# Formal Analysis of SPDM: Security Protocol and Data Model version 1.2

Full version v1.0, December 14, 2022

Cas Cremers
*CISPA Helmholtz Center for Information Security*
cremers@cispa.de

Alexander Dax
*CISPA Helmholtz Center for Information Security, Saarland University*
alexander.dax@cispa.de

Aurora Naska
*CISPA Helmholtz Center for Information Security, Saarland University*
aurora.naska@cispa.de

## Abstract

DMTF is a standards organization by major industry players in IT infrastructure including AMD, Alibaba, Broadcom, Cisco, Dell, Google, Huawei, IBM, Intel, Lenovo, and NVIDIA, which aims to enable interoperability, e.g., including cloud, virtualization, network, servers and storage. It is currently standardizing a security protocol called SPDM, which aims to secure communication over the wire and to enable device attestation, notably also explicitly catering for communicating hardware components.

The SPDM protocol inherits requirements and design ideas from IETF's TLS 1.3. However, its state machines and transcript handling are substantially different and more complex. While architecture, specification, and open-source libraries of the current versions of SPDM are publicly available, these include no significant security analysis of any kind.

In this work we develop the first formal model of the SPDM protocol, notably of the current version 1.2.1, and formally analyze its main security properties.

## 1 Introduction

The Distributed Management Task Force (DMTF) [18] is an industry standards organization for IT infrastructures (notably cloud, virtualization, network, servers and storage) whose board and members include AMD, Alibaba, Broadcom, Cisco, Dell, Google, Huawei, IBM, Intel, Lenovo, NVIDIA, Verizon, and VMware. One of its main goals is to provide common solutions that enable products in this domain to interoperate.

One of DMTF's most recent standards is the Security Protocol and Data Model (SPDM) security protocol for the broad device community, aiming to provide end-to-end trust for infrastructure including chip-to-chip, channels between components, and for various server platforms [39]. Many other standards groups have adopted SPDM, including Compute Express Link (CXL) [15], Peripheral Component Interconnect (PCI) [38], Mobile Industry Processor Interface (MIPI) [35], and the Trusted Computing Group (TCG) [41].

The current version, SPDM version 1.2.1 [19, 20], has two main goals: *securing communication over the wire*, and *device attestation*. With respect to securing communication over the wire, SPDM shares many high-level requirements with IETF's TLS 1.3 [40] and DTLS, including cipher suite and version negotiation, unilateral and mutual authentication based on certificates or preshared keys, and confidentiality based on key exchange and key update. In the context of SPDM, device attestation essentially amounts to support for a generic challenge-response mechanism, which can then be instantiated by specific stakeholders as required.

In its specification, SPDM is typically presented using an abstracted message sequence chart that suggests that it is essentially a single flow with six back-and-forth rounds, some of which are optional. However, the actual state machines defined by the standard are much more complex, and allow various types of loops, negotiations, optional flows, session resets, and delayed authentication processes. Furthermore, it uses message transcripts in a non-standard way, by using several different filtered transcripts. As a consequence, the complexity of its state machines exceeds even those of TLS 1.3.

In terms of security analysis, the only information that is available is in the white paper [19, p. 13–15], which includes a three-page "threat model" section. It describes a STRIDE analysis and lists mitigations such as "To prevent attacks, use one or more of these strategies as supported by the endpoint components: Stronger authentication schemes; Versions; Cryptographic algorithms".

Given the industry support behind this protocol, and its potential use across backend device vendors, this type of analysis may appear very limited compared to the successful approach used by IETF for TLS 1.3, which expressly invited interaction between design and formal analysis [37] and caught several potential flaws early in the design process.

In this work, we set out to provide a first formal analysis of the Security Protocol and Data Model (SPDM) protocol, and in particular focus on the current version 1.2.1 [20]. As we will show later, this is extremely challenging for a number of reasons: (i) While the protocol borrows heavily from TLS 1.3's design decisions, its main state machines and transcript handling are substantially different from, and more complex than, TLS 1.3. Notably, this means that the substantial analysis effort performed for TLS 1.3, e.g. [1, 3–6, 8, 9, 13, 14, 16, 17, 21–24, 26, 27, 30–32], provides absolutely no security guarantees for SPDM. Thus, all analysis has to be performed from scratch. (ii) The SPDM specification provides only very informal security goals and a high-level STRIDE analysis. (iii) A complete detailed modeling of the combination of complex transcripts and loops seems to stretch the limits of current formal protocol analysis tools.

Our main contributions are as follows:

- We construct the first formal model of the SPDM protocol, using the current version 1.2.1 as the reference point.
- We formally analyze our model using the TAMARIN prover [34], proving several of its core properties under specific threat models. Our models include device initialization, the four phases of the protocol, its three modes of key exchange and session setup, and the optional requests performed outside a secure session. Our results are the first substantial security analysis results for the SPDM protocol.
- Our formal modeling and analysis leads to several suggestions for next versions of the standard, as well as potential design pitfalls.

We provide all our annotated models for reproducibility, inspection, and extension, at [12].

**Paper Organization** This paper is organized as follows: In Section 2 we describe the main components of the SPDM protocol, and give an overview of the tool used for the formal analysis, the TAMARIN prover. In Section 3 we outline our formal models of SPDM and our design choices. Then, we formalize the security properties and show the results of our analysis in Section 4. Next, we discuss wider observations in Section 5, and conclude in Section 6. Finally, we provide details of the protocol's transcripts in Appendix A and a list of all the requests and responses of SPDM in Appendix B.

**Other Related Work** As stated previously, while the related TLS 1.3 protocol has been extensively investigated [1, 3–6, 8, 9, 13, 14, 16, 17, 21–24, 26, 27, 30–32], there currently is no meaningful security analysis of the SPDM protocol available beyond a three-page STRIDE analysis [19, p. 14].

One related work is [42], in which the authors consider what changes need to be made to the SPDM protocol to use post-quantum secure primitives. This largely involves selecting post-quantum secure versions of underlying primitives, except for the key exchange. Because SPDM's key exchange is based on Diffie-Hellman, for which currently no practical post-quantum secure drop-in replacement is known, the authors propose a new key exchange for SPDM based on a Key-Encapsulation Mechanism (KEM). They note that their key exchange design is similar to a KEM-based key exchange proposed for TLS 1.3, and leave security analysis to future work.

## 2 Background

In this section we provide background on the SPDM protocol in Section 2.1, and on the protocol analysis tool, the TAMARIN prover in Section 2.2.

### 2.1 Background on SPDM

Security Protocol and Data Model (SPDM) is a two-party protocol between a *Requester* that initiates the protocol and sends the first message and the *Responder*. SPDM aims to achieve two major security goals: *device attestation* and *authenticated, secure communication*. Enabling device attestation is explained in presentations as “the ability to attest various aspects of a device (Responder) such as firmware integrity and device identity” to an Requester. The second goal is again similar to TLS' security goals: establishing an authenticated, secure channel to exchange data between two parties over the wire.

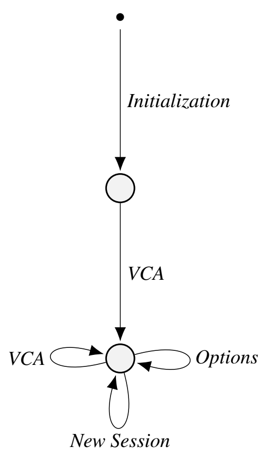
The protocol can be viewed as a composition of four phases: the *Device Initialization* phase, the *Version-Capabilities-Algorithms (VCA)* phase, then a phase we refer to as the *Options* phase, and finally the *Session* phase.

(i) **Device Initialization phase** The Device Initialization phase is performed outside of the protocol and is responsible for distributing to devices their cryptographic capabilities and the initial protocol implementation.

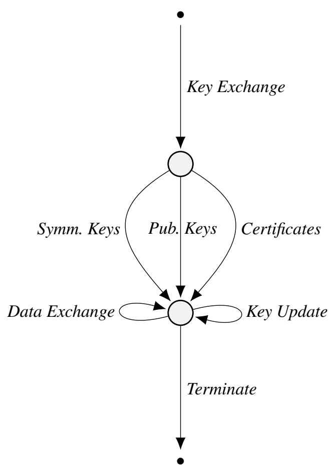
(ii) **VCA phase** The Version-Capabilities-Algorithms phase serves to initiate the protocol and negotiate ciphersuites and protocol versions between participants.

