Root-of-Trust Abstractions for Symbolic Analysis: Application to Attestation Protocols

Georgios Fotiadis[1], José Moreira[2], Thanassis Giannetsos[3], Liqun Chen[4], Peter B. Rønne[1], Mark D. Ryan[2], and Peter Y.A. Ryan[1]

[1] SnT, University of Luxembourg, Luxembourg
{georgios.fotiadis,peter.roenne,peter.ryan}@uni.lu
[2] School of Computer Science, University of Birmingham, United Kingdom
{j.moreira-sanchez,m.d.ryan}@cs.bham.ac.uk
[3] Ubitech Ltd., Digital Security & Trusted Computing Group, Greece
agiannetsos@ubitech.eu
[4] Surrey Centre for Cyber Security, University of Surrey, United Kingdom
liqun.chen@surrey.ac.uk

Abstract. A key component in building trusted computing services is a highly secure anchor that serves as a Root-of-Trust (RoT). There are several works that conduct formal analysis on the security of such commodity RoTs (or parts of it), and also a few ones devoted to verifying the trusted computing service as a whole. However, most of the existing schemes try to verify security without differentiating the internal cryptography mechanisms of the underlying hardware token from the client application cryptography. This approach limits, to some extent, the reasoning that can be made about the level of assurance of the overall system by automated reasoning tools. In this work, we present a methodology that enables the use of formal verification tools towards verifying complex protocols using trusted computing. The focus is on reasoning about the overall application security, provided from the integration of the RoT services, and how these can translate to larger systems when the underlying cryptographic engine is considered perfectly secure. Using the Tamarin prover, we demonstrate the feasibility of our approach by instantiating it for a TPM-based *remote attestation* service, which is one of the core security services needed in today's increased attack landscape.

Keywords: Trusted Computing · Remote attestation · TPM modelling · Formal Verification · Tamarin-prover · SAPiC.

1 Introduction

In the last years, academia and industry working groups have made substantial efforts towards realizing next-generation smart-connectivity "*Systems-of-Systems*". These systems have evolved from local, standalone systems into safe and secure solutions distributed over the continuum from cyber-physical end devices, to edge servers and cloud facilities. The core pillar in such ecosystems is the establishment of trust-aware *service graph chains*, comprising both resource-constrained devices, running at the edge, but also container-based technologies. The primary existing mechanism to establish trust is by leveraging the concept

of trusted computing, which addresses the need for verifiable evidence about a system and the integrity of its trusted computing base and, to this end, related specifications provide the foundational concepts such as *measured boot*, *sealed storage*, *platform authentication* and *remote attestation*.

An essential component in building such trusted computing services is a highly secure anchor (either software- or hardware-based) that serves as a Root-of-Trust (RoT), providing cryptographic functions, measuring and reporting the behavior of running software, and storing data securely. Examples include programmable Trusted Execution Environments (TEEs), and fixed-API devices, like the Trusted Platform Module (TPM) [24]. Such components are considered inherently secure elements and implement hardened protections (e.g., tamper-proof or side-channel protections), because any vulnerability on them can compromise the security assurance of the overall system. This sets the challenge ahead: *Because such RoTs by definition are trusted, all internal operations including handling of cryptographic data can be idealized. However, this does not directly translate to the overall application security that leverages such RoTs.*

Thus, formal verification of protocols and interfaces has become a fundamental process to identify bugs or misuse cases when building complex interacting systems, as several earlier works have shown, e.g., [23,6]. There are two noteworthy difficulties when considering formal verification in these settings. First, the inherent fact that protocol verification is an undecidable problem for unbounded number of protocol executions and unbounded size of the terms. And second, the additional difficulty faced by protocol verification tools when considering protocols with non-monotonic mutable global states, which may provoke numerous false attacks or failure to prove security; see [7]. Trusted computing protocols fall into both categories, because the non-volatile memory of the RoTs can be regarded as a state that may change between executions of the protocol.

Indeed, several works on symbolic verification of hardware RoTs that are part of other high-level functionalities take into account the usage of persistent state [2,7]. Some notable examples in the context of the TPM are the works by Shao et al., which cover specific subsets of TPM functionalities, such as Enhanced Authorization (EA) [23] or HMAC authorization [22], identifying misuse cases. Also, Xi et al. [30] and Wesemeyer et al. [29] conduct formal analysis and verification of the the Direct Anonymous Attestation (DAA) protocol of the TPM. On the other hand, Delaune et al. [10], propose a Horn-clause framework where they prove its soundness and use ProVerif to approximate the TPM internal state space, helping to address non-termination issues.

A recurrent characteristic when formally verifying the aforementioned scenarios is that there is usually no distinction between the cryptography used for self-consumption of the RoT (e.g., the mechanism it uses for secure storage) and cryptography that is provided specifically for the overall application. This has the inherent drawback that reasoning about the security of a large application or service forces reasoning about the internals of the security device itself, and thus limiting or hampering the scope of the reasoning that automated analysis tools can achieve.

Contribution. The main objective of the present work is to propose a methodology for proving security in scenarios based on services that make use of RoTs, by idealizing the internal functionalities of the security device, except those that provide explicit cryptographic functionalities for the service being offered. In order to illustrate our methodology, we concentrate on a class of remote attestation services based on the TPM. Even though we focus on a particular case of attestation, we build an abstract model for a subset of TPM primitives sufficient to implement the core functionalities of generic attestation services. From the perspective of formally verifying RoT-based applications, this model represents a means of reasoning about security and privacy (of offered services) without being bogged down by the intricacies of various crypto primitives considered in the different platforms. We conduct our analysis in the symbolic model of cryptography (Dolev-Yao adversary [11]) through the Tamarin prover [4] and its front-end SAPiC [17]. We define a number of security properties relevant for the considered scenario, and successfully verify them with this framework.

2 Background

In this section we summarize specific notions related to RoT, TPM and remote attestation that will be needed in our discussion to follow. For more details, we refer the reader to [24,25,26,3,21,14].

2.1 The TPM as a Root of Trust

The Trusted Computing Group (TCG) splits the responsibility of the RoT into three categories: measurement, storage and reporting [24]. The RoT for measurement is usually the first piece of BIOS code that is executed on the main processor during boot, and starts a chain of measurements. The RoT for storage and reporting are responsibilities assigned to the TPM, typically implemented as a tamper-resistant hardware embedded in a *host* device. The TPM and the host device form a *platform*.

A *measurement chain of trust* is a mechanism that allows to establish trust from the low-level RoT for measurement to a higher-level object, e.g., the OS. Each component in the chain measures the next component and these measurements are checked against reference values, typically provided by the platform manufacturer, before passing control to the next component. For instance, the RoT will measure the (remaining part of) the BIOS, the BIOS will measure the bootloader, and so on. Each component in the chain updates the TPM's Platform Configuration Registers (PCRs), which are a set of special registers that store the representation of the measurements as a hash chain.

