PSK mode. Surprisingly, the Responder accepts both requests of the network attacker as valid transitions and proceeds to compute the secure session keys thinking they were derived from the pre-shared symmetric key with the Requester. However, in reality, all keys are based on the Diffie-Hellman output computed with the network attacker in the certificate key exchange mode.

We define this type of attack as the *mode switch attack*, where the attacker changes the executed mode of the protocol, in this case from certificate to PSK mode at the Responder side. The mode switch is a specific form of *cross-protocol attack*, which allows the attacker to break security by misusing the

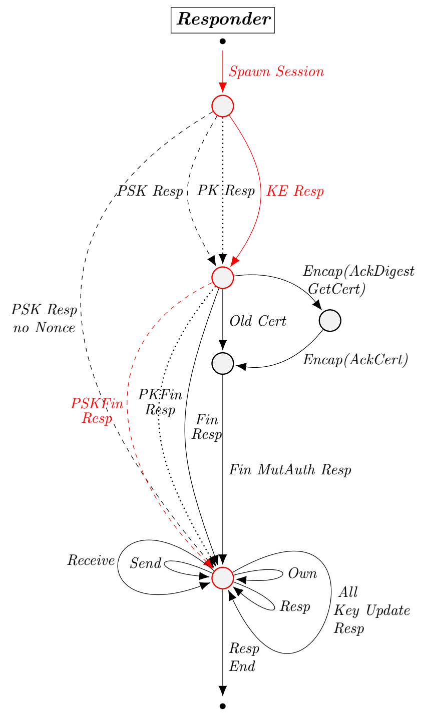


Figure 4: Our mode switch attack visualized on the state machine of the Responder. Dashed lines represents the normal transitions of the preshared symmetric key (PSK) mode, dotted lines the preshared public key (PK), and solid lines the certificate mode and shared transitions. The mode switch attack, highlighted in red, makes the Responder end in a mutually authenticated session with the attacker, using transitions from both the certificate mode and PSK mode.

execution of two two different key-exchange modes. This attack can only show up if we analyze the system as a whole by including all sub-protocols and their interactions, which is why it could not be discovered in previous work [12]. In Figure 4, we illustrate the attack transitions highlighted in red in the Responder's state machine.

4.2 Mode Switch Attack

In this section, we will look into the details of the mode-switch attack found by TAMARIN and explain how to execute this attack based on the SPDM 1.2.1 standard [21].

1) Setup in Certificate Mode

To bootstrap the protocol, the attacker and the Responder successfully complete the VCA phase (and optionally any parts of the device attestation). Then, the attacker sends a *KEY_EXCHANGE* request, indicating its intention to begin a session with the Responder in certificate mode. Upon receiving the request, the Responder performs the necessary computations to derive and store the Diffie-Hellman output, handshake secrets, and finished keys for the key exchange. Subsequently, the Responder computes the signature and HMAC of the transcript, and sends them in the *KEY_EXCHANGE_RSP* response.

Upon receiving the response, the attacker processes it as per the usual protocol operations. It computes the Diffie-Hellman output and derives the same secrets as the Responder. Up to this point, the attacker has strictly adhered to the protocol, and both parties are in the certificate mode.

2) Attacker switches to PSK Mode

At this point, the attacker decides to finish the protocol by sending the *PSK_FINISH* request of the PSK mode, instead of the *FINISH* request of the current certificate mode. To do this, the attacker performs the following steps:

1. *Compute transcript T2:* Concatenation of the transcript according to the standard, i.e., the message exchanged in the VCA phase (`VCA`) and the requests and responses of the key exchange (see

Appendix A for details). However, since the *PSK_EXCHANGE* and *PSK_EXCHANGE_RSP* are never issued, the transcript is degraded to the concatenation of `VCA`, `NULL`, `NULL`, *PSK_FINISH*.

2. *Compute the RequestorVerifyData:* Compute the HMAC of transcript T2 using the stored requester finished key fk_{Req}. Note, that finished keys are already calculated from the Diffie-Hellman output after receiving the key exchange response.
3. *Construct the request:* Prepend the protocol version and request code of *PSK_FINISH* to the HMAC of step 2).
4. *Encrypt the request:* Encrypt the request of step 3) using the requester directed handshake secret $handshake_{Req}$, and send the encrypted request to the Responder. Note again, that role directed handshake secrets are also calculated and stored after the key exchange response.

3) Responder accepts Mode Switch

Upon receiving the encrypted request, the Responder executes the following:

1. *Decrypt the request:* Retrieve the stored requester handshake secret $handshake_{Req}$ and decrypt the message. Since, the key computed locally by the Responder is derived in the same way as the Requester, the decryption is successful.
2. *Check correctness of request*: The request has a valid format and is issued correctly inside of a session.
3. *Calculate transcript T2:* Concatenate the transcript of the VCA phase and the executed requests and responses of the session, namely: `VCA`, `NULL`, `NULL`, *PSK_FINISH*.
4. *Verify the RequestorVerifyData:* Locally compute the HMAC of T2 using the stored requester finished key fk_{Req}. Since both, transcripts and keys, are the same, the HMAC passes the verification check.