(iii) **Options phase** The Options phase enables the parties to perform *unilateral Responder authentication* and attest aspects of the Responder device through so-called *measurements*, the mutable configurations of the Responder's device. As these requests depend on device initialization outside of the protocol and the goals of the Requester, we call this part of the protocol *options phase*. The Requester can skip directly to the session phase, by sending a key exchange request after the VCA.

(iv) **Session phase** The Session phase establishes a secure and/or authenticated communication session between the parties to exchange application data. The phase is constructed from three sub-phases, the *handshake*, the *application data*, and the *key update*. During the handshake, the parties derive the session key using a key exchange based on Diffie-Hellman or preshared symmetric keys. By default the key exchange provides unilateral authentication of the Responder, and to elevate the connection to mutual authentication, the Responder needs to explicitly request it. During the application data sub-phase, the parties exchange data encrypted using an *authenticated encryption with associated data* (AEAD) scheme as specified in the architecture whitepaper [19]. In addition, the standard specifies that the parties shall regularly perform a *key update* mechanism.



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



(b) High-level view of the sub-phases of each *New Session*.

Figure 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-phase interaction within the session phase. Note that multiple sessions can be spawned and executed concurrently for each SPDM protocol run.

In Figure 1 we show an overview of the four phases of SPDM and the sub-phases of each new session. In the rest of this section, we give a more in-depth description of the SPDM processes.

**Device Initialization** Before the start of the protocol, parties receive their initial protocol code and cryptographic material during the *device initialization*. This initial setup should be performed in a secure and trusted environment, e.g., at the device manufacturer.

The general setup includes: a unique device identifier, the SPDM protocol implementation itself, and information on supported protocol versions, relevant capabilities, and cryptographic algorithms. For the protocol version, DMTF additionally defines which *request* and *response codes* are required to be implemented. Request and response codes define format and type of the sent messages during the protocol and are listed in [20, p. 33–36]. Additionally, the initialization should at least include one of the following:

(i) preshared symmetric keys with another device (possibly multiple),
(ii) preshared public keys with another device, or
(iii) a public key pair, certificates over the public key, and a root of trust to verify certificates.

Option (i) and (ii) need to be set up with predetermined communication partners. For both options, there is no fixed upper limit of shared keys. For option (iii), a device can store up to 8 certificates in ASN.1 DER-encoded X.509 v3 format as defined in [11]. The initial one (certificate slot 0), however, should only be set or altered in a secure and trusted environment. For the other 7 slots, SPDM offers means to retrieve a certificate signing request [36] from a Responder, and set the certificate remotely using *GET_CSR* and *SET_CERTIFICATE*. While the specification states that *SET_CERTIFICATE* should only be issued in a *secure session*, there is no detailed information on what that entails.

Additionally, during the device initialization vendors can define and implement their own request and response, which is allowed through the optional "vendor defined functionality" of SPDM.

**VCA and Connection Establishment** In the VCA phase, the parties exchange protocol *versions*, discover the *capabilities* of their partner, and the supported *algorithms*. To bootstrap the protocol, the Requester sends an initial version request to learn which SPDM version is supported by the Responder. After receiving the response, the Requester decides on the version of the protocol run, which according to the specification should be the highest supported version of both parties.

Each party has a set of capabilities, which define the supported operations of the SPDM specifications, for instance, if a party does support certificates, it sends a set *CERT_CAP* flag. After exchanging capabilities, both parties store the common subset of the exchanged capabilities.

Finally, the parties exchange supported algorithms: concretely, this is a list of supported cryptographic algorithms, like signature or encryption schemes. Intuitively, the strongest available cryptographic algorithms should be chosen, but the standard does not prescribe a selection mechanism.

During the VCA phase, the parties also maintain a transcript of their conversation, i.e., all received and sent messages.

**Device attestation through Measurements** In practice, the attestation mechanism boils down to a challenge-response mechanism that is optionally authenticated: the requester can ask the responder for so-called *measurements*, sending 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.

In both public key settings, these measurements can optionally be authenticated by using digital signatures, if this capability is supported by the communicating parties. In the case of preshared symmetric keys, it is not possible to request measurements explicitly at this point, but they can be part of the PSK exchange later on.

**Runtime Responder Authentication** When establishing a connection using non-preshared keys for the first time, the parties do not have cryptographic information about each other to perform authentication mechanisms. To this end, SPDM allows the Requester to retrieve from their partner a public key and certificate, *i.e., option (iii) of device initialization*, to be used in the protocol run.

The Requester can then challenge the Responders knowledge of the private key associated to the certificate by requesting a signature of the communication transcript and a random Requester chosen nonce. Details on the transcript computation can be found in [20], line 355.

Additionally, if the certificate was stored already in a previous session, the Requester can instead request a digest of the Responder's certificate for comparison (*GET_DIGESTS*). The specification recommends to perform unilateral Responder authentication using *CHALLENGE* at least once before performing device attestation through measurements.

**Key Exchange** Parties that support digital signatures and public key cryptography can start a Diffie-Hellman based key exchange to derive the session secret. During this phase, they also authenticate each other either by a) using preshared public keys, b) stored certificates, e.g. from a previous protocol run, or c) explicitly requesting their partner's certificates. Figure 2 shows how the key exchange between Requester and Responder is executed in the latter case.

To start the session phase, the Requester generates an ephemeral public key pair, a session id, and a random nonce and sends them with a key exchange request to the Responder. Symmetrically, the Responder also generates their own ephemeral key pair, session id and random nonce and sends them in the reply with a signature and a *message authentication code* (MAC) of the transcript so far. A transcript is the concatenation of all exchanged messages in a specific order. For details on how the transcripts and the authentication keys (also called *finished keys*) are constructed, we refer to Figure 6.

Note that the Responder needs to indicate if they wish to mutually authenticate the Requester by sending a flag in their reply. At the end of the protocol, they derive session keys and enter the application data exchange phase.

**PSK Key Exchange** The preshared key exchange (PSK) is intended for parties that have been provisioned a preshared symmetric key, and want to bootstrap secure and authenticated data exchange. To this end, the handshake serves to verify and prove knowledge of the shared key, thus implicitly authenticating themselves, and derive a unique session secret.

The Requester shows its intent to start the key exchange by sending the *PSK_EXCHANGE* request with a 32-bit random nonce or counter. From the specifications, if the parties have multiple preshared secrets, the request includes the slot ID of the intended key. Upon processing this message, the Responder can decide to either contribute in the session key derivation with their own nonce or immediately derive the session key and enter the data exchange phase. In the latter case, the parties can skip the finish messages.

However, when the Responder replies with their own nonce, they also authenticate the transcript of the protocol so far. At this point, all subsequent messages (*including the PSK_EXCHANGE_RSP*) until the end of the protocol need to be encrypted using the handshake secret as stated in [20], line 478. When completing the key exchange finish, the parties derive the session keys from the transcript and the initial shared secret (see Figure 6).

**Mutual Authentication** Explicit mutual authentication needs to be requested by the Responder when the parties are initialized with certificates and preshared public keys, by sending a flag, *MutAuthRequested*, in their key exchange reply. Depending on this flag's value, they can trigger the authentication mechanism with encapsulated flow or directly skip to the finish. The latter is desirable when they have already obtained the certificate from another protocol run or the parties have preshared public keys.

