A Cryptographic View of Deep-Attestation, or how to do Provably-Secure Layer-Linking*

Ghada Arfaoui[1], Pierre-Alain Fouque[2], Thibaut Jacques[2], Pascal Lafourcade[3], Adina Nedelcu[1,2], Cristina Onete[4], and Léo Robert[3]

[1] Orange Labs
[2] IRISA, University of Rennes 1
[3] Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, LIMOS
[4] XLIM, University of Limoges

Abstract. Deep attestation is a particular case of remote attestation, *i.e.*, verifying the integrity of a platform with a remote verification server. We focus on the remote attestation of hypervisors and their hosted virtual machines (VM), for which two solutions are currently supported by ETSI. The first is single-channel attestation, requiring for each VM an attestation of that VM and the underlying hypervisor through the physical TPM. The second, multi-channel attestation, allows to attest VMs via virtual TPMs and separately from the hypervisor – this is faster and requires less overall attestations, but the server cannot verify the *link* between VM and hypervisor attestations, which comes for free for single-channel attestation.
We design a new approach to provide linked remote attestation which achieves the best of both worlds: we benefit from the efficiency of multi-channel attestation while simultaneously allowing attestations to be linked. Moreover, we formalize a security model for deep attestation and *prove* the security of our approach. Our contribution is agnostic of the precise underlying secure component (which could be instantiated as a TPM or something equivalent) and can be of independent interest. Finally, we implement our proposal using TPM 2.0 and vTPM (KVM/QEMU), and show that it is practical and efficient.

Keywords: deep attestation, layer linking, TPM, vTPM

1 Introduction

Network Function Virtualization (NFV) is a technology that promises to provide better versatility and efficiency in large-scale networks.

* This study was partially supported by the French ANR project ANR-18-CE39-0019 (MobiS5), the French government research program "Investissements d´Avenir" through the IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25), the IMobS3 Laboratory of Excellence (ANR-10-LABX-16-01), the French ANR project DECRYPT (ANR-18-CE39-0007) and SEVERITAS (ANR-20-CE39-0009).

The core idea is to move from architectures in which physical machines are set up to perform various roles in a network, to a design in which that configuration is done virtually. As such, a machine could be configured and re-configured at distance, and, by judicious use of virtual machines, it could perform a variety of roles within the network infrastructure.

Virtualized platforms are set up in layers, including the following basic components: physical resources, the virtualization layer and infrastructures, *virtualized network functions* (VNFs), and the NFV management and orchestration module.

At the bottom of the infrastructure are real, physical components, meant for computations, storage and physical network functions. The virtualization layer (also called *hypervisor*) manages the mapping between those physical components and virtual equivalents. As such, the virtualized network functions – hosted by virtual machines running inside the NFV infrastructure– never have direct access to the physical resources. Instead, the VNFs access the virtual resources. The NFV management and orchestration module runs the combined infrastructure, including: the lifecycle of the instantiated VNFs, resource allocation for VNFs, or overall management in view of particular, given network services.

Deep Attestation (DA). Virtualization enables efficient, versatile remote network configuration and administration; however, the fact that multiple virtual processes share resources can introduce hazards to security. One way to ensure that a component runs correctly is by using *attestation*. Attestation is a process complementary to authentication: whereas the latter allows a platform to prove that it is the entity it claims to be, the former ensures that the platform runs a trustworthy code, *i.e.*, it has not been breached. As described in [11], "*Attestation is the process through which a remote challenger can retrieve verifiable information regarding a platform's integrity state.*" A property can be for instance software integrity, geolocalisation, access control, etc.

Attestation relies on a *root of trust* (RoT), usually instantiated through a *trusted platform module* (TPM) – or an equivalent mechanism. The root of trust is responsible, amongst other things, for protecting sensitive cryptographic materials (such as private keys) and for running cryptographic operations in an isolated way. The virtualization layer (hypervisor) has direct access to the RoT, but the virtual machines it manages do not; instead they will have access to the RoT by means of *virtual Roots of Trust* (vRoTs). Virtual Roots of Trust are a combination of resources, some provided by the physical RoT, and other managed by the hypervisor, which directs and mediates access to the RoT.

In a nutshell, attestation is a process which allows an independent, remote verifier to check that a target platform still behaves in the desired way. This is done by first authenticating the RoT, then by comparing a measurement of the current state of the component to a presumably-correct state, as indicated in a *Root of Trust for Storage* (RTS).

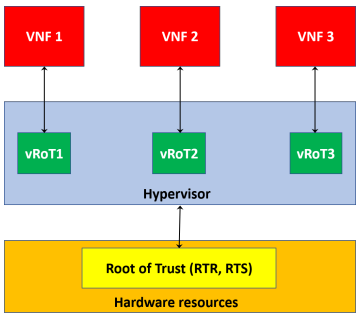

Fig. 1: The setup for DA.

In addition, a guarantee must be given of the correctness of the RTS, which is done by means of a *Root of Trust for Reporting* (RTR). The functionalities of RTS and RTR can effectively be provided by a TPM. A TPM is an example of implementation that could provide RTR and RTS by leveraging the specific tampering detection properties of its Platform Configuration Registers (PCR) and issuing signed reports, or *quotes*, of their content.