Following the described steps, the Responder accepts the request and believes they are at the end of the key exchange of the PSK mode. Therefore, they believe to have successfully authenticated the Requester (attacker), and compute the keys for the application data as normal.

To conclude the key exchange, the Responder composes and sends the response, and finally enters the application data phase. Symmetrically, the attacker accepts the response, derives the data secrets and enters the secure session.

To illustrate the attack, we provide a message sequence chart in Figure 7 in Appendix C .

Broken Mutual Authentication

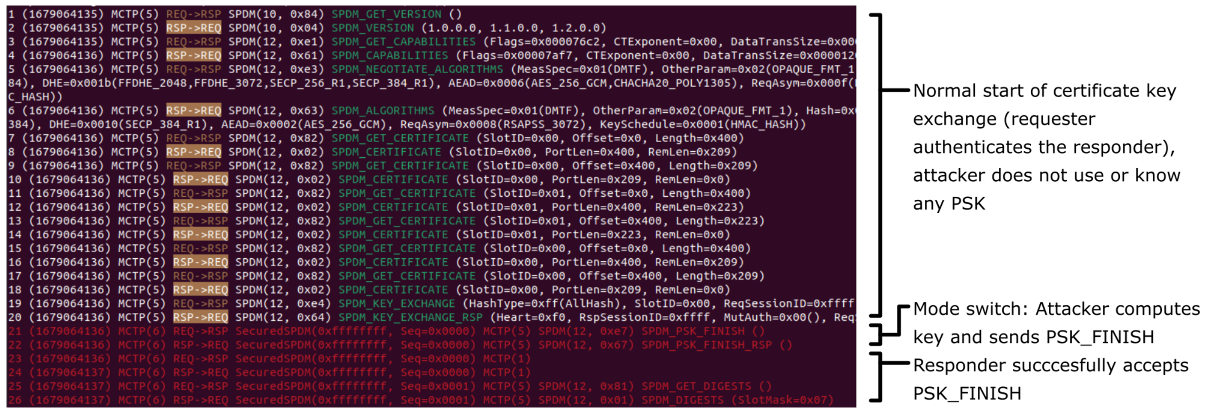
This attack on mutual authentication is made possible because a) the Responder has no knowledge of the protocol mode they are executing, and b) the keys used in PSK mode are already computed and stored from the previous step in the certificate mode.

It is crucial to note that during the initial part of the key exchange in the certificate mode, the handshake secrets and finished keys are all derived from the Diffie-Hellman output. However, after accepting the attacker's request in PSK mode, the Responder erroneously assumes that the computed finished keys and handshake secrets are derived from the PSK key. The computed finished keys act as an authentication credential in the PSK mode (see Section 2.4). In contrast, in the certificate mode, they are used to verify the integrity of the protocol transcript or detect the absence of a MITM attack. This discrepancy is a main cause that leads to the successful execution of the mode switch attack.

In summary, the attacker successfully circumvents authentication by exploiting the differences in authentication constructions between the two modes. By switching the context of the protocol, the attacker changes the meaning and usage of the computed secrets. As a result, this elevates a one-sided authenticated session to a mutually authenticated one, which ultimately breaks the protocol's security guarantee.