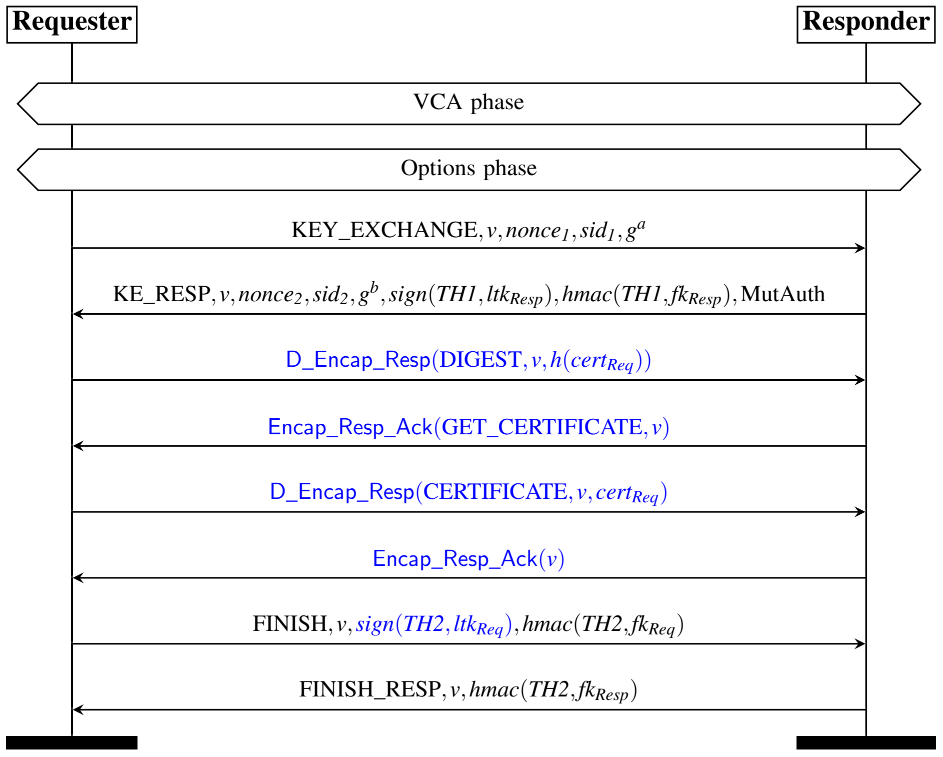

Requester
Responder
VCA phase
Options phase
KEY_EXCHANGE, $v, nonce_1, sid_1, g^a$
KE_RESP, $v, nonce_2, sid_2, g^b, sign(TH1, ltk_{Resp}), hmac(TH1, fk_{Resp})$, MutAuth
D_Encap_Resp(DIGEST, $v, h(cert_{Req})$)
Encap_Resp_Ack(GET_CERTIFICATE, $v$)
D_Encap_Resp(CERTIFICATE, $v, cert_{Req}$)
Encap_Resp_Ack($v$)
FINISH, $v, sign(TH2, ltk_{Req}), hmac(TH2, fk_{Req})$
FINISH_RESP, $v, hmac(TH2, fk_{Resp})$

Figure 2: 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*.

During a specific protocol run, the Requester role always has the initiative while the Responder role should only respond. However, this is not always desired and to ensure certain security goals, for instance, mutual authentication, the Responder needs to retrieve their partner's public key, and certificate. To this means, the protocol uses dedicated messages, e.g. , *ENCAPSULATED_REQUEST*, and an example information flow of the mutual authentication mechanism is shown in Figure 2.

**Key Update** When already in the application phase, the parties can update their session secrets instead of starting a new handshake. The update mechanism has two potential ways to update the keys: either the sender updates their own, or all the keys of the session.

To update their own key, the Requester sends a key update request (*KEY_UPDATE*) and upon acknowledgment forwards their session major secret, *msk* using a key derivation function *HKDF*: $msk_{i+1} = HKDF(msk_i, 'upd', '0')$. From this, the new encryption key is derived as $enckey = HMAC(msk_{i+1}, 'key')$. Then, to verify the key update it encrypts a request with the new key. Upon decrypting the message, the Responder deletes the old keys and encrypts the acknowledgment response. At this point the parties exit the key update mechanism and enter the application phase with the updated session major secret and new encryption key.

Note that when the Requester updates all keys, major secrets of both parties will be updated, and the Responder needs to encrypt the verify acknowledgement using their new encryption key. In addition, when the Responder wishes to update the secrets, the same protocol flow will be followed. However, instead of sending a key update request the latter uses the encapsulated messages e.g. *ENCAPSULATED_REQUEST*(*KEY_UPDATE*(...)).

**Threat modeling and mitigations** The publicly available SPDM documentation includes one part about threat modeling and mitigations in the architecture whitepaper [19, p. 13–15]. The threat model on page 13 draws the trust boundary between the requester and responder, across the communication channels; yet denotes the channels as "secure request" and "secure response". Pages 14 and 15 contain a STRIDE analysis table; we show an excerpt in Figure 3.

| **STRIDE category** | **Description** | **Justification mitigation** |
|---|---|---|
| Spoofing | Packets or messages without sequence numbers or timestamps can be captured and replayed in a wide variety of ways. Implement or use a communication protocol that supports anti-replay techniques, which investigate sequence numbers before timers, and strong integrity. | To prevent replay attacks, the Requester and Responder shall use a random nonce. |
| Information Disclosure | Custom authentication schemes are susceptible to common weaknesses, such as weak credential change management, credential equivalence, easily guessable credentials, absent credentials, downgrade authentication, or a weak credential change management system. Consider the impact and potential mitigations for your custom authentication scheme. | To prevent attacks, use one or more of these strategies as supported by the endpoint components: • Stronger authentication schemes • Versions • Cryptographic algorithms |

Figure 3: Excerpt: two out of ten rows in the STRIDE analysis from the SPDM architecture whitepaper [19, p. 14]

## 2.2 The Tamarin Prover

We set out to prove formal security guarantees for SPDM using the TAMARIN prover [34]. The TAMARIN prover is a state-of-the-art tool to verify and analyze complex security protocols. It is especially suited for protocols featuring various cryptographic primitives and constructions, like Diffie-Hellman, digital signatures, and symmetric encryption. TAMARIN operates in the symbolic model, where bit strings are abstracted to symbolic terms.

TAMARIN takes as input a model of the protocol, the description of an attacker and the property that we wish to prove. To formally model a protocol, we express its possible transitions using *Multiset Rewriting Rules* (MSR rules). Such a rule is usually written in the form $LHS \text{--}[\ Actions\ ]\rightarrow RHS$, and has three parts: a) a Left-Hand Side (LHS) which inputs *facts* needed to trigger the transition, b) action facts (Actions), to log an action label that is used to later on reference the transition, and c) a Right-Hand Side (RHS), which outputs the final state after the transition. In TAMARIN, facts are used to denote state knowledge, e.g., to represent the current state of a party in a protocol run. The In and Out facts are special built-in TAMARIN facts that model the receiving of a message and sending of one into the network. The fact Fr ensures that the variable in envelopes is instantiated uniquely. Additionally, TAMARIN distinguishes between two kinds of facts: *linear* and *persistent* facts. While linear facts, like

Fr, are consumed by rules, persistent facts, denoted with a “!”, will never be removed from a state. To give an intuition, consider the following rule:

$$\begin{array}{ll}
[\ !\mathsf{Device}(oid,\cdots),\mathsf{Fr}(ltk)\ ] & \}\ \text{LHS} \\
-[\ \mathsf{LongTermSecret}(oid,ltk)\ ]\!\rightarrow & \}\ \text{Actions} \\
[\ !\mathsf{LTK}(oid,ltk),!\mathsf{PK}(oid,pk(ltk)), & \multirow{2}{*}{$\Big\}\ \text{RHS}$} \\
\quad \mathsf{Out}(pk(ltk))\ ] &
\end{array}$$

The above rule is enabled if a device wants to create a new long-term key. Given a persistent Device fact, a unique long-term key *ltk* is created and bound to the *oid* of the device, by the action fact LongTermSecret.The rule creates two additional persistent facts, LTK and PK, adding the long-term key and its corresponding public key to the global state, respectively. With the Out fact, the public key is send out to the network, making it therefore “public”.

One can add rules to model the intended attacker, such as “leaking” the secret keys of an honest party to the network. TAMARIN provides a base attacker, informally known as Dolev-Yao, which has complete control over the network and can drop, inject and insert messages. For this, TAMARIN has another built-in fact K, which represents the current knowledge of the attacker/network.

To specify a property we write guarded first-order logic formulas such as:

$$\neg\exists\ oid\ ltk\ i\ j\ .\ \mathsf{LongTermSecret}(oid\ ,ltk\ )@i\ \wedge\ \mathsf{K}(ltk)@j$$

This is a reachability lemma, and expresses that there should not exist a trace of the model that includes the action fact LongTermSecret over some long-term key *ltk* at some time point *i* while the same key is known by the attacker at some other point *j*.

For each lemma, TAMARIN will either provide a proof of the property, an attack trace which falsifies it, or not terminate and timeout. In the latter case, users can use the interactive mode of the tool to debug the model and gain insight on how to guide the prover, for example by introducing additional helper lemmas, which can be seen as invariants. The analysis is then rerun with TAMARIN proving the helper lemmas first, and using them to prove the target property. In addition to helper lemmas, one can define *restrictions* on the models, to better capture realistic behaviors of the agents. Restrictions are specified as guarded first-order logic formulas as well and can help to reduce the search space of TAMARIN during a proof. For example with the following formula one would restrict each device to only be allowed one single long-term key.

$$\begin{aligned}
\forall\ oid\ ltk1\ ltk2\ i\ j\ .\ & \mathsf{LongTermSecret}(oid\ ,ltk1\ )@i\ \wedge \\
& \mathsf{LongTermSecret}(oid\ ,ltk2\ )@j \\
& \Rightarrow\ ltk1\ =\ ltk2
\end{aligned}$$

For more information we refer the reader to TAMARIN’s documentation [2].