The TPM contains an embedded key pair known as the Endorsement Key (EK) generated and certified by the platform manufacturer. The private part of the EK never leaves the TPM. This key pair uniquely identifies the platform. If this key pair is used to sign platform measurements it will compromise the platform privacy. Therefore, the TPM offers mechanisms to generate an arbitrary

number of Attestation Keys (AKs) that can be used to attest the platform state by signing the PCR contents. These AKs are generated in such a way that it can be ensured to an external verifier that the signature was generated by a legitimate TPM, without revealing the identity of the TPM. See Sec. 2.2 below.

Moreover, the TPM offers mechanisms to restrict access to TPM commands and objects by defining authorization policies. Most notably, in TPM 2.0, the Enhanced Authorization (EA) mechanism allows to define flexible, fine-grained authorization policies by combining a number of assertions through logical connectors. For instance, a system administrator could create a TPM key and associate with it a policy that allows the usage of that key when (i) the PCRs are in a given state, or (ii) a password is provided *and* the user is physically present. The authorization policy is stored, within the TPM object, as a hash chain called `authPolicy`. An authorization session is the mechanism used to pass in authorization values or policies, and to maintain state between subsequent commands. To load or use a TPM object a session must be created, and the user will indicate what assertions must be checked. The TPM checks the assertions and updates the session attribute `policyDigest` (a hash chain) if they succeed. If `policyDigest` matches the `authPolicy` for a given object, then access to that object is granted. We refer the reader to [24, §19.7] for the complete details. For illustration purposes, we provide an example in Appendix A.

2.2 Remote Attestation

Remote attestation [21,14] is a mechanism to provide evidence of the integrity status of a platform. It is typically realized as a challenge-response protocol that allows a third party (*verifier*) to obtain an authentic and timely report about the state of an untrusted, and potentially compromised, remote device (*prover*). The TPM allows implementing privacy-preserving remote attestation protocols. Remote attestation services are currently used in a variety of scenarios, ranging from attestation for isolated execution environments based on the –now outdated– Intel's Trusted Execution Technology [15], to more modern approaches used together with Intel's Software Guard Extensions, e.g., [28,16].

From a high-level perspective, a remote attestation protocol requires that the user first creates an AK that will be used to sign attestation reports (*quotes*). A quote is essentially composed of the contents stored in selected PCRs (which reflect the platform state) signed with with the AK. As commented above, the user has the ability to create as many AKs as they wish, but each AK is required to be certified by a third party called the Privacy Certification Authority (PCA). The certification process, detailed in Sec. 4, implies that the PCA knows the relationship between EK and AKs, but the PCA is trusted not to reveal this information, which would break the anonymity of the platform. A verifier can trust the platform if it successfully verifies that a quote is a valid signature over expected PCR values with an AK certified by a PCA.

We also note that the TCG has an alternative method for performing remote attestation without revealing the EK to a trusted third party, which is known as the Direct Anonymous Attestation (DAA) [5]. However, DAA works by design

only in conjunction with the TCG specification for TPMs. Since the aim in this paper is to formalize the notion of secure remote attestation in a more general context, we focus on the first approach, i.e., attestation using a PCA. Such protocols represent most of the real-world applications and they do not modify the core characteristics of the remote attestation service. Further, we argue that the formalization methods that we present here can be used as the basis for other hardware-based remote attestation instances, not only TPMs.

2.3 The Tamarin prover and SAPiC

For our modelling approach we have chosen the Tamarin prover [4,19] and its front-end SAPiC [17]. SAPiC allows modelling protocols in (a dialect of) the applied pi-calculus [1], and converts the processes specification into multiset rewrite rules (MSRs) that can be processed by Tamarin. Security properties can be expressed as first-order logic formulas. Everything that SAPiC does can be expressed in MSRs in Tamarin, but it provides an encoding (e.g., for locks, reliable channels, state handling) which is likely more concise than an ad-hoc modelling a user would come up with using MSRs. In this context, SAPiC has a better chance for termination. We refer the reader to Appendix B for a brief description of the SAPiC syntax; see [4,17] for the complete details.

Tamarin and SAPiC have already been used successfully for modelling TPM functionalities in existing works, e.g., [23,22,29], and they offer a convenient syntax for modelling protocols with global state. However, as mentioned above, it is rather challenging to model protocols with arbitrarily mutable global state, as it is required in the scenario presented in this paper. Therefore some technical alternatives and manual intervention have been adopted in order to define a realistic adversary. See Sec. 5 below.

3 A Methodology for Modelling protocols with RoTs

The key idea in our modelling approach is to consider a further layer of abstraction within the traditional symbolic model of cryptography [11] and idealize the internal functionalities of the RoT, except those providing cryptography to a consumer application like hashing, asymmetric encryption or signatures, capturing their intended semantics rather than their implementation. This is done by replacing cryptographic functionalities of the RoT with non-cryptographic mechanisms, for example using a combination of strong access control with channels not visible to the adversary where honest parties interact (private channels). We call this approach the *idealized model of cryptography*, which in addition to restricting the adversary capability in computing terms using only cryptographic function symbols, it also idealizes the internal operations of the RoT, assuming they are "perfect." We provide a comparison of the assumptions considered in the computational, symbolic and idealized models Table 1.

Thus using this approach, one has to prove, or assume two facts. First, the particular RoT under analysis implements securely those high-level functionalities that are part of a certain application or service. This can be proved under

Computational	**Symbolic**	**Idealized**
Messages are bitstrings, and the cryptographic primitives are functions from bitstrings to bitstrings.	Messages are terms in an algebra on cryptographic primitives defined as function symbols.	Messages are terms in an algebra, on exposed cryptographic primitives defined as function symbols.
The adversary is any probabilistic Turing machine with a running time polynomial in a security parameter.	The adversary is restricted to compute only using these primitives.	The adversary is restricted to compute only using these primitives, and to only interact with the RoT through its interface.
Cryptography is implemented securely with overwelming probability in the security parameter.	Cryptography is implemented securely.	Cryptography is implemented securely. Cryptographic operations of the RoT that are not exposed to the application are assumed to be secure.

Table 1. Assumptions considered in the three models.

some model of cryptography which might require significant effort, but this effort will be required to be done only once. Second, that the system is secure when we use an idealized version of the RoT instead of the real device. This task indeed cannot be reused, and needs to be done for each application or service considered. Idealizing cryptography used internally in a RoT allows to carry out an analysis of the cryptography that is relevant to the application itself more concisely, where this analysis can be supported by the use of a more broadly accessible set of tools than those used so far to analyze such applications. Further, it allows to address more complex protocols and larger use cases and compositions of Systems-of-Systems with current formal verification technologies.

The overall overview of our methodology is as follows:

i. Identify and select the subset of the RoT functionalities that apply to the service.
ii. Obtain an idealized model, and identify the best approach to model them. Assume or prove security of the idealized functionalities.
iii. Model the application-specific scenario using the idealized device.
iv. Define and model the set of security properties that we want to consider.