We consider the attestation of two types of components: virtual machines (VMs), such as VNFs, and the hypervisor managing them, whose underlying physical component includes a RoT providing an RTR and an RTS. This architecture is depicted in Figure 1.

To verify that the VMs and the hypervisor are running correctly, both these types of components must undergo remote attestation. Not only must they be attested in isolation, but a statement must be attested on the *layer-binding*: notably, we should know which VMs run on which hypervisor. This is known as *deep attestation* (DA). There are two typical ways of achieving deep attestation (as described by ETSI standardization documents [11]): single- and multi-channel VM-Based Deep Attestation.

Single/Multi-channel Deep Attestation. In single-channel deep attestation the attestation is run only between the remote verifier and the virtual machines. At each attestation, the VM (by querying its associated virtual TPM, or vTPM) provides not only an attestation for itself, but also the hypervisor it runs on.

Specifically the response forwarded by the VM to the remote verifier includes the (independent) attestation of the hypervisor, and the layer-binding attestation between the VM and its hypervisor. This is depicted in Figure 2, on the left-hand-side. Note that the quotes in this case are both obtained from the (slow) physical TPM.

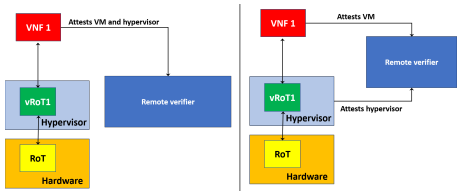

Fig. 2: Single vs multi-channel deep attestation

From the point of view of security, this solution is optimal; however, it scales poorly. Given as few as 1000 VMs running on top of the hypervisor, we would require that the hypervisor be attested 1000 times, once for each VM.

By contrast, in multi-channel deep attestation, the VMs are attested separately and independently from the hypervisor. In this scenario, the VMs attest to the remote verifier, thus proving they were not tampered with. Separately, the hypervisor also attests to the remote verifier. This can be seen on the right hand side of Figure 2. In this case, the efficiency is optimal: for 1000 VMs, we have 1000 VM-attestations and 1 hypervisor attestation. However, there is virtually no layer-binding between the VMs and their hypervisor: there is no guarantee that the VMs are really managed by the hypervisor. An attacker could therefore "convince" a party (such as the owner of the infrastructure) that a VM still exists on a given physical machine when it has, in fact, been removed.

Our solution. We take the middle path between single- and multi-channel deep attestation to obtain layer-binding between VMs and hypervisors with reasonable efficiency. Our solution is simple, yet elegant, using standard cryptography to ensure that a hypervisor's single attestation is linkable to any number of attestations of VMs managed by it. We give three contributions:

A cryptographic scheme. Our scheme ensures secure and efficient linked DA. The hypervisor and VMs each attest only once. However, we also embed a list of public keys (associated with the VMs managed by the hypervisor) *within* the hypervisor attestation, which is established by the root of trust. In order to authenticate the list of forwarded keys, we embed them into the attestation nonce, forwarded by the attestation server. If the hypervisor's attestation verifies, then the attestation server can link that hypervisor with the (subsequently attesting VMs) which use keys in the forwarded list. If the hypervisor's attestation fails, then the public keys cannot be trusted.

Provably secure authorized linked attestation. An important advantage of our approach is that we have a fully-formalized provable-security guarantee. We use a composition-based approach, constructing primitives that are increasingly stronger out of weaker ones. At the basis of our construction is a yea-or-nay *basic attestation scheme*, which we construct so that it is "secure" by assumption. Its functionality is simple: whenever a component is compromised, the basic attestation scheme tells us so by outputting a faulty attestation; whenever the component is honest, the basic attestation outputs a correct attestation. In other words, this basic attestation scheme is a compromise-oracle: when queried it (indirectly) produces a proof of whether a component has been tampered with or not.

Based on this assumption, we build a sequence of cryptographic mechanisms that add security against stronger adversaries. A first step is to build *authenticated attestation*: a scheme which allows us to authenticate the component that provides the attestation, and additionally ensures that this component's attestations always verify prior to corruption, but fails to verify as soon as a compromise occurs. Then, we consider *linked attestation*: a scheme that introduces the hypervisor-VM relationship described above, and permits not only the verification of individual attestations, but also (publicly) linking attestations.

We construct *authorized linked attestation*: a primitive that permits attestation protocols between a component and authorized attestation server. The latter will also then be able to link certain attestations together. Authorized linked attestation will have three properties: *authorization* (only an authorized server can query an attestation quote); *indistinguishability* (no Person-in-the-Middle adversary can know even a bit of a quote exchanged during a legitimate protocol with probability significantly better than $\frac{1}{2}$); and *linkability* (no one can fool an attestation server into believing that two unlinked components are in fact linked).

Implementation. We used a regular laptop equipped with TPM 2.0 (as a root of trust). We set up an architecture with one hypervisor and multiple VMs. The VMs used full virtual TPM as a virtual root of trust. We made over 100

experiments. This showed that our solution is more efficient that single channel approach and adds only insignificant charge (a hash function computation) compared to traditional multi-channel DA.

Our work is, to our best knowledge, the first that attempts to provide a sound cryptographic treatment of deep attestation. In many ways, this is much harder than designing the scheme that we present, because attestation is a generic term comprising an entire class of algorithms that have different goals. As such, we are only scratching the surface here, and believe that –aside from the real, and practical advantages of our presented construction– our cryptographic treatment, primitives, and proofs, may be of independent interest to this line of research.

