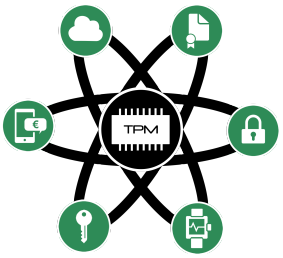

FutureTPM

D3.3

Second Report on Security Models for the TPM

Project number:	779391
Project acronym:	**FutureTPM**
Project title:	Future Proofing the Connected World: A Quantum-Resistant Trusted Platform Module
Project Start Date:	1st January, 2018
Duration:	36 months
Programme:	H2020-DS-LEIT-2017
Deliverable Type:	Report
Reference Number:	DS-LEIT-779391 / D3.3 / v1.1
Workpackage:	WP 3
Due Date:	31st December, 2019
Actual Submission Date:	3rd February, 2020
Responsible Organisation:	SUR
Editor:	K. Liang
Dissemination Level:	PU
Revision:	v1.1
Abstract:	In this report, we deliver two main parts related to the security modelling and models for the TPM. The first part is for the design of ideal functionalities and the second part is for cryptographic model.
Keywords:	TPM, model, ideal functionalities

The project FutureTPM has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779391.

Editor

K. Liang (SURREY)

Contributors (ordered according to beneficiary numbers)

Kaitai Liang, Liqun Chen (SURREY)
Anja Lehmann (IBM)
Sofianna Menesidou (UBITECH)
José Moreira (UB)
Georgios Fotiadis (UL)
Thanassis Giannetsos (DTU)

Disclaimer

Executive Summary

This deliverable reports on current progress towards designing and developing security models, general security framework and ideal functionalities for the Trusted Platform Module (TPM) and its related use cases.

Standing at the view point of cryptography for the TPM, we mainly tackle issues of security model for lattice-based Direct Anonymous Attestation (DAA) via the invention of a new property-based security model. For the security modelling, we initially and successfully design ideal functionalities for create/load scenario within TPM.

We make the previous outline approach for the definition of ideal functionalities more concrete to formally capture the security of TPM functionalities and their compositions. Our design can idealize all TPM functionalities except those that serve to provide cryptography to a consumer application, capturing their intended semantics rather than their implementation. This idealization is done by replacing cryptography with a Trusted Third party. We investigate the TPM commands and further merge them into our ideal functionalities. We present the first scenario (create/load TPM key) for our ideal functionalities design, and consider the integration of our model in the three use cases.

We report some progress towards the development of *cryptographic provably secure* definition, called property-based model. To do so, we define new symbols and oracles (concurrent and non-current versions) to cover the gap between stand-alone setting security and Universal Composability (UC) framework. We further propose a protocol which is able to convert a stand-alone secure design to be UC-secure. In this way, we show that our new design achieves all necessary features and security as the UC version does.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In D3.2 First Report on Security of the TPM, we identify a number of research challenges. In this Deliverable, we refine our research methodology on security modelling for the whole TPM and the cryptographic security notion for the DAA.

Ideal functionalities for TPM. We make the modelling approach proposed in D3.2 more concrete and taking first step to capture the security of TPM functionalities via the definition and design of the ideal functionalities. The core idea we use is to idealize the functionalities (except those serving to provide cryptography) to a consumer application via replacing cryptography used internally by the TPM by non-cryptographic approaches. The approaches are to combine the use of trusted third party, access control and private communication channels. Our design is able to allow that: the TPM functionalities can securely implement the ideal functionalities and, the system can be secure even if it leverages the ideal functionalities instead of the TPM functionalities. In other words, we equal the security of the use of ideal functionalities to the TPM functionalities. We provide general modelling framework, and concrete ideal functionalities for the first modelling scenario - create/load TPM key. We at last present some discussions on how to extend the outcomes to the three project use cases.

New cryptographic notion for DAA. In this deliverable, we introduce a new property based definition for Direct Anonymous Attestation (DAA) which is the main cryptographic functionality of TPM. In the literature, one believes that UC based model is able to capture all security notions of DAA. However, it is still unknown that how to merge the UC notion into property-based security setting. We close the gap between property-based and UC -based security of DAA by defining this new security model. In the model, we first extract four common security features and ”wrap” them into the security games with stand-alone and concurrent oracles. We further provide security analysis to show that the new model achieves the same security guarantees as the UC version in [19] via the combination of concurrent oracles and a ”wrapper” protocol.

1.1 Methodology

The above two aspects have so far been investigated and researched by independent teams, supporting fine-grained interactions with other Work Packages in the project as needed. Both of the aspects are complimentary to each other - ideal functionalities targets to non-cryptographic components while new cryptographic definition is for the DAA which is the main cryptographic

function of TPM. The reason of proposing a new cryptographic definition for DAA is because we have designed lattice-based DAA schemes which requires up-to-date security model for their security proof. Instead of delivering all-round security modelling for TPM, our strategy is to identify and model the most common and important functionalities (which are mainly related to the three use cases), for example, the ideal functionalities for TPM key create/load match the real need in the implementation of use cases, the new cryptographic definition also takes the common four features extracted from practical cases and current security definition literature. Our main philosophy in the security modelling is to "wrap" the security requirements (of TPM's functionalities) or adversarial actions (towards to TPM) into a "box" and further interact the box with "outside world". This philosophy may allow us to relate the modelling strongly with the current three use cases.

1.2 Structure of the Report

Chapter 2 overviews the modelling approach via a specific example, proposes the first modelling scenario for TPM and define the corresponding ideal functionalities. Extensions to use cases and the discussions are presented at the end. In Chapter 3, we mainly propose a new property-based model for DAA, and further prove how this new model fits the concurrent needs. We also show that the new model meets the same security as the UC DAA notion. Finally, Chapter 4 summarizes our findings, identifies remaining gaps, and scopes further research to be conducted.

Chapter 2

Security Modelling

In D3.2 [27], we outlined an approach to defining ideal functionalities for the TPM that can serve as both a foundation for the verification and validation of applications that rely on TPM for their security, and a specification of the TPM's security. This modelling approach has so far been left abstract; we now make it slightly more concrete, taking first steps towards formally capturing the security of the TPM's functionalities and their compositions.

The key feature of our model is that it idealizes all TPM functionalities except those that serve to *provide* cryptography to a consumer application (for example, hashing or asymmetric encryption), capturing their intended semantics rather than their implementation. This idealization is done by replacing cryptography with a Trusted Third Party (or a group of them), with strong access control, and with which honest parties interact through channels not visible to the adversary.[1]

Intuitively, idealizing cryptography that is used internally allows one to study the security of applications and systems that rely on the TPM by: i. checking that the TPM is used in a way that allows this idealization; and ii. analyzing only cryptography relevant to the application itself. In cases where the application uses the TPM in a way that does not allow the treatment of cryptography as ideal (for example, because it uses parameters that do not guarantee ideal security), then the model still allows reasoning about the application's security by modelling the TPM's internal cryptography in a more standard model, appropriate for the context.

Looking ahead to our example of *sealing* an encryption key to Platform Configuration Registers (PCRs), the idealization allows the analysis of the application to focus on whether the application itself uses the key securely, whilst assuming that nothing about the key leaks out (not even an encryption of it) when stored in the TPM's Protected Storage. As such, the model may in fact become useful beyond the TPM itself, since the same ideal security functionality could be realized in a different way (for example, implementing Protected Storage through actual secure storage–at an additional cost).

We also hope that this idealization will support the use–in analyzing applications that use only non-cryptographic feature of the TPM (such as protected storage and authentication session)–of tools more broadly accessible than those used so far to analyze such applications. In particular, we contend that using the TPM–or indeed any Trusted Execution Environment (TEE)–as a backend for many of the secure functionalities it offers should not require cryptographic expertise, and hope that the tools afforded by our idealization approach can be deployed more readily and with less demand on the end user than existing tools that have been used to analyze TPM functionalities in the past.

[1] These are often referred to in the symbolic literature as *private channels*.

Computational	Symbolic	Idealized
Assumptions: • Messages are bitstrings, and the cryptographic primitives are functions from bitstrings to bitstrings. • The adversary is any probabilistic Turing machine with a running time polynomial in a security parameter. • Cryptograpy is implemented securely with overwelming probability in the security parameter.	Assumptions: • Messages are terms on cryptographic primitives defined as function symbols. • The adversary is restricted to compute only using these primitives. • Cryptography is implemented securely.	Assumptions: • Messages are terms in an algebra, with exposed cryptographic primitives defined as function symbols. • The adversary is restricted to only call TPM commands. • Cryptography that is not exposed to applications is provided by a Trusted Third Party (or some equivalent model).

Table 2.1: Assumptions considered in the three models.

2.1 Overview of the Modelling Approach

In order to formally capture the intended semantics of TPM operations, the ideal functionalities replace the cryptography used internally by the TPM by non-cryptographic mechanisms–usually a combination of a Trusted Third Party, access control and private channels–wherever possible. In order to then obtain security result on a system that makes use of TPM functionalities, the following facts need to be assumed, proved or verified:

1. the TPM algorithms securely implement the ideal functionality, that is, we can substitute such ideal functionalities by the TPM without “losing too much security”; and
2. the system is “secure” when it uses the ideal functionalities instead of the TPM.

We envision that the former argument, that the TPM itself is “almost as secure” as our ideal functionalities, could be done as a cryptographic proof (in the computational model) of indistinguishability–or indifferentiability where relevant–given a specific adversary and trust model. This will be a complex argument, using techniques akin to those leveraged in Chapter 3, but would only need carried out once and for all.

By contrast, idealizing as much of the cryptography as possible without application-specific knowledge should make the second task, of proving that a specific application is “secure” for some application-specific notion of security, much more manageable, and accessible to designers and developers with less expertise. Since this task must be performed for each application or system–and perhaps for each combination of applications and systems, if they can interfere–simplifying it as much as possible is a worthy goal.

In the rest of this chapter, we focus discussions on partially defining a simplified ideal functionality, and discussing techniques and tools that could be leveraged to fulfill the second task.

2.1.1 An Example

Consider, for example, a scenario where Enhanced Authorization (EA) is used by an honest user to create a TPM-protected (non-primary) object *sealed* to a PCR state (that is, protected by a policy constructed by measuring the state of some subset of the PCRs).

In order to load this object back into the TPM after its creation, an *authorization session* object must first be constructed that also measures the state of the same subset of the PCRs. The policy object contains a digest (the *policy digest*), constructed iteratively through hash chaining the parameters of successive policy commands. When the load is performed, the constructed policy digest is compared to that stored in the object, and the load is allowed if and only if the digests are equal. As such, the load command *actual semantics* are that the object will be loaded if it is performed within an authorization session whose policy digest matches that of the loaded object.

However, the load command *intended semantics* (which we aim to capture in our *idealized semantics*, or *ideal functionality*) is that the object can only be loaded if the load command is performed in an authorization session in which the same sequence of policy commands has been performed as was performed when the object was created.

With overwhelming probability, these two semantics are equivalent (and this is indeed what an indistinguishability proof would demonstrate), but reasoning about complex systems up to overwhelming probability has so far remained impractical.

By contrast, studying the security of the same system in a setting where the TPM is assumed to follow its intended semantics would yield much easier proofs, or, when insecurity is detected, a much easier path to identifying root causes and vulnerabilities.

2.1.2 Modelling Ideal Cryptographic Functionalities

Traditional ways of modelling ideal cryptography often rely on global state under access control, in a way similar to Morris' language-based *local objects* or *sealing* [35].[2] Bengtson, Bhargavan, Fournet, Gordon and Maffeis [7] present a type-based approach to sealing (where values are sealed to a *property*, using refinement types), later extended by Fournet, Kohlweiss and Strub [31] to more complex type systems which, essentially, allow sealing of values to value-dependent properties. These type-based techniques, combined with some informally-enforced verification discipline, has been deployed to reason about complex ideal functionalities and their modular composition.

However, these techniques have not so far been applied to concepts of Trusted Computing, and the type-based approach, which is naturally closely tied to inductive structures, does not lend itself well to reasoning about the TPM's key management ecosystem, where storage hierarchies can become graphs without loss of security.

[2] We note the terminology clash, and highlight that Morris' notion of sealing is indeed quite different from the TPM concept of sealing a protect object to a PCR state. However, we also note that Morris' notion of sealing, and the notion of sealing a TPM object to a PCR state in fact achieve very similar goals, although their formal definition and the context in which they arose differ drastically.

Other formalisms and tools have been used to model properties of TPM, key management tokens and Application Programming Interfaces (APIs), and other Trusted Computing concepts.

- Approaches based on Abadi and Fournet's *applied π-calculus* [1] (including ProVerif [9]) are mature and stable, but do not provide appropriate support for non-monotonic global state,[3] as would be needed to model volatile TPM state (including PCRs, and object and session handles, for example) over TPM restarts.
- Approaches based on multiset rewrite rules (including Tamarin [3]) naturally support encodings of state, including non-monotonic updates, but only provide limited support for inductive structures.

Extensions and front-ends to tools from both approaches aim at implementing support for non-monotonic global state and more complex flows. These include StatVerif [41] and GSVerif [26], for ProVerif, and the SAPIC [36] frontend, for Tamarin.

In addition, tools from both approaches also include built-in support for some mechanisms that could be used in addition to, or instead of, type-based sealing. More specifically, ProVerif's notion of *private constructors and destructors* could be used as a native sealing mechanism to directly encode the ideas of Morris [35]. We further note that access control (a necessary component for sealing to properly capture authentication) can be enforced by scoping in the applied π-calculus.

Relation to existing models of TPM security.

We note that models of TPM security already exist in the tools discussed above, and do include the use of global states [43, 42, 41, 26]. However, none of these models differentiate between cryptography that is used internally to the TPM and cryptography that directly serves the security of the application. As a consequence, in those works, reasoning about application security requires reasoning about TPM internals, even in settings where they could be treated as opaque.

Our hope is that securely idealizing cryptography that is irrelevant to application security will eventually allow us to address much larger use cases that make use of the TPM. We first focus on smaller use cases.

2.1.3 Towards A Reference Ideal Functionality

The roadmap for implementing this model is therefore as follows

1. Identify and select the subset of TPM functionalities that we would like to idealize, focusing on the project Reference Scenarios. For the present document, we will focus on secure storage, create and load functionalities.
2. Obtain an idealized model for these functionalities, and identify the best approach to model them.
3. Define and model the set of security properties that we want to consider.

[3] We would like to remark that the applied π-calculus, indeed, *does* provide the ability to precisely model processes, including their non-monotonic changes. However, ProVerif does not, because of the abstractions it makes, so it can produce false counterexamples. Our point here is that there is no algorithmic way to prove arbitrary properties using the applied π-calculus.

4. Consider what modelling tool is best suited to provide an automated verification of these security properties.

In relation to the model of security properties, we note that the TPM Specification [44] lacks of formal models for what security properties a given functionality is intended to provide. Moreover, in many occasions, it lacks a rigurous definition, providing only some vague description which might be open to interpretation. A clear example of this occurs in [44, Section 19.7]:

> *"Enhanced authorization is a TPM capability that allows entity-creators or administrators to require specific tests or actions to be performed before an action can be completed. The specific policy is encapsulated in a value called an* authPolicy *that is associated with an entity"* [sic]

The "specific tests" and "specific actions" refers, obviously, to the assertions implemented by the policy commands, but this requires a formal refinement on the semantics of the security properties. An approach in this direction can be found in [43], where the "specific actions" and "specific tests" are formally modelled by security properties (ϕ_A and ϕ_T, respectively) through the usage of events triggered at specific points (creation and fulfillment of a policy assertion). Their model asks whether the trace of action events of a consumer application (Caller) with the TPM satisfies them. We provide here a simplified version of these security properties (see [43] for a complete description):

$$\phi_A = \forall h_S, t_0.\mathrm{UseObjectEvent}(O.\texttt{authPolicy}, h_S)@t_0 \Rightarrow \bigvee_{i=1}^{n}\bigwedge_{j=1}^{m_i}(\exists t_{i,j}.\mathrm{PolEvent}(\mathrm{AssertionLabel}_{i,j}, h_S, \mathrm{AssertionParams}_{i,j})@t_{i,j} \wedge t_{i,j} < t_{i,j-1}),$$

$$\phi_T = \phi_{T,1} \wedge \phi_{T,2} \wedge \cdots \wedge \phi_{T,k}.$$

Here, the security property ϕ_A states that a set of specific actions (e.g., signing with a specific key) must be completed ($\mathrm{PolEvent}$ triggered) before using an object ($\mathrm{UseEvent}$ triggered). These set of actions can be combined through AND and OR operations. On the other hand, the security property ϕ_T states that a set of specific tests (e.g., PCRs have a certain value) is performed before the object is used. The security properties $\phi_{T,i}$ are usually modelled as correspondence properties, and ensure that the tests required by specific assertion commands in the policy are fulfilled (e.g., ensure that the PCRs haven't been changed between session assertion and object usage). However, the approach from [43] still relies on taking into account the internal workings of the TPM cryptography, therefore further investigation needs to be done so that these properties can be abstracted from it and extended for their usage with the idealized TPM functionalities that we are interested in modelling.

2.2 First Modelling Scenario - Create/Load a TPM Key

We start our approach security modelling by presenting a first scenario that involves a sequence of TPM commands for creating and loading a TPM key. More precisely, we demonstrate the creation of a TPM key that is bound to a PCR policy. This is a simple scenario that is relevant to all use cases in this project and it will serve as a basis for modelling additional, more complicated TPM functionalities in the future. The TPM commands that are relevant in this scenario are the following:

1. `tpm.StartAuthSession`: for creating a trial, or a policy session.

2. `tpm.PolicyPCR`: to initialize the PCR policy.
3. `tpm.PolicyGetDigest`: to retrieve the policy digest value from a session.
4. `tpm.FlushContext`: to erase a session (or object) from TPM memory.
5. `tpm.Create`: to create a key under a parent key.
6. `tpm.Load`: to load the created key to the TPM.

Since we are focusing on PCR policies, the command `tpm.PCRExtend` is also used implicitly, in order to modify specific PCRs. We give our abstract description of these commands below, which are based on the TPM specification manuals [44, 45].

2.2.1 Abstract Description of TPM Commands

Creating a session: `tpmStartAuthSession`

Sessions are of the core components in various TPM functionalities. They can be used for encryption purposes, as well as in authorization and auditing mechanisms (see [44, section 19.5] for more details). The command `tpm.StartAuthSession` is used to initialize a session, meaning that it creates a session key which is typically used to encrypt sensitive information (encryption session), or to derive values that are used for authorization (authorization session). We restrict ourselves to authorization sessions for now, where we distinguish three such types, namely: HMAC, policy and trial sessions. We focus on trial and policy sessions, that are used in the context of EA authorization, which is a new feature in TPM 2.0, and describe the command `tpm.StartAuthSession` accordingly.

Policy sessions are used as an authorization mechanism by associating a policy to an object in such a way that a user can access an object if a specific policy is satisfied. This authorization is typically performed by verifying that certain digest values match. For policy sessions, we may assume that no session key is created since EA authorization is accomplished by verifying that a policy is satisfied and there is no need to involve the authorization value of an object (hence an encryption/decryption process). This simplifies the modelling, since in this case the command `tpm.StartAuthSession` will simply initialize a policy digest value, denoted by `polD`, as the zero-digest. If the policy session is associated to a PCR policy, then the TPM will also set the counter `pcrC` as zero. The command description is given in Fig. 2.1. We note here that policy sessions are generated after an object has been created.

```
tpm.StartAuthSession(sessionType)
1: create sessionH                                          //create session handle
2: create polD, pcrC                          //create policy digest value and PCR counter
3: polD ⟵ zeroDigest
4: if sessionType ≠ TRIAL then
5:    pcrC ⟵ 0
6: output(sessionH)
```

Figure 2.1: `tpm.StartAuthSession()` command for policy/trial sessions

Trial sessions are used for initializing a policy. They are required in order to generate the policy digest value that will be associated with an object at creation time. Thus, trial sessions are initiated before the creation of the object.