The strategy of idealizing as much of the cryptography as possible should make the task of proving that an application or a system is secure, for some specific notion of security, more manageable. Therefore, simplifying the RoT internal cryptographic components is a worthy consideration. To illustrate our methodology, we focus on an attestation protocol based on TPMs. We describe the complete scenario in Sec. 4 and we instantiate our methodology in Sec. 5.

4 Remote Attestation Using a PCA

The goal of the TPM-based remote attestation protocol presented in this section is to establish a secure communication channel between a Client and a Service Provider (SP), while ensuring the trust status of the Client. The protocol is based on [13], and it is a generic version of a network management protocol presented in [8,9]. There are four devices that participate in the protocol: the Client, a TPM embedded in the Client, the PCA Server, and the SP. See Fig. 1.

Notation. For an entity A, we denote A_{pub} and A_{priv} its public and private keys, respectively. We denote $\text{cert}_A(x)$ a certificate for object x issued by entity A, and $\text{sign}_A(y)$ a signature of y using private key A_{priv}. Also, for a TPM key k we denote k_{pub} and k_{priv} its public and private key parts, k_{name} its name, $k_{\text{authPolicy}}$ its EA policy, and k_{h} its TPM object handle [24, §15]. For clarity of exposition, we omit session management objects in the description of the protocol and command parameters.

First of all, the Client receives the PCA certificate, signed with the SP key and initializes the TPM by extending the PCR to the current firmware/software values. In order to achieve the establishment of a secure communication channel between the Client and the SP the following phases are executed: (1) The Client creates an AK using the TPM and this AK is certified by the PCA. (2) the Client creates a TLS key using the TPM, which is certified by the PCA and signed by the SP. (3) Finally, the Client uses the TLS key to establish a secure communication channel with the SP, for exchanging encrypted messages and attests its status. The three phases are described in detail below.

Phase 1: Creation and Certification of AK. In the first step, the Client creates an AK via the TPM, which is certified by the PCA in a similar way as in [13]. This initial AK is not bound to a PCR state through EA, because authorization for the AK will be accomplished with the certificate that will be issued by the PCA. The AK certification sub-protocol works as follows. Upon receiving the EK, the AK and a desired fully-qualified domain name (FQDN), the PCA creates a random *challenge* and protects it using the TPM command $\texttt{TPM2\_MakeCredential}(ek_{\text{pub}}; challenge; ak_{\text{name}})$, which outputs the values:

- *secret*: random *seed* encrypted with ek_{pub},
- *credBlob*: encrypt-then-MAC of the *challenge* using keys derived from *secret*.

The Client receives this pair of values and uses the TPM to execute:

$$\texttt{TPM2\_ActivateCredential}(ek_{\text{h}}; ak_{\text{h}}; credBlob; secret)$$

which outputs the *challenge* after performing some validation checks. Then, the Client sends it to the PCA. If the *challenge* matches, the PCA is convinced that the AK and EK are bound to a legitimate TPM and issues the AK certificate $\text{cert}_{\text{PCA}}(hak_{\text{pub}}; fqdni)$. See Fig. 1(a).

Phase 2. Creation and certification of the TLS key. In this phase, the Client creates a TLS signing key using the TPM, which is bound to PCR state through EA. At this point, the PCR have already been extended with appropriate firmware/software measurements using a succession of `TPM2_PCRExtend`s.

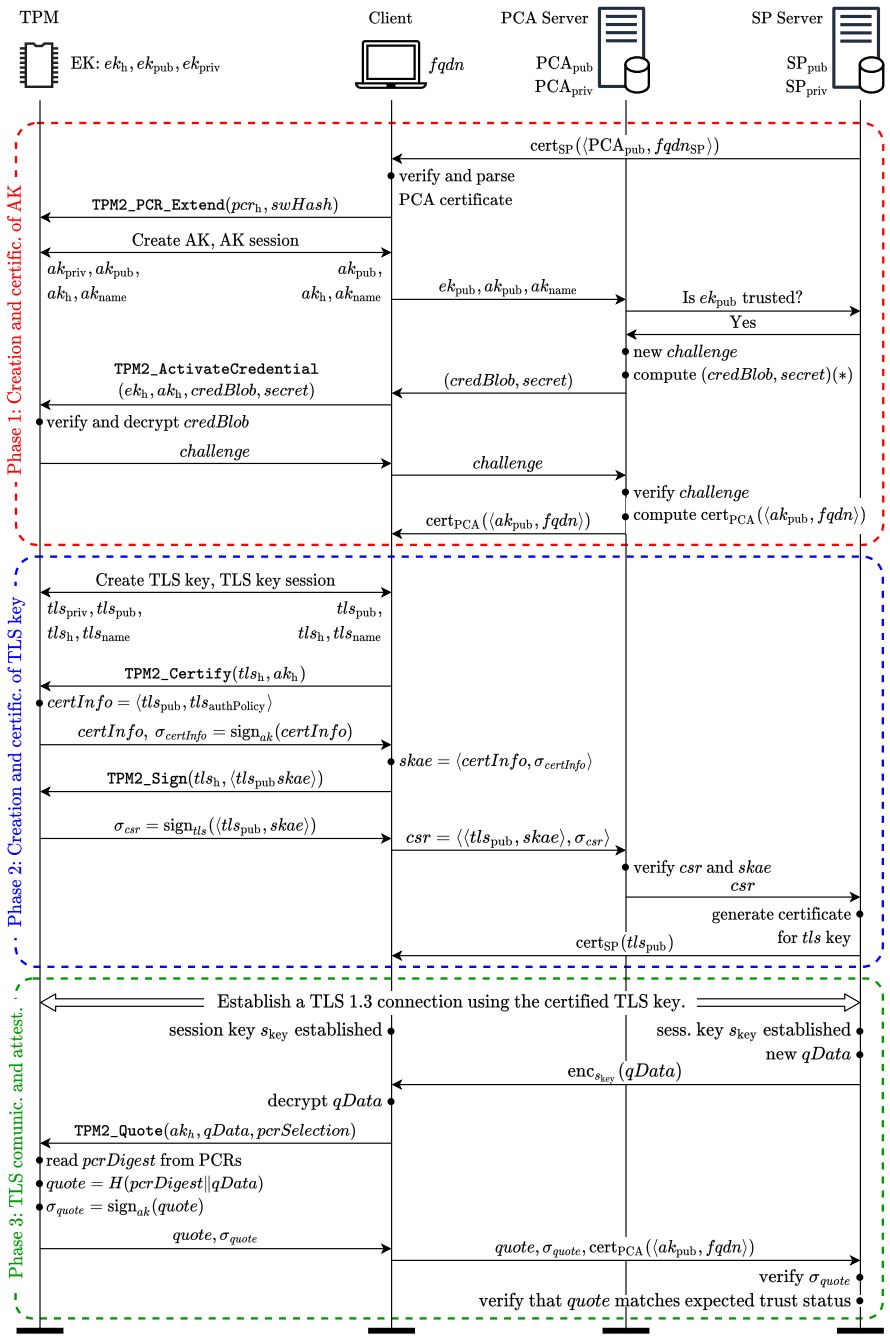

Fig. 1. Remote attestation protocol using a PCA. (*) The PCA can compute *credBlob* and *secret* on its own, or by calling the convenience command TPM2_MakeCredential(ek_{pub}; *challenge*; ak_{name}) on a local TPM.