Limitations. A first fundamental limitation is the fact that we assume, in our constructions, the existence of a basic attestation primitive that works infallibly like an oracle, telling us if a component is compromised or not. Attestation is a complex process, allowing for components to attest to many properties *e.g.*; "are you based in the US?". We explicitly require the strongest attestation possible, which takes into account the state of an entire system, and then has a way of telling whether it has been compromised or not.

We treat the existence of basic attestation as an assumption because we do not see a way of constructing it with cryptographic tools. The cryptography we put on top adds a lot of new properties: authenticity, confidentiality, authorization, linkability, but *not* the simple fact of distinguishing a compromised component from an honest one. Our result should therefore be interpreted as a need for such a scheme to exist, as in fact required by ETSI [11].

Another limitation of our scheme lies in the way we modeled our linked-attestation component. We consider classes of components which can be linked. At the registration of each piece of hardware, a number of subcomponents of each type is indicated – and (unique) keys are given to those components. As a result, we cannot account for having two hypervisors that manage the same VM on a given infrastructure. A future work could be to consider *multi-hypervisor VM* as introduced in [12].

Related Work. Many attacks have been recently reported on remote attestation mechanism [9] or 5G standards standards [13]. Many tools such as formal methods or cryptography can be used to model and prove the security of such standards. However, this lack of formalization must today be now addressed otherwise we will have more and more attacks. Provable cryptography is a nice solution to solve this problem since it allows to better understand the security model, what is the adversary goal and its means, which oracle can he query. Some cryptographic primitives have already be nicely formalized such as Direct Anonymous Attestation (DAA) which enables remote authentication of a trusted computer (TPM for instance) while preserving the privacy of the platform's user in [8] by Brickell et al. It is a group signature without the feature that a signature can be opened, *i.e.*, the anonymity is not revocable. Such primitive are well described using cryptography as a variant of signature scheme. However, provable cryptography has also been used successfully to formalize security protocols

as authenticated key exchange [6, 10]. This is precisely our goal here to model the different security components independently and to compose them to prove the security of a new security mechanism. Indeed, the attestation server must authenticate the whole platform, *i.e.*, the hypervisor and the NFV running on top. This problem has been addressed by others in the context of secure boot or for instance in [5], where the authors propose an attestation mechanism for swarms of device softwares in IoT and embedded environment. Software attestation is different from remote attestation, as said in [4] since it cannot rely on cryptographic secrets to authenticate the prover device. The first to have take into account deep attestation are Lauer and Kuntze in [14] but their solution miss a security proof and a rigorous analysis.

2 Towards authorized linked attestation

Our core contribution provides layer-binding in deep attestation. Cryptographically, we view this as a new primitive, which we call *authorized linked attestation*, built in steps from increasingly-stronger primitives. We use a basic-attestation primitive as a black box. This primitive is an abstraction of the algorithm by which a single party (like a component of a virtualized platform) generates an attestation of its state, given a fresh, honestly-generated nonce. To ease notation, we assume that all the registers are attested at each attestation, and that the property we are attesting is that the entire component has not been compromised.

Authenticated attestation builds on basic attestation by associating parties with identities. The attestation must now no longer indicate whether the party is compromised: it must also authenticate the component. One step further, the linked-attestation primitive built from authenticated attestation will allow two different components to (a) attest their own states; (b) provide auxiliary material that will make two separate attestations linkable.

We also add a new party into the system: the attestation server that serves as a verifier. We then *compose* the linked-attestation primitive with a unilaterally-authenticated authenticated key-exchange protocol, which will authenticate the attestation server and permit the attestation itself to remain confidential with respect to a Person-in-the-Middle (PitM) adversary.

2.1 Basic attestation

During basic attestation a single honest party is generated. This party can be later compromised. A quote-generation algorithm will output a quote if the party is still honest at that time, or a special symbol if it is not. Finally a (public) verification algorithm will yield 1 (the component is honest) or 0 (otherwise).

Note that a party such as the one we describe could correspond in practice to a combination of two parties: a virtual entity (like a VM or the hypervisor) and an underlying, uncorruptible, secure part (the TPM), which actually generates

the quote. At this stage, we importantly do not associate these entities with keys as authentication will only appear in our next step (Section 2.2).

What we want to capture, formalized by the security of basic attestation, is the minimal assumption that a compromised component will always yield an attestation that will fail the verification. This is why, when basic attestation is run for a compromised component, it will yield the special symbol $\aleph$. We also demand correctness: when a non-$\aleph$ quote is generated, the latter will automatically verify. Our basic attestation component thus becomes the minimal non-cryptographic assumption that we need to make to prove our scheme secure.