## 3 Formal Model of SPDM v1.2

In this section we describe our formal 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:

A Device initialization and the VCA phase (Section 3.1),
B Options phase (Section 3.2),
C Session handling (Section 3.3),
D Three different key exchanges (Section 3.4), and
E Application data exchange and termination (Section 3.5).

We use these components to create four models:

- **Device Attestation**, includes device initialization, VCA and the options phase (A,B).

- **Key Exchange Certificate**, includes device initialization, VCA, the key agreement with certificates, and application data (A,C,D1,E).
- **Key Exchange Preshared Public Keys**, includes device initialization, VCA, the key agreement with public keys, and application data (A,C,D2,E).
- **Key Exchange Preshared Symmetric Keys**, includes device initialization, VCA, the key agreement with symmetric keys, and application data (A,C,D3,E).

Finally, in Section 3.6, we expand on the threat models considered in our analysis.

## 3.1 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: preshared symmetric keys (PSK), preshared 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 4a.

**Certificates** SPDM specifies using certificates as defined in [11], 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 $!\mathsf{RootCA}(key, pk(key))$. In the same way a device with identifier *id* is represented by $!\texttt{Device}(id, \ldots)$, where we use ... 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:

$$\begin{aligned}
&[\ !\texttt{Device}(id,\ldots), \mathsf{Fr}(ltk), !\mathsf{RootCA}(key,pk(key))\ ]\\
&-[\ \texttt{HonestlyGenerated}(id,ltk,pk(ltk)),\ldots\ ]\!\rightarrow\\
&[\ !\texttt{Cert}(id,pk(ltk),sign(<id,pk(ltk)>,key)),\ \mathsf{Out}(pk(ltk),sign(<id,pk(ltk)>,key),\ldots\ ]
\end{aligned}$$

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 4, 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:

$$\begin{aligned}
&[\ !\texttt{Device}(idReq,\ldots), !\texttt{Device}(idRes,\ldots), \mathsf{Fr}(\ tid),\ldots\ ]\\
&-[\ \mathsf{StartThread}(tid\ ,idReq\ ,idRes)\ ,\ \texttt{Channel}(idReq\ ,idRes)\ ]\!\rightarrow\\
&[\ \mathsf{StateReq}(tid\ ,idReq\ ,idRes\ ,\ldots,\text{'GETVERSION'}),\ \mathsf{Out}(<\text{'GET\_VERSION'},\text{'1'}>)\ ]
\end{aligned}$$

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 send 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 parties state and need to be updated with every message sent and received. For the details on the structure of the 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. At this point, all previous sessions and data related to that Requester and Responder conversation thread is not accessible anymore, i.e., no further transitions are allowed in the old thread. This is modeled as a restriction in TAMARIN:

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

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 threat $tid_1$ is replaced by a new $tid_2$, there cannot be any other transition being executed in the old thread $tid_1$.

## 3.2 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 3.1.

**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 i 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 is required to request the full certificate of the Responder and verify it using the root of trust stored at the persistent state of the Requester.

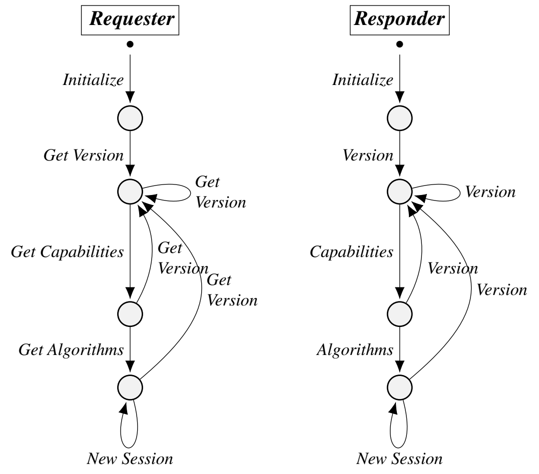
When modeling *CHALLENGE* and *CHALLENGE_AUTH*, we implicitly require that the Requester issued a *GET_CERTIFICATE* request at some previous point, as it is the only means for the Requester to learn the public key of the Responder in the certificate setting.

**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.

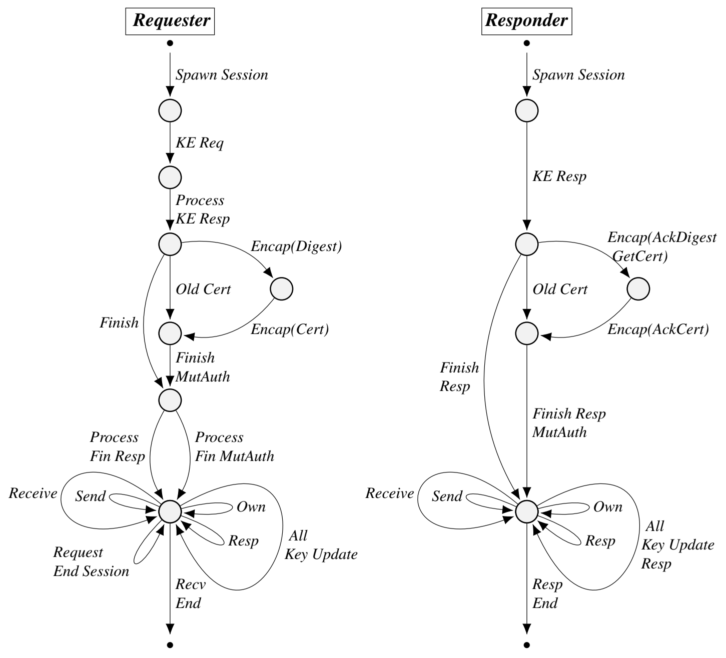
## 3.3 Session phase

Our formal analysis of the sessions includes the three sub-phases of a session, namely the handshake, the application data exchange loop with key update and the termination. Depending on the type of key exchange performed during the handshake, the analysis is spread across three models. In Figure 1b we give an overview of the session's sub-phases state machines.

In our models, we allow for an unbounded number of parallel sessions, each independent from another and at different sub-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 we will see later, this transition is represented by the *Spawn Session* edge. The core of the rule is the following:



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



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

Figure 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.

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

The result of this modeling choice is that the parties can have multiple independent sessions from the main conversation thread and data can be shared 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 handshake sub-phase, the parties' states transition to the application sub-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*.

## 3.4 Handshake sub-phase

**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 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 their random values. When sending the message, the Requester updates their transcript by adding the current message, stores the private key and goes in the state of waiting for the response. To encode this transition, we use the multi-set rewriting rule as follows:

$$\begin{array}{l}
[\ \texttt{KeyExchangeReq}(\mathit{sid},\mathit{tid},\ldots,\text{'NULL'},\mathit{tscript},\text{'START\_KE'}),\ \mathsf{Fr}(\ \mathit{nonce}),\mathsf{Fr}(\ \mathit{privK}),\ldots\ ] \\
-[\ \texttt{CurrentThread}(\mathit{tid}\ ,\mathit{idReq}\ ,\mathit{idRes})\ ,\ \mathsf{KETranscriptR}(\mathit{tscript})\ ]\!\to \\
[\ \mathsf{Out}(<\mathit{KEY\_EXCHANGE},\mathit{nonce},\mathsf{g}^{\mathit{privK}},\ldots>), \\
\texttt{KeyExchangeReq}(\mathit{sid},\mathit{tid},\ldots,\mathit{privK},\mathit{new\_tscript},\text{'WAIT\_RESP'})\ ]
\end{array}$$

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 3.3, 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 2), and c) send a finish request with mutual authentication if the certificate was provided in a previous session. In any of the cases, the parties cannot go back the protocol steps or change their choice within the same session. In the end, we model the two ways to finish the key exchange. The difference between the two being that in mutual authentication the Requester has to sign the protocol transcript and certificate digests, while in the unilateral case not.

In Figure 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 4a we show the state machine of the Device Initialization and VCA phase which are shared across all models, and in Figure 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 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 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.