4.3 Attacking the Implementation

Upon investigating the standard, we verified that it indeed allows a network attacker to execute the mode switch attack described earlier. To further validate this vulnerability, we inspected the actual open-source reference implementation of SPDM, `libspdm` [23]. We performed experiments to demonstrate the attack on `libspdm` version 2.3.1 (released on January 10, 2023) [27], and the vulnerability was confirmed to be present in versions as far back as 1.0.0 (released on December 21, 2021). In the following, we outline the steps taken during our assessment to verify the existence of the attack in the reference implementation.



```
1 (1679064135) MCTP(5) REQ->RSP SPDM(10, 0x84) SPDM_GET_VERSION ()
2 (1679064135) MCTP(5) RSP->REQ SPDM(10, 0x04) SPDM_VERSION (1.0.0.0, 1.1.0.0, 1.2.0.0)
3 (1679064135) MCTP(5) REQ->RSP SPDM(12, 0xe1) SPDM_GET_CAPABILITIES (Flags=0x000076c2, CTExponent=0x00, DataTransSize=0x00
4 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x61) SPDM_CAPABILITIES (Flags=0x00007af7, CTExponent=0x00, DataTransSize=0x00012
5 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0xe3) SPDM_NEGOTIATE_ALGORITHMS (MeasSpec=0x01(DMTF), OtherParam=0x02(OPAQUE_FMT_1
84), DHE=0x001b(FFDHE_2048,FFDHE_3072,SECP_256_R1,SECP_384_R1), AEAD=0x0006(AES_256_GCM,CHACHA20_POLY1305), ReqAsym=0x000f(
C_HASH))
6 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x63) SPDM_ALGORITHMS (MeasSpec=0x01(DMTF), OtherParam=0x02(OPAQUE_FMT_1), Hash=0x
384), DHE=0x0010(SECP_384_R1), AEAD=0x0002(AES_256_GCM), ReqAsym=0x0008(RSAPSS_3072), KeySchedule=0x0001(HMAC_HASH))
7 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x00, Offset=0x0, Length=0x400)
8 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x00, PortLen=0x400, RemLen=0x209)
9 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x00, Offset=0x400, Length=0x209)
10 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x00, PortLen=0x209, RemLen=0x0)
11 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x01, Offset=0x0, Length=0x400)
12 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x01, PortLen=0x400, RemLen=0x223)
13 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x01, Offset=0x400, Length=0x223)
14 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x01, PortLen=0x223, RemLen=0x0)
15 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x00, Offset=0x0, Length=0x400)
16 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x00, PortLen=0x400, RemLen=0x209)
17 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0x82) SPDM_GET_CERTIFICATE (SlotID=0x00, Offset=0x400, Length=0x209)
18 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x02) SPDM_CERTIFICATE (SlotID=0x00, PortLen=0x209, RemLen=0x0)
19 (1679064136) MCTP(5) REQ->RSP SPDM(12, 0xe4) SPDM_KEY_EXCHANGE (HashType=0xff(AllHash), SlotID=0x00, ReqSessionID=0xffff
20 (1679064136) MCTP(5) RSP->REQ SPDM(12, 0x64) SPDM_KEY_EXCHANGE_RSP (Heart=0xf0, RspSessionID=0xffff, MutAuth=0x00(), Req
21 (1679064136) MCTP(6) REQ->RSP SecuredSPDM(0xffffffff, Seq=0x0000) MCTP(5) SPDM(12, 0xe7) SPDM_PSK_FINISH ()
22 (1679064136) MCTP(6) RSP->REQ SecuredSPDM(0xffffffff, Seq=0x0000) MCTP(5) SPDM(12, 0x67) SPDM_PSK_FINISH_RSP ()
23 (1679064136) MCTP(6) REQ->RSP SecuredSPDM(0xffffffff, Seq=0x0000) MCTP(1)
24 (1679064137) MCTP(6) RSP->REQ SecuredSPDM(0xffffffff, Seq=0x0000) MCTP(1)
25 (1679064137) MCTP(6) REQ->RSP SecuredSPDM(0xffffffff, Seq=0x0001) MCTP(5) SPDM(12, 0x81) SPDM_GET_DIGESTS ()
26 (1679064137) MCTP(6) RSP->REQ SecuredSPDM(0xffffffff, Seq=0x0001) MCTP(5) SPDM(12, 0x01) SPDM_DIGESTS (SlotMask=0x07)
```

Normal start of certificate key exchange (requester authenticates the responder), attacker does not use or know any PSK
Mode switch: Attacker computes key and sends PSK_FINISH
Responder succcesfully accepts PSK_FINISH

Figure 5: View of our attack from the SPDM Emulator. We manually gave the last few lines a red color to indicate the exchanges after the mode switch. The attacker starts the handshake in key exchange with certificates and then switches to preshared symmetric keys (PSK). The victim (the Responder) successfully accepts *PSK_FINISH* and completes the mutually authenticated PSK mode, despite the fact that the attacker (as Requester) does not know any pre-shared key. In the end, the attacker learns the session keys for the mutually authenticated session with the Responder, and can now continue communicating within the session as if they were fully authenticated.

Experimental Setup

We use the SPDM emulator open source library, `spdm-emu` [26] to emulate both endpoints, a Requester and a Responder. We use `spdm-dump` [25] to format and interpret the messages in the SPDM communication. We implement the attacker by re-using code from the Requester's library. More specifically, we modify the main communication file `libspdm_req_communication.c` by swapping the function call of the *FINISH* request in the certificate mode with the *PSK_FINISH* request from the pre-shared key mode. The Responder's library remains unmodified, since it is an honest party that strictly follows the protocol.

To execute the attack, we run the emulators of the attacker and Responder with the required capabilities. These capabilities entail a) performing a key exchange, b) support of certificates and pre-shared keys, as well as c) other helper flags for normal protocol execution, like the capability of encryption. At the end of the protocol run, we inspect the collected logs and confirm the successful completion of the protocol where both parties use the application data secrets to send encrypted requests inside the secure session. We show in Figure 5 the execution trace between the attacker and the Responder from the logs of the SPDM emulator.

5 Restoring Authentication

In this section, we outline our recommendation for addressing the mode switch vulnerability in both the SPDM standard and its reference implementation. Furthermore, we describe the modeling of our suggested solution in Tamarin. We present and summarize the properties proven using Tamarin and provide an overview of our analysis results.