Formalization. We consider an environment parametrized by a security parameter λ, in which we have a single party P. This party keeps track of a single *attribute*, namely a *compromise bit* γ originally set to 0. Once this bit is flipped to 1, it can never go back to 0. We define a primitive `BasicAtt` as a tuple of algorithms: (`aBSetup`, `aBAttest`, `aBVerif`):

- $\texttt{aBSetup}(1^\lambda) \rightarrow \mathsf{ppar}$: on input the security 1^λ (in unary), this algorithm outputs some public parameters ppar.
- $\texttt{aBAttest}(\mathsf{ppar}) \rightarrow \mathsf{quote}$: on input the public parameters ppar, if $\mathsf{P}.\gamma = 0$, then this algorithm outputs an attestation quote $\mathsf{quote} \neq \aleph$ for P, and if $\mathsf{P}.\gamma = 1$, then it outputs $\aleph$.
- $\texttt{aBVerif}(\mathsf{ppar}, (\mathsf{quote} \cup \aleph)) \rightarrow 0 \cup 1$: on input public parameters ppar and a value that is either a quote denoted quote or a special symbol $\aleph$, this algorithm outputs a bit. By convention, an output of 0 means the attestation fails, while if the output is 1, the attestation succeeds. We require by construction that for all ppar: $\texttt{aBVerif}(\cdot, \cdot, \aleph) = 0$.

This primitive is also depicted in Figure 3. We assume that if $\mathsf{P}.\gamma = 0$ and $\mathsf{quote} \rightarrow \texttt{aBAttest}(\mathsf{ppar})$, then $\texttt{aBVerif}(\mathsf{ppar}, \mathsf{quote}) = 1$.

Security. The only security we demand from this primitive is that, if a party is compromised, then its attestation will always fail. This will happen by construction (since this is an assumed primitive) and is embedded in the security model. The adversary $\mathcal{A}$ will play a game against a challenger $\mathcal{G}$. Initially, the challenger sets the system up by running `aBSetup` to output ppar. This value is given to $\mathcal{A}$ as well. The unique party is generated, such that its corrupt bit is set to 1 ($\mathsf{P}.\gamma = 0$).

Since $\mathcal{A}$ now has ppar, it can now run the `aBAttest` and `aBVerif` algorithms. In addition, it has access to the $\mathsf{OBAttest}$ oracle: $\mathsf{OBAttest}() \rightarrow (\mathsf{quote} \cup \aleph)$. This oracle calls the $\texttt{aBAttest}()$ algorithm for the (corrupted) party P and returns

Target T		**Appraiser**
Setup phase: $\texttt{aBSetup}(1^\lambda) \rightarrow \mathsf{ppar}$		
$\texttt{aBAttest}(\mathsf{ppar}) \rightarrow \mathsf{quote}$	$\xrightarrow{\mathsf{quote}}$	$\texttt{aBVerif}(\mathsf{ppar}, \mathsf{quote}) \rightarrow 0$ if T compromised ($\mathsf{T}.\gamma = 1$) $\rightarrow 1$ if T uncompromised ($\mathsf{T}.\gamma = 0$)

Fig. 3: Basic attestation description with an honestly-generated target. Notice that there is no authentication involved.

the output to the adversary $\mathcal{A}$. The challenger stores the result in a database DB. The adversary wins if, and only if, there exists a quote in DB (possibly with $\mathsf{quote} = \aleph$) such that $\mathtt{aBVerif}(\mathsf{ppar}, \mathsf{quote}) = 1$. Note that by construction our basic attestation primitive is secure, since once the compromise bit is set, the output is $\aleph$, which always yields $\mathtt{aBVerif}(\mathsf{ppar}, \aleph) = 0$.

Basic attestation in reality. One may wonder at this point what our purpose might be in constructing a security model for a primitive that is by definition correct and secure. We need that security model in our reductions: we will use the attestation primitive to build stronger, linked attestation, and then we will want to make the argument that if an attacker can break the larger primitive, it will also break the smaller primitive. As the smaller primitive is secure by design, this is not possible, and hence, the larger primitive is also secure.

2.2 Authenticated Attestation

Basic attestation acts as a foolproof way of telling whether a device is compromised or not. However, the security it provides is very weak. For one thing, it has no authentication guarantees, so potentially one could use a quote that was honestly generated for an honest component to attest a compromised one. Another problem that is more subtle concerns the way components are compromised. Because the basic quotes described in the previous section have no timestamp, nor specific freshness, we cannot take into account adaptive tampering. In the security notion, the party generating the quote is either honest or compromised from the beginning. Yet, ideally we would like a primitive that ensures that a party can start out as honest (and all the quotes generated at that time verify as correct), and later be compromised (and all the quotes generated after that moment will fail). We can do this by deploying cryptographic solutions (two of which are described after the formal definition of the primitive we want to achieve).

A relevant question is why we did not include these security aspects in the basic attestation primitive considered above. To answer this, recall that we have constructed the basic attestation tool to be secure by design. As such, it is an assumption, rather than a solution. To additionally suppose that this primitive also ensures authentication, for instance would be to go against the principle of using minimal assumptions.

Formalization. The precise formalization of this primitive is in Appendix A. We consider an environment containing up to N parties. The parties keep track of the compromise bit γ used also for basic attestation, and a pair of public and secret keys denoted, for each party P, $\mathsf{P.pk}$ (the public key) and $\mathsf{P.sk}$ (the private key). Intuitively, the security we require for this primitive will be that a valid authenticated quote for a party P and fresh auxiliary information (used as nonce) is hard to forge by an adversary which knows all the the public information, can register and compromise users, and query an attestation oracle that returns a valid quote or $\aleph$. In particular, in a secure scheme,verification should fail if either the authentication or the attestation fails.