Let $tls_{priv}; tls_{pub}$ be the generated TLS key pair. The Client executes the command $\texttt{TPM2\_Certify}(tls_h; ak_h)$, which attests the TLS key, in order to vouch that it is protected by a genuine, but unidentified TPM. This attestation is a signature $_{certInfo}$ by the AK over information that describes the key, which we abstract as the tuple $certInfo = \langle tls_{pub}; tls_{authPolicy} \rangle$. The signed tuple, plus some additional information, is known as Subject Key Attestation Evidence (SKAE), and it is an X.509 certificate extension [27], $skae = \langle certInfo; \ _{certInfo} \rangle$.

Next, the Client creates a Certification Signing Request (CSR) for the TLS key. The CSR is composed of the message $\langle tls_{pub}; skae \rangle$ and the signature on this message with the private part of the TLS key. This is done using the command:

$$_{csr} = \texttt{TPM2\_Sign}(tls_h; \langle tls_{pub}; skae \rangle):$$

The value $csr = \langle \langle tls_{pub}; skae \rangle; \ _{csr} \rangle$ is sent to the PCA, which verifies both csr and $skae$ signatures, using tls_{pub} and the AK certificate, and forwards CSR to the SP. The SP issues $\text{cert}_{SP}(tls_{pub})$, which is sent to the Client. See Fig. 1(b).

Phase 3. TLS communication and device attestation. The last step is where the secure communication channel between the Client and the SP is established. The Client, and the SP execute the TLS 1.3 protocol [20] using the TLS key that was created in the previous step, to establish a common symmetric encryption session key s_{key}. The SP encrypts some random data $qData$ with s_{key} and sends the ciphertext to the Client. Upon receipt, the Client decrypts the ciphertext obtaining $qData$. For the attestation part, the Client executes:

$$(quote; signature) = \texttt{TPM2\_Quote}(ak_h; qData; pcrSelection);$$

to quote the PCR referenced in $pcrSelection$, where $quote = H(pcrDigest \ k \ qData)$ and $signature = \text{sign}_{ak_{priv}}(quote)$. The pair $(quote; signature)$ and the AK certificate are sent to the SP, who verifies the signature and the PCR values that reflect the trust status of the platform. See Fig. 1(c).

5 Application of the Modelling Methodology

In this section we apply the modelling methodology, specifically Steps i.–iii., discussed in Sec. 3, to the use case presented in Sec. 4. We address the verification of security properties (Step iv.) in Sec. 6. Once again, we recall that the objective is to come up with high-level abstractions of the relevant RoT commands that will allow us to model those functionalities that do not provide direct cryptographic operations to the application.

Adversarial model. We consider the usual Dolev-Yao model [11] for an adversary sitting between the Client and the two servers (PCA, SP). This is because we assume that there is a trusted infrastructure supporting the interactions between the two servers and it is not a viable target for the adversary. Consequently, we consider three processes in our model, the Client, the TPM and the Server, representing both the PCA and the SP, and the adversary is allowed to

monitor and interfere in the communication between the Client and the Server. However, the communication between the Client and the TPM is treated independently. More precisely, we allow the adversary a certain degree of control over the communication channel between the Client and the TPM. For example, the adversary can behave as a passive adversary, in order to capture more relaxed trust assumptions, such as the case where the Client has malware installed.

i. Identify and select the subset of the RoT functionalities. Since the focus is on attestation services, we concentrate on a specific subset of commands that are common in most reference scenarios and application domains in relation with this functionality. Such commands are related to object creation, authorization sessions, EA, PCR extension, remote attestation, and cryptographic operations. Concretely, the commands that we have considered are:

- Key and session management: `TPM2_StartAuthSession`, `TPM2_PolicyPCR`, `TPM2_PCRExtend`, `TPM2_Create`
- Attestation: `TPM2_MakeCredential`, `TPM2_ActivateCredential`, `TPM2_Sign`, `TPM2_Certify`, `TPM2_Quote`

These commands provide TPM functionalities that are crucial in establishing chains of trust in generic heterogeneous environments. Our intuition is that the models that we will describe for these commands will serve as both a basis for reasoning about the security of a wide set of TPM-based applications and for reasoning about the security of the TPM's mechanisms themselves.

ii. Obtain an idealized model. Our models for the TPM commands mentioned above are presented in Fig. 2 using SAPiC syntax (see Appendix B). The cryptographic operations that we need to abstract are the creation of hash chains, which are used for updating PCR and policy digest values, secure storage, as well as object and session management in general. We describe our modelling key points to achieve this.