Policy assertion: `tpmPolicyPCR`

The command calculates the policy based on specific PCR. In particular it provides a link between a PCR and a policy session that was previously created with the `tpm.StartAuthSession` command, by updating the policy digest value using the digest in the specified PCR. It takes as input the handle `sessionH` of a session, the handle `pcrH` of the PCR to be included in the policy digest calculation and the expected value `digestV` of the PCR. If the session type is `POLICY`, the TPM will retrieve the PCR digest value `pcrD` referenced in `pcrH` and will check if it matches the value `digestV` that is given by the Caller. If the two digests match, the TPM will update the policy digest value that is referenced in the handle `sessionH` according to the relation:

$$\texttt{polD}_{\texttt{new}} \longleftarrow \texttt{H}(\texttt{polD}_{\texttt{old}} || \text{`TPM-CC-PolicyPCR'} || \texttt{pcrD})$$

where `TPM-CC-PolicyPCR` is the command code of the command `tpm.PolicyPCR`, which specifies the command that caused the update for the policy digest value. Furthermore, the TPM will also update the counter `pcrC` with the corresponding value of the counter `pcrUpdateCounter` of the PCR that was included in the calculation. If the session type is `TRIAL`, the TPM will perform no check for the given `digestV` value and no update for the PCR counter `pcrC`. It will simply update the policy digest according to the above relation. In both cases, the command returns `SUCCESS` or `FAIL`, depending on whether it was properly executed or not. The process of `tpm.PolicyPCR` is described in Fig. 2.2. For more details, see [45, p. 248].

```
tpm.PolicyPCR(sessionH, pcrH, digestV)
 1: get polD, sessionType from sessionH
 2: if sessionType == TRIAL then
 3:    polD ⟵ H(polD||'TPM-CC-PolicyPCR'||digestV)
 4:    output(SUCCESS)
 5: if sessionType == POLICY then
 6:    get pcrD, pcrUpdateCounter from pcrH
 7:    if pcrD == digestV then
 8:       polD ⟵ H(polD||'TPM-CC-PolicyPCR'||pcrD)
 9:       pcrC ⟵ pcrUpdateCounter
10:       output(SUCCESS)
11:    else output(FAIL)
tpm.PolicyGetDigest(sessionH)
 1: get polD from sessionH
 2: output(polD)
```

Figure 2.2: `tpm.PolicyPCR()` command for PCR policy evaluation

Note that `tpm.PolicyPCR` does not return the policy digest value of a policy session. In order to obtain this value, the command `tpm.PolicyGetDigest` must be executed. The Caller provides as input the session handle `sessionH` and the command returns the policy digest value that is currently stored in this session (see also Fig. 2.2). In order to remove a session and its context from the TPM memory, the command `tpm.FlushContext` must be executed with input the handle of the session to be removed.

Create a Key: `tpmCreate`

This command creates a (child) key under a parent key in an existing hierarchy. It takes as input the following parameters:

- `parH`: The handle of the parent key in the hierarchy, under which the new key will be created. It contains the `seedValue`, which will be used in the Key Derivation Function (KDF) for generating the (child) key, as well as other information related to the parent key.
- `inSensitive`: Contains sensitive data related to the key that will be created. It includes the value `userAuth`, which is given by the Caller and will be set as the initial authorization value `authValue` for the new key, as well as other additional sensitive information. The `inSensitive` parameter is wrapped using the parent key and in addition it may be optionally encrypted using session-based encryption [44][4].
- `inPublic`: This is a public area template that contains information and attributes of the new key, such as the type (symmetric/asymmetric), the type of authorization to access the new key, the `authPolicy` value, which will be set as the policy digest value `polD` of a policy session to be linked with the key.
- `outsideInfo`: Additional data that provide a link between the Creator and the new key. They will be included in the `creationData` parameter in the output.
- `creationPCR`: The selection of PCR that will be included in the key creation process and will represent the state of the TPM when the object was created.

```
tpm.Create(parH, inSensitive, inPublic, outsideInfo, pcrSelection)
 1: result ⟵ ValidationCheck(parH)                    //validation check for parent key
 2: authCheck ⟵ Authorization()                  //authorization check for accessing parH
 3: if (authCheck == TRUE)  and  (result == SUCCESS) then
 4:    decrypt inSensitive                        //session-based encryption and parent key
 5:    get userAuth from inSensitive
 6:    get seedValue, parName, parQN from parH
 7:    authValue ⟵ userAuth
 8:    k ⟵ KDF(seedValue, authValue)
 9:    copy inPublic to outPublic and authValue, seedValue to outSensitive
10:    outPrivate ⟵ E(outSensitive)              //session-based encryption and parent key
11:    get pcrD from creationPCR
12:    copy H(creationPCR, pcrD), parName, parQN, outsideInfo to creationData
13:    creationHash ⟵ H(creationData)
14:    create creationTicket
15:    output(outPrivate, outPublic, creationData, creationHash, creationTicket)
16: else output(FAIL)
```

Figure 2.3: `tpm.Create()` command for creating a key under a parent key in a hierarchy

The command will generate a (child) key in a hierarchy under the parent key with handle `parH`. The TPM will first verify that the parent key referenced in `parH` handle is indeed a parent key,

[4]This is done in order to ensure that even if an adversary has control of the parent key, has no control of the sensitive area of its child keys.

meaning that it is a storage key that can wrap a child key. Additionally, proper authorization is needed for accessing the parent key, using an authorization session. The generated key will have an authorization value `authValue` and will be bound to a policy session with policy digest `polD`, which is set as the key `authPolicy` value. In particular, the command outputs the following parameters:

- `outPrivate`: The private area of the new key. This contains the `authValue` of the key, the private key (if the created key is asymmetric), the `seedValue` that was used in the KDF function for generating the key and other sensitive information. This field is usually encrypted using the corresponding encryption method that was used in the encryption of `inSensitive`.
- `outPublic`: The public portion of the newly created object, containing in general the values of `inPublic`, including the `authPolicy` digest.
- `creationData`: Contains data that describe the state of the TPM at the time of the key creation. It includes a digest of the PCR selection and values `pcrD` that were used in the creation process, as specified in `creationPCR`. It also includes certain parts from `outsideInfo`, the name and Qualified Name (QN)[5] of the parent key.
- `creationHash`: Represents the digest value that is created as the hash of `creationData`.
- `creationTicket`: A creation ticket that is used in order to verify that the created key, specifically the `creationData` of the key, was indeed created by a valid, authenticated TPM.

An abstract description of the command `tpm.Create` is presented in Fig. 2.3. For a more detailed analysis, we refer to [44, 45].

Load a Key: `tpmLoad`

This command loads a key to the TPM. It takes as input the handle of the parent key `parH`, the private portion `inPrivate` of the key to be loaded (this was returned as `outPrivate` by the command `tpm.Create`) and the public portion `inPublic` of the key. The later will be used in order to create the name of the key. The command outputs the handle `kH` of the loaded key, as well as the name of the key, which is the hash of its public portion, concatenated with the name of the hash algorithm that was used and is specified in `inPublic`. The process is described in Fig. 2.4. For more information, see [45, p.54].

For the key to be loaded, first a verification process is needed to test whether the key referenced in the parent handle `parH` is indeed the parent key and an authorization for using the parent key. An additional authorization process is needed in order to load the desired TPM key. This can be performed with HMAC or policy-session.

2.3 Ideal Functionalities

We give a simple example in order to realize our notion of an ideal functionality. Let us consider the TPM command `tpm.PCRExtend()`. This is used when a Caller wants to extend a digest value in a selected PCR. More precisely, the Caller executes the command with input a PCR handle

[5] The qualified name of a key is the digest of all of the names of all of the ancestor keys back to the Primary Seed at the root of the hierarchy [44].

```
tpm.Load(parH, inPrivate, inPublic)
 1: result ⟵ ValidationCheck(parH)                    //validation check for parent key
 2: parentAuthCheck ⟵ Authorization()          //authorization check for accessing parH
 3: if (parentAuthCheck == TRUE)  and  (result == SUCCESS) then
 4:    decrypt inPrivate                          //session-based encryption and parent key
 5:    get authValue
 6:    authCheck ⟵ Authorization()          //authorization for loading key (HMAC/policy
    session)
 7:    if authCheck == TRUE then
 8:       create kH                                                  //create key handle
 9:       name ⟵ (nameAlg||H(inPublic))
10:       output(kH, name)
11:    else output(FAIL)
12: else output(FAIL)
```

Figure 2.4: `tpm.Load()` command for loading a key into the TPM

`pcrH` and a digest value `digestV`, which is the digest to be extended to the PCR that is specified by the numeric selector `pcrNum` lying in `pcrH`. The TPM will update the current PCR value with index `pcrNum` with a new one, using the following relation:

$$\texttt{PCRDigest}_{\texttt{new}}[\texttt{pcrNum}] \longleftarrow \texttt{H}(\texttt{PCRDigest}_{\texttt{old}}[\texttt{pcrNum}] \parallel \texttt{digestV})$$

where `H()` denotes a hash function. Each PCR is also linked to a specific PCR counter, denoted by `pcrUpdateCounter`, which captures the number of updates that were performed. This extra functionality is applied as a defensive mechanism against Time-of-Check Time of Use (TOCTOU) attacks [43]. Thus, whenever the `PCRDigest` value is updated, the `pcrUpdateCounter` will increment by one. The command performs a test on the input data to check if a PCR update can be done. It returns `SUCCESS`, if it has successfully updated the PCR value and `FAIL` in any other case. An abstract version of the `tpm.PCRExtend` command that captures the above actions is presented in Fig. 2.5, while the complete description is given in [45, p.199].

```
tpm.PCRExtend(pcrH, digestV)
1: get pcrNum from pcrH
2: validationCheck ⟵ ValidateInput()          //check if operation is allowed for the
   specific PCR
3: if validationCheck == SUCCESS then
4:    PCRDigest_new[pcrNum] ⟵ H(PCRDigest_old[pcrNum] || digestV)
5:    pcrUpdateCounter ⟵ pcrUpdateCounter + 1
6:    output(SUCCESS)
7: else output(FAIL)
```

Figure 2.5: `tpm.PCRExtend()` command

PCR values are the result of measuring platform or software state [2]. It is clear from the above description that the `tpm.PCRExtend` command creates a chain of hash values, in other words a chain of measurements, where each measurement corresponds to the current state of the software.

This PCR functionality is also used in authorization mechanisms, particularly in policy-based authorization for allowing the use of objects. This is achieved by the command `tpm.PolicyPCR`, which was described in the previous section. The intention of the TPM in the case of authorization based on PCRs is to ensure that a Caller can use an object that is bound to a PCR, if and only if he possesses the latest digest value in a specific PCR, which can only be read under proper authorization. This suggests the following security property:

Security Property (PCR). An adversary cannot use an object that is bound to a specific PCR, unless he can reconstruct the PCR hash chain.

A model for the command `tpm.PCRExtend` in stateful applied pi calculus is presented in [43]. In order to capture this security property in our model, we present the ideal functionality for the command `tpm.PCRExtend`, which is denoted by $\mathcal{F}_{\texttt{tpmPCRExtend}}$ and is described in Fig. 2.6. In this ideal functionality, instead of calculating each time a hash of the previous PCR value and the digest to be extended, equivalently we keep a record of all PCR extensions by appending each new digest, provided by the Caller, to a list called `PCRList`. This procedure is executed by the command `append` in line 3, of Fig. 2.6. The syntax

```
append <'PCRList', pcrH>, digestV;
```

means that the TPM will look for the the list of PCR extensions that is referenced in the PCR handle `pcrH` and will append the value `digestV` to this list and thus the length of this list will be automatically incremented. This is the main point of difference from the modelling approach that is followed in [43], in which a PCR value is updated by a hash of the previous PCR value and a new digest that is provided by the Caller.

$\mathcal{F}_{\texttt{tpmPCRExtend}} :=$

```
1: in(<'PCRExtend', digestV>);
2: lock pcrH;
3: append <'PCRList', pcrH>, digestV;          //append digest value to PCR list
4: unlock pcrH
```

Figure 2.6: Ideal functionality $\mathcal{F}_{\texttt{tpmPCRExtend}}()$

The benefit of this modelling approach is that it captures the security property mentioned above and allows us to abstract away from cryptographic operations, in particular the involvement of a hash function. Note that in the modelling of $\mathcal{F}_{\texttt{tpmPCRExtend}}$, no counter is needed, since the number of PCR updates corresponds to the length of the list `PCRList`.

2.3.1 Ideal Functionalities for Create/Load Scenario

Ideal functionality for creating a session: $\mathcal{F}_{\texttt{tpmStartAuthSession}}$

In Fig. 2.7 we present the ideal functionality for creating an authorization session of type `TRIAL` or `POLICY`. The main purpose of this functionality is to create and initialize a list of arbitrary size, for the policy digest value. The difference from the actual `tpm.StartAuthSession`, as well as the modelling in [43], is that the policy evaluation will be represented as a list of digests instead of a single digest value, which as we will see later on corresponds to the PCR measurement.

$\mathcal{F}_{\texttt{tpmStartAuthSession}} :=$

```
1: in(<'StartAuthSession', sType>);
2: new sessionH; lock sessionH;                                  //sType = TRIAL/POLICY
3: insert <'POLDList', sessionH>, empty[];   //policy digest list set to empty list
4: insert <'SESSIONType', sessionH>, sType;
5: out(sessionH);
6: unlock sessionH
```

Figure 2.7: Ideal functionality $\mathcal{F}_{\texttt{tpmStartAuthSession}}()$

Ideal functionality for policy assertion: $\mathcal{F}_{\texttt{tpmPolicyPCR}}$

The ideal functionality $\mathcal{F}_{\texttt{tpmPolicyPCR}}$ will calculate the policy based on PCR values. The Caller provides a session handle `sessionH` and a list of digests `digestL[]` as input, which is the expected PCR list. If the session handle references a policy session, the TPM will compare this list to the actual PCR list and if there is a match, it will append the `digestL[]` to the policy digest list `polL[]`. If the session is a trial session, the TPM will append the `digestL[]` to the policy digest list `polL[]` directly, without performing any check. This process is described in Fig. 2.8, along with the ideal functionality $\mathcal{F}_{\texttt{tpmPolicyGetDigest}}$, which outputs the policy digest list `polL[]`.

$\mathcal{F}_{\texttt{tpmPolicyPCR}} :=$

```
1: in(<'PolicyPCR', sessionH, digestL[]>);
2: lock sessionH; lock pcrH;
3: lookup <'SESSIONType', sessionH> as sType in;
4: lookup <'POLDList', sessionH> as polL[] in;          //get the policy digest list
5: if sType == TRIAL then
6:   append <'POLDList', sessionH>, digestL[];
7:   unlock sessionH; unlock pcrH
8: if sType == POLICY then
9:   lookup <'PCRList', pcrH> as pcrL[] in;                        //get the PCR list
10:  if digestL[] == pcrL[] then
11:    append <'POLDList', sessionH>, digestL[];   //policy digest list as PCR list
12:    unlock sessionH; unlock pcrH
13:  else unlock sessionH; unlock pcrH
```

$\mathcal{F}_{\texttt{tpmPolicyGetDigest}} :=$

```
1: in(<'PolicyGetDigest', sessionH>);
2: lock sessionH;
3: lookup <'POLDList', pcrH> as polL[] in;
4: out(polL[]);
5: unlock sessionH
```

Figure 2.8: Ideal functionality $\mathcal{F}_{\texttt{tpmPolicyPCR}}()$

Note that in the ideal functionality $\mathcal{F}_{\texttt{tpmPolicyPCR}}$ there is no need to add a counter that measures the **PCR** updates, since this is captured by the length of the PCR list. Any comparison between lists of different length will fail to update the policy.

Ideal functionality for creating a key: $\mathcal{F}_{\texttt{tpmCreate}}$

The ideal functionality for creating a key is summarized in Fig. 2.9. This model considers only the assignment of the authorization policy to the newly created key and assumes that a new key `k` is securely generated. Hence we are focusing on the link of the key with a policy session. In addition and in contrast to the actual TPM command `tpm.Create`, we assume that the ideal functionality $\mathcal{F}_{\texttt{tpmCreate}}$ creates and returns the handle `kH` of the new object, following the modelling of Shao et al. [43]. In reality, the handle of the key is created and returned by the `tpm.Load` command, as described in the previous section.

$\mathcal{F}_{\texttt{tpmCreate}} :=$

```
1: in(<'Create', polL[]>);
2: new k; new kH;
3: lock kH;
4: insert <'AUTHPOLICY', kH>, polL[];  //initialize authorization policy of the key
5: out(kH);
6: unlock kH
```

Figure 2.9: Ideal functionality $\mathcal{F}_{\texttt{tpmCreate}}()$

Ideal functionality for loading a key: $\mathcal{F}_{\texttt{tpmLoad}}$

Based on the model for creating the key, the ideal functionality $\mathcal{F}_{\texttt{tpmLoad}}$ takes as input the handle `kH` of the key to be loaded and the handle `sessionH` of the policy session. The TPM will retrieve the authorization policy of the key, which is the list `authPolicy[]` and it will compare it with the policy digest list `polL[]` that is stored in the corresponding session. If the two lists match, the TPM will load the key (`event Load(k)`). This process is described in Fig. 2.10.

$\mathcal{F}_{\texttt{tpmLoad}} :=$

```
1: in(<'Load', kH, sessionH>);
2: lock kH; lock sessionH;
3: lookup <'POLDList', sessionH> as polL[] in
4: lookup <'AUTHPOLICY', kH> as authPolicy[] in
5: if authPolicy[] == polL[] then
6:    event Load(k);
7:    unlock kH; unlock sessionH
8: else unlock kH; unlock sessionH
```

Figure 2.10: Ideal functionality $\mathcal{F}_{\texttt{tpmLoad}}()$

2.3.2 Create/Load Process Command Flow

The command flow that represents the process of creating a new key and loading it in the TPM, using the ideal functionalities, is summarized in Fig. 2.11. The Caller executes the command $\mathcal{F}_{\texttt{tpmStartAuthSession}}(\texttt{TRIAL})$ in order to create a trial policy, with a policy digest list `polL[]` initialized to an empty list by the TPM. The TPM returns the handle `tsH` of the trial session and the

Caller runs the command $\mathcal{F}_{\texttt{tpmPolicyPCR}}(\texttt{tsH, digestL[]})$ in order to append the list digestL[] to the policy digest list polL[]. He retrieves the updated polL[] via $\mathcal{F}_{\texttt{tpmPolicyGetDigest}}(\texttt{tsH})$ and removes the trial session from the TPM memory by executing $\mathcal{F}_{\texttt{tpmFlushContext}}(\texttt{tsH})$.

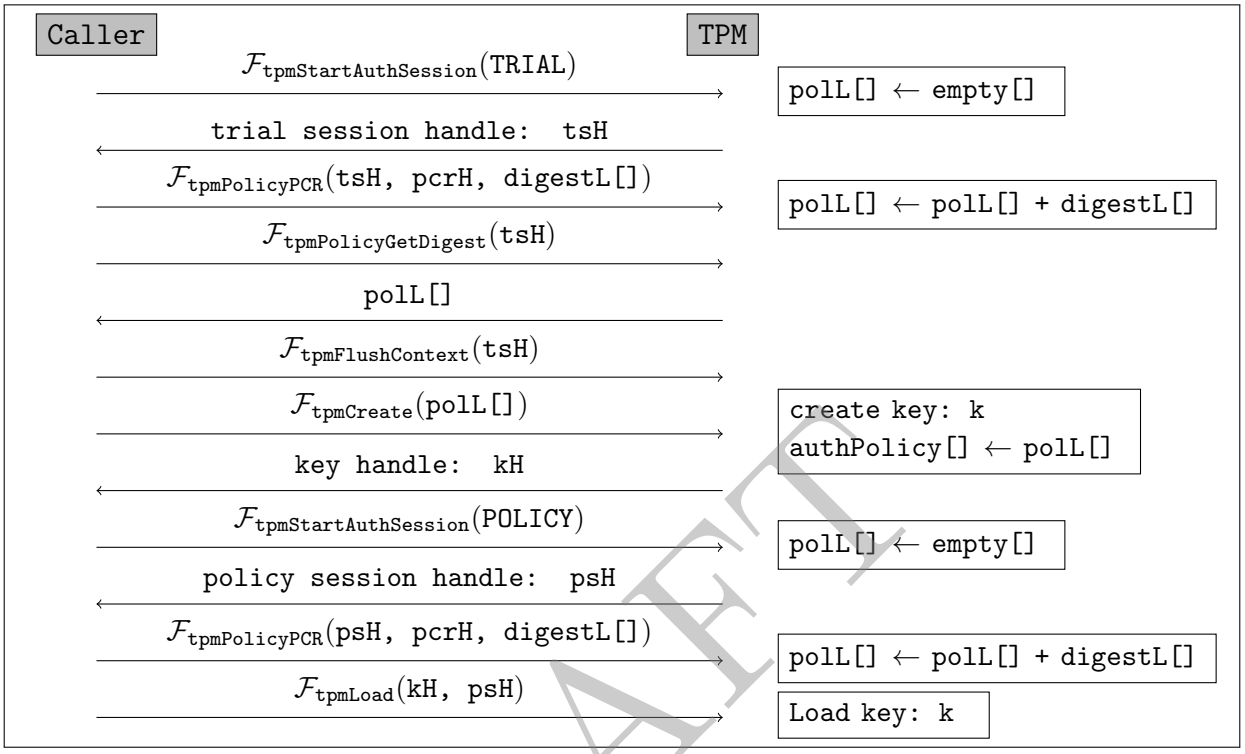

Figure 2.11: Create/Load an object in TPM

The Caller runs the command $\mathcal{F}_{\texttt{tpmCreate}}(\texttt{polL[]})$ in order for the TPM to generate the a key with authorization policy authPolicy[] set as polL[]. Then the TPM will return the key handle kH to the Caller. The next step is to create the policy session. The Caller executes the ideal functionality $\mathcal{F}_{\texttt{tpmStartAuthSession}}(\texttt{POLICY})$ and receives the policy session handle psH by the TPM. Then the command $\mathcal{F}_{\texttt{tpmPolicyPCR}}(\texttt{psH, digestL[]})$ is executed, in order to evaluate the policy, or in other words, in order to append digestL[] to the policy digest list polL[]. Now the Caller can load the key into the TPM by running the command $\mathcal{F}_{\texttt{tpmLoad}}(\texttt{kH, psH})$. The key will be successfully loaded to the TPM if the authorization policy list of the key matches the policy digest list of the corresponding policy session.