Construction. We construct an authenticated attestation scheme out of basic authentication, a large set of nonces $\mathcal{N} := \{0,1\}^\ell$ (with ℓ chosen as a function of the security parameter λ), and an EUF-CMA-secure signature scheme $\mathtt{Sig} = (\mathtt{aSigKGen}, \mathtt{aSigSign}, \mathtt{aSigVerif})$. We thus instantiate $\mathcal{AUX} := \mathcal{N}$, and our `AuthAtt` scheme is as follows:

- $\mathtt{aAuthSetup}(1^\lambda) \rightarrow \mathsf{ppar}$: this algorithm runs $\mathtt{aBSetup}(1^\lambda)$ a number N of times, outputting $\mathsf{ppar}_1, \mathsf{ppar}_2, \ldots, \mathsf{ppar}_N$. Each time ppar_i is created, a party handle P_i is also created (it will be the party associated with the instance of `BasicAtt` run for those parameters). It sets $\mathsf{ppar} := (\mathsf{ppar}_1, \mathsf{ppar}_2, \ldots, \mathsf{ppar}_N, N)$, and outputs this value.
- $\mathtt{aAuthKGen}(\mathsf{P_i}) \rightarrow (\mathsf{P_i.pk}, \mathsf{P_i.sk})$: it keeps a counter (starting from 0), which indicates how many times this algorithm has been run. If at the time this algorithm is queried $\mathsf{counter} < N$, then `aAuthKGen` runs `aSigKGen` as a black box and outputs the resulting $(\mathsf{pk}, \mathsf{sk})$ (public and private) keys. It sets $\mathsf{P_i.pk} := \mathsf{pk}$ and $\mathsf{P_i.sk} := \mathsf{sk}$. Party $\mathsf{P_i}$ is then initialized with these keys.
- $\mathtt{aAuthAttest}(\mathsf{ppar}, \mathsf{P.sk}, R) \rightarrow \mathsf{authQuote} \cup \aleph$: on input the public parameters ppar, a private key $\mathsf{P.sk}$ of a party P (which has already been registered), and a value $R \xleftarrow{\$} \mathcal{N}$, this algorithm first runs $\mathsf{quote} \leftarrow \mathtt{aBAttest}(\mathsf{ppar})$, then the algorithm signs $\sigma \leftarrow \mathtt{aSigSign}(\mathsf{P.sk}, (\mathsf{quote}, R))$, that is, it signs a concatenation of the nonce and the obtained quote. The output of this algorithm is $\mathsf{authQuote} := (\mathsf{quote}, \sigma)$. If the required party or key does not exist, the value $\aleph$ is output by default. If $\mathsf{quote} = \aleph$, then we instantiate $\mathsf{authQuote} = \aleph$.
- $\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{P.pk}, R, (\mathsf{authQuote} \cup \aleph)) \rightarrow 0 \cup 1$: on input public parameters ppar, a public key $\mathsf{P.pk}$ of a party P, an auxiliary value $R \in \mathcal{N}$, this algorithm first checks if the last input is $\aleph$; if so, the algorithm outputs 0 by default. Else, the algorithm parses $\mathsf{authQuote} = (\mathsf{quote}, \sigma)$ (with $\mathsf{quote} \neq \aleph$ by construction), then runs $b \leftarrow \mathtt{aSigVerif}(\mathsf{P.pk}, \mathsf{quote}, \sigma)$ and $d \leftarrow \mathtt{aBVerif}(\mathsf{ppar}, \mathsf{quote})$. The algorithm outputs $b \wedge d$ as its response. Notably, 1 is output if, and only if, the signature and the basic attestation verify concomitantly.

Theorem 21 (Secure Authenticated Attestation) *The* `AuthAtt` *scheme is secure assuming that (1)* `BasicAtt` *scheme is secure (2) the size of* $\mathcal{N}$ *is large and (3) the* `Sig` *signature scheme is EUF-CMA secure.*

The proof is given in Appendix A

2.3 Linked Attestation

Authenticated attestation allows the attestation of one (out of many) components, based on that component's unique secret key. If we define now parties as being either VMs or hypervisors, the notion of authenticated attestation suffices to capture the basic guarantees of multi-channel deep-attestation. However, in this paper our goal is to allow parties to *link* their attestations (a hypervisor's attestation should, *e.g.*, , be linkable to that of a number of VMs also hosted on that platform).

Target T	Appraiser
Setup phase: $\mathtt{aAuthSetup}(1^\lambda) \to \mathsf{ppar}$	
$\mathtt{aAuthKGen} \to (\mathsf{T.pk}, \mathsf{T.sk})$	
$\mathtt{aAuthAttest}(\mathsf{ppar}, \mathsf{T.sk}, \mathsf{aux}) \to (\mathsf{quote}, \sigma) \xrightarrow{\mathsf{authQuote}=(\mathsf{quote},\sigma)}$	$\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{T.pk}, \mathsf{aux}, (\mathsf{quote}, \sigma))$
	$\to 0$ if T compromised ($\mathsf{authQuote} = \aleph$ or σ invalid)
	$\to 1$ if T uncompromised ($\mathsf{authQuote} \neq \aleph$ and σ valid)

Fig. 4: Authenticated attestation built upon basic attestation (Figure 3).

In this section we describe our next primitive: linked attestation. The latter takes place in an environment where several parties are registered in a linked way – this corresponds to a single platform. A first step is platform registration, by which several parties are linked on the same underlying hardware. Each entity later generates a linkable attestation – verifiable on its own, and linkable with other linkable attestations.