- **Modelling hash chains.** PCR values are the results of measurements of a platform or software state. When the command $\texttt{TPM2\_PCRExtend}(pcr_{\mathrm{h}}; v)$ is executed, the TPM extends the contents of the PCR referenced by pcr_{h} as $pcrDigest = H(pcrDigest\ k\ v)$. Applying `TPM2_PCRExtend` iteratively, a single digest is obtained, representing the sequence of measurements of the platform state. An object can be *sealed* to a PCR value, meaning that it can only be accessed when the PCR has been appropriately extended to that particular value. As such, the actual semantics to get access to the object is performed by comparing a digest value stored in the object blob with a PCR hash value. However, the intended semantics is that the object can only be accessed if a certain number of PCR extensions have been executed with corresponding specified values. Therefore, in our idealized approach, we need to find an alternative description for the PCR extension that captures that intended semantics. The idea is to keep a record of all PCR extensions in a list pcr_{list} as global, mutable state, and append each new digest provided by the Caller to this list. More concretely, the idealization of the process of

```
let StartAuthSession =
  in('TPM2_StartAuthSession');
  new ~s_h;
  lock 'device';
  insert ⟨'policyDigest', ~s_h⟩, nil;
  out(~s_h);
  unlock 'device'
```

```
let Create =
  in(⟨'TPM2_Create', authPolicy⟩);
  new ~k_h;
  new ~k_priv;
  lock 'device';
  let k_pub = pk(~k_priv) in
  insert ⟨'authPolicy', ~k_h⟩, authPolicy;
  insert ⟨'privPart', ~k_h⟩, k_priv;
  insert ⟨'pubPart', ~k_h⟩, k_pub;
  out(⟨~k_h, k_pub⟩);
  unlock 'device'
```

```
let Sign =
  in(⟨'TPM2_Sign', k_sh, k_h, m⟩);
  lock 'device';
  lookup ⟨'policyDigest', k_sh⟩ as k_pd in
  lookup ⟨'authPolicy', k_h⟩ as k_ap in
  if k_pd = k_ap then
    lookup ⟨'privPart', k_h⟩ as k_priv in
    out(sign_{k_priv}(m));
    unlock 'device'
  else
    unlock 'device'
```

```
let Certify =
  in(⟨'TPM2_Certify', k_sh, k_h, ak_h⟩);
  lock 'device';
  lookup ⟨'policyDigest', k_sh⟩ as k_pd in
  lookup ⟨'authPolicy', k_h⟩ as k_ap in
  if k_pd = k_ap then
    lookup ⟨'pubPart', k_h⟩ as k_pub in
    lookup ⟨'privPart', ak_h⟩ as ak_priv in
    out(sign_{ak_priv}(⟨k_pub, k_ap⟩));
    unlock 'device'
  else
    unlock 'device'
```

```
let PCR_Extend =
  in(⟨'TPM2_PCR_Extend', value⟩);
  lock 'device';
  lookup 'PCR' as pcr_list in
  insert 'PCR', ⟨value, pcr_list⟩;
  unlock 'device'
```

```
let PolicyPCR =
  in(⟨'TPM2_PolicyPCR', s_h⟩);
  lock 'device';
  lookup 'PCR' as pcr_list in
  lookup ⟨'policyDigest', s_h⟩ as pd_list in
  insert ⟨'policyDigest', s_h⟩,
      ⟨pcr_list, 'TPM_CC_PolicyPCR', pd_list⟩;
  unlock 'device'
```

```
let ActivateCredential =
  in(⟨'TPM2_ActivateCredential', k_sh, k_h, ak_sh, ak_h, credBlob⟩);
  lock 'device';
  lookup ⟨'policyDigest', ak_sh⟩ as ak_pd in
  lookup ⟨'authPolicy', ak_h⟩ as ak_ap in
  if ak_pd = ak_ap then
    lookup ⟨'pubPart', ak_h⟩ as a_pub in
    lookup ⟨'privPart', k_h⟩ as k_priv in
    if verifyCredential(ak_pub, k_priv, credBlob) = true then
      let challenge = activateCredential(ak_pub, k_priv, credBlob) in
      out(challenge);
      unlock 'device'
    else
      unlock 'device'
  else
    unlock 'device'
```

```
let Quote =
  in(⟨'TPM2_Quote', k_sh, k_h, qData⟩)
  lock 'device';
  lookup 'PCR' as pcr_list in
  lookup ⟨'policyDigest', k_sh⟩ as k_pd in
  lookup ⟨'authPolicy', k_h⟩ as k_ap in
  if k_pd = k_ap then
    lookup ⟨'privPart', k_h⟩ as k_priv in
    out(sign_{k_priv}(⟨qData, pcr_list⟩));
    unlock 'device'
  else
    unlock 'device'
```

Fig. 2. Idealized TPM commands (sketch). See Appendix B for an overview of SAPiC syntax.

extending a value v_n to pcr_{list}, translates to:

$$\text{append}(v_n; pcr_{list}) \;)\; pcr_{list} = (\underbrace{v_1; v_2; \ldots; v_{n-1}}_{\text{previous } pcr_{list}}; v_n)$$

Then, the adversary cannot gain access to a TPM object that is sealed to PCR, unless he extends the pcr_{list} with the corresponding values in the intended order, and not only any combination of extensions leading to the same hash value. We apply the same idea to the case of policy digests, which consists of hash chains similar to PCRs.

– **Usage of equational theories.** We model cryptographic equational theories as usual, e.g., for the signature generation and verification we use the Tamarin built-in function symbols `pk/1`, `sign/2` and `verify/3` satisfying

$\mathtt{verify}(\mathtt{sign}(m; k); m; \mathit{pk}(k)) = \mathit{true}$. The command `TPM2_MakeCredential`, discussed above, is used to protect a randomly generated value *challenge*, with (a variation of) the encrypt-then-MAC scheme. The seed for the symmetric keys is protected with the public part of the EK. Note that this command does not make use of any TPM protected object, and it relies entirely on public components. Therefore we do not need to model the TPM side for this command. On the other hand, `TPM2_ActivateCredential` will retrieve the value *challenge* after being provided with the appropriate private keys and execute some internal checks. These commands can be abstracted by the equational theory with function symbols `pk/1`, `makeCredential/3`, `activateCredential/3` and `verifyCredential/3`. For an object name n and a private key k it must be satisfied that:

$$\mathtt{verifyCredential}(n; k; \mathtt{makeCredential}(\mathtt{pk}(k); \mathit{challenge}; n)) = \mathrm{true}$$
$$\mathtt{activateCredential}(n; k; \mathtt{makeCredential}(\mathtt{pk}(k); \mathit{challenge}; n)) = \mathit{challenge}.$$

The first equation verifies the correctness of *challenge*, hence emulates the HMAC operation. The second retrieves the *challenge*, which models the symmetric decryption process.

– **Secure storage and object management.** As a resource-constrained device, the TPM offers the possibility of offloading memory contents into the host in the form of encrypted blob. The object can then be loaded again if appropriate authorization is provided. We idealize this functionality by assuming that the TPM has unlimited memory space. This is achieved by using as many memory cells as objects or sessions are required. Therefore, we do not model the command `TPM2_Load`, and assume that objects are readily available through their handles after invoking `TPM2_Create`. In a real scenario a user might want to offload memory objects (e.g., when the TPM is switched off). But if a security property holds without offloading memory objects it will also hold when these objects are offloaded, since the adversary will have access to less objects. The communication to/from the TPM needs to be carefully addressed, since if this channel is made fully available to the adversary it would be an overestimation of his capabilities: he could detach, interact with and reattach the TPM at his will at any time. Therefore we need to implement access control (e.g., through private channels) when using some critical commands. This is discussed below.

iii. Model the application-specific scenario using the idealized device. As commented above, in our model we aggregate the PCA and the SP servers depicted in Fig. 1 as a single Server process. The main purpose is to capture the communication between the three processes (Client, TPM, Server) in a way that replicates the real-world interactions and modes of operation, to the best possible extent, according to the proposed adversarial model. Modelling the channel between the Client and the TPM as a "standard" private channel in applied pi-calculus caused various technical difficulties for the tool in completing the proofs. For this reason, we have considered several strategies in SAPiC that allow the adversary to observe, but not interfere in that channel:

```
let Client = (//sending TPM command
let tpm_send_cmd =
    ⟨CmdCode; param_1; ::: param_k⟩ in
event TPM_SendCmd(tpm_send_cmd);
out(tpm_send_cmd);
P)
```

```
let TPM = (//receiving TPM command
let tpm_recv_cmd =
    ⟨CmdCode; param_1; ::: param_k⟩ in
in(tpm_recv_cmd);
event TPM_RecvCmd(tpm_recv_cmd);
Q) //TPM processess command here.
```

Fig. 3. Tamarin restrictions for sending/receiving TPM commands

– **Tamarin restrictions.** Used in order to limit the capabilities of the adversary. We use the templates for sending/receiving TPM commands depicted in Fig. 3, and the following restriction on the execution traces:

$$\begin{aligned} &\forall c, i.\ \text{TPM\_RecvCmd}(c)@i && \text{(RestrictionTPMCommand)}\\ &\quad \Rightarrow ((\exists j.\ \text{TPM\_SendCmd}(c)@j \wedge (j < i))\\ &\qquad \wedge \neg(\exists k.\ \text{TPM\_RecvCmd}(c)@k \wedge \neg(k = i))) \end{aligned}$$

This ensures that in order for a TPM to receive and process a message, an (injective) TPM call must have been executed by the Client process. This forbids an external adversary from calling the TPM arbitrarily, but it allows an internal (e.g., malware) adversary access to the TPM.