2.4 Outcomes to Use Case and Extensions

The process of creating and loading a key into the TPM is present in all the use cases that the FutureTPM project focuses on, although the generated keys have different attributes and objectives. The model we have described in the previous sections is a generic model for creating/loading a key that is sealed into a specific PCR. Hence we may need to adjust our model, depending on the properties and purposes of the keys that are generated in each use case. We briefly outline the main points to consider when integrating our model in the context of the three use cases, as those were described in D4.1 [29] and D6.1 [28]. We only highlight here that although the keys that are generated in the use cases might have different usage, the authorization of their usage is performed via the EA mechanism and more precisely using policy sessions based on PCRs. This justifies our decision of starting our modelling with the create/load functionality.

Reference Scenario 1 – Secure Mobile Wallet and Payments

This use case focuses in the provision of enhanced security properties of the TPM (sealing, unsealing and key generation). To protect sensitive data (bearer token, financial token, transaction history), this scenario uses a randomly generated key, and then protects this key with a password. The password data blob is then sealed to specific PCR values that reflect the platform state. Fast IDentity Online (FIDO) Universal 2nd Factor (U2F) authentication is used as an extra layer of security, in order to cover Android devices without a TPM, but the particularities of this mechanism are outside the scope of the use case. More concretely, the authorization value `authValue` of the key is set as the password (see Fig. 2.3), and the loading process of such password requires policy-based authorization using the command `tpm.PolicyPCR` to evaluate the policy, which coincides with our model. That is, the key can be loaded if the password is known, or if the authorization policy `authPolicy` of the password matches specific PCR values that contain the sealed blob. The platform state, and hence the extended PCR value used for the authorization policy, is based on the Control Flow Graph (CFG) of the Android software state and device integrity measurements (see D6.1 [28] for details). The create and loading actions in this use case clearly match our initial set of TPM functionalities intended to be modeled.

Reference Scenario 2 – Activity Tracking

One of the core functionalities in this use case is the implementation of the DAA scheme which ensures the establishment of a secure and anonymous private channel for the interaction between the S5PersonalTracker, which represents the host device that includes a TPM and the S5Tracker Analytics Engine, which acts as the Verifier. In D4.1 [29], the description of the TPM commands for the activity tracking use case suggests that two TPM keys are created during the execution of the `DAA-Sign` and `DAA-Verify` sub-protocols of the DAA scheme. Both keys are bound to specific PCR values that correspond to measurements of the CFGs that were described in D3.2 [27]. The loading of these keys requires the verification of the corresponding PCR state and hence implying the use of PCR policy sessions for authorization.

Reference Scenario 3 – Device Management

This use case is concerned with the protection of the communication keys between the network devices and the Network Management System (NMS), and with the routing of network traffic according to the trust state of such devices. During the setup phase, each device creates an Attestation Key (AK) which is certified by the NMS, returned back to the device and retrieved by using the appropriate set of TPM commands [29]. This process is the line with the operations performed by a Privacy Certification Authority (CA), where the trusted entity (the NMS in this case) needs to be trusted and manages the Endorsement Key (EK)s of the devices, which for this particular use case, also allows creating whitelists of trusted devices. Further, a Transport Layer Security (TLS) key needs to be created in each device in order to establish a trusted channel with the NMS. This key will be certified by the above mentioned AK. According to the description presented in D4.1 [29], the creation of the TLS key requires that the `authPolicy` is asserted with a PCR policy, based on PCR extensions that correspond to the current system state. In this case, the system state refers to the whitelist of allowed software to run in the device. Hence, if the state is different, the device will be unable to load the TLS key and perform the handshake with the NMS, which will be interpreted by that entity as an untrusted state, and will take the appropppriate measures. We note that this scenario involves multiple loads of the TLS key (and to a lesser extent, the AK) where a verification of the authorization policy against the system's state

is required. Clearly, this justifies our initial modelling approach of TPM functionalities. We refer to D4.1 [29] and D6.1 [28] for a more detailed description and analysis on the interaction of the TPM, the host device and the NMS.

Chapter 3

Cryptography for the TPM

In this chapter we are going to introduce a new property-based definition for Direct Anonymous Attestation (DAA) which is the main functionality of TPM. As one may know, UC based model is able to capture all security notions of DAA. However, it still has outstanding gaps on how to merge it into property based security setting. Motivating by some existing problems in the current DAA security definitions, we close the gap between property-based and UC-based security models of DAA via the new defined model implying the UC definition of Camenisch et al. [19]. We further provide security analysis to state that the new model provides the same security guarantees as the UC version in [19]. Technical overview and details will be given below.

3.1 A New Property-Based Definition Of DAA

3.1.1 Introduction

The ever growing ecosystem of devices connected to the internet poses an insurmountable task for security professionals aiming to secure such devices. Acknowledging both the finite number of trained professionals and shortcomings of software solutions, the industry standardisation body, The Trusted Computing Group (TCG), looks to trusted hardware as an axiom of security with its TPM chip. In 2004 the TCG standardised the first DAA protocol in the TPM 1.2 specification [46]. Now present in more than 500 million devices, and with support for multiple DAA schemes in the TPM 2.0 standardisation [47], DAA is one of the most widely deployed protocols of such complexity. It has been adopted as a basis for EPID [16], Intel's recommendation as an industry standard for attestation of trusted systems in the Internet of Things. The standardisation of DAA has also been made by ISO as ISO/IEC 11889 international standards [32, 34]. Furthermore, it is under consideration by FIDO for their specification of anonymous attestation [18]. In addition to these standards, DAA has applications for anonymous subscription [38, 37] and vehicle communication (V2X) [40, 48].

DAA allows a TPM chip, embedded in a host computer, to make attestations about the state of the host system. The TPM is used as a root of trust to create signatures on the integrity of the host machine's hardware and software configuration. Such attestations convince a remote verifier that the platform (host computer and TPM) with whom it communicates is in a healthy state. Crucially, attestations must be anonymous so that a verifier is able to check that a signature originates from a legitimate platform without learning its identity. An additional feature of DAA is that platforms are able to make signatures linkable so that a verifier knows when two signatures have come from the same anonymous platform. This 'linkability' is steered by the use of a basename where signatures signed using the same basename are linkable whereas those created with a fresh

basename are not.
As well as a number of platforms and verifiers, DAA includes an issuer who decides whether a platform can join (obtain signing credentials) or not. In practise this might be a service provider like Google or a government authority for example. Once joined, the platform can sign messages using a credential given by the issuer. These signatures might, for example, allow access to one of the issuer's services. For those familiar with the group signature setting, this process of joining, issuance, and signing (attesting) holds many similarities. However, in DAA the TPM and Host are required to *jointly* create signatures so that neither of them can make valid signatures without the involvement of the other. With the added complexity that the TPM and Host can be in different corruption states this makes defining DAA security a real challenge.

Problems with existing security definitions

Alarmingly, in spite of wide deployment and a large body of work, from 2004-2016 there existed no sound and comprehensive security model of DAA. Indeed, to date, there exists no such property-based[1] definition and all previous attempts are demonstrably flawed. The shortcomings of DAA security definitions not only represent a theoretical problem but undermine schemes already deployed in practice. In particular, a DAA scheme that allows one to forge attestations (as it does not exclude the 'trivial' TPM credential $(1, 1, 1, 1)$) has even been standardised in ISO/IEC 20008-2 [33, 25]. There are several non-trivial challenges in designing security definitions for DAA, however there are two significant obstacles that have, time and again, lead to flaws in security.
One of the most tricky aspects is the notion of TPM 'identity'. Unlike related protocols such as group signatures, where parties posses a certified public/secret key pair, the identity of a TPM is not clearly defined since there is no public key related to the secret key used in signature generation. This has long been the Achilles heel of DAA security models since it is crucial to be able to determine whether a given signature stems from some legitimate TPM. This is particularly important when one wants to prevent the existence of valid signatures that do not come from any legitimate TPM. It is then necessary to be able determine if a given signature stems from any honest *or corrupt* TPM. The works [15, 24] skirt around this issue of corrupt TPM identification by assuming that corrupt TPMs already have their secret keys exposed in a public list RogueList. The assumption that a corrupt TPM's signing key is automatically publicly visible in some list is a very strong one. Moreover, a real life adversary has no obligation to publish a TPM's signing key if it were able to obtain one; it may choose simply to create forgeries with it. Furthermore, the works of [15, 24] do not allow for a notion of correctness in their RogueList process. The most recent work on property-based DAA by Bernard et al. [8] make a hidden assumption that all TPM secret keys can be extracted from their signatures. Then a TPM's secret key can be used to determine whether a given signature came from that TPM. Unfortunately this extraction would take exponential time for their accompanying scheme and is not formally defined in their work.
Another aspect of DAA, which adds to the complexity of security models, is that a platform is made up of two entities, a TPM and host, who can be in different corruption states. This calls for extra consideration when defining the unforgeability of DAA since here we require that a corrupt host cannot produce a valid signature without the involvement of an honest TPM. For example, in standard signature schemes, unforgeability is only concerned with preventing forgeries under entirely honest users. In DAA however, an honest TPM who's host is corrupt produces signatures that we also want to be unforgeable. Therefore, our unforgeability definition must also cover these 'mixed' corrupt settings. This has been one of the major sources of complexity for DAA security

[1] Note that 'property-based' is often used interchangeably with 'game-based' in the DAA literature.

over the years. One work [8] tries to combat this complexity by considering the TPM and Host as one party so that both platform components are in the same corruption state at all times. They then argue how to bootstrap this notion of 'Pre-DAA' security to (full) DAA security in which the TPM and Host can be in different corruption states as in the real world. Unfortunately, due to the absence of a full security proof, this lifting argument is only made informally. As shown in [19], the argument for unforgeability cannot be lifted (under the same assumptions) to the full DAA setting. These two subtleties of DAA have been the pitfalls leading to many flawed security models over the years and overcoming these constitutes one of the contributions of this work.
In addition to these two intrinsic difficulties with defining DAA security, the most recent work of Bernard et al. [8] further suffers a trivial security flaw by assuming that the issuer registers its public key with an extractable proof of correctness or that there is a trusted setup; something not true for their scheme. This permits a corrupt issuer to choose its key in a malformed way, allowing it to win the non-frameability game. Also we observe that their model only permits the adversary a single call to the challenge oracle in their anonymity game allowing for a trivially non-anonymous scheme to be proven secure (see full paper for details []). For a comprehensive overview of the flaws in all existing property-based DAA definitions we refer the reader to Section 2.1 of [19].
Owing to apparent difficulties with defining property-based DAA, Camenisch et al. [19] turn to the UC model, formulating a DAA functionality to capture security. The UC model [22] provides a watertight framework for defining security. One creates an 'ideal functionality', performing the protocol in a perfectly secure way, and then proves a given instantiation indistinguishable from this functionality. The work of [19] stands as the only solid security definition to date and forms the basis of a subsequent line of work [20, 17]. The later papers consider an add-on feature in which some security properties are required to hold even if the TPM is subverted. For the sake of clear presentation our work is in line with the TCG's foundational assumption that TPMs cannot be subverted. We thus refer to the original UC model in [19]. In retrospect, the success of the UC definition over previous property-based ones is unsurprising given the framework's ability to model complex protocols like DAA whose intended use involves composition with other cryptographic protocols. In this way, one could argue that UC is the correct starting point for defining DAA security. Unfortunately, this definition is very hard to work with, even for those familiar with UC, where DAA proofs require as many as 17 game-hops. Consequently, researchers have continued to work with somewhat adapted property-based definitions that are still flawed.
Note that we have a UC based security notion which formally captures all the security notions of DAA, the next big step is to bring back these UC based security notions to the familiar property-based security setting. This would greatly simplify the proofs of DAA security and should form the basis for future work regarding DAA. However, to be absolutely sure that the property-based security definition is solid, unlike prior works, it is desirable to have a mechanism providing a 'sanity-check' that the property-based definition implies the UC-based definition.

Our Contribution

In this work we affirmatively close the gap between property-based and UC-based security models of DAA by providing a property-based security model that, after a syntactic re-structuring, implies the UC definition of Camenisch et al. [19]. This provides a meaningful 'sanity-check' that our security model is both sound and comprehensive.
Our first contribution is to define a property-based model of DAA. Our model is based on a completely different design philosophy to other property-based works. Indeed, rather than patching the known flaws in preceding works, we extract the essential security properties of the UC definition, and reformulate them using property-based experiments. In particular, these properties are

correctness, anonymity, unforgeability, and non-frameability. Since the UC functionality achieves security through a series of checks and conditions that may be interleaved across multiple interfaces this turns out to be a non-trivial task (For example see [19] for the UC-based security model). Some of these checks may even contribute towards several distinct security properties, and therefore, it is even unclear how many properties the UC-based security model implicitly embeds. We thus consider the reformulation of these properties as separate experiments to be a valuable contribution on its own.
Our second contribution is to show there can be no doubt that our model provides the same security guarantees as the UC definition of [19]. This is done as follows. We begin by considering a scheme, secure in our property-based model. We then create, via a syntactical re-structuring, a 'wrapper protocol' which allows for a direct proof of indistinguishability between this wrapper protocol and the ideal functionality in the UC framework. Importantly, this wrapping process is merely cosmetic and does not affect the security of the protocol. Indeed, Theorem 1 provides a 'sanity check' that our model is sound by providing the necessary conditions for the property-based DAA algorithms to imply UC security (when wrapped). This puts a positive end to a long journey of flawed property-based models proposed since 2004.
Herein, we give a more detailed overview of the techniques that we employed in this work.

Technical Overview

Comprehensive Security Definitions. When designing each experiment we aim to extract the same guarantees that the UC-DAA functionality provides for a particular security notion. This is a non-trivial task since in the ideal functionality $\mathcal{F}_{\mathsf{daa}}$, it is often the case that numerous checks across several of its interfaces interleave to achieve a given security notion. Thus, we must unravel $\mathcal{F}_{\mathsf{daa}}$ and split its security guarantees into individual properties. We do this for each of the five properties mentioned above. In Section 3.1.1 we highlight two main pitfalls with defining property-based DAA security. We now briefly describe the techniques used in this work to overcome each of these obstacles. Interestingly both of these challenges are most acute in the unforgeability game. Here, no adversary should be able to create more unlinkable signatures than he has corrupt TPMs, nor can he produce a signature which identifies to an honest TPM who was not involved in its creation. Crucially, this property should hold even when the platform's host machine is corrupt but the TPM remains honest. Whilst the solutions we present do lead to a sound notion of unforgeability, these techniques have wider reaching benefits for the whole model. However, the unforgeability property best motivates their necessity and exhibits their elegance, so we focus on the unforgeability game below. There are several approaches explored in the literature to tackle this heavy security notion. We take the most recent property-based model of Bernard et al. [8] as a motivating example which becomes unstuck in the following ways.

- Due to the complexity introduced by the TPM and host being in different corruption states, it is attractive to first consider an unforgeability game in which both parties are in the same corruption state. One must then provide a 'lifting' argument to bootstrap this weaker notion to a stronger one where the TPM and host can be in different corruption states. However, due to the sensitive nature of the unforgeability notion, such lifting does not exist in general, under the same hardness assumptions. Indeed, this is where the work of [8] brakes down. Details of the attack on this scheme can be found in Section 2.2 of [19].

- A fundamental feature of the unforgeability game is to ensure that a valid signature can only be generated by a legitimate TPM. In particular this requires that given a signature one must be able to determine whether it has come from any honest *or corrupt* TPM who has 'joined'.

This latter case is where the difficulty lies since it is not clear how the challenger can check this against corrupt TPMs for which he doesn't know the secret key. This is exactly due to the absence of a notion of TPM identity since there is no public key associated with a TPM's secret key. [8] makes some progress toward this problem by using extracted secret keys from join transcripts to identify signatures coming from corrupt TPMs. Unfortunately, the extractability of secret keys from each join transcript is something not assumed in the accompanying scheme nor is formally defined in the security model.

To overcome this first problem of great complexity introduced by the presence of a TPM and host that can be in different corrupt states, we accompany our security experiments with a detailed set of oracles (see Figure A.1) used to simulate honest parties for the adversary. Where interactive protocols are considered, our oracle definitions allow the adversary to make queries on *individual protocol messages* rather than having to complete the full protocol before moving on. For example, the adversary is not obliged to complete a signing protocol with one TPM before beginning another with a different TPM. This is done by adapting the notation used in group signatures [4], [13] to the DAA setting. The granularity of these oracle definitions is essential for capturing all meaningful adversarial behaviour, something only attempted in [8], though there it was for the vastly simplified setting where the TPM and host are assumed to be in the same corruption state. Although this slightly adds to the complexity of oracle definitions, we are rewarded with a much more succinct set of security experiments which we can be sure capture the multitude of corruption settings in DAA. Indeed, our model comprises fewer security experiments than the most recent attempt of Barnard et al. in which a vastly simplified 'pre-DAA' is considered.
To tackle the second issue of TPM identification we import the auxiliary algorithms Ext and $\mathsf{Identify}$ from [19] where they are used informally. We formalise them for the first time, giving definitions which display their necessary interdependence. Ext is used to extract TPM secret keys from a TPM's communication with an (honest) issuer. We then use an $\mathsf{Identify}_{\mathsf{T}}$ algorithm which determines whether a given TPM key was used in a given join transcript. This algorithm borrows from the ideas of [8] and is used in this work as an intermediate step in determining whether a given signature was signed using a TPM that has joined. This algorithm tells us whether a TPM with that key has joined. $\mathsf{Identify}$ can then be used to determine whether a signature was generated using this secret key. Thus, we formalise the method of determining whether a valid signatures was signed by a TPM that has joined. Our definition of these algorithms formalises, for the first time, the process of extraction, avoiding the pitfalls of [8] and making way for a more succinct, intuitive security notion. In particular this resolves the long-standing issue of defining a TPM identity since, unlike in group signatures, there is no notion of user identity in DAA.
Besides overhauling the unforgeability definition, we make small, but essential alterations to the anonymity and non-frameability games. Here, we allow the adversary to choose the issuer's secret key. Since schemes like EPID do not assume a trusted setup nor require the issuer to register its public key with an extractable proof of correctness, we have to model this correctly. We also allow the adversary multiple calls to the challenge oracle in the anonymity experiment. Failure to do so in [8] lead to a definition allowing for trivially non-anonymous schemes to be proven secure. This is demonstrated in the Appendix of the full paper [].