5.1 Fixing the Protocol

As stated before, the mode switch attack is made possible due to the lack of proper separation of the protocol state among the different modes of key exchanges, and the lack of an explicit check on the allowed transitions in each mode. This implies that mitigating solutions can be recommended depending on either of these issues.

A naive solution to address this issue is to rename and store the finished keys and handshake secrets separately for different modes in distinct variables of the state. Consequently, the Responder would have no finished keys in the PSK mode if these keys were initially calculated in the certificate mode. Assuming that the protocol checks for keys not being null before their usage, the attack would no longer be possible. However, a more complete solution would require that the entire state of the protocol is completely separated for each of the different modes.

On the other hand, the protocol could explicitly specify the allowed transitions, and check that the message sequences within the same session belong to a consistent mode, i.e., checking the mode throughout the handshake.

Proposed Solution

We recommend that the standard explicitly states the allowed handshake transitions for each of the modes, as well as the entry conditions that can trigger those request, e.g., if the Requester and Responder start the key exchange with certificates, they should also end in the same mode. This means that the protocol should store the exact key exchange mode and check each transition within that session to belong to the correct mode.

While studying the code of the official SPDM reference implementation [23], we noticed a boolean flag, *use_psk*, which was mainly used by the Requester to decide which mode of the key exchange to trigger. Following this, we recommend a minimal change to implement in `libspdm`, where the Responder also uses this flag to implement the restrictions we recommended on the standard. We define the checks of the Responder upon receiving a finish request:

1. *PSK_FINISH* and *use_psk* (Allowed)
2. *PSK_FINISH* and $\neg$(*use_psk*) (Disallow)
3. *FINISH* and *use_psk* (Disallow)
4. *FINISH* and $\neg$(*use_psk*) (Allowed)

While the fixed version of the protocol implements the minimal change, it would be prudent engineering practice to implement that the protocol stores and checks the exact mode, instead of a single boolean value. This way, also potential cross-protocol attacks or interleaving between the certificate and PK mode are also considered.

5.2 Formally Modeling the SPDM Fix

To gain trust in the concrete fix for the protocol, we integrate the proposed solution into our model of SPDM.

As in the minimal implemented fix on `libspdm`, we introduce an additional variable called *use_psk* to the state of the parties. At the beginning of the handshake, both Requester and Responder will set the value of this parameter based on the mode they started executing. Specifically, they will set it to `TRUE` after successfully executing *PSK_EXCHANGE* and *PSK_EXCHANGE_RSP*, and `FALSE` after completing *KEY_EXCHANGE* and *KEY_EXCHANGE_RSP*.

We then impose restrictions on the rules of the parties to check the value of the *use_psk* parameter before issuing or accepting a finish request. By doing so, we ensure that the correct mode is maintained throughout the handshake.

To achieve this, we add an Eq action fact that checks that *use_psk* is equal to `TRUE` for the PSK mode and `FALSE` for certificate and pre-shared public keys modes. This corresponds to $\mathsf{Eq}(\mathit{use\_psk}, \mathtt{TRUE})$ and $\mathsf{Eq}(\mathit{use\_psk}, \mathtt{FALSE})$, respectively. Then, the Eq action fact is restricted by the standard equality restriction in Tamarin:

$$\begin{array}{l}\mathsf{restriction}\ \mathit{equality}:\\ \forall\ x\ y\ \sharp i\ .\ \mathsf{Eq}(x\ ,y\ )@\sharp i\ \Rightarrow\ (x\ =\ y)\end{array}$$

5.3 Analysis Summary

By applying the proposed fix and modeling it in Tamarin, we successfully verified all main properties related to our model of SPDM as a whole, as outlined in Section 3.4. Our analysis implies the absence of the previously discovered mode switch attack. We verified 13 lemmas to show the main security properties, 14 helper lemmas, and 15 sanity traces to verify the correctness of our model.

The verification process was performed on a computing cluster with an Intel(R) Xeon(R) CPU E5-4650L 2.60GHz processor, 756GB of RAM, and 4 threads per Tamarin call. The results were achieved using the Tamarin prover version 1.7.1 on the develop branch[2]. It is important to note that due to the substantial size of our Tamarin SPDM model, each lemma execution required between 3 to 20 minutes.

[2]Git commit `3e882554d671eebc6288d0e11ed6e89ce0f87b26` in the Tamarin prover repository.

We show in Figure 6 the proving time of the main security properties, while 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 [13].

Property	#Helper Lemma	Fix	Runtime (s)
Responder Authentication	-	✓	229
Measurement Integrity Cert	9	✓	251
Measurement Integrity PK	7	✓	232
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

Figure 6: Formal analysis results of the SPDM TAMARIN models. The properties and their helper lemmas are all proven automatically.