– **Usage of the public channel.** Whenever it is possible, we output the response of the TPM to the public channel (e.g., when it produces public keys). To some extent, this overestimates the capabilities of the adversary, as it assumes that he can listen to the communication channel between the TPM and the Client. Still, the adversary does not have access to internal secrets such as the EK or private parts of the objects. That is, in most cases, we can assume that the output of the TPM is available to the adversary to prove the security properties we are interested in.
– **Direct usage of multiset rewrite rules.** The SAPiC calculus has an advanced feature which allows direct access to the multiset rewrite system of Tamarin. In order to emulate asynchronous message transfers between the Client and the TPM, we perform the following updates in Fig. 3:

replace out(tpm_send_cmd) with [ ]—[ ] !! [CmdNameIn(tpm_send_cmd)]
replace in(tpm_recv_cmd) with [CmdNameIn(tpm_send_cmd)]—[ ] !! [ ]

The state fact CmdNameIn(tpm_send_cmd) is produced by the Client, consumed by the TPM process, and it is not available to the adversary at any time point. A similar approach can be devised for the TPM outputs, i.e., the TPM produces a state fact that is consumed by the Client process.

Source code. Our model is split in three parts, corresponding to the three phases of the protocol from Sec. 4, namely AK certification, TLS certification and attestation. The SAPiC models are available at [12].

6 Verification

We address our last step (Step. iv.) by describing a number of high-level security properties, defined for the protocol that is presented in Sec. 4. These properties are expressed as first-order logic formulas using Tamarin lemmas [4] and they focus on achieving integrity, confidentiality and attestation in the different phases. The notation "Ev($p_1;:::;p_n$)@t" below stands for an execution of event fact with name "Ev" with parameters $p_1;:::;p_n$ executed at time t.

Sanity-check properties: We model a number of properties to guarantee the existence of execution traces that reach the end of each possible branch in the protocol. Whereas these reachability properties do not encode any security guarantee in particular, they are required in order to ensure a non-trivial verification of the remaining correspondence properties. That is, a property of the form Event2) Event1 will be trivially satisfied if Event2 is never reached. Therefore, for a given party A we define a number of events AFinish_i(), where i ranges over the number of branches of the process that defines party A. We then define the following collections of exists-trace lemmas (one for each branch):

$$9t_1: \text{AFinish\_}i()@t_1: \qquad \text{(Reachability)}$$

Availability of keys at honest processes: We define a number of properties that ensure that all honest parties have initial access to the trusted key material required, so that they can build a chain of trust and successfully complete each phase of the protocol. This property can be treated as a premise in our models. We define the event HasKey($label;pid;k$), where $label$ is the key identifier (e.g., 'EK', 'AK'), pid is a unique process identifier, and k is the key value. This event is launched at the start of the process representing each principal.

$$8label;pid_1;pid_2;k_1;k_2;t_1;t_2: (pid_1 \neq pid_2) \wedge \text{HasKey}(label;pid_1;k_1)@t_1 \wedge \text{HasKey}(label;pid_2;k_2)@t_2 \ ) \ (k_1 = k_2): \qquad \text{(KeyAvailability)}$$

Key freshness: This property ensures that the created key material is not reused as a new key during the execution of the protocol. This applies both to asymmetric keys (e.g., AK public and private parts), as well as to symmetric keys (e.g., session keys). We define the event GenerateKey($label;tid;k$), where $label$ is the key identifier, tid stands for thread execution of the principal process, and k is the value of the generated key. The property is captured by the lemma:

$$8label;tid_1;tid_2;k;t_1;t_2: \text{GenerateKey}(label;tid_1;k)@t_1 \wedge \text{GenerateKey}(label;tid_2;k)@t_2 \ ) \ (tid_1 = tid_2): \qquad \text{(KeyFreshness)}$$

Key secrecy: This property ensures that the sensitive key material, namely private or symmetric session keys, is not available to the adversary. Similarly, we define the lemma SecretKey($label;k$), where $label$ is the key identifier and k represents its value. This event is launched after the corresponding key material has been created in each processes. This is modeled through the lemma:

$$8label;k;t_1: \text{SecretKey}(label;k)@t_1 \ ) \ : (9t_2: K(k)@t_2); \qquad \text{(KeySecrecy)}$$

where $K(:::)$ is the event fact denoting adversary knowledge.

Authentication: We consider the agreement property from Lowe's hierarchy [18] in the parameters exchanged in each phase of the protocol. Whenever it is possible (e.g., for a session key), we also require injective, mutual agreement. As customary, we encode this property through the usage of the events ARunning($id_A; id_B; pars$) and BCommit($id_B; id_A; params$), where the former is placed on the party A being authenticated, and the latter is placed in party B, to whom the authentication is being made. The placement of these events has some flexibility, but not all placements are correct. The ARunning event must be placed *before* party A sends its last message, and the BCommit event must be placed *after* party B receives and verifies the last message sent by A. The variable $pars$ capture the set of parameters which are being agreed. The corresponding lemma has the following form in its injective version:

$$\begin{aligned} &\forall id_A; id_B; pars; t_1 : \text{BCommit}(id_B; id_A; pars)@t_1 \;) \\ &\quad ((\exists t_2 : \text{ARunning}(id_A; id_B; pars)@t_2 \wedge (t_2 < t_1)) \wedge \\ &\quad \neg(\exists id'_B; id'_A; t_3 : \text{BCommit}(id'_B; id'_A; pars)@t_3 \wedge \neg(t_3 = t_1))) : \end{aligned} \quad \text{(Agreement)}$$

In case the agreement is mutual, we require an additional lemma where the roles of A and B are reversed.