Implying the UC definition. The remaining task it to check that our new definition indeed captures all desirable aspects of DAA security. We give a natural confirmation of this by proving that our model implies the UC-DAA definition of [19]. Theorem 1 is a statement of this result.
To build confidence in any property-based definition of DAA, one should be able to claim that it at least provides, in essence, the same security guarantees as the UC-DAA definition of [19]. The use of 'essence' here is a strategic one since clearly no property-based definition can provide

the composability guarantees as one in the UC model. Nevertheless, existing works [21] have successfully shown how a new property-based definition can be re-formulated or 'wrapped' to imply an existing UC definition. Crucially, this process should not affect the security guarantees of the new definition but should merely serve to make a direct comparison between them possible. Indeed, one can view this merely as a cosmetic repackaging of the property-based algorithms to create a UC protocol that can be compared to an existing UC functionality.
Proving that a property-based definition implies one formulated in the UC framework presents a number of challenges. First and foremost, the UC framework [22] assumes by default a completely asynchronous network in which only one party can be activated at any given time. As a consequence of this and DAA's interactive protocols, the authors of [19] were forced to split the join and sign interfaces of their ideal functionality each into two phases: a request and proceed phase. This has the knock on effect that schemes proven in the UC model must also contain these split phases even though any implementation must eventually combine them to create a single join protocol and a single signing protocol. Indeed, the ability to define intuitive, single join and signing protocols in our new model is very attractive for scheme design and something we consider a valuable and non-trivial contribution. We define a wrapper protocol Π_{DAA} in Section 3.1.3 that matches the compact, intuitive join and sign algorithms of our model to the split interfaces of the UC DAA functionality. Our main result, Theorem 1, shows that protocol Π_{DAA} realises the UC functionality $\mathcal{F}_{\mathsf{daa}}$ [19]. This is done by a series of game hops where indistinguishably between games is shown by reducing to the assumed security properties of our new model. Thus, this result confirms that our property based model captures the full essence of DAA security and does not suffer from the flaws of previous property-based models.

3.1.2 Syntax and Security Model

We first develop the syntax in this section and then present the *property-based* security definition of a DAA scheme in Section 3.1.2. Due to the complex nature of the security notion for DAA, in this section, we also prepare some oracles and auxiliary algorithms needed to formally define the security requirements.
Before describing the security model and its technicalities we informally discuss how DAA works and the desired security properties. In a DAA scheme, we have four main entities: a number of TPMs, a number of hosts, an issuer, and a number of verifiers. A TPM and a host together form a platform which performs the *join protocol* with the issuer who decides if the platform is allowed to become a member. Once a member, the TPM and host (i.e., platform) can together sign messages with respect to a basename bsn. If a platform signs with $\mathsf{bsn} = \perp$ or a fresh basename, the signature must be anonymous and unlinkable to previous signatures. That is, any verifier can check that the signature stems from a legitimate platform via a deterministic verify algorithm, but the signature does not leak any information about the identity of the signer. Only when the platform signs repeatedly with the same basename $\mathsf{bsn} \neq \perp$, it will be clear that the resulting signatures were created by the same platform, which can be publicly tested via a (deterministic) link algorithm.
The desirable security properties of a DAA scheme can be described informally as follows:

Correctness: When an honest platform generates a signature on message m and basename bsn, an honest verifier will accept this signature. Furthermore, two signatures σ_1 and σ_2 generated by the same honest platform using the same basename $\mathsf{bsn} \neq \perp$ should link according to the link algorithm. To an honest verifier, it should not matter in which order two signatures are supplied when testing their linkability.
Anonymity: An adversary that is given two signatures, w.r.t. two different basenames or $\mathsf{bsn} = \perp$,

cannot distinguish whether both signatures were created by one honest platform, or whether two different honest platforms created the signatures. We require this property to hold even when the issuer is corrupt.

Unforgeability: No adversary can produce more unlinkable signatures than he has corrupt TPMs and cannot produce a signature that identifies to an honest TPM who never took part in its signing. In particular, this captures the scenario where a platform with an honest TPM and corrupt host cannot produce a signature attesting to the fact that the platform is in the 'right' configuration. This property only holds when the issuer is honest.

Non-frameability: No adversary can create signatures on a message m w.r.t. basename bsn that links to a signature created by an honest platform, when this honest platform never signed m w.r.t. bsn. We require this property to hold even when the issuer is corrupt.

Syntax

We begin by providing the syntax of DAA. The following syntax represents the algorithms run by the parties involved in a DAA scheme: an *issuer* $\mathcal{I}$ which controls who can obtain signing credentials; a set of *platforms* each comprised of a *TPM* and *Host*, and a set of verifiers who may request attestations from a platform. Formally, a DAA scheme DAA consists of the following (interactive) PPT algorithms:

$\mathsf{Setup}(1^\lambda) \to \mathsf{param}$: The set-up algorithm takes as input the security parameter 1^λ and outputs a description of the system parameters param.

$\mathsf{TPM.Kg}(\mathsf{param}) \to (\mathsf{esk}, \mathsf{epk})$: The TPM key generation algorithm takes as input the system parameters param and outputs an endorsement secret/public key pair $(\mathsf{esk}, \mathsf{epk})$ for a TPM.

$\mathsf{I.Kg}(\mathsf{param}) \to (\mathsf{isk}, \mathsf{ipk}, \mathsf{ML})$: The issuer key generation algorithm takes as input the system parameters param and outputs an issuer secret/public key pair $(\mathsf{isk}, \mathsf{ipk})$ for the issuer $\mathcal{I}$ and initialises an empty members list ML.

$\mathsf{IpkVf}(\mathsf{ipk}, \mathsf{param}) \to 1/0$: The issuer public key verification algorithm takes as input the issuer's public key and the system parameters and outputs 1 or 0 to indicate accept or reject.

$\langle \mathsf{JoinTPM}(\mathsf{esk}), \mathsf{JoinH}(\mathsf{epk}, \mathsf{ipk}), \mathsf{Issue}(\mathsf{isk}, \mathsf{epk}, \mathsf{ML}) \rangle$: This is an interactive join protocol between a TPM, host, and issuer. At the end of the interactive protocol, the algorithms output a TPM signing key sk, a credential cre, and an updated members list $\mathsf{ML} \leftarrow \mathsf{ML} \cup \{\mathsf{epk}\}$, respectively.

$\langle \mathsf{SignTPM}(\mathsf{sk}, \mathsf{m}, \mathsf{bsn}), \mathsf{SignH}(\mathsf{cre}, \mathsf{m}, \mathsf{bsn}) \rangle$ This is an interactive sign protocol between a TPM with signing key sk, message m, and a basename bsn and host with credential cre, message m, and basename bsn as inputs. At the end of the interactive protocol, the algorithms output $\perp$ and σ, respectively.

$\mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{RL}) \to 1/0$: The deterministic verification algorithm takes as input the issuer's public key ipk, a signature σ, a message m, a basename bsn, and a revocation list RL, and returns 1 or 0 to indicate accept or reject.

$\mathsf{Link}(\mathsf{ipk}, \sigma, \sigma', \mathsf{m}, \mathsf{m}', \mathsf{bsn}) \to 1/0$: The deterministic linking algorithm takes as input the issuer's public key ipk, two signatures σ, σ' w.r.t two messages m, m', and a common basename bsn. The algorithm returns 1 or 0 to indicate accept or reject.

The correctness and security requirements for DAA will be formalized via an experiment between an adversary and a challenger in the following Section 3.1.2. Below, we provide some details on the above syntax so as to help understand DAA and the following sections better. We present them in a sequence or remarks.

Remark 1 (Significance of endorsement keys). The endorsement keys $(\mathsf{esk}, \mathsf{epk})$ of the TPM are mainly used to create an authenticated channel between a TPM and the issuer during the join protocol. This is necessary since the TPM can only communicate with the issuer via the (possibly malicious) host. However, as in other advanced signature schemes such as group signatures [5, 6], this will be implicit in the scheme for simplicity.

Remark 2 (Significance of members list ML). The issuer manages the members list ML created by the issuer key generation algorithm I.Kg. The list ML keeps track of which TPMs have joined through the interactive join protocol. We will always assume membership management is done by including the endorsement public keys of the TPMs to ML and make it explicit in the syntax. This is how membership is managed in practice and makes use of the authentication that the endorsement keys allow.

Remark 3 (Key-based revocation). Our syntax captures the notion of *key-based revocation*. As one of the popular methods in the literature, e.g., [10, 19] and preference of the TCG, we model *key-based revocation* where we include the signing secret keys sk of corrupted TPMs in RL to revoke signatures. More discussions will be provided in the following section.

Security Model

Before we formally state our security definition, we define several oracles and auxiliary algorithms, $\mathsf{Identify}$ and Ext, which allow for a more succinct and intuitive description of our security definition.

Oracles. Since DAA must capture various security notions such as anonymity, unforgeability, and non-frameability, it would be helpful to abstract what the adversary can do in each of these property-based security definitions in the form of oracle access. In this section, we define several oracles that help us simplify the property-based definition provided in Section 3.1.2.

Concurrent Oracle Access. It is crucial that oracles simulating a DAA protocol to an adversary allow him the same concurrent access as he would have in real life. For example, an adversary in control of two host machines might begin a join session using one of its honest TPMs whilst using the other to commence a signing session. This example demonstrates that oracles simulating part of an interactive protocol (joining or signing) can be interleaved with other oracles by the adversary. In other words, he should be allowed to make a query that corresponds to the first move of an interactive protocol and receive the corresponding reply. Then, he should be able to query an entirely different oracle with no obligation to complete the protocol he started with the first. Only oracles defined in this way capture, with sufficient granularity, the true nature of a real life adversary's power in the DAA setting. Of equal importance is that concurrent adversarial access is the default setting in the UC framework used to define DAA security in [19]. Thus, in order that our property-based definition implies the same level of security, we must define oracles that give the adversary equivalent power. We follow the standard notation for defining such oracles as used in group signatures [4], [13]. The algorithms comprising a protocol are modelled as interactive Turing machines (ITIs) which take as input a protocol message M_{in} and an internal state st and output a message M_{out}, a new internal state st', and a decision $\mathsf{dec} \in \{\mathsf{cont}, \mathsf{accept}, \mathsf{reject}\}$. We cut the protocol into pieces where each 'piece' is defined by the receiving of a message and then the sending of a response message. We say that an ITI (algorithm) run as part of a protocol

List label	List description	Entry Format
HT	Identities of honest TPMs with corrupt host.	i
HP	Identities of honest TPMs with honest host *i.e.,* honest platform.	i
CP	Identities of corrupt TPMs with corrupt host *i.e.,* corrupt platform.	i
ESK	Secret TPM endorsement keys.	esk_i
EPK	Public TPM endorsement keys.	epk_i
HSK	Honest TPM (secret) signing keys.	sk_i
CSK	Corrupt TPM signing keys (extracted)	sk_i
CRE	Signing credentials.	cre_i
SL	Signatures obtained from the signing oracle SignO.	$(i, \mathsf{m}, \mathsf{bsn})$
CL	Queries made to the challenge oracle ChalO.	(i_0, i_1, bsn).
TS	Communication transcript of calls to IssueO.	Scheme dependent

Table 3.1: Global lists maintained by oracles.

returns cont if the protocol is still going on, reject if some check failed or an invalid input was received, and accept if everything runs as expected and there is no next step for that ITI in the protocol.

Oracle Descriptions. As mentioned above, the oracles are an abstraction of the adversary's ability in the property-based security definition. For example, we may want an adversary to be able to obtain signatures from honest platforms. In the descriptions below, we omit the input protocol messages M_{in} for readability and adopt the same convention when referring to the oracles in the security games and proofs but note that they are included in the formal definitions of Figure A.1.
The following global lists, initially set to empty, are maintained by the oracles: HT is a list of honest TPMs (whose host is corrupt); HP is a list of TPMs which are part of an entirely honest platform $\mathcal{P}$, *i.e.,* the host is also honest; CP is a list of corrupt TPMs which are part of an entirely corrupt platform; ESK and EPK are lists of secret and public TPM endorsement keys respectively; HSK is a list of honest TPM signing keys; CRE is a list of signing credentials; SL is a list of signatures obtained from the SignO oracle; CL records the identity pair and list of basenames sent in queries to ChalO; TS is a list of communication transcripts of calls to the IssueO oracle; CSK is a list of corrupt TPM's secret keys extracted . We summarise these for reference in Table 3.1.
The following is an informal explanation of the seven oracles which will be used in our property-based definition. Note that some of the following oracles take an extra party parameter as input. The party parameter may take one of the values in $\{\mathcal{M}, \mathcal{P}\}$ (TPM or entire platform) and is used to indicate the honest party which the oracle is to play the part of (where it is unclear).

AddHonO$(\cdot;\mathsf{party})$: The *add honest party* oracle on input a TPM identity i, creates an honest TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{M}$) or a platform containing TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{P}$). It generates a pair of TPM endorsement keys $(\mathsf{esk}, \mathsf{epk})$ for the TPM. Finally, it returns the TPM's public endorsement key epk.

AddCrptO$(\cdot, \mathsf{epk}; \mathsf{party})$: The *add corrupt party* oracle on input a TPM identity i and an arbitrary string epk, creates a corrupt TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{M}$) or platform containing TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{P}$) and sets the TPM's public endorsement key as epk. It returns 1 to indicate completion of this task.

InitO–P$(\cdot)$[2]: The *initialise platform* oracle on input a TPM identity i, creates a signing key sk and credential cre for the platform with TPM i. This is done by retrieving the TPM endorsement

[2] We note that this is a special oracle only used to define correctness of the DAA scheme.

keys $(\mathsf{esk}, \mathsf{epk})$ and running the entire interactive join protocol on behalf of the TPM, host and issuer. The oracle returns 1 to indicate completion of this task.

JoinO$(\cdot; \mathsf{party})$: The *join party* oracle on input a TPM identity i, allows an adversary to perform the join protocol with an honest TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{M}$) or a platform containing TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{P}$). If $\mathsf{party} = \mathcal{M}$, then the oracle plays the role of the honest TPM $\mathcal{M}_i$ and executes the interactive join protocol with the adversary by running the JoinTPM algorithm. If $\mathsf{party} = \mathcal{P}$, then the oracle plays the role of the honest platform and executes the interactive join protocol with the adversary by running the JoinTPM and JoinH algorithms. The oracle returns 1 to indicate completion of this task.

IssueO$(\cdot)$: The *issue* oracle on input a TPM identity i, allows an adversary to perform the join protocol with an issuer to obtain signing credentials. The oracle runs the Issue algorithm to create a credential cre for the platform with TPM i. If TPM i is corrupt, then the oracle stores the transcript TS$[i]$ of the communication with the adversary. Finally, the oracle returns cre.

SignO$(\cdot, \cdot, \cdot, \cdot; \mathsf{party})$: The *signing* oracle on input a TPM identity i, a message m, a basename bsn, generates a signature made by an honest TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{M}$) or a platform containing TPM $\mathcal{M}_i$ (if $\mathsf{party} = \mathcal{P}$). If $\mathsf{party} = \mathcal{M}$, the oracle adds an entry $(i, \mathsf{m}, \mathsf{bsn}, *)$ to the signing queries list SL and returns 1 to indicate completion of this task. Here, '$*$' captures the fact that the TPM (played by the oracle) does *not* see the final signature. If $\mathsf{party} = \mathcal{P}$, the oracle adds an entry $(i, \mathsf{m}, \mathsf{bsn}, \sigma)$ to SL and returns the signature σ.

ChalO$(\cdot, \cdot, \cdot, \cdot; b)$: The *challenge* oracle on input two TPM identities i_0 and i_1, a message m and basename bsn, generates a signature on message m and basename bsn under identity i_b. The oracle returns the signature σ.

IdentifyO$(\cdot, \cdot, \cdot, \cdot)$: The *identify* oracle, on input a TPM identity i, a signature σ, message m, and basename bsn, runs Identify$(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{HSK}[i])$[3]. The oracle returns the resulting output bit.

Non-Concurrent Oracle Access. Building on the intuitive explanation of the oracles, we present in Figure 3.1 the formal oracle definitions with non-concurrent access. We opted to make our oracle definitions short and accessible so that the proofs of the property-based definition would remain simple (relatively to the proofs in the UC setting). In Figure 3.1, we assume that the adversary cannot interleave the interactive protocols run between the JoinO, IssueO, and SignO oracles. We also assume that algorithms JoinTPM, JoinH, Issue, and SignTPM output some special symbol indicating failure of execution and say that the algorithm "terminates" when it outputs a valid value. A more stringent formalization will be provided for the concurrent setting in Appendix A, Figure A.1.

Concurrent Oracle Access. While the property-based definition defined through the oracles with non-concurrent access is very handy and intuitive, unfortunately, it does not imply UC-security. To this end, we generalize the oracles in Figure 3.1 to capture concurrent access by using the notion of stateful oracles as done by Bellare et al. [6] and Bootle et al. [14] for groups signatures. The formal description is provided in Appendix A, Figure A.1. As can be seen there, the adversary can interleave oracle calls and interact with the oracles in a concurrent fashion; this is captured by slightly modifying the interactive oracles JoinO, IssueO, and SignO to take an extra input called the protocol message M_{in}. More details will be provided in Appendix A. Although these may look quite

[3] This algorithm determines whether a given signature was signed using a given key. See Definition 1 for a formal definition of this algorithm.

AddHonO$(i;$ party$)$
- **If** $i \in$ HT $\cup$ HP $\cup$ CT $\cup$ CP, then **Return** $\perp$.
- **If** party $= \mathcal{M}$, then HT $\leftarrow$ HT $\cup \{i\}$.
- **If** party $= \mathcal{P}$, then HP $\leftarrow$ HP $\cup \{i\}$.
- (esk, epk) $\leftarrow$ TPM.Kg(param).
- (ESK$[i]$, EPK$[i]$) $:=$ (esk, epk).
- **Return** EPK$[i]$.

AddCrptO$(i,$ epk; party$)$
- **If** $i \in$ HT $\cup$ HP $\cup$ CT $\cup$ CP, then **Return** $\perp$.
- **If** party $= \mathcal{M}$, then CT $\leftarrow$ CT $\cup \{i\}$.
- **If** party $= \mathcal{P}$, then CP $\leftarrow$ CP $\cup \{i\}$.
- (ESK$[i]$, EPK$[i]$) $:=$ ($\perp$, epk).
- **Return** accept.

JoinO$(i;$ party$)$
– **If** party $= \mathcal{P}$:
- **If** $i \notin$ HP or CRE$[i] \neq \perp$, then **Return** $\perp$.
- Run $\langle$JoinTPM(ESK$[i]$), JoinH(EPK$[i]$, ipk), $\star\rangle$, where $\star$ is an arbitrary algorithm run by the adversary.
- If JoinTPM terminates, then HSK$[i] :=$ sk$_i$.
- If JoinH terminates, then CRE$[i] :=$ cre$_i$.
- **Return** 1.

– **If** party $= \mathcal{M}$:
- **If** $i \notin$ HT, then **Return** $\perp$.
- Run $\langle$JoinTPM(ESK$[i]$), $\star$, $\star\rangle$, where $\star$ is an arbitrary algorithm run by the adversary.
- If JoinTPM terminates, HSK$[i] :=$ sk$_i$.
- **Return** 1.

IssueO(i)
– **If** $i \in$ HT:
- Run $\langle$JoinTPM(ESK$[i]$), $\star$, Issue(isk, EPK$[i]$, ML)$\rangle$, where $\star$ is an arbitrary algorithm run by the adversary.
- **If** Issue terminates, then CRE$[i] :=$ cre$_i$.
- **Return** CRE$[i]$.

– **If** $i \in$ CP:
- Run $\langle \star, \star,$ Issue(isk, EPK$[i]$, ML)$\rangle$, where $\star$ is an arbitrary algorithm run by the adversary.
- **If** Issue terminates, then CRE$[i] :=$ cre$_i$, and record the view of Issue in the transcript list TS$[i]$.
- **Return** CRE$[i]$.

InitO–P(i)
- **If** $i \notin$ HP, then **Return** $\perp$.
- Run $\langle$JoinTPM(ESK$[i]$), JoinH(EPK$[i]$, ipk), Issue(isk, EPK$[i]$, ML)$\rangle$ to obtain HSK$[i]$, CRE$[i]$, and an updated members list ML respectively.
- **Return** 1.

SignO$(i,$ m, bsn; party$)$
– **If** party $= \mathcal{P}$:
- **If** $i \notin$ HP or CRE$[i] = \perp$ or HSK$[i] = \perp$, then **Return** $\perp$.
- Run $\langle$SignTPM(HSK$[i]$, m, bsn), SignH(CRE$[i]$, m, bsn)$\rangle$ to obtain signature σ.
- SL $\leftarrow$ SL $\cup \{(i,$ m, bsn, $\sigma)\}$.
- **Return** σ.

– If party $= \mathcal{M}$:
- **If** $i \notin$ HT or HSK$[i] = \perp$, then **Return** $\perp$.
- Run $\langle$SignTPM(HSK$[i]$, m, bsn), $\star\rangle$, where $\star$ is an arbitrary algorithm run by the adversary.
- **If** SignTPM terminates, then SL $\leftarrow$ SL $\cup \{(i,$ m, bsn, $\perp)\}$.
- **Return** 1.