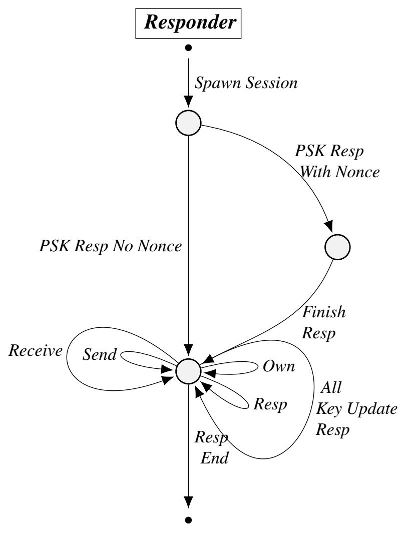
**Preshared public keys** The key agreement using preshared public keys is a slight variation of our certificate model. Here, the parties do not need to exchange certificates, but rather only verify their partner's knowledge of the keys provisioned to both devices before the start of the protocol. From the certificate model, we made the following two changes: a) removed the encapsulated mutual authentication, and b) used preshared public keys instead of certificates to create signatures of the transcript. In total we needed 8 rules to capture the transitions of the handshake.

**Preshared symmetric keys** We modeled the two options to perform the handshake of parties who have been provisioned with preshared 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 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 preshared symmetric key agreement with 9 TAMARIN rules.

**Transcripts** During the key agreement, the parties need to sign and/or authenticate the transcript of the conversation. The transcript 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 preshared private keys case, items 2 and 4 are not included. Note that the transcript only includes at any time the already issued and the current message, e.g. when the Responder signs the transcript in the key exchange response it will only include up to item 3. For more details on the transcripts, see Appendix A.2.

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 handshake sub-phase, and update their content on every new message



***Details of rules that are specific to this mode:***

***PSK Resp No Nonce** (PSK_EXCHANGE_RSP):*
- *Receive PSK_EXCHANGE request, $m_1$*
- *Derive major secrets and message keys*
- *Send message $m_2$:*
  $m_2 = \text{"KE\_Resp"}, v, sid, NULL, hmac(TH1, fk_{Resp})$
- *Enter Application Data*

***PSK Resp With Nonce** (PSK_EXCHANGE_RSP):*
- *Receive PSK_EXCHANGE request, $m_1$*
- *Generate nonce $n_2$*
- *Send message $m_2$:*
  $m_2 = \text{"KE\_Resp"}, v, sid, n_2, hmac(TH1, fk_{Resp})$

***Finish Resp** (PSK_FINISH_RSP):*
- *Receive PSK_FINISH request, $m_3$*
- *Verify Transcript of Requester*
- *Derive major secrets and message keys*
- *Send message $m_4$: $m_4 = \text{"FIN\_Resp"}, v$*
- *Enter Application Data*

Figure 5: Detailed state machine of the Responder in the Key Exchange with Preshared 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 sub-phase, shared across models. The state machines of the Requester are similar with the additional edges that process the responses of the Responder.

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:

$$\begin{aligned}
&\forall\, ke\_transcript\ i.\ \mathsf{KETranscriptR}(ke\_transcript)@i\\
&\Rightarrow (\exists\, m1\ m2\ m3\ m4.\\
&\qquad ke\_transcript = <\text{'Get\_Key\_Exchange'}, m1>\\
&\qquad\quad + <\text{'Key\_Exchange\_Resp'}, m2>\\
&\qquad\quad + <\text{'Finish'}, m3> + <\text{'Finish\_Resp'}, m4>)
\end{aligned}$$

The lemma states that all KETranscriptR labels at time *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.

**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 preshared 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 [28] and [29]. 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 6 we can see how the entire key derivation is performed by the Requester.

## 3.5 Application Data sub-phase

The application data phase starts at the end of the key agreement, as shown in Figure 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.

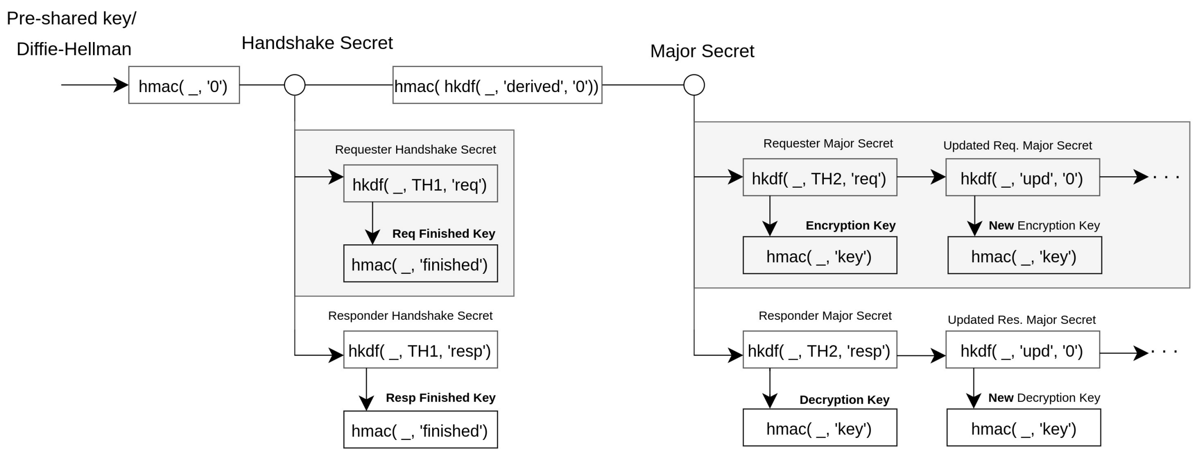


Figure 6: Derivation of finished key, session keys and update of message keys for Requester, symmetrically for the Responder. The _ is a placeholder for the input secret in the function, e.g., the handshake secret is calculated as $hmac(key_{preshared}, '0')$ in the preshared key setting. For the certificate 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 Section 3.4 on how the transcripts change for other cases.

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.

## 3.6 Threat Models

**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 then pose the questions on whether such an attacker can break the authentication guarantees in the certificate mode of the key agreement.

In the other key agreements, with preshared public and symmetric keys, we assume the secret keys are not compromised.

**Compromised Session key** We also run our models against another attacker that compromises the session keys of the parties. To model this behavior, we add a rule that takes as input the session state of one of the parties and leaks to the network the session key. As before, the attacker also has the full power of the built-in attacker. Under this threat model, we aim to prove the key secrecy and forward secrecy of a session.

## 4 Formal Analysis using TAMARIN

We now turn to formalizing and proving the security properties of the described SPDM components in the previous section. Our analysis is expanded across four models: a) the device attestation, b) the key exchange with certificates, c) key exchange with preshared public keys, and d) key exchange with preshared symmetric keys. In these models, we prove the following guarantees:

- Unilateral Responder Authentication
- Measurements Integrity and Authentication
- Mutual Authentication
- Secrecy of handshake key
- Forward secrecy (in restricted model)

In this section we focus on these properties before discussing our wider observations in Section 5.

## 4.1 Unilateral Responder Authentication

The main authentication property of SPDM is authentication of the Responder by the Requester; by default, this is unilateral authentication: only the Requester obtains a guarantee. This guarantee can be obtained in two distinct ways:

1. Responder Authentication with Public Key Cryptography and Certificates *before* a key exchange, or
2. Responder Authentication with Public Key Cryptography and Certificates *during* a key exchange (if mutual authentication is disabled)

We base our authentication formalizations on the work by Lowe [33], which states that if a party A in some role $role_A$ executes a run of the protocol with partner B and agrees on some data $ds$, then there must be a party B in the role of the partner $role_B$ running the protocol and agreeing on the same data $ds$.

In the context of SPDM and public keys, the identity of a party is by default equated with its public key. Thus, establishing that a Requester is communicating with the intended Responder amounts to verifying that there is thread of the protocol executed by the owner of the public key. Since we also model compromised keys, there exists the possibility that a Requester tries to authenticate a Responder whose private key is known to the network adversary, who in turn can emulate the responses without really running the protocol. Thus, we specify the authentication property as: for each successful challenge of an Requester, we expect that (i) if the challenged public key belongs to an honest party, (ii) there exists a Responder that owns the expected public key and is running the protocol. Formally this is specified as:

**Definition 1** (Responder Authentication 1)**.**

$$\begin{aligned}
&\forall\ tid1\ id1\ id2\ pk2\ ltk2\ i\ j\ .\\
&\mathsf{CommitChallenge}(tid1, id1, id2, pk2)@i\\
&\wedge\ \mathsf{HonestlyGenerated}(id2, ltk2, pk2)@j\\
&\Rightarrow\ (\exists\ tid2\ t\ .\ t < i\ \wedge\ \mathsf{RunningChallenge}(tid2, id2, ltk2)\ @t\ )
\end{aligned}$$

For protocols without loops, such lemmas are usually trivially proven by TAMARIN. However, in the case of the complex loops of SPDM, its backwards search may unroll loops arbitrarily often, which can cause non-termination. To help TAMARIN to prove this property, we needed to manually specify two additional helper lemmas. The first specifies that any certificate used during the challenge is either produced honestly or is attacker generated. The second lemma encodes that a Challenge query can only be issued after a successful completion of *GET_CERTIFICATE* and *CERTIFICATE* which TAMARIN can prove inductively.