Effort

Modeling and analyzing SPDM 1.2 as a whole took approximately 3-4 months of person work, starting from the models in [12]. Throughout the process, we encountered challenges and iteratively made decisions on various aspects of the models. After the discovery of the reported mode switch attack, we revised the protocol model to include the fix, which in turn lead to additional effort to re-establish proofs and helper lemmas. Our final model consists of roughly 3800 lines of TAMARIN code, including 71 TAMARIN rules, 11 restrictions and 42 analyzed lemmas, which is more than double the size of the largest model in [12].

6 Discussion

We identify two main lessons we learned from the SPDM analysis that we can directly apply to other formal analysis studies, and discuss the remaining pitfalls in SPDM's design.

6.1 Modeling Protocol Roles as Processes versus Modeling Transitions

In our model of SPDM as a whole, we used TAMARIN's rewrite rules to accurately model all state-machine transitions as described in the standard. If we had instead used a process-based specification language, e.g., as in ProVerif or SAPIC[+], it would have been natural to model the responder's PSK code as a sequential process that first processes *PSK Req*, and then *PSK Finish*; and similarly for the certificate mode. The resulting model would have not detected our attack, as the only way to process *PSK Finish* would be directly after *PSK Req*. In this sense, our attack showcases how very subtle modeling choices directly impact which attacks are covered.

Our findings also suggest to revisit previous work on formally verifying other large case studies such as those for EMV, Bluetooth, and others [8, 19, 40, 14, 5, 6, 4] and check whether the models implicitly abstract away possible transitions in the state machines of the protocol. As a general take-away, ideally models should capture the real intent of the parties and their decisions on the next protocol steps, instead of condensed processes to simplify modeling and proof search.

6.2 Domain separation as a practice

Domain separation of cryptographic keys is the practice of ensuring that keys derived for different purposes use distinct inputs (e.g., by using tags) in their key derivation functions. This often simplifies reasoning about the protocol's security, since it makes the probability of key-collisions across different key types negligible. This is considered standard good practice in modern security protocol engineering, and consistently used in modern protocols such as TLS 1.3. However, SPDM 1.2 does not use this approach.

Instead, SPDM relies on transcripts used for the key derivation that are distinct for all different modes of key exchange. Since these differences occur deep within the transcript, making it difficult for Tamarin to reason about them effectively, we lifted the tagging from the transcript to the key exchange in our models (see Section 3.3.)

While SPDM's approach achieves similar results to classical domain separation, its guarantees are harder to understand and analyze. For reliable security, we recommend that SPDM and future standards use domain separation for keys and – even more – include tagging in uses of cryptographic primitives to decrease complexity and simplify analysis.

6.3 Remaining SPDM pitfalls

We note that the design pitfalls highlighted by [12] remain unaddressed as of now. We claim that two of the unattended pitfalls are especially relevant. First, the vendor defined requests remain under-specified, and a weakly designed vendor request can break most of the desired security guarantees. Second, the authentication properties desired by the standard remain unclear in terms of what actually should be authenticated.

Provisioning Protocol Secrets

Initial provisioning of protocol secrets is left to the vendor, but in addition the SPDM 1.2 standard still offers means to provision certificates even after the initial setup. While [12] already pointed out that the restrictions on provisioning certificates are too lax in the current standard, DMTF introduced even more options to supply devices with cryptographic secrets within a protocol run in SPDM version 1.3. In this version they allow to set an indefinite amount of pre-shared public keys. We want to stress that cryptographic secrets should only be set and replaced within well-defined protocol procedures that allow for thorough analysis.

Certificates Accessible across SPDM Connections

During our talk with the developers, the point was raised that the parties potentially maintain the received certificates across different SPDM connections in the same storage place to reduce the memory usage. This can be the case especially for devices with limited memory, where multiple connections with the same partner would otherwise require storing duplicates of the same certificates. Notably, in the current SPDM design the Requester can decide the certificate with whom they authenticate themselves to the Responder. Since the certificates only optionally contain the device identifier, one can impersonate other devices by starting a protocol pretending to have one OID but authenticating with their own certificate stored by the Responder in another protocol execution. Another potential problem, is that both the pre-shared public keys and certificates are identified by the same identifier and having access to other certificates outside of the protocol run can lead to variations of the mode switch attack between these two modes.

Lastly, these certificates can also be maintained even after the protocol reset, and the only time the certificate validity is checked, is when it is stored. This means that SPDM needs to develop policies on deleting old and expired certificates or means to actively check them before verifying signatures.

Advanced Security Guarantees

As SPDM is supposed to be a general solution to low-level trusted communication, the discussion of the desired security guarantees is rather sparse. We think there should be an open discussion on the desired security properties.