ChalO$(i_0, i_1,$ bsn, m; $b)$
- **If** $i_{b'} \notin$ HP or CRE$[i_{b'}] = \perp$ or HSK$[i_{b'}] = \perp$ for $b' \in \{0, 1\}$, then **Return** $\perp$.
- **If** $|$CL$| \geq 1$ and $(i_0, i_1, *) \notin$ CL, then **Return** $\perp$.
- Run $\langle$SignTPM(HSK$[i_b]$, m, bsn), SignH(CRE$[i_b]$, m, bsn)$\rangle$ to obtain challenge signature σ.
- CL $\leftarrow$ CL $\cup \{(i_0, i_1,$ bsn$)\}$.
- **Return** σ.

IdentifyO$(i, \sigma,$ m, bsn$)$
- If $i \notin$ HT $\cup$ HP, then **Return** $\perp$.
- $b \leftarrow$ Identify$(\sigma,$ m, bsn, HSK$[i])$.
- **Return** b.

Figure 3.1: Oracles with non-concurrent access.

different from Figure 3.1, we emphasize that in essence they both capture the aforementioned intuitive oracle description, and the only difference lies in how we model concurrent access.

Remark 4 (Considerations for oracle use)*.* The oracle definitions given in Figure 3.1 are designed to provide the reader with a clear intuition as to the function of each of them so that one might use them in a security proof. Whilst simple, the use of these oracles should be made with careful attention to the following details, which are covered comprehensively by the more granular definitions of Appendix A. One should allow an adversary, calling one of the interactive oracles, to discontinue that interaction before the interaction is complete or even begin communicating with another oracle before replying to the first. To do so, Figure A.1 uses stateful oracles as used by Bellare et al. in [6]. This notation covers the possibility that an adversary may mix and match inputs and outputs from different oracles. Also note that the interactive algorithms (SignO, JoinO, IssueO) additionally take as input a protocol message M_{in}. We omit these in Figure 3.1 for readability but they can be found in the full oracle definitions of Appendix A.

Auxiliary Algorithms. In the unforgeability requirement of DAA it will be necessary to know given a signature σ, message m, basename bsn, whether this signature was produced by a TPM who has joined. Precisely, was this signature signed with a key that was earlier used in a join session (and thus on which signing credentials have been issued). We formalise this notion by defining an extraction algorithm Ext which, given a join transcript and a set of trap-doored parameters, can extract the TPM secret key used in that transcript. Then, given a TPM key and a join transcript, we also define an $\mathsf{Identify}_\mathsf{T}$ algorithm which determines whether the key was used in a given transcript or not. Finally, a third algorithm Identify is used to determine whether a given signature was generated using a given TPM key. The interdependencies of these three algorithms defined in Definition 1 provide us a framework for determining whether a signature was created using a TPM who has joined or not.
In the following definition we denote the space of all TPM secret keys $\mathcal{K}$.

Definition 1. *For any DAA scheme* DAA*, we require the existence of auxiliary algorithms* SimSetup*, and deterministic algorithms* Ext*,* Identify*, and* $\mathsf{Identify}_\mathsf{T}$ *with the following properties where the oracles provided to the adversary* $\mathcal{A}$ *are defined in Figure A.1:*

1. *For all PPT adversaries* $\mathcal{A}$*, if we run* $(\mathsf{param}, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$ *and* $\mathsf{param}' \leftarrow \mathsf{Setup}(1^\lambda)$*, then we have*

$$\left|\Pr[\mathcal{A}(1^\lambda, \mathsf{param}) \to 1] - \Pr[\mathcal{A}(1^\lambda, \mathsf{param}') \to 1]\right| = \mathsf{negl}(\lambda),$$

   where the probability is taken over the randomness used by SimSetup *and* Setup*.*

2. *For all* ipk *such that* $\mathsf{IpkVf}(\mathsf{ipk}, \mathsf{param}) = 1$*, where* $\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda)$ *or* $\mathsf{param} \leftarrow \mathsf{SimSetup}(1^\lambda)$*, then given a signature* σ *on some message* m *and basename* bsn *such that* $\mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}) = 1$*, there exists a unique* $\mathsf{sk} \in \mathcal{K}$ *such that* $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{sk}) = 1$*.*

3. *For all* $(\mathsf{param}, \mathsf{td}) \in \mathsf{SimSetup}(1^\lambda)$*,* $(\mathsf{ipk}, \mathsf{isk}, \mathsf{ML}) \in \mathsf{I.Kg}(\mathsf{param})$*, and PPT adversary* $\mathcal{A}$ *with oracle access to* $(\mathsf{AddHonO}(\cdot; \mathcal{M})$*,* $\mathsf{AddCrptO}(\cdot, \cdot; \mathcal{P})$*,* $\mathsf{JoinO}(\cdot; \mathcal{M})$*,* $\mathsf{SignO}(\cdot, \cdot, \cdot; \mathcal{M})$*,* $\mathsf{IssueO}(\cdot))$*, then for any* i *such that* $\mathsf{CRE}[i] \neq \perp$ *we have:*

   - $\mathsf{Identify}_\mathsf{T}(\mathsf{TS}[i], \mathsf{sk}) = 1$*, where* $\mathsf{sk} \leftarrow \mathsf{Ext}(\mathsf{param}, \mathsf{td}, \mathsf{ipk}, \mathsf{isk}, \mathsf{TS}[i])$*, and*
   - *there is a unique* $\mathsf{sk} \in \mathcal{K}$ *such that* $\mathsf{Identify}_\mathsf{T}(\mathsf{TS}[i], \mathsf{sk}) = 1$*.*

Unforgeability using Identify: In the unforgeability game, we allow the adversary to make join requests to the honest issuer on behalf of a corrupt host or entirely corrupt platform. The adversary

will eventually output a signature which it claims wasn't signed using any TPM that has joined or that originates from an honest TPM who never signed it. In order to check this claim, we extract (using Ext) the TPM secret key from the join transcript of each corrupt platform that joins. By definition of $\mathsf{Identify}_{\mathsf{T}}$, we know that an extracted key is exactly the one used in the given join transcript. We then run Identify with the given signature on each of the honest TPM signing keys and also the extracted secret keys (for the corrupt TPMs) in turn. If the signature doesn't identify to any of these then the adversary wins the game. This precisely captures the property that no adversary can make valid signatures without using a key for which signing credentials have been issued. Readers familiar with the group signature setting will note the similarity of this notion to traceability. Finally, the adversary also wins if the signature identifies to an honest TPM who never signed it.

Key-based revocation: To instantiate secret key-based revocation we assume that the Identify algorithm is run as a subroutine of Verify to check that the given signature doesn't identify to any of the compromised TPM keys listed in the revocation list RL. Precisely, in order to verify a signature σ with respect to a message m, basename bsn, and revocation list RL one does the following.

1. Set $b \leftarrow \mathsf{Verify}(\sigma, \mathsf{m}, \mathsf{bsn}, \emptyset)$, where $\emptyset$ is the empty list.
2. For each $\mathsf{sk}_i \in \mathsf{RL}$, set $b_i \leftarrow \mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{sk}_i)$.
3. If $b = 0$ or $\exists\, i$ such that $b_i = 1$, then verification fails.
4. Else, verification passes.

We note that this 'hard-coding' of the Verify algorithm might appear quite restrictive. However, this assumption allows for a smooth proof of Theorem 1. Whilst this may be an artifact of the proof, we don't consider it to be a strong assumption since all existing secure DAA schemes satisfy this. We further note that this assumption naturally imposes the desired security properties on the revocation process. For example, a signature cannot pass verification if it identifies to some key on the revocation list. Furthermore, a previously invalid signature cannot be made valid by adding extra keys to the revocation list. Finally, if one only requires stand-alone security, i.e., security which does not necessarily imply UC security, it may be more intuitive and allow for more flexible scheme design if one formulated the above revocation guarantees as an experiment instead of hard-coding Verify in this way.

Security Definitions. Here we define the properties of a secure DAA scheme: correctness, anonymity, unforgeability, and non-frameability. The experiments referred to in the following definitions are presented in Figure 3.2.

Correctness. We require that signatures produced by honest platforms are accepted by the Verify algorithm. In addition, we require that two signatures produced by the same platform with the same basename $\mathsf{bsn} \neq \perp$ are linked w.r.t. Link. Finally, Link is required to be symmetric.

Definition 2. *A DAA scheme* DAA *is said to be correct if for all $\lambda \in \mathbb{N}$, the advantage,*

$$\mathsf{Adv}^{\mathit{Corr}}_{\mathsf{DAA},\mathcal{A}}(\lambda) := \mathit{Pr}[\mathsf{Exp}^{\mathsf{Corr}}_{\mathsf{DAA},\mathcal{A}}(\lambda) = 1]$$

is negligible (in λ) for all PPT adversaries $\mathcal{A}$.

Anonymity. It is essential that attestations made by a platform do not reveal anything about its identity. Precisely, we require that no adversary should be able to determine which of two honest platforms has produced a given signature, even if he controls the issuer. Note that anonymity can only be preserved for fully honest platforms. Clearly, a corrupt host can append identifying information to any signature to reveal the platform's identity since it is the only one able to interact directly with the outside world.

Experiment: $\mathsf{Exp}^{\mathsf{Corr}}_{\mathsf{DAA},\mathcal{A}}(\lambda)$
– $\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda)$; $\mathsf{HP}, \mathsf{EPK}, \mathsf{ESK}, \mathsf{CRE}, \mathsf{HSK} := \emptyset$.
– $(\mathsf{isk}, \mathsf{ipk}, \mathsf{ML}) \leftarrow \mathsf{I.Kg}(\mathsf{param})$
– **If** $\mathsf{IpkVf}(\mathsf{ipk}, \mathsf{param}) = 0$, then **Return** 0.
– $(i, \mathsf{m}_0, \mathsf{m}_1, \mathsf{bsn}) \leftarrow \mathcal{A}(\mathsf{ipk}; \mathsf{AddHonO}(\cdot; \mathcal{P}), \mathsf{InitO–P}(\cdot))$
– $\sigma_b \leftarrow \mathsf{SignO}(i, \mathsf{m}_b, \mathsf{bsn}; \mathcal{P})$ for $b \in \{0, 1\}$
– **If** $\mathsf{Verify}(\mathsf{ipk}, \sigma_0, \mathsf{m}_0, \mathsf{bsn}, \emptyset) = 0$ or $\mathsf{Verify}(\mathsf{ipk}, \sigma_1, \mathsf{m}_1, \mathsf{bsn}, \emptyset) = 0$, then **Return** 0.
– **If** $\mathsf{bsn} \neq \perp$ and $\mathsf{Link}(\mathsf{ipk}, \sigma_0, \sigma_1, \mathsf{m}_0, \mathsf{m}_1, \mathsf{bsn}) = 0$ or $\mathsf{Link}(\mathsf{ipk}, \sigma_1, \sigma_0, \mathsf{m}_0, \mathsf{m}_1, \mathsf{bsn}) = 0$, then **Return** 1.
– **Return** 0.

$\mathsf{Exp}^{\mathsf{Anon–b}}_{\mathsf{DAA},\mathcal{A}}(\lambda)$
– $\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda)$; $\mathsf{HP}, \mathsf{EPK}, \mathsf{ESK}, \mathsf{CRE}, \mathsf{HSK}, \mathsf{SL}, \mathsf{CL} := \emptyset$.
– $(\mathsf{ipk}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{param})$.
– **If** $\mathsf{IpkVf}(\mathsf{ipk}, \mathsf{param}) = 0$, then **Return** 0.
– $b^* \leftarrow \mathcal{A}(\mathsf{param}, \mathsf{st}; \mathsf{AddHonO}(\cdot; \mathcal{P}), \mathsf{JoinO}(\cdot; \mathcal{P}), \mathsf{SignO}(\cdot, \cdot, \cdot; \mathcal{P}), \mathsf{ChalO}(\cdot, \cdot, \cdot, \cdot; b))$.
– **If** $\exists (i, \mathsf{m}, \mathsf{bsn}, \sigma) \in \mathsf{SL}$ s.t. $\mathsf{bsn} \neq \perp$ and $(i, *, \mathsf{bsn}) \in \mathsf{CL}$ or $(*, i, \mathsf{bsn}) \in \mathsf{CL}$ then **Return** 0.
– **If** $b^* = b$, then **Return** 1.
– **Return** 0.

$\mathsf{Exp}^{\mathsf{UF}}_{\mathsf{DAA},\mathcal{A}}(\lambda)$
– $(\mathsf{param}, \mathsf{td}) \leftarrow \mathsf{SimSetup}(1^\lambda)$; $\mathsf{HT}, \mathsf{CP}, \mathsf{EPK}, \mathsf{ESK}, \mathsf{CRE}, \mathsf{HSK}, \mathsf{SL} := \emptyset$.
– $(\mathsf{ipk}, \mathsf{isk}, \mathsf{ML}) \leftarrow \mathsf{I.Kg}(\mathsf{param})$.
– $(\sigma, \mathsf{m}, \mathsf{bsn}) \leftarrow \mathcal{A}(\mathsf{param}, \mathsf{ipk}; \mathsf{AddHonO}(\cdot; \mathcal{M}), \mathsf{AddCrptO}(\cdot, \cdot; \mathcal{P}), \mathsf{JoinO}(\cdot; \mathcal{M}), \mathsf{IssueO}(\cdot), \mathsf{SignO}(\cdot, \cdot, \cdot; \mathcal{M}), \mathsf{IdentifyO}(\cdot, \cdot, \cdot, \cdot))$.
– **If** $\mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}, \emptyset) = 0$, then **Return** 0.
– For all $i \in \mathsf{CP}$, run $\mathsf{sk}_i \leftarrow \mathsf{Ext}(\mathsf{param}, \mathsf{td}, \mathsf{ipk}, \mathsf{isk}, \mathsf{TS}[i])$, and set $\mathsf{sk}_i \leftarrow \mathsf{CSK}[i]$.
– **If** $\exists i \in \mathsf{HT}$ s.t $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{HSK}[i]) = 1$ and $(i, \mathsf{m}, \mathsf{bsn}, *) \notin \mathsf{SL}$, then **Return** 1.
– **If** $\forall i \in \mathsf{CP}$, $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{CSK}[i]) = 0$, then **Return** 1.
– **Return** 0.

$\mathsf{Exp}^{\mathsf{NF}}_{\mathsf{DAA},\mathcal{A}}(\lambda)$
– $\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda)$; $\mathsf{HP}, \mathsf{EPK}, \mathsf{ESK}, \mathsf{CRE}, \mathsf{HSK}, \mathsf{SL} := \emptyset$.
– $(\mathsf{ipk}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{param})$.
– **If** $\mathsf{IpkVf}(\mathsf{ipk}, \mathsf{param}) = 0$, then **Return** 0.
– $(i, \sigma, \mathsf{m}, \mathsf{bsn}) \leftarrow \mathcal{A}(\mathsf{param}, \mathsf{st}; \mathsf{AddHonO}(\cdot; \mathcal{P}), \mathsf{JoinO}(\cdot; \mathcal{P}), \mathsf{SignO}(\cdot, \cdot, \cdot; \mathcal{P}), \mathsf{IdentifyO}(\cdot, \cdot, \cdot, \cdot))$.
– **If** $\mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}, \emptyset) = 0$, then **Return** 0.
– **If** $i \notin \mathsf{HP}$ or $(i, \mathsf{m}, \mathsf{bsn}, \sigma) \in \mathsf{SL}$, then **Return** 0.
– **If** $\exists\, (i, \mathsf{m}^*, \mathsf{bsn}, \sigma^*) \in \mathsf{SL}$ s.t. $\mathsf{Link}(\sigma, \sigma^*, \mathsf{m}, \mathsf{m}^*, \mathsf{bsn}) = 1$, then **Return** 1.
– **If** $\mathsf{bsn} = \perp$, $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{HSK}[i]) = 1$, and $(i, \mathsf{m}, \mathsf{bsn}, *) \notin \mathsf{SL}$, then **Return** 1.
– **Return** 0.

Figure 3.2: Security experiments for property-based definition of DAA.

The adversary's aim is to guess which of two TPMs (of his choice) has signed a chosen message and basename. The powers afforded to $\mathcal{A}$ include control of the issuer and so his first action must be to output the issuer's public key. Secondly, he is given oracle access allowing him to create honest platforms which he can instruct to join and request signatures from. $\mathcal{A}$ may also call the challenge oracle ChalO on two honest platforms, and a sequence of message and a basename pairs on which he receives signatures signed under one of those identities (fixed for all queries). He must then output a bit b^* to indicate which of the two TPMs he thinks has signed the signatures returned by ChalO. $\mathcal{A}$ wins the game if he guesses correctly. We indicate the shared state of the two adversarial actions using st. Note that we do not explicitly give $\mathcal{A}$ the power to create corrupt TPMs and platforms since he controls the issuer and can therefore simulate corrupt TPMs and platforms internally.
Since we instantiate our model with the Identify algorithm which tells us whether a given signature was created under a given secret key, any user can trivially determine whether a signature was

created under their own secret key. Furthermore, $\mathcal{A}$ cannot query both the signing oracle and the challenger with the same identity and basename. Otherwise, $\mathcal{A}$ could simply win the game by using the Link algorithm.

Remark 5 (Anonymity with subverted TPM). There is one line of work which defines anonymity in the presence of a subverted TPM [20], [17], and only the model [17] allows one to construct efficient schemes. Whilst the anonymity definition there is stronger, one can only model a specific form of 'isolated' corruption in which only the code of a TPM can be specified by the adversary who does not have continued access to the TPM. For the sake of simplicity and of focusing on conventional schemes we do not consider anonymity of an honest host with corrupt TPM in this work.

Remark 6 (Alternative definition for anonymity). The reader will note that the anonymity experiment defined in Figure 3.2 is in the 'left-or-right' style where the challenger chooses between one of two TPMs supplied by the adversary who must determine which is the signatory. In the full paper [] we present an alternative 'real-or-random' variant in which the adversary supplies only a single TPM identity and the challenger either answers signing queries using that platform or creates a fresh TPM (once and for all) and uses this TPM to answer the queries. The adversary must then decide whether signatures were produced by the platform he chose or by the fresh one. We show the equivalence of these two notions as long as $\mathcal{A}$ is required to output a proof of knowledge of his secret key. We emphasise their equivalence since the 'real-or-random' variant will be useful for our UC implication.

Definition 3 (Anonymity of DAA)**.** *A DAA scheme* DAA *is said to be anonymous if and only if for all $\lambda \in \mathbb{N}$, the advantage*

$$\mathsf{Adv}^{\mathsf{Anon}}_{\mathsf{DAA},\mathcal{A}}(\lambda) := |\Pr[\mathsf{Exp}^{\mathsf{Anon\text{-}0}}_{\mathsf{DAA},\mathcal{A}} = 1] - \Pr[\mathsf{Exp}^{\mathsf{Anon\text{-}1}}_{\mathsf{DAA},\mathcal{A}} = 1]|$$

is negligible (in λ) for all PPT adversaries $\mathcal{A}$.

Unforgeability. Unforgeability ensures that no adversary can produce more unlinkable signatures than he has corrupt TPMs and cannot produce a signature that identifies to an honest TPM who never took part in its signing. This property is unique in that it can only hold when the issuer is honest, else a corrupt issuer simply creates dummy users which cannot be identified by the challenger. Unforgeability must be maintained for platforms containing an honest TPM, even if the host is corrupt and says that no adversary can create a signature that identifies to an honest TPM who never signed it *or* that does not identify to any TPM (honest or corrupt) that has joined. The challenger **C** begins by generating the issuer's secret and public key. Next, the adversary $\mathcal{A}$ makes a round of oracle queries. These allow him to create honest TPMs, create corrupt platforms, join using an honest TPM or corrupt platform, use an honest TPM to create signatures, and determine whether a given signature was signed by a given TPM. The adversary must then output a signature on a message and basename of his choice. The Ext algorithm is used by the challenger to extract secret keys for each of the corrupt TPM's that $\mathcal{A}$ is using to join. The challenger then checks that the output signature is valid. Next, the challenger checks whether the signature identifies to any of the honest TPMs (for which **C** knows sk) but was never signed by them. If so, $\mathcal{A}$ wins the game. Finally, $\mathcal{A}$ also wins the game if the output signature does not identify to any of the TPMs that have joined (honest or corrupt).
After creating a corrupt TPM, an adversary can simulate the part of an honest host himself. Thus, where there is a corrupt TPM, we assume the host to be corrupt also. Similarly, when a TPM is honest we assume the host to be corrupt. This is also because we do not require unforgeability to hold for a corrupt TPM and honest host.