While these two lemmas suffice to prove authentication of the responder, we were able to prove additional properties over the state of the involved parties, making the proofs terminate faster using less proof steps. We used a total of 5 helper lemmas to prove responder authentication before the key exchange (and used a total of 14 helper lemmas for responder authentication during the key exchange.)

The formalization of **Responder Authentication 2** is similar but uses action facts from mutual authentication as in Section 4.3.

## 4.2 Measurements

Another stated goal of SPDM is the integrity and authentication of device measurements. Similarly, to the unilateral responder authentication we prove the following: For a Requester querying for measurements with the public key of an honest Responder (i) the Requester is talking to its intended Responder and (ii) the measurements send by the Responder are also correctly received by the Requester. In TAMARIN the guarantee is specified as follows:

**Definition 2** (Measurement Authentication)**.**

$$\begin{aligned}
&\forall\, tid1\ id1\ id2\ pk2\ ltk2\ sign\ i\ j\ .\\
&\mathsf{CommitMeasurement}(tid1, id1, id2, pk2, sign)@i\\
&\wedge\ \mathsf{HonestlyGenerated}(id2, ltk2, pk2)@j\\
&\Rightarrow\ (\exists\, tid2\ t\ .\ t < i\ \wedge\ \mathsf{RunningMeasurement}(tid2, id2, ltk2, sign)\ @t\ )
\end{aligned}$$

The Requester commits to the received measurements from Responder with public key *pk2* in thread *tid1* once it verifies the signature *sign* of the Responder over the transcript. This implies that the Responder has a running thread *tid2* where it has indeed sent the measurements and the same signature *sign*. Note that the transcript contains both the measurement request issued initially as well as the Responder's reply, in addition to the VCA phase transcript.

We were able to prove this property using the same 5 helper lemmas of the unilateral authentication in the device attestation model as described in Section 4.1.

## 4.3 Mutual Authentication

In our analysis we prove mutual authentication of the three modes of the handshake sub-phase, namely with certificates, preshared public keys and preshared symmetric keys.

Informally, in mutual authentication the parties will verify the identity of each other in both directions and agree on the shared data. To express this in our model, we define the two agents in the roles of the *Requester* and *Responder*. Since we want to prove agreement, the data *ds* includes the entire *transcript* of the protocol (*TH2*) together with the exchanged nonces, and the handshake secret. Then, for the commit and running conversation thread of a party we specify two action facts CommitMutAuth and RunningMutAuth. The commit fact is used at the end of the protocol, respectively for the Requester after receiving the *FINISH_RSP*, (i.e., the last message of the key agreement), and for the Responder after receiving the *FINISH* request.

**Definition 3** (Mutual Authentication 1)**.** We define mutual authentication for models with preshared public keys and certificates as follows, where we require agreement over the used public keys:

$$\begin{aligned}
&\forall\, sid1\ tid1\ pk1\ pk2\ secret\ TH2\ role1\ id2\ ltk2\ i\ j\ .\\
&\mathsf{CommitMutAuth}(sid1, tid1, pk1, pk2, secret, TH2, role1\ )@i\\
&\wedge\ \mathsf{HonestlyGenerated}(id2, ltk2, pk2)@j\\
&\Rightarrow\ (\exists\, sid2\ tid2\ role2\ t\ .\ t < i\ \wedge\ not(role1 = role2) \wedge\\
&\mathsf{RunningMutAuth}(sid2, tid2, pk2, pk1, secret, TH2, role2\ )\ @t\ )
\end{aligned}$$

Note that for unilateral responder authentication in the certificate setting, the Responder does not commit on any data, as it did not request any certificates if mutual authentication was disabled.

Similarly, we proved a variation of this property, called *Mutual Authentication 2*, for the preshared symmetric keys models, where we removed the public keys from the action facts. However, in this model parties can execute the protocol with themselves if they are allowed to use the same key in both roles of Requester and Responder. We captured the reflection property in our model, and incorporated the check that preshared keys should be used only in one direction. In the modified version, we then proved the following mutual authentication property:

**Definition 4** (Mutual Authentication 2)**.** We define mutual authentication for models with preshared symmetric keys as:

$$\begin{aligned}
&\forall\, sid1\ tid1\ secret\ TH2\ role1\ i\ .\\
&\mathsf{CommitMutAuth}(sid1, tid1, secret, TH2, role1\ )@i\\
&\Rightarrow\ (\exists\, sid2\ tid2\ role2\ t\ .\ t < i\ \wedge\ not(role1 = role2) \wedge\\
&\mathsf{RunningMutAuth}(sid2, tid2, secret, TH2, role2\ )\ @t\ )
\end{aligned}$$

Overall, the mutual authentication properties required 14 helper lemmas for the certificate model, 12 for the preshared public keys and 9 for the preshared symmetric key.

## 4.4 Secrecy of Handshake Key

We want to prove that if a party finishes the key exchange with an honest partner, then the attacker does not know the handshake secret. In this property, we consider both the perspective of the Requester and the Responder.

**Definition 5** (Handshake Secrecy)**.** For the Requester, we define handshake secrecy as:

$$\begin{aligned}
&\forall\ sid\ tid\ idReq\ idRes\ pkRes\ secret\ id\ ltk\ i\ j\ .\\
&\mathsf{SesssionMajorSecretReq}(sid, tid, idReq, idRes, pkRes, secret)@i\\
&\wedge\ \mathsf{HonestlyGenerated}(id, ltk, pkRes)@j\\
&\Rightarrow\ \neg(\exists\ t\ .\ \mathsf{K}(secret)@t)
\end{aligned}$$

This formalizes that for any session *sid* that the Requester with identifier *idReq* starts with intended partner *idRes*, if the public key of the partner *pkRes* is honestly generated, then the attacker does not learn the handshake secret *secret* at any timepoint *t*.

Note this property does not hold for the Responder side in the handshake with certificates, because the Responder might not authenticate their partner. In these cases, the attacker can impersonate a Requester to the Responder and compute the handshake secret. In our model with certificates, we could prove that indeed this property does hold: a counterexample trace exists when the Responder finishes the entire protocol without a Requester.

The best case where we can prove secrecy of the Responder's handshake, is when the parties are mutually authenticated. For the preshared models, this is rather trivial since the attacker does not know any of the preshared secrets. For the certificate model, we need to modify the lemma, such that we reason only for the cases when the Responder performs the mutual authentication steps. All the lemmas where automatically proven in TAMARIN, using 13 helper lemmas in the certificate model, 12 in the preshared public keys, and 9 in the preshared symmetric keys.

## 4.5 Forward Secrecy

Forward secrecy, or *perfect forward secrecy*, is the security guarantee that previous keys in a conversation are secure even when the attacker compromises the parties' keys in the future [7]. A typical way to achieve forward secrecy within a session is by using a key derivation function (*KDF*) to derive a new key from the old one. However, this requires that the old keys are deleted regularly and the root key is not leaked to the attacker at the time of the compromise. Looking at the key update mechanism in SPDM (see Figure 6), one can notice the same pattern of how the parties forward their keys within a session. More precisely, the major secrets are updated as $major\_secret_{i+1} = HKDF(major\_secret_i, \dots)$. This creates a chain of keys where the attacker cannot compute previous keys $major\_secret_i$ even if the future ones are known, unless they reverse a one-way function.

In TAMARIN we expressed this property that once the Requester *idReq* updates the current secret from *s1* to *s2* in session *sid1*, and the attacker knows the previous key *s1*, but has not compromised their partner *idRes*, then it must have compromised the Requester before the update.