Transfer of information as generated: Such lemmas ensure that infomation generated at different stages in the protocol, such as certificates or attestation reports, are received at the destination process as generated by the process of origin. We model this property through the events GenerateValue($label; v$) and ReceiveValue($label; v$), executed at the sender and receiver side respectively:

$$\begin{aligned} &\forall label; v; t_1; t_2 : \text{ReceiveValue}(label; v)@t_1 \;) \\ &\qquad (\exists t_2 : \text{GenerateValue}(label; v)@t_2 \wedge (t_2 < t_1)) : \end{aligned} \quad \text{(CorrectTransfer)}$$

No reuse of key: This property ensures that a specific key is used only once in its intended context. We consider this property mainly used for single-use session keys. We therefore define the event UseKey($label; k$) and lemma

$$\forall label; k; t_1; t_2 : \text{UseKey}(label; k)@t_1 \wedge \text{UseKey}(label; k)@t_2 \;) \; (t_1 = t_2) : \quad \text{(NoKeyReuse)}$$

No attestation of corrupted Client: This property signifies that a Client that is in a corrupted (untrusted) state will not be able to perform a successful attestation. Note that the PCR value of the Client's TPM will be extended to an expected reference value only if its configuration is in a trusted state. SAPiC offers a non-deterministic branching construct which we use to model the fact that the Client might be in a trusted or untrusted state, but the remaining parties in the model do not know a priori which one. We define the event Corrupted(id_A), which is launched at the beginning of the corrupted branch,

	Phase 1		Phase 2		Phase 3	
Property	Objects	Steps	Objects	Steps	Objects	Steps
Reachability	–	37	–	103	–	18
KeyAvailability	EK, AK, PCA_{pub}	1378	AK, TLS, SP_{pub}	537	AK, SP_{pub}	24
KeyFreshness	AK	4	TLS	4	s_{key}; *qData*	4
KeySecrecy	EK, AK, PCA_{priv}	12	TLS_{priv}, SP_{priv}	10	AK, s_{key}	6
Agreement	$cert_{PCA}$(AK), *challenge*	403	AK, TLS, *swHash*	2987	s_{key} *qData*	2237
CorrectTransfer	$cert_{PCA}$(AK)	383	CSR	406	*quote*	9
NoKeyReuse	n/a	n/a	n/a	n/a	s_{key}	8
Corrupted	n/a	n/a	n/a	n/a	–	1699

Table 2. Summary of verified properties in SAPiC/Tamarin

and reuse the event ServerCommit($id_B; id_A; pars$), launched after a successful verification of the Client attestation report. We express this property as follows

$$\forall id_A; t_1 : \text{Corrupted}(id_A)@t_1 \;) \\ : (\exists id_B; pars; t_2 : \text{ServerCommit}(id_B; id_A; pars)@t_2 \wedge : (t_1 < t_2))) : \\ \text{(Corrupted)}$$

Table 2 summarizes the results of our analysis using SAPiC/Tamarin and the objects to which each property is related, and the number of steps to prove the corresponding lemmas. The simulations have been executed in a VM 3 cores, 4GB RAM on Intel(R) Core(TM) i5-4570 @ 3.20GHz. The tool has been able to successfully verify them as expected, showing the feasibility of our approach in abstracting the functionalities of the TPM as a RoT for reporting.

7 Conclusion

In this paper, we introduce a new formal verification methodology, in which our focus is to idealize the internal functionalities of the RoT in such a way that we exclude the cryptographic actions carried out by the RoT and replace them with non-cryptographic approaches. This idealized model of cryptography allows for a more effective verification process, especially when complex protocols and extensive use case scenarios are considered. We formalized the notion of secure remote attestation towards trust aware service graph chains, in "Systems-of-Systems", and verified Tamarin security proofs showing that our models satisfy the three key security properties that entail secure remote attestation and execution: integrity, confidentiality, and secure measurement. Furthermore, in order to model this service, we also considered additional TPM processes such as the creation of TPM keys, the Enhanced Authorization (EA) mechanism, the management of the Platform Configuration Registers (PCRs), and the creation and management of policy sesssions. We argue that the included TPM commands cover a wide range of TPM-based applications and hence they can serve as a baseline for modelling additional TPM functionalities, in various application domains in the literature. Finally, we believe that our methodology of idealizing the cryptography can be used as an extensible verification methodology that enables rigorous reasoning about the security properties of a RoT in general.

References

1. Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 104–115, London (UK), January 2001. ACM.
2. Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark D Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, 2014.
3. Will Arthur and David Challener. *A practical guide to TPM 2.0: using the Trusted Platform Module in the new age of security*. Apress, 2015.
4. David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. Tamarin prover (v. 1.6.0), September 2020. https://tamarin-prover.github.io/.
5. Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick D. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145. ACM, 2004.
6. Liqun Chen and Mark Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *International Workshop on Formal Aspects in Security and Trust*, volume 5983 of *LNCS*, pages 201–216, Eindhoven, The Netherlands, November 2009. Springer.
7. Vincent Cheval, Véronique Cortier, and Mathieu Turuani. A little more conversation, a little less action, a lot more satisfaction: Global states in ProVerif. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 344–358. IEEE, 2018.
8. The FutureTPM Consortium. FutureTPM use case and system requirements. Deliverable D1.1, FutureTPM, June 2018.
9. The FutureTPM Consortium. Demonstrators implementation report – first release. Deliverable D6.3, FutureTPM, April 2020.
10. Stephanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel. Formal Analysis of Protocols Based on TPM State Registers. In *IEEE Computer Security Foundations Symposium*, pages 66–80. IEEE, Jun 2011.
11. Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.
12. Anonymized for peer review. Repository for SAPiC/Tamarin models. https://drive.google.com/drive/folders/1OJAkL6QF6Ulm7b2maxga-anBO4cX1xHI, 2021.
13. Ken Goldman. Attestation Protocols. Technical report, IBM, December 2017. https://www.ibm.com/developerworks/library/l-trusted-boot-openPOWER-trs/index.html.
14. Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Proc. ACM Workshop on Scalable Trusted Computing (STC)*, page 21–24, Alexandria, VA, November 2006. ACM.
15. James Greene. Intel Trusted Execution Technology: Hardware-based Technology for Enhancing Server Platform Security, 2013. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf.
16. Zhou Hongwei, Ke Zhipeng, Zhang Yuchen, Wu Dangyang, and Yuan Jinhui. TSGX: Defeating SGX Side Channel Attack with Support of TPM. In *Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, pages 22–24. IEEE, April 2021.

17. Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. *Journal of Computer Security*, 24(5):583–616, 2016.
18. Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.
19. Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on CAV*, pages 696–701. Springer, 2013.
20. Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.
21. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium (USENIX Security)*, San Diego, CA, August 2004. USENIX Association.
22. Jianxiong Shao, Yu Qin, and Dengguo Feng. Formal analysis of HMAC authorisation in the TPM2.0 specification. *IET Information Security*, 12(2):133–140, March 2018.
23. Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. Formal analysis of enhanced authorization in the TPM 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 273–284. ACM, 2015.
24. Trusted Computing Group (TCG). TPM 2.0 library specification - part 1: Architecture. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf.
25. Trusted Computing Group (TCG). TPM 2.0 library specification - part 2: Structures. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part2_Structures_pub.pdf.
26. Trusted Computing Group (TCG). TPM 2.0 library specification - part 3: Commands - code. Available at: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_code_pub.pdf.
27. Trusted Computing Group (TCG). TCG Infrastructure Workgroup Subject Key Attestation Evidence Extension. Available at: https://trustedcomputinggroup.org/wp-content/uploads/IWG_SKAE_Extension_1-00.pdf.
28. Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proc. ACM Conf. Data and Appl. Security Privacy (CODASPY)*, pages 261–268, New York, NY, USA, Mar 2017. ACM.
29. Stephan Wesemeyer, Christopher J. P. Newton, Helen Treharne, Liqun Chen, Ralf Sasse, and Jorden Whitefield. Formal analysis and implementation of a TPM 2.0-based direct anonymous attestation scheme. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 784–798. ACM, 2020.
30. Li Xi and Dengguo Feng. Formal Analysis of DAA-Related APIs in TPM 2.0. In *Network and System Security*, pages 421–434, Cham, Switzerland, Nov 2015. Springer.