Definition 4 (Unforgeability of DAA)**.** *A DAA scheme* DAA *is said to be unforgeable if and only if for all* $\lambda \in \mathbb{N}$*, the advantage,*

$$\mathsf{Adv}^{UF}_{\mathsf{DAA},\mathcal{A}}(\lambda) := Pr[\mathsf{Exp}^{\mathsf{UF}}_{\mathsf{DAA},\mathcal{A}}(\lambda) = 1]$$

is negligible (in λ*) for all PPT adversaries* $\mathcal{A}$*.*

Remark 7 (Absence of revocation list in unforgeability experiment)*.* We note that our unforgeability experiment makes no mention of revocation lists. This might seem strange to the reader, however the security properties relating to the revocation process are covered by the assumption that Verify is hard-coded to internally run Identify on each secret key of RL. Indeed, this is proven formally via our UC implication Theorem 1. One can see this intuitively in the following way. Consider a modified unforgeability experiment where winning conditions depend on a non-empty revocation list as done in security experiments of group signature schemes such as [11, 39]. It is easy to see that winning outputs of such an experiment would also constitute a winning output of our current same *assuming* that Verify is hard-coded in this way.

Remark 8 (Redundancies omissible for stand-alone security)*.* At this point it is important to note that there are a few features of the property-based security model which, while necessary for Theorem 1 to hold, may not appear to provide meaningful security enhancements if one only requires stand-alone security.
First, the non-frameability experiment prevents an adversary from creating a signature under $\mathsf{bsn} = \perp$ which identifies to some honest platform who never signed it, even when the issuer is corrupt. In practice, this kind of 'framing via secret key' is never noticed since one would need the secret key of the honest platform in order to test that the signature identifies. Therefore, the effective result of such an attack is for an adversary to make valid signatures. However, if the issuer is corrupt, the adversary can make valid signatures anyway by internally running a corrupt platform.
In both the unforgeability game and the non-frameability game, the adversary is provided with an $\mathsf{IdentifyO}(\cdot,\cdot,\cdot,\cdot)$ oracle which allows him to determine whether a given signature was signed by a particular TPM. On first glance, this seems redundant since considering only the security experiments defined in Figure 3.2, all signatures are made through signing queries or by the adversary himself so he can always determine the TPM who signed a signature without needing to call the $\mathsf{IdentifyO}(\cdot,\cdot,\cdot,\cdot)$ oracle. However, this oracle turns out to be necessary to simulate the Verify interface of $\mathcal{F}_{\mathsf{daa}}$ in the proof of Theorem 1. In this simulation, the simulator needs to know whether the signature provided for verification comes from one of the honest TPMs or from somewhere else. This is necessary since he wants to determine whether the input signature constitutes some kind of forgery or framing. He must first compile a list of all existing TPMs to which the signature identifies. Since he doesn't know the secret keys of honest TPMs (he created these using calls to his property-based oracles) he cannot test for such identifications. Instead, he calls the $\mathsf{IdentifyO}(\cdot,\cdot,\cdot,\cdot)$ with each honest TPM in order to determine this. Note that our model (and $\mathcal{F}_{\mathsf{daa}}$) only guarantees standard (not strong) existential unforgeability. Otherwise, when provided a signature σ on some message m for verification, the simulator could check whether any of the honest TPMs had signed m and if σ was not what the signing oracle returned then this would constitute a forgery of a *strong* unforgeability experiment.
We note that one could omit the above features of the experiments when proving the security of a scheme without losing any meaningful security. However, Theorem 1 regarding UC implication would no longer hold.

Non-Frameability. This ensures that one cannot create signatures on messages that an honest platform never signed but that link to signatures the platform did create. As in anonymity, we

allow the issuer to be corrupt and can only guarantee security for an entirely honest platform. In particular, honest TPMs with a corrupt host cannot be protected from framing. This is because the host controls the output of signatures and could also run a corrupt TPM that had joined and and use the corrupt TPM's key to create signatures instead of using the honest TPM's contribution. The resulting signatures can't be protected from framing since they use the corrupt TPM's signing key.
The adversary $\mathcal{A}$ first chooses the issuer's secret key and announces the corresponding public key since all further actions will depend on this value. Next, he can make queries to oracles that allow him to create honest platforms, join using an honest platform, request signatures from an honest platform on a message and basename of his choice, and determine whether a given signature was signed by a given TPM. He must then output a TPM identity i and a valid signature σ on a message m and basename bsn that isn't in the signing queries list. $\mathcal{A}$ wins the non-frameability game if the output identity is part of an honest platform and there exists a signature signed by that platform which links to the output signature. $\mathcal{A}$ can also win if the output signature, with $\mathsf{bsn} = \perp$, identifies to some honest platform who never signed it.
Note that we do not explicitly allow $\mathcal{A}$ to create corrupt TPMs and platforms since he controls the issuer and can simulate these internally as in the anonymity game.

Definition 5 (Non-frameability of DAA)**.** *A DAA scheme* DAA *is said to be non-frameable if and only if for all $\lambda \in \mathbb{N}$, the advantage,*

$$\mathsf{Adv}^{NF}_{\mathsf{DAA},\mathcal{A}}(\lambda) := Pr[\mathsf{Exp}^{\mathsf{NF}}_{\mathsf{DAA},\mathcal{A}}(\lambda) = 1],$$

is negligible (in λ) for all PPT adversaries $\mathcal{A}$.

Remark 9 (Absence of revocation list in non-frameability experiment)*.* As with our unforgeability experiment, we demonstrate that although our non-frameability experiment makes no mention of revocation, the assumption that Verify is hard-coded to run $\mathsf{Identify}$ on each value in RL covers all desirable properties relating to revocation within non-frameability. Indeed, this is proven via our UC implication Theorem 1. Below we give the intuition as to why this is true.
Consider a modified non-frameability experiment in which the adversary may additionally output a revocation list RL and we check that his signature is valid by running $\mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{RL})$. Note that this is what is done in the security experiments of group signatures with verifier local revocation [12, 39]. One can easily show that if $\mathcal{A}$ can give a winning output in this modified experiment, it gives us one for the current experiment *assuming* that Verify is hard-coded as described.

Now that we have formalised all the security requirements of DAA we can define the overall DAA security. The definition acknowledges three extra assumptions. Firstly, the existence of the auxiliary algorithms $\mathsf{SimSetup}$, $\mathsf{Identify}$, $\mathsf{Identify}_\mathsf{T}$, and Ext defined in Definition 1. Secondly, that KBR is instantiated using a Verify algorithm that internally runs $\mathsf{Identify}$ on each key in the revocation list. Finally, we assume signatures link if and only if they identify to a common signing key. Including this last condition allows for much more succinct property definitions.

Definition 6 (DAA security)**.** *A DAA scheme* DAA *is said to be secure if and only if it is correct, anonymous, unforgeable, non-frameable, and the following hold:*

1. *There exist auxiliary algorithms* $\mathsf{SimSetup}$*,* $\mathsf{Identify}$*,* $\mathsf{Identify}_\mathsf{T}$*, and online-extraction algorithm* Ext *as defined in Definition 1.*

2. *We assume that key-based revocation is instantiated by a verification algorithm that implicitly runs* $\mathsf{Identify}$ *on each secret key of the provided revocation list.*

3. *Given two valid signature-message-basename tuples* $(\sigma, \mathsf{m}, \mathsf{bsn})$ *and* $(\sigma', \mathsf{m}', \mathsf{bsn})$ *for a common basename* $\mathsf{bsn} \neq \perp$*, it holds that* $\mathsf{Link}(\sigma, \sigma', \mathsf{m}, \mathsf{m}', \mathsf{bsn}) = 1$ *if and only if there exists an input* sk *such that* $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{sk}) = \mathsf{Identify}(\sigma', \mathsf{m}', \mathsf{bsn}, \mathsf{sk}) = 1$.

3.1.3 Property-based to UC Model Implication

Here we construct the DAA wrapper protocol Π_{DAA} shown in Figures 3.3 and 3.4 which provides a generic transformation of a property-based DAA scheme DAA to one compatible with the universal composability framework.

This transformation is essentially a syntactic re-structuring. The main function of the wrapper is to define the interfaces of Π_{DAA} so that they match those of $\mathcal{F}_{\mathsf{daa}}$ [19] (See Appendix of full paper [] for the $\mathcal{F}_{\mathsf{daa}}$ functionality). Figures 3.3 and 3.4 show how one can use the algorithms of DAA to construct the abstract scheme Π_{DAA}. The next step is then to prove that this abstract scheme realises the DAA functionality $\mathcal{F}_{\mathsf{daa}}$ assuming that DAA is secure according to Definition 6.

Additionally, for the interactive protocols join and sign, the wrapper uses the following convention. Where interactive protocols occur (joining and signing), messages output by the respective algorithms are enhanced (by the wrapper) with algorithm and session specific identifiers. For example, if JoinH sends a request $\mathsf{M}_{\mathsf{Issue}}$ to join with $\mathcal{M}_i$ to the issuer, then the wrapper sends $(\mathsf{JOIN}, \mathsf{sid}, \mathsf{jsid}, \mathcal{M}_i, \mathsf{M}_{\mathsf{Issue}})$ to $\mathcal{I}$. Here sid is the session identifier for that run of the protocol whereas jsid is the join session identifier specific to that platform's join session. This syntax allows us to distinguish between protocol messages and objects added by the wrapper. Furthermore, the wrapper assumes that the TPM manufacturer has already embedded all TPMs with endorsement keys and has registered the list of public endorsement keys with $\mathcal{F}_{\mathsf{ca}}$. This is equivalent to having already run TPM.Kg and registering all epk's with $\mathcal{F}_{\mathsf{ca}}$. To make the protocol Π_{DAA} more readable we omit explicit calls to the sub-functionalities with sub-session IDs etc. and simply say e.g., issuer $\mathcal{I}$ registers its public key with $\mathcal{F}_{\mathsf{ca}}$.

In the theorem which follows we use three hybrid functionalities. These are artifacts of the way in which we prove things in the UC model and have been used in prior works. Their purpose is to provide idealised forms of functionalities that we do not consider cryptographically in our work but whose existence we depend on. $\mathcal{F}_{auth^*}$ models the authenticated channel between TPM and issuer; this is established via an authentication protocol running in parallel with the join protocol and whose details are outside the scope of this work. $\mathcal{F}_{\mathsf{ca}}$ represents the certificate authority with whom the issuer registers its public key and the TPM manufacturer maintains a public list of existing TPM endorsement public keys. Finally, $\mathcal{F}_{\mathsf{crs}}^{D}$ is the common reference string functionality which gives all parties a set of up-to-date system parameters. For formal definitions of $\mathcal{F}_{\mathsf{ca}}$, $\mathcal{F}_{\mathsf{crs}}^{D}$ and $\mathcal{F}_{auth^*}$ we refer the reader to [23], [22], and [19] respectively. We assume, as in the the UC DAA definition, that the host and TPM can communicate directly, meaning that they have an authenticated and perfectly secure channel. This models the physical proximity of the host and TPM forming the platform: if the host is honest an adversary can neither alter nor read their internal communication, or even notice that communication is happening.

We now present our main result showing that the wrapper protocol, if secure according to the new model, enjoys the same security guarantees as provided by the UC definition. There are two additional conditions that a (wrapped) DAA scheme, secure in the property-based model, must satisfy in order to achieve UC security. Firstly, the issuer must register its public key using an extractable proof of knowledge of its secret key. Secondly, the host is assumed to begin and end the interactive join and signing protocols. All of these conditions are met by existing DAA schemes except the first which, where omitted, has lead to security flaws.

Theorem 1. Let DAA be a DAA scheme that is secure according to Definition 6. Then Π_{DAA}

Setup:

1. $\mathcal{I}$ upon input $(\text{SETUP}, \text{sid})$:
   - Checks that $\text{sid} = (\mathcal{I}, \text{sid}')$ for some sid'.
   - Retrieves $\text{param} \leftarrow \mathcal{F}_{\text{crs}}^{D}$.
   - Runs $(\text{ipk}, \text{isk}, \text{ML}) \leftarrow \text{I.Kg}(\text{param})$ and store $(\text{isk}, \text{ML}, \text{sid})$
   - Registers ipk with $\mathcal{F}_{\text{ca}}$.
   - Outputs $(\text{SETUPDONE}, \text{sid})$.

Join Request:

1. $\mathcal{H}_j$ upon input $(\text{JOIN}, \text{sid}, \text{jsid}, \mathcal{M}_i)$:
   - Parses $\text{sid} = (\mathcal{I}, \text{sid}')$, retrieves ipk and epk_i from $\mathcal{F}_{\text{ca}}$.
   - Sets $\text{st}^i_{\text{JoinH}} := (\text{epk}_i, \text{ipk})$, $\text{dec}^i_{\text{JoinH}} := \text{cont}$.
   - Runs $(\text{st}^j_{\text{JoinH}}, \text{M}_{\text{Issue}}, \text{dec}^j_{\text{JoinH}}) \leftarrow \text{JoinH}(\text{st}^i_{\text{JoinH}}; \perp)$.
   - Sends $(\text{JOIN}, \text{sid}, \text{jsid}, (\mathcal{M}_i, \text{M}_{\text{Issue}}))$ to $\mathcal{I}$.
2. $\mathcal{I}$ upon receiving $(\text{JOIN}, \text{sid}, \text{jsid}, (\mathcal{M}_i, \text{M}_{\text{Issue}}))$ from $\mathcal{H}_j$:
   - Creates a join request record $\langle \text{sid}, \text{jsid}, \mathcal{M}_i, \mathcal{H}_j, \text{M}_{\text{Issue}}, \rangle$.
   - Outputs $(\text{JOINPROCEED}, \text{sid}, \text{jsid}, \mathcal{M}_i)$.

Join Proceed:

1. $\mathcal{I}$ upon input $(\text{JOINPROCEED}, \text{sid}, \text{jsid})$:
   - Retrieves both the join record $\langle \text{sid}, \text{jsid}, \mathcal{M}_i, \mathcal{H}_j, \text{M}_{\text{Issue}} \rangle$ and (isk, ML) from its records. It may also retrieve epk_i from $\mathcal{F}_{\text{ca}}$ if not included in M_{Issue}.
   - Sets $\text{st}^i_{\text{Issue}} := (\text{isk}, \text{epk}_i, \text{ML})$, $\text{dec}^i_{\text{Issue}} := \text{cont}$.
   - Runs $(\text{st}^i_{\text{Issue}}, \text{M}_{\text{JoinH}}, \text{dec}^i_{\text{Issue}}) \leftarrow \text{Issue}(\text{st}^i_{\text{Issue}}; \text{M}_{\text{Issue}})$.
   - Sends $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinH}})$ to $\mathcal{H}_j$.

2.($\mathcal{H}$) If $\mathcal{H}_j$ receives a message of the form $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinH}})$ from $\mathcal{I}$, $\mathcal{M}_i$, or $\mathcal{F}_{auth^*}$:
   - If $\text{dec}^i_{\text{JoinH}} \neq \text{cont}$, does nothing. records.
   - Runs $(\text{st}^i_{\text{JoinH}}, \text{M}_{\text{JoinTPM/Issue}}, \text{dec}^i_{\text{JoinH}}) \leftarrow \text{JoinH}(\text{st}^i_{\text{JoinH}}, \text{M}_{\text{JoinH}})$.
   - If $\text{dec}^i_{\text{JoinH}} = \text{accept}$, sets $\text{cre}_i := \text{st}^i_{\text{JoinH}}$ and outputs $(\text{JOINED}, \text{sid}, \text{jsid})$.
   - If $\text{dec}^i_{\text{JoinH}} = \text{cont}$, sends $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinTPM/Issue}})$ to $\mathcal{M}_i/\mathcal{I}$.

2.($\mathcal{M}$) If $\mathcal{M}_i$ receives a message of the form $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinTPM}})$ from $\mathcal{H}_j$:
   - If $\text{st}^i_{\text{JoinTPM}}$ is undefined, retrieves esk_i from its records and sets $\text{st}^i_{\text{JoinTPM}} := \text{esk}_i$, $\text{dec}^i_{\text{JoinTPM}} := \text{cont}$.
   - If $\text{dec}^i_{\text{JoinTPM}} \neq \text{cont}$, does nothing.
   - Runs $(\text{st}^i_{\text{JoinTPM}}, \text{M}_{\text{JoinH/Issue}}, \text{dec}^i_{\text{JoinTPM}}) \leftarrow \text{JoinTPM}(\text{st}^i_{\text{JoinTPM}}; \text{M}_{\text{JoinTPM}})$.
   - If $\text{dec}^i_{\text{JoinTPM}} = \text{accept}$, sets $\text{sk}_i := \text{st}^i_{\text{JoinTPM}}$.
   - Sends $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinH/Issue}})$ to $\mathcal{H}_j$ directly or via $\mathcal{F}_{auth^*}$ respectively.

2.($\mathcal{I}$) If $\mathcal{I}$ receives a message of the form $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{Issue}})$ from $\mathcal{H}_j$ or $\mathcal{F}_{auth^*}$:
   - If $\text{dec}^i_{\text{Issue}} \neq \text{cont}$, does nothing.
   - Runs $(\text{st}^i_{\text{Issue}}, \text{M}_{\text{JoinH}}, \text{dec}^i_{\text{Issue}}) \leftarrow \text{Issue}(\text{st}^i_{\text{Issue}}; \text{M}_{\text{Issue}})$
   - If $\text{dec}^i_{\text{Issue}} = \text{accept}$, sets $\text{ML} = \text{st}^i_{\text{Issue}}$.
   - Sends $(\text{JOIN}, \text{sid}, \text{jsid}, \text{M}_{\text{JoinH}})$ to $\mathcal{H}_j$

Figure 3.3: DAA protocol Π_{DAA}

Sign Request:

1. $\mathcal{H}_j$ upon input $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathcal{M}_i, \mathsf{m}, \mathsf{bsn})$:
   - Creates a sign record $\langle \mathsf{sid}, \mathsf{ssid}, \mathcal{M}_i, \mathsf{m}, \mathsf{bsn} \rangle$.
   - Retrieves cre_j from its records.
   - Sets $\mathsf{st}^i_{\mathsf{SignH}} := (\mathsf{cre}_i, \mathsf{m}, \mathsf{bsn})$, $\mathsf{dec}^i_{\mathsf{SignH}} := \mathsf{cont}$.
   - Runs $(\mathsf{st}^i_{\mathsf{SignH}}, \mathsf{M}_{\mathsf{SignTPM}}, \mathsf{dec}^i_{\mathsf{SignH}}) \leftarrow \mathsf{SignH}(\mathsf{st}^i_{\mathsf{SignH}}; \mathsf{M}_{\mathsf{SignH}})$.
   - Sends $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignTPM}})$ to $\mathcal{M}_i$.
2. $\mathcal{M}_i$ upon receiving $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignTPM}})$ from $\mathcal{H}_j$:
   - Creates a sign record $\langle \mathsf{sid}, \mathsf{ssid}, \mathcal{H}_j, \mathsf{m}, \mathsf{bsn} \rangle$.
   - Outputs $(\mathsf{SIGNPROCEED}, \mathsf{sid}, \mathsf{ssid}, \mathsf{m}, \mathsf{bsn})$.

Sign Proceed:

1. $\mathcal{M}_i$ upon input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, \mathsf{ssid})$:
   - Retrieves the sign record $\langle \mathsf{sid}, \mathsf{ssid}, \mathcal{H}_j, \mathsf{m}, \mathsf{bsn} \rangle$ and sk_i from its records.
   - Sets $\mathsf{st}^i_{\mathsf{SignTPM}} := (\mathsf{sk}_i, \mathsf{m}, \mathsf{bsn})$ and $\mathsf{dec}^i_{\mathsf{SignTPM}} := \mathsf{cont}$.
   - Runs $(\mathsf{st}^i_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^i_{\mathsf{SignTPM}}) \leftarrow \mathsf{SignTPM}(\mathsf{st}^i_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignTPM}})$.
   - Sends $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignH}})$ to $\mathcal{H}_j$.

2.($\mathcal{M}_i$) If $\mathcal{M}_i$ receives a message of the form $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignTPM}})$ from $\mathcal{H}_j$:
   - If $\mathsf{dec}^i_{\mathsf{SignTPM}} \neq \mathsf{cont}$, does nothing.
   - Runs $(\mathsf{st}^i_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^i_{\mathsf{SignTPM}}) \leftarrow \mathsf{SignH}(\mathsf{st}^i_{\mathsf{SignTPM}}; \mathsf{M}_{\mathsf{SignTPM}})$.
   - Sends $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignH}})$ to $\mathcal{H}_j$.