7 Responsible Disclosure

After TAMARIN found the attack on our model of the SPDM standard versions 1.1 - 1.2.1 [21], we confirmed that this attack was indeed possible on the corresponding reference implementation versions 1.0.0 - 2.3.1 [23] and created a proof-of-concept script (using the libspdm emulator scripts [26]) that implements the attack: an attacker that knows no keying material can get any Responder to accept a *PSK_FINISH_RSP* message. We wrote two brief reports, one for the standard and one for the reference implementation, including our proof-of-concept code. In both of our reports, we recommended that the particular transition should be disallowed, i.e., *PSK_EXCHANGE_RSP* should only be accepted if the previous step was the start of the PSK mode. DMTF confirmed that this was a critical vulnerability on the reference implementation and incorporated fixes into the new version of SPDM's specification [22] to check the correct protocol mode before accepting a finish request (in particular, Section 10.17, line 595 and Section 10.19, line 645). For the reference implementation, CVE [17] was assigned with CVSS severity score of 9.0 (Critical) and directly attributed to our findings. Approximately two months after the report was sent, DMTF released a patch, and ended the embargo period. Following this, we presented more details of our work to several of the DMTF SPDM working group members. We actively investigated alternative implementations of libspdm, but out of those accessible to us, only two supported the PSK mode: the reference implementation [23] and a libspdm implementation in RUST [24]. Both of these were vulnerable to our attack before the patch.

8 Conclusions

In this work, we performed the first *full* formal analysis of SPDM 1.2 using the TAMARIN prover, which finds a critical attack in the standard, that also worked practically on the official reference implementation [17]. Our work directly caused DMTF to modify the new version of the standard, DMTF's reference implementation, and the Rust implementation, by incorporating our fix. We formally proved that the fixed standard is no longer vulnerable to the cross-protocol attack, and we provide all our models at [13].

Note that TAMARIN could only find the attack because (i) we modeled SPDM as a whole, considering the interaction between its sub-protocols, and (ii) we accurately modeled the state machine transitions as described in the standard, instead of using a process-based specification. As we pointed out in Section 6.1, the latter may be a reason to revisit existing formal models of large protocol standards, which may implicitly make assumptions on state machines that are not guaranteed by the actual standard, thereby potentially missing attacks.

More generally, our work shows again that the interaction between seemingly secure protocol components can introduce critical attacks, and analysis of the larger systems as a whole is required to obtain higher assurance and more reliable guarantees.

References

[1] Martin R. Albrecht, Sofía Celi, Benjamin Dowling, and Daniel Jones. Practically-exploitable cryptographic vulnerabilities in Matrix. *IACR Cryptol. ePrint Arch.*, page 485, 2023.

[2] 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.

[3] Renan CA Alves, Bruno C Albertini, and Marcos A Simplicio Jr. Benchmarking the Security Protocol and Data Model (SPDM) for component authentication. *arXiv preprint arXiv:2307.06456*, 2023.

[4] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2018.

[5] 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)*, pages 179–194, 2021.

[6] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, fix, verify. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1766–1781. IEEE, 2021.

[7] 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.

[8] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 483–502, 2017.

[9] 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.

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Symposium on Foundations of Computer Science*. IEEE, 2001.

[11] Stefan Ciobâca and Véronique Cortier. Protocol composition for arbitrary primitives. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 322–336. IEEE, 2010.

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

[13] Cas Cremers, Alexander Dax, and Aurora Naska. SPDM Composition Models. `https://github.com/ComprehensiveSPDM/TamarinSPDMAnalysis`, December 2024.

[14] 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.

[15] 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 *Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[16] 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 *USENIX Security Symposium (USENIX Security)*, 2020.

[17] CVE. CVE of mode switch attack on the SPDM reference implementaton. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-31127`, May 2023. accessed: 2023-06-20.

[18] 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.

[19] 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*, pages 113–129. Springer, 2012.

[20] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium (USENIX Security)*, 2015.

[21] DMTF. DSP0274: 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`, Jun 2022. accessed: 2025-01-06.

[22] 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`, May 2023. accessed: 2025-01-06.

[23] DMTF. Open source code of libspdm. `https://github.com/DMTF/libspdm`, March 2023. accessed: 2025-01-06.

[24] DMTF. Open source code of libspdm in Rust. `https://github.com/jyao1/rust-spdm`, March 2023. accessed: 2025-01-06.

[25] DMTF. Open source code of spdm-dump, version 2.3.1. `https://github.com/DMTF/spdm-dump/tree/2.3.1`, March 2023. accessed: 2025-01-06.

[26] DMTF. Open source code of spdm-emu, version 2.3.1. `https://github.com/DMTF/spdm-emu/tree/2.3.1`, March 2023. accessed: 2025-01-06.

[27] DMTF. Vulnerable open source code of libspdm v2.3.1. `https://github.com/DMTF/libspdm/releases/tag/2.3.1`, January 2023. accessed: 2025-01-06.

[28] 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`, 2024. accessed: 2025-01-06.

[29] Sébastien Gondron and Sebastian Mödersheim. Vertical Composition and Sound Payload Abstraction for Stateful Protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021.

[30] Thomas Groß and Sebastian Modersheim. Vertical protocol composition. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2011.

[31] Joshua D Guttman. Cryptographic protocol composition via the authentication tests. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2009.