Although our application scenario is that of linking VM and hypervisor attestations, we make our framework more generic than that. Instead of just two types of components, we consider linkable sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_L$, which resemble equivalence classes. These sets are defined such that any party in one set (say $\mathsf{P}_{\mathcal{S}_1}$) can produce an attestation that is linked to attestations produced by parties in sets $\mathcal{S}_2, \ldots, \mathcal{S}_L$. We write $\mathsf{P} \diamond \mathsf{Q}$ to say that two parties are linked. The relation is reflexive ($\mathsf{P} \diamond \mathsf{P}$), symmetric (if $\mathsf{P} \diamond \mathsf{Q}$, then $\mathsf{Q} \diamond \mathsf{P}$), and transitive (if $\mathsf{P} \diamond \mathsf{Q}$ and $\mathsf{Q} \diamond \mathsf{R}$, then $\mathsf{P} \diamond \mathsf{R}$).

We formalize a linked-attestation scheme $\mathtt{LinkedAtt}$ as a tuple of algorithms $\mathtt{LinkedAtt} = (\mathtt{aLSetup}, \mathtt{aLReg}, \mathtt{aLAttest}, \mathtt{aLVerif}, \mathtt{aLLink})$, defined for some auxiliary set $\mathcal{AUX}$. The detailed formalization is given in Appendix B.

The setup algorithm outputs public parameters ppar, including the maximal number L of sets considered for linking. One can register platforms including subsets of components of each type: this algorithm generates keys for each party. A linked attestation algorithm produces a linked quote $\mathsf{linkedQuote}$ and an auxiliary linking value lkaux. Finally, the verification algorithm checks the attestation in each individual $\mathsf{linkedQuote}$ and the linking algorithm outputs 1 if several linked attestations seem to belong to the same registered platform, and 0 otherwise. This syntax is also depicted in Figure 5.

The security of linked attestation informally states that an adversary, which has Person-in-the-Middle capabilities and can compromise devices at will, cannot make it appear that two devices are linked when they are not, in fact, so.

A significant limitation on the adversary's capabilities is that compromising a device will not leak its private keys (which are assumed to be held by a TPM). However, the adversary will gain a limited oracle access to those keys upon compromising the device. The limitations to those queries follow rules of access to an actual TPM.

More formally, we define the security of *linked attestation* as a game $\mathtt{LinkSec}_{\lambda,\mathsf{F}}$ parametrized by a security parameter λ and a set of functions F, which we call the *permitted key-access functions*. The adversary wins if it is able to make attes-

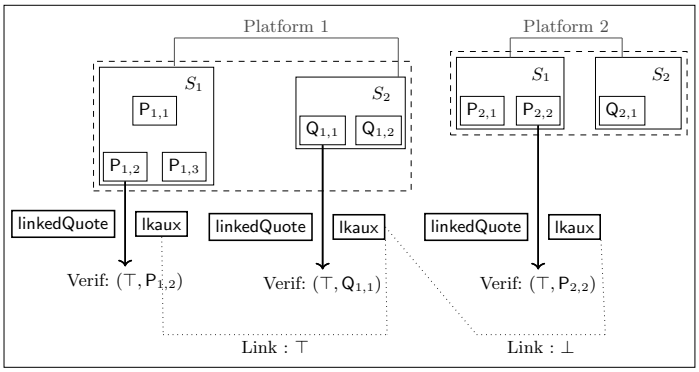

Fig. 5: Linked attestation primitive. The dashed line indicates a platform under the same registration. In this example, both platforms are composed of two subsets (namely S_1 and S_2). There are a total of three quote verifications $(\mathsf{P}_{1,2}, \mathsf{Q}_{1,1}, \mathsf{P}_{2,2})$. The link verification outputs true when the devices are registered under the same platform and false otherwise.

tations stored in $\mathcal{L}_{\mathsf{Att}}$ for parties registered on different platforms (P and Q) link. However, at this point the adversary is constrained to a change-one-change-all kind of game: it cannot, for instance, append an lkaux component of its choice to an honestly-generated linkedQuote, nor vice-versa.

In the security game, the adversary registers platforms and can compromise some of their components. When a component is compromised, the adversary gets oracle access to a set of permitted functions of the component's private key. As a result, the strength of the security proof depends on the function space F. The more functions the adversary is able to query once it compromises a component, the more security our primitive is able to provide. However, note that we cannot give the adversary access to *some* functions, such as the identity function on the component's private key.

Construction. We provide a construction for platforms that have two types of components: virtual machines (VMs) and their managing hypervisor. Thus, in our instantiation, $L = 2$. We use an authenticated attestation scheme (`aAuthSetup`, `aAuthKGen`, `aAuthAttest`, `aAuthVerif`) as a black box. The basic construction is depicted in Figure 6. During setup, our linked-attestation scheme first runs `aAuthSetup` and outputs ppar and $L = 2$. Note that by construction `aAuthSetup` must output a number N, denoting the maximal number of parties that can be set up. This counter will represent a global maximum to parties of all types that will exist in our ecosystem. Following setup, one can register a subset of VMs together with a hypervisor. The algorithm runs the key-generation algorithm `aAuthKGen` of the underlying authenticated attestation scheme for each party,