2.($\mathcal{H}$) If $\mathcal{H}_j$ receives a message of the form $(\mathsf{SIGN}, \mathsf{sid}, \mathsf{ssid}, \mathsf{M}_{\mathsf{SignH}})$ from tpm_i:
   - If $\mathsf{dec}^i_{\mathsf{SignH}} \neq \mathsf{cont}$, does nothing.
   - Runs $(\mathsf{st}^i_{\mathsf{SignH}}, \mathsf{M}_{\mathsf{SignTPM}}, \mathsf{dec}^i_{\mathsf{SignH}}) \leftarrow \mathsf{SignH}(\mathsf{st}^i_{\mathsf{SignTPM}}; \mathsf{M}_{\mathsf{SignTPM}})$.
   - If $\mathsf{dec}^i_{\mathsf{SignH}} = \mathsf{accept}$, sets $\sigma := \mathsf{st}^i_{\mathsf{SignH}}$.
   - Outputs $(\mathsf{SIGNATURE}, \mathsf{sid}, \mathsf{ssid}, \sigma)$.

Verify:

1. $\mathcal{V}$ upon input $(\mathsf{VERIFY}, \mathsf{sid}, \mathsf{m}, \mathsf{bsn}, \sigma, \mathsf{RL})$:
   - Sets $f \leftarrow \mathsf{Verify}(\mathsf{ipk}, \sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{RL})$.
   - Outputs $(\mathsf{VERIFIED}, \mathsf{sid}, f)$.

Link:

1. $\mathcal{V}$ upon input $(\mathsf{LINK}, \mathsf{sid}, \sigma, \mathsf{m}, \sigma', \mathsf{m}', \mathsf{bsn})$ with $\mathsf{bsn} \neq \perp$:
   - Sets $f \leftarrow \mathsf{Link}(\sigma, \mathsf{m}, \sigma', \mathsf{m}', \mathsf{bsn})$
   - Outputs $(\mathsf{LINK}, \mathsf{sid}, f)$.

Figure 3.4: DAA protocol Π_{DAA}

presented in Figures 3.3 and 3.4 securely realises $\mathcal{F}_{\mathsf{daa}}$ in the $(\mathcal{F}_{auth^*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}^{D})$-hybrid random oracle model if the following hold:

1. The issuer registers its public key using an online-extractable proof of knowledge $\pi_{\mathcal{I}}$ of the underlying secret key, with some certificate authority.

2. The host initializes and terminates the interactive join and signing protocols.

As noted in [19], obtaining composable security comes with the caveat that where knowledge extractors are used to prove security, one is restricted to on-line (straight-line) extractors [30] only. This is because rewinding does not work in the UC setting since the environment will observe the repeat adversarial behaviour induced by rewinding his state. Note that Ext is only well defined with respect to Definition 1 for on-line extraction [30]. Whilst we believe our security definition could be adapted to support extraction by rewinding also, this technique is not supported in the UC framework and so such a scheme could never obtain composable security. Since our primary aim is to design a property-based definition that implies the UC definition, we assume all proofs of knowledge used in the join protocol and by the issuer to register his public key are on-line (straight-line) extractable. Whilst [19] argue that the simulator used in security proofs could equally use rewinding, they note that one could only achieve stand-alone security. Furthermore, one must be careful to limit the number of simultaneous join sessions to be logarithmic in the security parameter to avoid exponential blow-up in the reduction algorithm's runtime when extracting witnesses by rewinding.
In order to prove that the DAA scheme Π_{DAA} referred to in Theorem 1 satisfies the UC DAA definition of [19] we proceed as follows. We must show that no environment $\mathcal{E}$ can distinguish the real world, in which it is working with Π_{DAA} and adversary $\mathcal{A}$ from the ideal world in which it uses the DAA ideal functionality $\mathcal{F}_{\mathsf{daa}}$ as defined in [19] (see Appendix of full paper for details []) with simulator $\mathcal{S}$. We begin with the real world protocol Π_{DAA}. Next we define an entity $\mathcal{C}$, which we call the challenger, who runs this real world protocol for all parties. $\mathcal{C}$ is then split into two parts; a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$. We first have a useless functionality which forwards all inputs from honest parties to the simulator and then relays the output back to those honest parties. Over a series of game hops we modify $\mathcal{S}$ and $\mathcal{F}$ so that $\mathcal{F}$ handles more of the inputs from honest parties on its own. At the end we are left with the ideal functionality $\mathcal{F}_{\mathsf{daa}}$ and a satisfying simulator.
Due to space limitations, we present the full details of the complex security proof including intermediate functionalities and simulators in the Appendix of the full paper [] . Here we give a brief overview of the game hops used and the techniques used to prove indistinguishably between each hop.

Proof Sketch of Theorem 1

Game 1: This is the real world protocol.

Game 2: The challenger $\mathcal{C}$ now receives all inputs and simulates the real world for all honest parties. It also simulates the hybrid functionalities honestly. By construction this is equivalent to the previous game.

Game 3: We now split $\mathcal{C}$ into two parts: a 'dummy' functionality $\mathcal{F}$ and a simulator $\mathcal{S}$. $\mathcal{F}$ simply forwards all inputs from honest parties to $\mathcal{S}$. $\mathcal{S}$ simulates the real world protocol and relays the outputs back to $\mathcal{F}$ who in turn passes them onto $\mathcal{E}$. This is simply a restructuring of the previous game and is thus equivalent to the previous game.

Game 4: In this game we let our intermediate $\mathcal{F}$ handle the Setup related interfaces using the procedure specifed in $\mathcal{F}_{\mathsf{daa}}$. Consequently, $\mathcal{F}$ expects to receive the algorithms (ukgen, sig, ver, link,

identify) from the simulator. For ukgen, ver, link, and identify, $\mathcal{S}$ can simply provide the algorithms from the real-world protocol, where it omits the revocation check from ver. The sig algorithm, though, must contain the issuer's private key, so $\mathcal{S}$ has to be able to get that value.
When $\mathcal{I}$ is honest, $\mathcal{S}$ will receive a message from $\mathcal{F}$ asking for the algorithms, which informs $\mathcal{S}$ what is happening and allows him to start simulating the issuer. Since $\mathcal{S}$ is running the issuer, it knows its secret key and sets the sig algorithm accordingly.
When $\mathcal{I}$ is corrupt, $\mathcal{S}$ starts the simulation when the issuer registers his key with $\mathcal{F}_{\mathsf{ca}}$ that is controlled by the simulator. Since the public key includes an (on-line extractable) proof of knowledge of the issuer's secret key, $\mathcal{S}$ can extract the secret key from there and define the sig algorithm accordingly. By the simulation soundness of the SPK, this game hop is indistinguishable for the adversary.

Game 5: $\mathcal{F}$ now handles the verify and link queries using the provided algorithms ver and link from the previous game, rather than forwarding the queries to $\mathcal{S}$. We do not let $\mathcal{F}$ perform the additional checks (Check (**ix**) - Check (**xvii**)) done by $\mathcal{F}_{\mathsf{daa}}$, though, but add these only later. For Check (**xiii**), $\mathcal{F}$ rejects a signature when a matching $\mathsf{sk}' \in \mathsf{RL}$ is found, but does not exclude honest TPMs from this check yet.
Because verify and link do not involve network traffic, the simulator does not have to simulate traffic either, we must only make sure the outputs do not change. The verification algorithm ver that $\mathcal{F}$ uses is almost equal to the real world protocol. The only difference is that ver used by $\mathcal{F}$ does not contain the revocation check. $\mathcal{F}$ now performs this check separately. Since, by an assumption of Definition 6, key-based revocation is instantiated precisely by running the Identify algorithm on each key in RL this separation will not change the verification outcome.
For linking, $\mathcal{F}$ executes the Link algorithm that $\mathcal{S}$ supplied, which is equivalent to the real world algorithm, so the outcome will clearly be equivalent.

Game 6: In this step we change $\mathcal{F}$ to also handle the join-related interfaces, meaning it will receive the inputs and generate the outputs. We let $\mathcal{F}$ run the same procedure as $\mathcal{F}_{\mathsf{daa}}$, but again omit the additional checks (Check (**ii**)- Check (**iv**)).
We have to ensure that $\mathcal{F}$ outputs the same values as the real world does. As the join interfaces do not output crypto values, but only messages like 'start' and 'complete', we simply have to guarantee that whenever the real world protocol would reach a certain output, the functionality also allows that output, and vice versa.
For the direction from the real world to the functionality this is clearly given, since $\mathcal{F}$ does not perform additional checks and thus will always proceed for any input it receives from $\mathcal{S}$. For all outputs triggered by $\mathcal{F}$, the simulator has to give explicit approval which allows $\mathcal{S}$ to block any output by $\mathcal{F}$ if the real world protocol would not proceed at a certain point.
If the host and/or TPM are honest, the simulator knows the identities $\mathcal{M}_i$, $\mathcal{H}_j$ and correctly uses them towards $\mathcal{F}$ as well as in the simulation. If both, the TPM and host are corrupt but the issuer is honest, then $\mathcal{S}$ cannot determine the identity of the host, as the host does not authenticate towards the issuer in the real world. This does not impact the real world simulation of the issuer, but the simulator has to choose an arbitrary corrupt host $\mathcal{H}_j$ now when invoking $\mathcal{F}$ with the JOIN call. This will only result in a different host being stored in the ML list in $\mathcal{F}$, but $\mathcal{F}$ never uses this identity when the corresponding TPM is corrupt.
In the final join interface JOINCOMPLETE, the simulator has to provide signing key sk of the TPM. When the TPM is honest, $\mathcal{S}$ knows the key anyway and if the TPM is corrupt, $\mathcal{S}$ extracts the signing key from the join transcript using Ext (see Definition 1). Note that $\mathcal{F}$ sets $\mathsf{sk} \leftarrow \perp$ when both the TPM and host are honest. However, this has no impact yet, as the signatures are still created by the simulator and the verify and link interfaces of $\mathcal{F}$ do not run the additional checks that make use of the internally stored records and keys. Overall, this game hop is indistinguishable by the

simulation soundness of join transcripts.

Games 7, 8, 9, 10: Over the next four game hops, we gradually modify $\mathcal{F}$ so that it eventually handles all signing queries instead of merely forwarding them to $\mathcal{S}$. As before, $\mathcal{F}$ will use the (sign) interfaces from $\mathcal{F}_{\mathsf{daa}}$, with the difference that it does not perform the Checks (**v**)-(**viii**) which we add in a later game.

When the TPM or the host is corrupt, $\mathcal{S}$ has to provide the signature value, which it takes from the real world simulation, and thus perfectly mimics the real world output. When both the TPM and the host are honest, $\mathcal{F}$ creates the signatures internally in an unlinkable way; it chooses a new sk per basename and TPM, or per signature when $\mathsf{bsn} = \perp$ and then runs the sig algorithm for that fresh key. $\mathcal{F}$ keeps the internally chosen keys $\langle \mathcal{M}_i, \mathcal{H}_j, \mathsf{bsn}, \mathsf{sk}_i \rangle$ in a list DomaninKeys to ensure consistency if a TPM wishes to reuse the basename

This change is indistinguishable under the assumed anonymity of protocol Π_{DAA}. Suppose an environment can distinguish a signature by an honest party with the sk it joined with from a signature by the same party but with a different sk. Then one can use such an environment to win the 'real-or-random' anonymity game described in the Appendix of the full paper []. To see this, we consider a sequence of sub-games where, in one game, a signature is generated using the true sk used in the joining of that TPM and in the next game that same signature is created by sampling a fresh secret key. To show that these two sub-games are indistinguishable, we embed the challenge oracle $\mathsf{ChalO}^{ror}(\cdot,\cdot,\cdot)$ into the simulator so that the signature $\mathcal{S}$ returns is the one obtained on querying the $\mathsf{ChalO}^{ror}(\cdot,\cdot,\cdot)$ oracle. If the environment thinks it is observing the second sub-game, it must be that we are in the 'random' case, else we are in the 'real case'. Thus we can use the distinguishing power of the environment to win the real-or-random anonymity game. By the equivalence of these two anonymity notions (see Appendix of full paper []), we can then win the 'left-or-right' anonymity game. Since protocol Π_{DAA} is assumed to satisfy the left-or-right notion of anonymity, this change is indistinguishable.

Another important modification made to $\mathcal{F}$ is a check that all honestly generated signatures verify and identify correctly. Once again, we reduce to a property of real-world protocol Π_{DAA}; its correctness.

The last important modification to $\mathcal{F}$ is that for honest platforms, $\mathcal{F}$ now checks that a newly generated signature doesn't identify to the signing key of some other TPM. Since there is a unique value sk which identifies to each valid signature, this matching key corresponds to some honest platform. We then argue that due to the exponential size of the key space, the probability than one samples the same key as some other TPM is negligible.

Game 11: When verifying signatures, $\mathcal{F}$ now rejects any signature identifying to two different signing keys. By assumption that there is a unique TPM signing key to which a valid signature identifies, this new check does not alter the verification outcome.

Game 12: $\mathcal{F}$ now rejects signatures that do not identify to a TPM secret key used in an earlier join session. Importantly this check is only triggered if the issuer is also honest. In order to show that this check will not trigger, we demonstrate how such a triggering signature constitutes a game winning forgery in the unforgeability game defined in Figure 3.2. In particular, our reductionist simulates Game 12 to the environment by making use of $\mathsf{IssueO}(\cdot)$ and the other oracles available to it in the unforgeability game. Since Π_{DAA} is assumed unforgeable, Game 12 is indistinguishable from Game 11.

Game 13: $\mathcal{F}$'s verify interface now rejects signatures that identify to some honest TPM who never signed them. We show that if an environment is able to distinguish Games 12 and 13, it must be able to create a signature triggering this check which constitutes a game-winning output in the unforgeability game. Since Π_{DAA} is assumed unforgeable, Game 13 is indistinguishable from Game 12.

Game 14: $\mathcal{F}$ now introduces a new check that disallows signatures identifying to some honest platform who never signed them, even when the issuer is corrupt. We show that if there is a distinguisher who can distinguish between Games 13 and 14, then one can construct a reductionist who wins the non-framability game described in Figure 2. We consider the cases when this check triggers for $\mathsf{bsn} \neq \perp$ and $\mathsf{bsn} = \perp$ separately, showing that neither situations occur with more than negligible probability if Π_{DAA} is non-frameable and unforgeable as assumed.

Game 15: Previously, $\mathcal{F}$ rejected any signature that identified to some signing key on the revocation list RL. Now such signatures are only rejected if they additionally identify to no honest platforms. To show that this is indistinguishable from the previous game, we must show that signatures rejected by the revocation list check that also identify to some honest platform occur with negligible probability. In other words, the additional constraint of this check affects the outcome of verification with negligible probability. We assume that there is a distinguisher, able to tell the difference between Games 14 and 15. We construct a reductionist $\mathcal{R}$ who uses this distinguisher to win the anonymity experiment defined in Figure 3.2. Once this reductionist observes a signature rejected by this new check, he learns the TPM key of an honest platform. He can then call the challenge oracle with this platform and some other randomly chosen honest platform. If the challenge signature identifies to this newly discovered key then it must have come from that platform. If not, it came from the second platform. Since protocol Π_{DAA} is assumed to be anonymous, Game 13 is indistinguishable from Game 14.

Game 16: F now puts requirements on the link algorithm. We show that these requirements do not change the output of the Link interface. The introduction of this new check would only change the output distribution in the following two cases. Firstly, two signatures that previously linked now do not. This would happen if $\mathsf{Link}(\sigma, \sigma', \mathsf{m}, \mathsf{m}', \mathsf{bsn}) = 1$ but either $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{sk}) = 0$ or $\mathsf{Identify}(\sigma', \mathsf{m}', \mathsf{bsn}, \mathsf{sk}) = 0$. Alternatively two signatures that previously did not link but now do, *i.e.,* $\mathsf{Link}(\sigma, \sigma', \mathsf{m}, \mathsf{m}', \mathsf{bsn}) = 0$ but $\mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{sk}) = \mathsf{Identify}(\sigma', \mathsf{m}', \mathsf{bsn}, \mathsf{sk}) = 1$. By assumption, (see Definition 6) neither of these situations can occur since it is assumed that signatures have the property that they link if and only if there is some value sk such that both signatures identify to sk.

The functionality $\mathcal{F}$ described in Game 15 is exactly $\mathcal{F}_{\mathsf{daa}}$ as defined in [19]. Since we have shown the indistinguishability of each game hop, we have that protocol Π_{DAA} realizes $\mathcal{F}_{\mathsf{daa}}$. □

Till here, we has presented all the technical contents. In summary, we have extracted four common features and further reflected them into the new property-based definition. The new model implies the UC version and meanwhile maintains the same security level.

Chapter 4

Conclusion

In this report, we define ideal functionalities for TPM in the context of create/load TPM key and a new cryptographic security notion–property-based model–for DAA.

For the design of ideal functionalities for TPM, we target to idealize the functionalities which do not provide cryptographic functions. To do so, we combine the use of trusted third party, access control and private channels. Via the security modelling, we now guarantee that the security of TPM can be reduced to the secure implementation of the ideal functionalities. All still points to the need for the TPM to be modelled and analysed as a whole, rather than as the sum of independent parts, in order to properly capture possible interactions with outside surroundings, the prime and first example of which is the create/load TPM key. We further provide discussions on the extension for the current outputs to support the three use cases.

Since DAA is the main cryptographic algorithm of TPM and we have designed and developed some lattice-based DAA algorithms (in the previous deliverable), we here propose a new security model for DAA in the property-based setting. The new model contains four necessary features supporting both stand-alone and concurrent security oracles. The model can be further extended to imply the UC notion, guaranteeing the security level of UC, via the use of wrapper protocol.

The above two contributions are complementary. The first design is for the non-cryptographic functionalities (being used to interact TPM's functionalities with outside surroundings) while the second modelling is designed for DAA which is the main crypto-function of TPM. Both of the modelling are built on top of the extraction of most common and important functionalities and features of TPM. In this way, our modelling can cover and support most of TPM practical use (w.r.t. the use cases).

We may consider to extend the ideal functionalities design to other crucial functionalities of TPM. But this will mainly depend on the practical requirements and deployment of the three use cases (i.e., D6.2).

Chapter 5

List of Abbreviations

AK Attestation Key

API Application Programming Interface

CA Certification Authority

EA Enhanced Authorization

EK Endorsement Key

FIDO Fast IDentity Online

CFG Control Flow Graph

DAA Direct Anonymous Attestation

HMAC Hash-Based Message Authentication Code

HSM Hardware Security Module

KDF Key Derivation Function

NMS Network Management System

PCR Platform Configuration Register

QN Qualified Name

TCG Trusted Computing Group

TEE Trusted Execution Environment

TLS Transport Layer Security

TOCTOU Time-of-Check Time of Use

TPM Trusted Platform Module

TTP Trusted Third Party

UC Universal Composability

U2F Universal 2nd Factor

WP Work Package

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] 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. https://link.springer.com/content/pdf/10.1007%2F978-1-4302-6584-9.pdf.

[3] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. Tamarin prover (v. 1.4.1), January 2019. https://tamarin-prover.github.io/.

[4] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 614–629. Springer, 2003.

[5] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *Eurocrypt*, pages 614–629. Springer, 2003.

[6] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 136–153, 2005.

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 17–32, 2008.

[8] David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel P. Smart, and Bogdan Warinschi. Anonymous attestation with user-controlled linkability. *Int. J. Inf. Sec.*, 12(3):219–249, 2013.

[9] Bruno Blanchet, Vincent Cheval, and Marc Sylvestre. Proverif (v. 2.00), 2018. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[10] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 168–177. ACM, 2004.

[11] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 168–177, 2004.

[12] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *ACM CCS 2004*, pages 168–177. ACM, 2004.

[13] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. Foundations of fully dynamic group signatures. In *ACNS 2016*, volume 9696 of *LNCS*, pages 117–136. Springer, 2016.

[14] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Essam Ghadafi, and Jens Groth. Foundations of fully dynamic group signatures. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 117–136, 2016.

[15] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int. J. Inf. Sec.*, 8(5):315–330, 2009.

[16] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

[17] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 901–920, 2017.

[18] Jan Camenisch, Manu Drijvers, Alec Edgington, Anja Lehmann, Rolf Lindemann, and Rainer Urian. FIDO ECDAA algorithm, implementation draft. https://fidoalliance.org/specs/fido-uaf-v1.1-id-20170202/fido-ecdaa-algorithm-v1.1-id-20170202.html.

[19] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, pages 234–264, 2016.