**Definition 6** (Forward Secrecy)**.** In our models, we define forward secrecy as:

$$\begin{aligned}
&\forall\ sid1\ idReq\ idRes\ s1\ s2\ i\ j\ .\\
&\mathsf{OwnKeyUpdate}(sid1, idReq, idRes, s1, s2\ , \dots)@i\\
&\wedge\ \mathsf{K}(s1)@j\\
&\wedge\ \neg(\exists\ sid2\ k\ .\ \mathsf{CompromiseParty}(sid2, idReq, idRes, \dots)@k)\\
&\Rightarrow\ (\exists\ t\ .\ \mathsf{CompromiseParty}(sid1, idReq, idRes, \dots)@t\ \wedge\ t < i)\ )
\end{aligned}$$

Here, the secrets *s1* and *s2* are the role-oriented major secrets. The property was proven from the perspective of both participants, conversely for Responder major secret and Requester major secret.

When trying to obtain the proof for this property, we encountered several difficulties in the unbounded model. The main issues were the large state of the protocol, the key size, and the multiple transitions with loops in the protocol. As a result, the memory of the machine was not enough, and TAMARIN's interface could not process the protocol graphs to provide an intuition on the how to help the tool. In the end, we had to restrict the models in a way that the parties could only perform each action within a session only once, e.g. , a single key update request. In this restricted model, we could prove this intermediate step to forward secrecy by guiding the tool with a proof oracle and helper lemmas.

In total, we proved 3 additional helper lemmas for key secrecy in the absence of compromise, and 18 structural ones.

| Model | Notes | Property | Result | Runtime (s) | Helper Lemmas |
|---|---|---|---|---|---|
| Device Attestation | | Responder Authentication 1 | ✓ | 20 | 5 |
| | | Measurement Authentication | ✓ | 25 | 5 |
| Certificates | | Responder Authentication 2 | ✓ | 76 | 14 |
| | | Mutual Authentication 1 | ✓ | 187 | 14 |
| | | Handshake Secrecy | ✓ | 208 | 13 |
| Preshared Public Keys | | Mutual Authentication 1 | ✓ | 58 | 12 |
| | | Handshake Secrecy | ✓ | 18 | 12 |
| | Restricted Model | Forward Secrecy | ✓ | 30 | 21 |
| Preshared Symmetric Keys | | Mutual Authentication 2 | ✓ | 17 | 9 |
| | | Handshake Secrecy | ✓ | 5 | 9 |

Table 1: **Summary of formal analysis results in TAMARIN.** Our proofs and their helper lemmas are obtained automatically, except for Forward Secrecy property which required us to guide the tool with an oracle. The runtime shows the time it takes for the tool to prove the main property.

## 4.6 Analysis Summary

In Table 1 we summarize the main security guarantees that we proved in our models. TAMARIN automatically proves all guarantees in this section and their helper lemmas, with the exception of the complex forward secrecy guarantee whose proof was guided by an oracle that we manually specified. In addition, TAMARIN automatically proved 54 sanity lemmas in total, showing that our models execute correctly and capture the intended protocol flow.

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. The execution time per property proven spans from 5s (PSK model, handshake secrecy) to 3m28s (certificate model, handshake secrecy).

## 5 Results and Discussion

In the preceding, we have focused on the properties we could formally prove, and thus Table 1 contains positive results of properties that hold within our formal model. However, our modeling and analysis did yield several other observations.

## 5.1 Potential design pitfalls

While our analysis suggests that SPDM achieves its main design goals for basic threat models, we note that the current design still has some design pitfalls, which we hope can be resolved in future versions.

**Session ID size and optional responder nonce in PSK mode** Under some circumstances, the replay protection in the protocol becomes entirely dependent on the uniqueness of the session ID. The size of the session ID share of each party is two bytes, which is insufficient in cryptographic terms to prevent replays; there exist practical attacks that exploit such small ranges.

For example, the Responder nonce is optional. This is particularly relevant for the PSK mode: if the responder provides no nonce during a PSK mode session, the initial messages could be replayed in the future by an attacker to a session that uses the same session ID (due to the small range) to trigger a session with the same session keys. This can have two consequences: (a) the Responder can mistakenly assume there is a recent Requester request, and (b) if a specific session key is compromised (e.g. through data loss or cryptanalysis), this can be used together with the reply to impersonate the Requester indefinitely to the Responder. We recommend requiring the Responder nonce in this case, or at least documenting the security consequences of omitting it.

**Device reset may lead to counter reuse** Instead of random nonces, SPDM allows the use of counters at certain phases of the protocol. The standard explicitly lists that (a) counters should not be repeated or reset during the lifetime of a device, and (b) devices can be reset, which typically leads to loss of all volatile state on the device. In practice these claims might be at odds or unrealistic unless counters are stored in non-volatile memory. It seems prudent to reduce the dependence on counter uniqueness.

**No restrictions on vendor-defined request/response** For flexibility, the standard explicitly allows for vendor-defined request/response definitions with only little restrictions. If vendor-defined mechanisms can reuse long-term or ephemeral secrets from the protocol, they can break the security guarantees of the core design. We recommend these are disallowed from reusing secrets from the protocol, and instead are just handled as requests over the standard data transfer managed by the SPDM core.

**Authentication of keys versus device authentication** Currently, all authentication essentially authenticates keys, not devices or their identifiers. Notably, while in the future SPDM seems to aim for binding to hardware object identifiers (OID), there is currently no mechanism that can check that OID identifiers are bound to keys, nor are OID identifiers included in transcripts or confirmations. The standard suggests to include OID identifiers in certificates, which would effectively lift them into transcripts, but only for the public key modes. This leaves the PSK mode unsolved, notably in the case that PSKs are shared among more than two devices. We recommend explicitly including OIDs in the transcripts.

**No default deny-all for remotely setting certificates** The standard includes a feature to remotely set trusted certifications for parties. This feature has the potential to be misused and violate all security goals. It seems prudent to require a default policy that this feature cannot be used, and must require authentication and specific access policies.

**Setting certificates** From the specification we do not find any restriction on which Requester devices are allowed to provision which certificates to which Responder. It only states that it should be performed in a secure environment.

Specification [20], Section 10.25.2:

```
For Slot 0 provisioning, the Requester should issue SET_CERTIFICATE only in a trusted environment (such as a secure
manufacturing environment). For slots 1-7, if the provisioning happens in a trusted environment, the Requester should
issue SET_CERTIFICATE inside a secure session.
```

There may be a connection to the *GET_CSR* request code, which allows the Requester to request a certificate signing request (CSR) from the Responder.

Specification [20], Section 10.25.1:

```
The resulting CSR contained in a successful CSR response will have to be signed by an appropriate Certificate Authority.
The details of the Public Key Infrastructure used to verify and sign the CSR, and make the final certificate available for
provisioning are outside the scope of this specification.
```

Thus, the standard currently does not seem to specify whether a CSR is needed to set a certificate (or to which extent the CSR is verified), or if those two requests are independent.

## 5.2 Further observations

**Previous responder authentication does not necessarily strengthen device attestation guarantees** The specification suggests that it could be interesting to authenticate the Responder before requesting its measurements.

Specification [20], 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, in our threat models we could not find any need for this, as proofs about measurement integrity and its authentication do not seem to depend on a previous Responder authentication. Notably, prior authentication still allows attestation without integrity protection.

## 5.3 Limitations and Future Work

**Unified model** To analyze SPDM at the desired level of detail, we had to split all possible flows into four models. Ideally, we would verify all security properties using a general, unified model. However, in realistic deployments of the protocol we do not expect the usage of multiple handshake modes simultaneously, therefore only two of the models would need to be merged, i.e., the device attestation and a handshake mode. At the moment this seems beyond reach of any existing methodology and would require advances in the verification methods.

**Missing Functionality** As already discussed Section 5, we did not model all possible functionalities of SPDM. While some parts of the functionality of SPDM lie outside the scope of the symbolic model, e.g. sending messages in chunks, others are still underspecified. See Appendix B to see all request and response codes of SPDM and which we included in our current models.

**Cryptographic Primitives** We used standard symbolic models of the cryptographic primitives, i.e., the classical Dolev-Yao modeling of primitives. Recently, effort was put into making the models of cryptographic primitives (E.g. a signature scheme) much more fine-grained, and capable of reflecting properties of the concrete instantiations (E.g. the IETF version of Ed25519) [10, 25]. SPDM uses digital signatures, AEAD, and hashes in multiple phases of the protocol. Possible future work could include making the models more tailored towards the concrete schemes used by SPDM. We experimented with an over-approximation model where we used broken cryptographic primitives (e.g. information leaking hashes, signatures that are verified by anything), we could observe that authentication properties (like measurement integrity) break. This seems natural and we used this model as a sanity check. Nevertheless, downgrade attacks are a real risk, and have been shown on older versions of TLS, forcing the honest participants to use cryptography that modern attackers can break. While this model was a first step to model downgrade attacks, we leave a detailed downgrade analysis as future work.