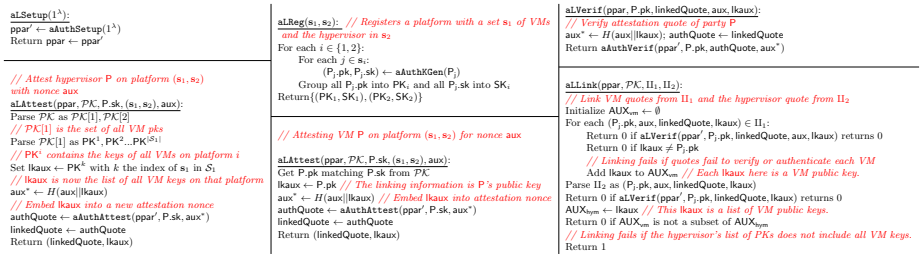

Fig. 6: Our linked attestation scheme for platforms with 2 types of components: VMs (stored in $\mathcal{S}_1$) and hypervisors (stored in $\mathcal{S}_2$). Each type of component attests via a different `aLAttest` algorithm, the main difference between them being that the hypervisor embeds a list of public keys in its nonce.

independently (note that this also ensures that the total number of parties remains at most N). Finally, keys are grouped by types of parties: keys of VMs are output in a set of public keys PK_1 and the key of the hypervisor is output as PK_2.

The VMs and hypervisor generate linked attestations differently. The hypervisor first fetches the public keys of all the components registered with it on the same platform. It computes a new nonce as the hash of two concatenated values: the original auxiliary value aux and the list of the public keys. The component then runs `aAuthAttest` on the public parameters, this new nonce, and its private key, outputting the authenticated quote. By contrast, when a VM attests, it computes a new nonce from the original auxiliary value aux and (only) its own public key. The authenticated quote obtained as a result is provided as the VM's linked quote.

A VM (or a set of VMs) are considered to be linked to a hypervisor if, and only if, the following conditions hold simultaneously: (1) the attestations of all the purportedly-linked parties verify individually (if we run `aAuthVerif` it returns 1 for each individual attestation); (2) the public key that was successfully used to verify each of the VMs' attestation is part of the auxiliary value lkaux forwarded by the hypervisor.

Security. We prove (in AppendixB) the security of our scheme with respect to a single permitted function, $\mathsf{F}_{\mathsf{Sign}}$ that takes in input a message M from a message space $\mathcal{M}$ and outputs, when queried for a compromised party P, a signature on the message M with the private key P.sk. We demand that the message space $\mathcal{M}$ be disjoint from the range of *any* basic attestation scheme.

Theorem 22 (Secure Linked Attestation) *The* `LinkedAtt` *scheme is secure assuming that* `AuthAtt` *scheme is secure and that the hash function H is collision resistant.*

2.4 Authorized Linked Attestation

So far, attestation has been viewed as a primitive, run by a single party (which can be of various types) and outputting an attestation. However, one of the most important requirements of attestation is that the actual quote only be given to authorized parties – which we call *attestation servers* [14].

We will define an *authorized linked attestation* protocol, which allows an attestation server to act as a verification party in the attestation procedures. The same server will also be the one to generate the auxiliary values required for the attestation (this provides freshness to the protocol). The server will also be responsible for linking multiple attestations.

Intuition. We provide a full formalization of authorized linked attestation below. However, we also believe it is useful to first give an intuitive understanding of what this primitive *is* and the security it wants to achieve.

In authorized linked attestation we consider a (single) attestation server S and platforms consisting of several types of components (as shown for linked attestation). The server will keep track of an evolving state, which is initially empty. However, as the server starts to attest various components, at every execution of the authorized attestation protocol, the server will output a verdict (indicating whether the component's individual attestation has failed or succeeded) and may – or may not – update its internal state. Intuitively, the state is meant to contain the *linking information* provided by each of the attesting components. After a number of attestations have taken place, the server might have enough information in its state to decide whether some of the components are linked or not.

The security notion we require for authorized linked attestation is threefold: (1) we require that parties only provide attestation guarantees to the actual attestation server; (2) we require that the contents of the attestation be actually indistinguishable from random for all unauthorized parties; (3) we require a similar kind of linking security as demanded in linked attestation see Section 2.3. However, as opposed to linked attestation, the adversary in this case can also play a Person-in-the-Middle role between honest components and the honest server, or it may attempt to replay messages or impersonate one or both parties. Finally, the adversary will be able to have oracle access to the secret key of any compromised component (this oracle access is parametrized in terms of a function space F of allowed functions).

Formalization. The complete formalization of authorized linked attestation is given in Appendix C. Components on platforms are either VMs or hypervisor. In addition, we consider a S, which stores a tuple consisting of a public and a private key $\mathsf{S.pk} = \mathsf{pk}$ and $\mathsf{S.sk} = \mathsf{sk}$ respectively, and a state $\mathsf{S.st}$. Parties interact with each other in sessions, which are run by an *instance* of the server and an *instance* of a given component. Instances of each party use that party's long-term public and private keys, as well as potential local randomness, such as instance-specific nonces. An instance of a component and an instance of the server are *partnered* if they essentially run the same session (formally, if they

share a session identifier, which consists of the concatenation of a number of session-specific values).