[20] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted tpms. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 427–461, 2017.

[21] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Uc-secure non-interactive public-key encryption. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 217–233, 2017.

[22] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.

[23] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219, 2004.

[24] Liqun Chen. A DAA scheme requiring less TPM resources. *IACR Cryptology ePrint Archive*, 2010:8, 2010.

[25] Liqun Chen, Dan Page, and Nigel P. Smart. On the design and implementation of an efficient DAA scheme. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, pages 223–237, 2010.

[26] Vincent Cheval, Véronique Cortier, and Mathieu Turuani. Global states verif (gsverif), August 2018. https://sites.google.com/site/globalstatesverif/.

[27] The FutureTPM Consortium. First report on the security of the TPM. Deliverable D3.2, June 2019.

[28] The FutureTPM Consortium. Technical integration points and testing plan. Deliverable D6.1, July 2019.

[29] The FutureTPM Consortium. Threat modelling & risk assessment methodology. Deliverable D4.1, February 2019.

[30] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, pages 152–168, 2005.

[31] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350, 2011.

[32] ISO, International Organization for Standardization. ISO/IEC 11889: Information technology - Trusted platform module library, 2009.

[33] ISO, International Organization for Standardization. ISO/IEC 20008-2: Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key, 2013.

[34] ISO, International Organization for Standardization. ISO/IEC 11889: Information technology - Trusted platform module library, 2015.

[35] James H. Morris Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

[36] Steve Kremer and Robert Kunnemann. Sapic - a stateful applied pi calculus. http://sapic.gforge.inria.fr/.

[37] Vireshwar Kumar, He Li, Noah Luther, Pranav Asokan, Jung-Min "Jerry" Park, Kaigui Bian, Martin B. H. Weiss, and Taieb Znati. Direct anonymous attestation with efficient verifier-local revocation for subscription system. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 567–574, 2018.

[38] Michael Z. Lee, Alan M. Dunn, Jonathan Katz, Brent Waters, and Emmett Witchel. Anonpass: Practical anonymous subscriptions. *IEEE Security & Privacy*, 12(3):20–27, 2014.

[39] Toru Nakanishi and Nobuo Funabiki. Verifier-local revocation group signature schemes with backward unlinkability from bilinear maps. In *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, pages 533–548, 2005.

[40] Jonathan Petit, Florian Schaub, Michael Feiri, and Frank Kargl. Pseudonym schemes in vehicular networks: A survey. *IEEE Communications Surveys and Tutorials*, 17(1):228–255, 2015.

[41] Eike Ritter, Joshua Philipps, Bruno Blanchet, Vincent Cheval, and Marc Sylvestre. Statverif (v. 1.97pl1.2), August 2019. https://sec.cs.bham.ac.uk/research/StatVerif.

[42] 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.

[43] 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.

[44] TCG. TPM 2.0 library specification - part 1: Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf.

[45] TCG. TPM 2.0 library specification - part 3: Commands - code. https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-2-Structures-01.38.pdf.

[46] Trusted Computing Group. TPM main specification version 1.2, 2004.

[47] Trusted Computing Group. Trusted platform module library specification, family “2.0”, 2014.

[48] Jorden Whitefield, Liqun Chen, Thanassis Giannetsos, Steve Schneider, and Helen Treharne. Privacy-enhanced capabilities for vanets using direct anonymous attestation. In *2017 IEEE Vehicular Networking Conference, VNC 2017, Torino, Italy, November 27-29, 2017*, pages 123–130, 2017.

Appendix A

Concurrent Oracle Definitions

It is crucial that oracles simulating a DAA protocol to an adversary allow him the same concurrent access as he would have in real life. For example, an adversary in control of two host machines might begin a join session using one of its honest TPMs whilst using the other to commence a signing session. This example demonstrates that oracles simulating part of an interactive protocol (joining or signing) can be interleaved with other oracles by the adversary. In other words, he should be allowed to make a query that corresponds to the first move of an interactive protocol and receive the corresponding reply. Then, he should be able to query an entirely different oracle with no obligation to complete the protocol he started with the first. Only oracles defined in this way capture, with sufficient granularity, the true nature of a real life adversary's power in the DAA setting. In the main body of this work Figure 3.1 presents simplified, non-concurrent versions of these oracles. We choose this presentation structure so as to provide intuition on the function of these oracles. However, we reiterate the important considerations one must bare in mind when referring to the simplified oracles of Figure 3.1. In particular, one must allow an adversary to interleave calls to interactive oracles so that he is not obliged to finish an interaction with one before beginning one with another. Note that the full definitions in Figure A.1 cover this concurrency explicitly.

Of equal importance is that concurrent adversarial access is the default setting in the UC framework used to define DAA security in [19]. Thus, in order that our property-based definition implies the same level of security, we must define oracles that give the adversary equivalent power. We follow the standard notation for defining such oracles as used in group signatures [4], [13]. The algorithms comprising a protocol are modelled as interactive Turing machines (ITIs) which take as input a protocol message M_{in} and an internal state st and output a message M_{out}, a new internal state st', and a decision $\mathsf{dec} \in \{\mathsf{cont}, \mathsf{accept}, \mathsf{reject}\}$. We cut the protocol into pieces where each 'piece' is defined by the receiving of a message and then the sending of a response message. We say that an ITI (algorithm) run as part of a protocol returns cont if the protocol is still going on, reject if some check failed or an invalid input was received, and accept if everything runs as expected and there is no next step for that ITI in the protocol.
Figure A.1 provides formal definition for these oracles.

InitO–P(i)
- **If** $i \notin$ HP, then **Return** $\perp$.
- CRE$[i] := \perp$, $\mathsf{dec}^i_{\mathsf{JoinTPM}} :=$ cont, $\mathsf{dec}^i_{\mathsf{Issue}} :=$ cont.
- $\mathsf{st}^i_{\mathsf{JoinTPM}} := (\mathsf{ESK}[i])$
- $\mathsf{st}^i_{\mathsf{JoinH}} := (\mathsf{EPK}[i], \mathsf{ipk})$
- $\mathsf{st}^i_{\mathsf{Issue}} := (\mathsf{isk}, \mathsf{EPK}[i], \mathsf{ML})$
- $(\mathsf{st}^i_{\mathsf{JoinH}}, \mathsf{M}_{\mathsf{Issue}}, \mathsf{dec}^i_{\mathsf{JoinH}}) \leftarrow \mathsf{JoinH}(\mathsf{st}^i_{\mathsf{JoinH}}; \perp)$.
- **While** $(\mathsf{dec}^i_{\mathsf{JoinTPM}} \wedge \mathsf{dec}^i_{\mathsf{JoinH}} \wedge \mathsf{dec}^i_{\mathsf{Issue}} =$ cont$)$ **do**
  - $(\mathsf{st}^i_{\mathsf{Issue}}, \mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{Issue}}) \leftarrow \mathsf{Issue}(\mathsf{st}^i_{\mathsf{Issue}}; \mathsf{M}_{\mathsf{Issue}})$.
  - $(\mathsf{st}^i_{\mathsf{JoinH}}, \mathsf{M}_{\mathsf{JoinTPM}}, \mathsf{dec}^i_{\mathsf{JoinH}}) \leftarrow \mathsf{JoinH}(\mathsf{st}^i_{\mathsf{JoinH}}; \mathsf{M}_{\mathsf{JoinH}})$.
  - $(\mathsf{st}^i_{\mathsf{JoinTPM}}, \mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{JoinTPM}}) \leftarrow \mathsf{JoinTPM}(\mathsf{st}^i_{\mathsf{JoinTPM}}; M_{\mathsf{JoinTPM}})$.
  - $(\mathsf{st}^i_{\mathsf{JoinH}}, \mathsf{M}_{\mathsf{Issue}}, \mathsf{dec}^i_{\mathsf{JoinH}}) \leftarrow \mathsf{JoinH}(\mathsf{st}^i_{\mathsf{JoinH}}; \mathsf{M}_{\mathsf{JoinH}})$.
- **If** $(\mathsf{dec}^i_{\mathsf{JoinTPM}} \vee \mathsf{dec}^i_{\mathsf{JoinH}} =$ accept$)$:
  - **If** $\mathsf{dec}^i_{\mathsf{JoinH}} =$ accept, then CRE$[i] := \mathsf{st}^i_{\mathsf{JoinH}}$.
  - **If** $\mathsf{dec}^i_{\mathsf{JoinTPM}} =$ accept, then HSK$[i] := \mathsf{st}^i_{\mathsf{JoinTPM}}$.
- **Return** 1.

JoinO(i, M_{in}; party)
– **If** party $= \mathcal{P}$:
- **If** $i \notin$ HP or CRE$[i] \neq \perp$, then **Return** $\perp$.
- **If** not defined, set $\mathsf{dec}^i_{\mathsf{JoinH}} = \mathsf{dec}^i_{\mathsf{JoinTPM}} =$ cont.
- **If** $\mathsf{dec}^i_{\mathsf{JoinH}} \neq$ cont, then **Return** $\perp$.
- **If** $\mathsf{st}^i_{\mathsf{JoinH}}$ is undefined
  - $\mathsf{st}^i_{\mathsf{JoinH}} := (\mathsf{EPK}[i], \mathsf{ipk})$.
- **If** $\mathsf{st}^i_{\mathsf{JoinTPM}}$ is undefined:
  - $\mathsf{st}^i_{\mathsf{JoinTPM}} := (\mathsf{ESK}[i])$.
- $(\mathsf{st}^i_{\mathsf{JoinH}}, \mathsf{M}_{\mathsf{JoinTPM}}, \mathsf{dec}^i_{\mathsf{JoinH}}) \leftarrow \mathsf{JoinH}(\mathsf{st}^i_{\mathsf{JoinH}}; \mathsf{M}_{in})$.
- **While** $(\mathsf{dec}^i_{\mathsf{JoinTPM}} \wedge \mathsf{dec}^i_{\mathsf{JoinH}} =$ cont$)$ **do**
  - $(\mathsf{st}^i_{\mathsf{JoinTPM}}, \mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{JoinTPM}}) \leftarrow \mathsf{JoinTPM}(\mathsf{st}^i_{\mathsf{JoinTPM}}; \mathsf{M}_{\mathsf{JoinTPM}})$
  - $(\mathsf{st}^i_{\mathsf{JoinH}}, \mathsf{M}_{\mathsf{Issue}}, \mathsf{dec}^i_{\mathsf{JoinH}}) \leftarrow \mathsf{JoinH}(\mathsf{st}^i_{\mathsf{JoinH}}; \mathsf{M}_{\mathsf{JoinH}})$.
- **If** $(\mathsf{dec}^i_{\mathsf{JoinTPM}} \vee \mathsf{dec}^i_{\mathsf{JoinH}} =$ accept$)$:
  - **If** $\mathsf{dec}^i_{\mathsf{JoinTPM}} =$ accept, then HSK$[i] := \mathsf{st}^i_{\mathsf{JoinTPM}}$.
  - **If** $\mathsf{dec}^i_{\mathsf{JoinH}} =$ accept, then CRE$[i] := \mathsf{st}^i_{\mathsf{JoinH}}$.
- **Return** $(\mathsf{M}_{\mathsf{Issue}}, \mathsf{dec}^i_{\mathsf{JoinH}})$.

– **If** party $= \mathcal{M}$:
- **If** $i \notin$ HT, then **Return** $\perp$.
- **If** $\mathsf{dec}^i_{\mathsf{JoinTPM}}$ is undefined, set $\mathsf{dec}^i_{\mathsf{JoinTPM}} :=$ cont.
- **If** $\mathsf{dec}^i_{\mathsf{JoinTPM}} \neq$ cont, then **Return** $\perp$.
- **If** $\mathsf{st}^i_{\mathsf{JoinTPM}}$ is undefined:
  - $\mathsf{st}^i_{\mathsf{JoinTPM}} := (\mathsf{ESK}[i])$.
- $(\mathsf{st}^i_{\mathsf{JoinTPM}}, \mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{JoinTPM}}) \leftarrow \mathsf{JoinTPM}(\mathsf{st}^i_{\mathsf{JoinTPM}}; \mathsf{M}_{in})$.
- **If** $\mathsf{dec}^i_{\mathsf{JoinTPM}} =$ accept, then
  - HSK$[i] := \mathsf{st}^i_{\mathsf{JoinTPM}}$.
- **Return** $(\mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{JoinTPM}})$.

IssueO(i, M_{in})
- **If** $i \notin$ HT $\cup$ CP, then **Return** $\perp$.
- **If** $\mathsf{dec}^i_{\mathsf{Issue}}$ is undefined, set $\mathsf{dec}^i_{\mathsf{Issue}} =$ cont.
- If $\mathsf{dec}^i_{\mathsf{Issue}} \neq$ cont, then **Return** $\perp$.
- **If** $\mathsf{st}^i_{\mathsf{Issue}}$ is undefined
  - $\mathsf{st}^i_{\mathsf{Issue}} := (\mathsf{isk}, \mathsf{EPK}[i], \mathsf{ML})$.
- $(\mathsf{st}^i_{\mathsf{Issue}}, \mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{Issue}}) \leftarrow \mathsf{Issue}(\mathsf{st}^i_{\mathsf{Issue}}; \mathsf{M}_{in})$.
- **If** $i \in$ CP, then TS$[i] \leftarrow$ TS$[i] \cup \{(\mathsf{M}_{in}, \mathsf{M}_{\mathsf{JoinH}})\}$.
- **If** $\mathsf{dec}^i_{\mathsf{Issue}} =$ accept, then
  - CRE$[i] := \mathsf{M}_{\mathsf{JoinH}}$.
- **Return** $(\mathsf{M}_{\mathsf{JoinH}}, \mathsf{dec}^i_{\mathsf{Issue}})$.

IdentifyO($i, \sigma, \mathsf{m}, \mathsf{bsn}$)
- If $i \notin$ HT $\cup$ HP, then **Return** $\perp$.
- $b \leftarrow \mathsf{Identify}(\sigma, \mathsf{m}, \mathsf{bsn}, \mathsf{HSK}[i])$.
- **Return** b.

AddHonO(i; party)
- **If** $i \in$ HT $\cup$ HP $\cup$ CT $\cup$ CP, then **Return** $\perp$.
- **If** party $= \mathcal{M}$, then HT $\leftarrow$ HT $\cup \{i\}$.
- **If** party $= \mathcal{P}$, then HP $\leftarrow$ HP $\cup \{i\}$.
- $(\mathsf{esk}, \mathsf{epk}) \leftarrow \mathsf{TPM.Kg}(\mathsf{param})$.
- $(\mathsf{ESK}[i], \mathsf{EPK}[i]) := (\mathsf{esk}, \mathsf{epk})$.
- **Return** EPK$[i]$.

AddCrptO(i, epk; party)
- **If** $i \in$ HT $\cup$ HP $\cup$ CT $\cup$ CP, then **Return** $\perp$.
- **If** party $= \mathcal{M}$ Then CT $\leftarrow$ CT $\cup \{i\}$.
- **If** party $= \mathcal{P}$ Then CP $\leftarrow$ CP $\cup \{i\}$.
- $(\mathsf{ESK}[i], \mathsf{EPK}[i]) := (\perp, \mathsf{epk})$.
- **Return** accept.

SignO($i, \mathsf{m}, \mathsf{bsn}, \mathsf{M}_{in}$; party)
– **If** party $= \mathcal{P}$:
- **If** $i \notin$ HP or CRE$[i] = \perp$ or HSK$[i] = \perp$, then **Return** $\perp$.
- $\mathsf{st}^i_{\mathsf{SignH}} := (\mathsf{CRE}[i], \mathsf{m}, \mathsf{bsn})$.
- $\mathsf{st}^i_{\mathsf{SignTPM}} := (\mathsf{HSK}[i], \mathsf{m}, \mathsf{bsn})$.
- $\mathsf{dec}^i_{\mathsf{SignH}} :=$ cont.
- $\mathsf{dec}^i_{\mathsf{SignTPM}} :=$ cont.
- $\mathsf{M}_{\mathsf{SignH}} := \perp$.
- **While** $(\mathsf{dec}^i_{\mathsf{SignH}} =$ cont$)$ **do**
  - $(\mathsf{st}^i_{\mathsf{SignH}}, \mathsf{M}_{\mathsf{SignTPM}}, \mathsf{dec}^i_{\mathsf{SignH}}) \leftarrow \mathsf{SignH}(\mathsf{st}^i_{\mathsf{SignH}}; \mathsf{M}_{\mathsf{SignH}})$.
  - $(\mathsf{st}^i_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^i_{\mathsf{SignTPM}}) \leftarrow \mathsf{SignTPM}(\mathsf{st}^i_{\mathsf{SignTPM}}; \mathsf{M}_{\mathsf{SignTPM}})$.
- **If** $\mathsf{dec}^i_{\mathsf{SignH}} =$ accept
  - $\sigma \leftarrow \mathsf{st}^i_{\mathsf{SignH}}$.
  - SL $\leftarrow$ SL $\cup \{i, \mathsf{m}, \mathsf{bsn}, \sigma\}$.
- **Return** σ.

– **If** party $= \mathcal{M}$:
- **If** $i \notin$ HT or HSK$[i] = \perp$, then **Return** $\perp$.
- **If** undefined, $\mathsf{dec}^i_{\mathsf{SignTPM}} :=$ cont.
- **If** $\mathsf{dec}^i_{\mathsf{SignTPM}} \neq$ cont, then **Return** $\perp$.
- **If** $\mathsf{st}^i_{\mathsf{SignTPM}}$ is undefined
  - $\mathsf{st}^i_{\mathsf{SignTPM}} := (\mathsf{HSK}[i], \mathsf{m}, \mathsf{bsn})$.
- $(\mathsf{st}^i_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^i_{\mathsf{SignTPM}}) \leftarrow \mathsf{SignTPM}(\mathsf{st}^i_{\mathsf{SignTPM}}; \mathsf{M}_{in})$.
- **If** $\mathsf{dec}^i_{\mathsf{SignTPM}} =$ accept
  - SL $\leftarrow$ SL $\cup \{i, \mathsf{m}, \mathsf{bsn}, \perp\}$.
- **Return** $(\mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^i_{\mathsf{SignTPM}})$.

ChalO($i_0, i_1, \mathsf{bsn}, \mathsf{m}; b$)
- **If** $i'_b \notin$ HP or CRE$[i'_b] = \perp$ or HSK$[i'_b] = \perp$ for $b' \in \{0, 1\}$ then **Return** $\perp$.
- **If** $|\mathsf{CL}| \geq 1$ and $(i_0, i_1, *) \notin$ CL, then **Return** $\perp$.
- $\mathsf{st}^{i_b}_{\mathsf{SignH}} := (\mathsf{CRE}[i_b], \mathsf{m}, \mathsf{bsn})$.
- $\mathsf{st}^{i_b}_{\mathsf{SignTPM}} := (\mathsf{HSK}[i_b], \mathsf{m}, \mathsf{bsn})$.
- $\mathsf{dec}^{i_b}_{\mathsf{SignTPM}}, \mathsf{dec}^{i_b}_{\mathsf{SignH}} :=$ cont.
- $\mathsf{M}_{\mathsf{SignH}} := (\mathsf{CRE}[i_b], \mathsf{m}, \mathsf{bsn})$.
- **While** $(\mathsf{dec}^{i_b}_{\mathsf{SignTPM}} \wedge \mathsf{dec}^{i_b}_{\mathsf{SignTPM}} =$ cont$)$ **do**
  - $(\mathsf{st}^{i_b}_{\mathsf{SignH}}, \mathsf{M}_{\mathsf{SignTPM}}, \mathsf{dec}^{i_b}_{\mathsf{SignH}}) \leftarrow \mathsf{SignH}(\mathsf{st}^{i_b}_{\mathsf{SignH}}; \mathsf{M}_{\mathsf{SignH}})$.
  - $(\mathsf{st}^{i_b}_{\mathsf{SignTPM}}, \mathsf{M}_{\mathsf{SignH}}, \mathsf{dec}^{i_b}_{\mathsf{SignTPM}}) \leftarrow \mathsf{SignTPM}(\mathsf{st}^{i_b}_{\mathsf{SignTPM}}; \mathsf{M}_{\mathsf{SignTPM}})$.
- **If** $\mathsf{dec}^{i_b}_{\mathsf{SignH}} =$ accept
  - $\sigma \leftarrow \mathsf{st}^{i_b}_{\mathsf{SignH}}$.
- CL $\leftarrow$ CL $\cup \{(i_0, i_1, \mathsf{bsn})\}$.
- **Return** σ.

Figure A.1: Details of the oracles used in the security games.