[32] Andreas V. Hess, Sebastian Alexander Mödersheim, and Achim D. Brucker. Stateful protocol composition. In *ESORICS (1)*, volume 11098 of *Lecture Notes in Computer Science*, pages 427–446. Springer, 2018.

[33] Felix Linker, Ralf Sasse, and David A. Basin. A formal analysis of apple's imessage PQ3 protocol. *IACR Cryptol. ePrint Arch.*, page 1395, 2024.

[34] 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 (CAV)*, 2013.

[35] 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.

[36] Jules van Thoor, Joeri de Ruiter, and Erik Poll. Learning state machines of TLS 1.3 implementations. *Bachelor thesis. Radboud University*, 2018.

[37] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

[38] Jiewen Yao, Anas Hlayhel, and Krystian Matusiewicz. Post Quantum KEM authentication in SPDM for secure session establishment. *Design & Test*, 2023.

[39] Jiewen Yao, Krystian Matusiewicz, and Vincent Zimmer. Post Quantum Design in SPDM for Device Authentication and Key Establishment. *Cryptography*, 6(4):48, 2022.

[40] Jingjing Zhang, Lin Yang, Weipeng Cao, and Qiang Wang. Formal analysis of 5G EAP-TLS authentication protocol using proverif. *IEEE access*, 8:23674–23688, 2020.

A Calculating Transcripts and Secrets

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 certificates 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 in a similar way. 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)**

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).

Handshake Secrets The handshake secret is computed from the initial shared secret between the parties key_{init} and a zero-filled array ($Salt_0$): $handshake = \text{HMAC}(Salt_0, key_{init})$. 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 $handshake_{Req} = \text{HKDF}(handshake, \text{"req"}, TH1, \cdots)$. The role oriented secrets serve to compute the HMAC keys, so-called finished keys *fk*, of the *ResponderVerifyData* and *RequestorVerifyData*: $fk_{Req} = \text{HKDF}(handshake_{Req}, \text{"finished"}, \cdots)$.

Application Data secrets The session's secrets are computed from the handshake secret and the final transcript of the key exchange by the parties first deriving a master secret from the handshake secret: $master = \text{HMAC}(\text{HKDF}(handshake, \text{"derive"}, \cdots), \cdots)$. Then, they compute role directed major secrets, $major_{Req} = \text{HKDF}(master, TH2, \cdots)$. In the end, to derive the encryption keys the parties compute: $key_{Req} = \text{HKDF}(major_{Req}, \text{"key"}, \cdots)$.

To update the keys of a session, they update the role directed major secrets as $newmajor_{Req} = \text{HKDF}(oldmajor_{Req}, \text{"traffic"}, \cdots)$, and derive the encryption keys as before.

B Targeted analysis of the attack scenario

Tamarin found our mode-switch attack automatically in a specific branch of its proof search for mutual authentication. However, due to the complexity of other branches in its search for mutual authentication, it does not find the attack in reasonable time (4 hours) when analyzing this property in its automatic mode, as it gets stuck in other branches. Once Tamarin had found the attack, we wrote a dedicated exists trace lemma that more narrowly guides Tamarin to this branch, enabling it to efficiently and fully automatically reconstruct the counterexample, which we describe in details next. That is, the following lemma serves as a targeted analysis of the part of the search space where TAMARIN found the attack. If TAMARIN verifies the below lemma, it means the mode switch attack is possible, and TAMARIN will automatically produce the attack trace.

Definition 4 (Targeted lemma for our mode switch attack)**.**

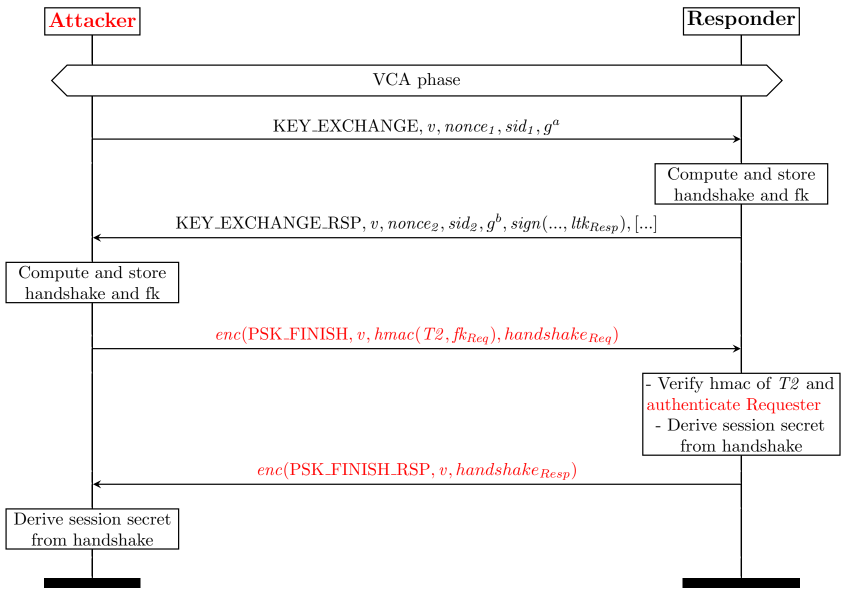
$$
\begin{aligned}
&\exists\ \mathit{sid2}\ \mathit{tid2}\ \mathit{oid1}\ \mathit{oid2}\ \mathit{secret}\ \sharp j1\ \sharp j2\ .\\
&\wedge\ \mathsf{RespKeyExchangeCert}(\mathit{sid2}\ \mathit{tid2}, \mathit{oid1}, \mathit{oid2}, \mathit{secret})@\sharp j1\\
&\wedge\ \mathsf{RespAcceptPSKFinish}(\mathit{sid2}\ \mathit{tid2}, \mathit{oid1}, \mathit{oid2}, \mathit{secret})@\sharp j2\\
&\wedge\ \neg(\exists\ \mathit{someoid}\ \mathit{ltk}\ \mathit{pk}\ \sharp t\ .\ \mathsf{Attacker}(\mathit{someoid}, \mathit{ltk}, \mathit{pk})@\sharp t)\\
&\wedge\ \neg(\exists\ \mathit{tid1}\ \sharp k\ .\ \mathsf{ReqStartProt}(\mathit{tid1}, \mathit{oid1}, \mathit{oid2})@\sharp k)
\end{aligned}
$$