A Create a TPM key with PCR policy

We show a simplified example on the usage of EA. We note that, for clarity, we are omitting many details on TPM internals and TPM objects; see [24,25,26,3].

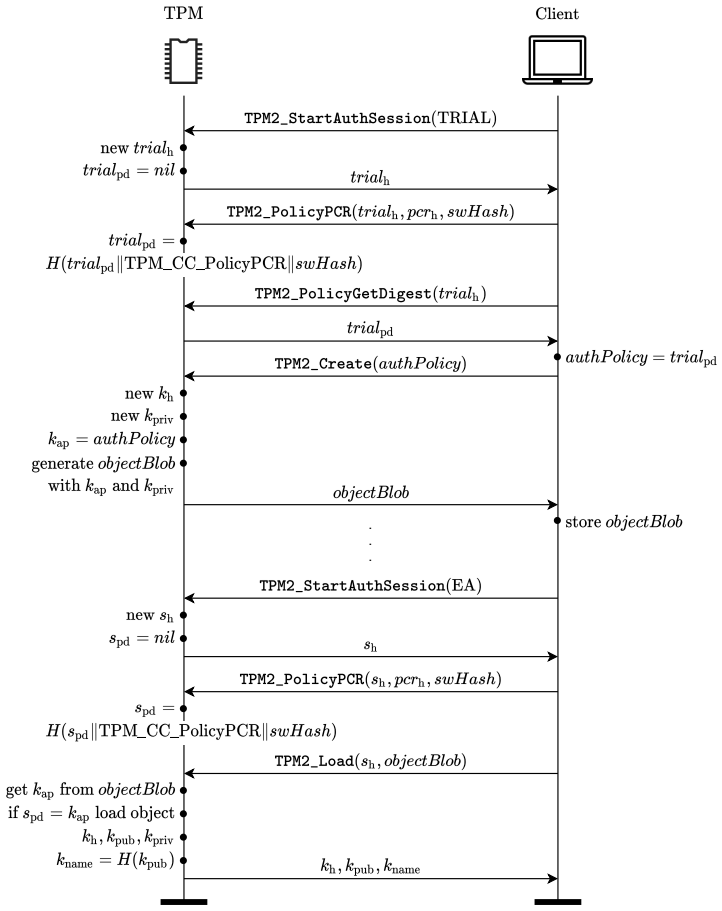

Fig. 4. Create a TPM key with PCR policy

In brief, the user execute the command TPM2_StartAuthSession(TRIAL) in order to initiate a fresh trial session. The TPM creates a handle $trial_{\mathrm{sh}}$ for this session, initiates the policy digest $trial_{\mathrm{pd}}$ to zero and returns $trial_{\mathrm{sh}}$ to the user. The user executes the command TPM2_PolicyPCR($trial_{\mathrm{sh}}; pcr_{\mathrm{h}}; swHash$), which asks the TPM to update the policy digest of the trial session as:

$$trial_{\mathrm{pd}} = H(trial_{\mathrm{pd}} k \text{'TPM\_CC\_PolicyPCR'} k swHash);$$

where $H(\,)$ is a has function. The user executes TPM2_PolicyGetDigest($trial_{\mathrm{sh}}$) in order to obtain $trial_{\mathrm{pd}}$ and calls TPM2_Create($trial_{pd}$). The TPM will create a new object (key) $k = (key_{priv}; key_{pub})$ with authorization policy $key_{\mathrm{ap}} = trial_{\mathrm{pd}}$ and returns to the user the protected object blob. Now the user creates a policy session s with TPM2_StartAuthSession(EA) and the TPM sets spd = nil and returns the handle s_{h} to the Client. The Client executes TPM2_PolicyPCR($\mathsf{s}_{\mathrm{h}}; pcr_h; swHash$)

in order to update the policy digest s_{pd}, and then TPM2_Load(s_{h};*objectBlob*) and obtains $k_{\text{h}}; k_{\text{pub}}; k_{\text{name}}$. This procedure is summarized in Fig. 4.

B SAPiC Syntax

Fig. 5 describes the SAPiC syntax. The syntax allows to define a protocol as a process. It is then translated into a set of rules that adhere to the semantics of the calculus, which is a dialect of the applied pi-calculus [1], and comprises an *order-sorted term algebra* with infinite sets of publicly known names PN, freshly generated names FN, and variables V. It also comprises a signature , i.e., a set of function symbols, each with an arity. The messages are elements of a set of terms T over PN, FN, and V, built by applying the function symbols in . Notation: $n\ 2\ FN; x\ 2\ V; M; N\ 2\ T; F\ 2\ F$. As opposed to the applied pi-calculus [1], SAPiC's input construct in$(M; N); P$ performs pattern matching instead of variable binding. See [4] for the complete details.

$\langle M; N \rangle ::= x; y; z \in \mathcal{V}$	variables
$\mid p \in PN$	public names
$\mid n \in FN$	fresh names
$\mid f(M_1; :::; M_n)$ s.t. $f \in$ of arity n	function application
$\langle P; Q \rangle ::=$	processes
$\mid 0$	terminal (null) process
$\mid P \mid Q$	parallel execution of processes P and Q
$\mid !P$	replication of process P
$\mid \ n; P$	binds n to a new fresh value in process P
$\mid$ out$(M; N); P$	outputs message N to channel M
$\mid$ in$(M; N); P$	inputs message N to channel M
$\mid$ if $P\,red$ then P [else Q]	P if predicate $P\,red$ holds; else Q
$\mid$ event $F; P \quad F \in \mathcal{F}$	executes event (action fact) F
$\mid P + Q$	non-deterministic choice
$\mid$ insert $M; N; P$	inserts N at memory cell M
$\mid$ delete $M; P$	deletes content of memory cell M
$\mid$ lookup M as x in P [else Q]	if M exists, bind it to x in P; else Q
$\mid$ lock $M; P$	gain exclusive access to cell M
$\mid$ unlock $M; P$	waive exclusive access to cell M
$\mid [L] -[A]\rightarrow [R]; P \quad (L; R; A \in \mathcal{F}^*)$	provides access to Tamarin MSRs

Fig. 5. SAPiC syntax