## 6 Conclusion

DMTF's SPDM is a complex protocol that uses TLS 1.3-like elements, but in a sufficiently different way that it does not automatically inherit any of the guarantees proven for TLS 1.3. For a protocol with such a wide industry support, it has seen surprisingly little security analysis.

We performed the first in-depth analysis of this protocol, proving the core properties of the protocol in a symbolic model that pushes the limits of what is possible with current tools like TAMARIN. Our models include all main modes, and various possible security guarantees; we thus proved the absence of a substantial class of attacks. Beyond our observations and suggestions, we expect that a computational analysis of the actual transmission layer would yield further insights.

## Acknowledgements

We are grateful to Justine Sauvage for contributions to the early stages of the SPDM models, fruitful discussions, and valuable feedback.

## References

[1] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. The privacy of the TLS 1.3 protocol. *Proc. Priv. Enhancing Technol.*, 2019(4):190–210, 2019.

[2] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. Documentation of the Tamarin Prover, 2022. https://tamarin-prover.github.io/#documentation.

[3] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In *CRYPTO (1)*, volume 9814 of *Lecture Notes in Computer Science*, pages 247–276. Springer, 2016.

[4] 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*, pages 506–525. IEEE Computer Society, 2016.

[5] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. miTLS: Verifying Protocol Implementations against Real-World Attacks. *IEEE Secur. Priv.*, 14(6):18–25, 2016.

[6] Bruno Blanchet. Composition theorems for CryptoVerif and application to TLS 1.3. In *CSF*, pages 16–30. IEEE Computer Society, 2018.

[7] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for authentication and key establishment*. Springer, second edition, 2020.

[8] Jacqueline Brendel, Marc Fischlin, and Felix Günther. Breakdown Resilience of Key Exchange Protocols: NewHope, TLS 1.3, and Hybrids. In *ESORICS (2)*, volume 11736 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2019.

[9] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. *IACR Cryptol. ePrint Arch.*, page 467, 2021.

[10] 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. Cryptology ePrint Archive, Paper 2022/1314, 2022. `https://eprint.iacr.org/2022/1314`.

[11] 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. Technical report, 2008.

[12] Cas Cremers, Alexander Dax, and Aurora Naska. Tamarin models and analysis scripts to reproduce the results in this paper, 2022. `https://github.com/AnalysisSPDM/FormalModel`.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788, 2017.

[14] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485. IEEE, 2016.

[15] CXL. Compute Express Link homepage, 2022. `https://computeexpresslink.org/`.

[16] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. On the concrete security of TLS 1.3 PSK mode. In *EUROCRYPT (2)*, volume 13276 of *Lecture Notes in Computer Science*, pages 876–906. Springer, 2022.

[17] Denis Diemert and Tibor Jager. On the tight security of TLS 1.3: Theoretically sound cryptographic parameters for real-world deployments. *J. Cryptol.*, 34(3):30, 2021.

[18] DMTF. DMTF website. `https://www.dmtf.org/`. accessed: 2022-10-09.

[19] 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`, Oct 2022. accessed: 2022-10-09.

[20] DMTF. DSP2058: Security Protocol and Data Model (SPDM) Architecture White Paper, Version 1.2.1. `https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.2.1.pdf`, Jun 2022. accessed: 2022-10-09.

[21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *CCS*, pages 1197–1210. ACM, 2015.

[22] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *J. Cryptol.*, 34(4):37, 2021.

[23] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *EuroS&P*, pages 60–75. IEEE, 2017.

[24] 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*, pages 452–469. IEEE Computer Society, 2016.

[25] 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.

[26] 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 *CCS*, pages 1185–1196. ACM, 2015.

[27] Hugo Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In *CCS*, pages 1438–1450. ACM, 2016.

[28] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. Technical report, 1997.

[29] Hugo Krawczyk and Pasi Eronen. HMAC-based extract-and-expand key derivation function (HKDF). Technical report, 2010.

[30] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. *IACR Cryptol. ePrint Arch.*, page 978, 2015.

[31] 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. *IEEE Trans. Dependable Secur. Comput.*, 16(2):272–286, 2019.

[32] 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*, pages 486–505. IEEE Computer Society, 2016.

[33] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.

[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)*, pages 696–701. Springer, 2013.

[35] MIPI. MIPI homepage, 2022. https://www.mipi.org/.

[36] M. Nystrom and B. Kaliski. RFC 2986: PKCS #10: Certification Request Syntax Specification Version 1.7, 2000. https://www.rfc-editor.org/rfc/rfc2986 (accessed: 2022-10-09).

[37] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of TLS. In *SSR*, volume 10074 of *Lecture Notes in Computer Science*, pages 160–186. Springer, 2016.

[38] PCI-SIG. PCI-SIG homepage, 2022. https://pcisig.com/.

[39] Scott Phuong. End-to-End Infrastucture Security: Security Protocol and Data Model. https://www.dmtf.org/sites/default/files/SPDM_1.0_Keynote_APTS.pdf, 2019. accessed: 2022-10-09.

[40] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, August 2018. https://tools.ietf.org/html/rfc8446 (accessed: 2022-10-09).

[41] TCG. Trusted Computing Group homepage, 2022. https://trustedcomputinggroup.org/.

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

## A Transcripts

### A.1 Transcripts for the Options Phase

**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)

**Transcripts for Measurement** The transcript for measurements is as follows:

– VCA , *GET_MEASUREMENTS*.* , *MEASUREMENTS*.*

## A.2 Transcripts during Key Agreement

**Transcript for HMAC in Preshared Symmetric Keys** In the preshared symmetric leys, the parties do not include the digest of the certificates or public keys. In addition, some of the 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 follows:

– *GET_VERSION*.*
– *VERSION*.*
– *GET_CAPABILITIES*.* (if issued)
– *CAPABILITIES*.* (if issued)
– *NEGOTIATE_ALGORITHMS*.* (if issued)
– *ALGORITHMS*.* (if issued)
– *PSK_EXCHANGE*.*
– *PSK_EXCHANGE_RSP*.* (current message of Responder, except *ResponderVerifyData*)
– *PSK_FINISH*.* (current message of Initiator, except *RequestorVerifyData*)

**Transcript for HMAC in Key Exchange** In key exchange the parties send the HMAC of the transcript during all messages except the *KEY_EXCHANGE* request. 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)

**Transcript for Signature in Key Exchange** The signature appended in 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 defined as follows:

– 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)

**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: a) TH1-used to derive the role directed secrets in the handshake phase, and b) TH2- used to derive session secrets.

Transcript TH1 for Key-Exchange (respectively Preshared Keys):

– VCA
– Hash of the Responder certificate
– *KEY_EXCHANGE*.*
– *KEY_EXCHANGE_RSP*.* (except *ResponderVerifyData*)

Transcript TH1 for Preshared Symmetric Keys:

– VCA
– *PSK_EXCHANGE*.*
– *PSK_EXCHANGE_RSP*.* (except *ResponderVerifyData*)

Transcript for TH2 for Key-Exchange:

– VCA
– Hash of the Responder certificate/public key
– *KEY_EXCHANGE*.*
– *KEY_EXCHANGE_RSP*.* (except *ResponderVerifyData*)
– (Hash of the Requester certificate/public key) (if mutual authentication)
– *FINISH*.*
– *FINISH_RSP*.*

Transcript for TH2 for Preshared Symmetric Keys:

– VCA
– *PSK_EXCHANGE*.*
– *PSK_EXCHANGE_RSP*.* (except *ResponderVerifyData*)
– *PSK_FINISH*.* (if issued)
– *PSK_FINISH_RSP*.* if issued

## B Request and Response Codes

| 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 5 |
| *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 5 |
| *SET_CERTIFICATE & SET_CERTIFICATE_RSP* | ✗ | See Discussion in Section 5 |
| *CHUNK_SEND & CHUNK_SEND_ACK* | ✗ | out of scope for TAMARIN |
| *CHUNK_GET & CHUNK_RESPONSE* | ✗ | out of scope for TAMARIN |

Table 2: List of all Request and Response codes in SPDM. Details can be found in [20]