In the above lemma, a Responder with *oid2* in session *sid2* establishes the handshake secret *secret* with a partner identifier *oid1* by first responding to the key exchange in certificate mode, RespKeyExchangeCert, and then accepting a finish request in PSK mode, RespAcceptPSKFinish. In this protocol execution, we clarify that the attacker has not compromised any parties, and that there does not exists any Requester with identifier *oid1* that has started the protocol, ReqStartProt.

Lemmas of the above type are useful during analysis to analyze particular sub-cases. Notably, they help to quickly reconstruct attacks or check if a fix works. After a fix seems to work in the narrow scenario, the modeler can proceed to attempt the desired full property without any limitations.

C Mode Switch MSC

Figure 7 contains shows the attack described in Section 4 as a message sequence chart.



Attacker
Responder
VCA phase
KEY_EXCHANGE, v, $nonce_1$, sid_1, g^a
Compute and store handshake and fk
KEY_EXCHANGE_RSP, v, $nonce_2$, sid_2, g^b, $sign(..., ltk_{Resp})$, [...]
Compute and store handshake and fk
enc(PSK_FINISH, v, $hmac(T2, fk_{Req})$, $handshake_{Req}$)
- Verify hmac of $T2$ and authenticate Requester
- Derive session secret from handshake
enc(PSK_FINISH_RSP, v, $handshake_{Resp}$)
Derive session secret from handshake

Figure 7: Message sequence diagram of the discovered mode switch attack on any responder. The attacker and Responder generate fresh DH keys, respectively (a, g^a) and (b, g^b), and compute the handshake and finished keys from the DH output, g^{ab}. The attacker then changes the mode of the key exchange, thus making the Responder believe the keys was derived from the pre-shared symmetric key and elevating the connection to mutually-authenticated. Messages in red highlight where the attack deviates from normal behavior.

D Model statistics

Table 1 highlights a significant increase in the number of rules and proof steps in the new SPDM model compared to its predecessors. Notably, our SPDM model incorporates 71 rules producing 371 unique

	LoC	#Rules	#Sources	#Lemmas source	secrecy	attestation	authentication	sanity
refl_attack	804	22	85	1	2	/	1	9
preshared_psk	1109	33	101	1	2	/	2	9
preshared_pk	1412	32	106	1	7	/	5	9
key_exchange	1798	41	129	1	2	/	6	14
device_attestation	981	29	91	1	/	9	/	12
Our SPDM model	3768	71	371*	1	6	12	15	15

Table 1: SPDM models from prior analysis [12] compared to our holistic model.
* = 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.

sources derived from from 28 cases. Sources are precomputations performed by Tamarin to enable its backwards search.

In addition to the increased model size compared to previous work, the complexity of proving each desired property also increased significantly. For instance, our SPDM 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 of our new analysis.

E Helper Lemmas for Mutual authentication

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 from Section 3.3 they belong. The lemmas for the Responder are similar.

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 d) *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 b) *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 c) *certificate and key origin lemmas.*

$$\begin{aligned}
&\forall\ tidI\ oidI\ oidR\ pkR\ digestR\ \sharp i\ .\\
&\mathsf{IStoredCert}(tidI, oidI, oidR, pkR, digestR\ )@\sharp i\\
&\wedge \neg(pkR = NULL)\\
&\Rightarrow\ \exists(someoid\ \sharp j\ .\mathsf{GenDeviceCert}(someoid, pkR\ )@\sharp j \wedge \sharp j < \sharp i\ )
\end{aligned}$$

F Changelog

- Version 1.0, December 18, 2024: Initial version.