Authorized linking attestation is defined as the tuple $\texttt{ALA} = (\mathsf{ASetup}, \mathsf{AReg}, \mathsf{AAttest}, \texttt{aALink})$. The first, second, and last of these are algorithms, while $\mathsf{AAttest}$ is a protocol. The setup algorithm generates parameters (keys and public system values) for all the involved parties. The registration algorithm allows the VMs and hypervisor on a single platform (defined as sets $\mathbf{s}_1$ for the VMs on the platform and $\mathbf{s}_2$ for the hypervisor) to be associated with each other. For administration purposes, the public keys of all VMs on a platform (*i.e.*, , all VMs in some $\mathbf{s}_1$) and respectively the public key of the platform's hypervisor (the hypervisor in the corresponding $\mathbf{s}_2$) are stored respectively in subsets $\mathsf{PK}_1, \mathsf{PK}_2$ ($i = 1$ for VMs and $i = 2$ for the hypervisor). Together all the subsets PK_i for all the components form a set $\mathcal{PK}[i]$ (for $i = 1, 2$).

The authorized attestation protocol is run by an instance of a component and an instant of the server, yielding, for the component, an acceptance bit (corresponding to the authentication of its partner as the authorized server) and for the server, a tuple $\mathsf{verdict}, \mathsf{S.st}$: the verdict $\mathsf{verdict}$ is 1 or 0 depending on whether the component attested successfully or not, and the state is an update of the server's current internal state. Finally, the server state can be used on a subset of components in the $\texttt{aALink}$ algorithm, yielding either 1 (the components are linked) or 0 otherwise.

Construction. Our construction of the $\texttt{ALA}$ primitive can be seen in the Figure 7. We consider the existence on an underlying $\texttt{LinkedAtt}$ scheme that we use for the $\texttt{aLSetup}$, $\texttt{aLReg}$ and $\texttt{aLLink}$ in a straightforward manner. However, the $\texttt{aLAttest}$ algorithm is no longer a primitive, but a protocol between two instances of two parties, P and Q. For simplicity of exposition, we assume that the instance of Q is the server attesting the component identified by P.

The protocol proceeds as follows. First, P and Q execute the TLS protocol, with P playing the role of the client and Q playing the role of the server. The role of the TLS protocol is two-fold: first, P authenticates the server, so that they can determine whether this party is allowed to obtain attestation data. Second, it leads to the establishment of a secure channel, such that the following messages can be passed on in a secure manner. Once the traffic key(s) established, the protocol continues as follows. First, the server uniformly randomly

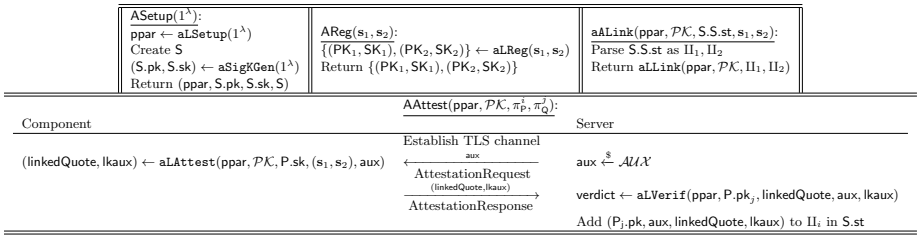

Fig. 7: Our authorized linked attestation scheme for 2 types of components.

samples a nonce aux, which is embedded in the first message of the protocol, AttestationRequest. In response, the party P executes the aLAttest algorithm and the output, consisting of a linkedQuote and the linkage information lkaux, is then sent to the server. The server will subsequently update his state.

In order for two components to be linked by the server successfully, the following conditions have to be met. First, the two components' attestation must be valid (their associated verdicts equals 1). Second, the two lkaux must be subsets of each other; essentially, the key that the VM used as part of its attestation must be found in the lkaux provided by the hypervisor.

We note that if the server has at some point accepted the attestation of a component (thus updating its state to add the linking information), and if later a failed attestation occurs with respect to that component, the server updates state as follows: it ignores the linking information provided in the second attestation; and it removes prior linking information provided by that component.

Security. There are three fundamental properties we want ALA schemes to have: an authenticity guarantee for the attestation server (authorization); a confidentiality guarantee for the contents of the attestation (indistinguishability); and a linkability guarantee for honestly-behaving components (linking-security). The first notion, authorization, captures the fact that before reaching an accepting state, a (non-server) party must be sure that it is speaking to the legitimate server (game $\mathtt{AuthSec}_{\lambda,\mathsf{F}}$). The second notion, indistinguishability, essentially covers Person-in-the-Middle confidentiality for the attestation protocol (game $\mathtt{AuthInd}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$). The last property, linking-security, refers to the fact that no PitM adversary with the ability to compromise components can convince an attestation server that a component is linked to another if that is not the case in reality (game $\mathtt{AuthLink}_{\lambda,\mathsf{F}}$). Although this last property might seem similar to the security notion for our linked attestation primitive, there is one important difference between the two: in linked attestation the adversary has access to essentially two ways to generate an attestation (depending on whether the component is honest or compromised), whereas in *authorized linked attestation* the adversary will have more leeway in combining attestation material across sessions. The stronger adversary in this section will thus make for a stronger primitive in the end. The three security games are defined in Appendix D.

Theorem 23 *Our construction is* $\mathtt{AuthSec}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$ *secure if the TLS protocol provides server authentication:* $\Pr[\mathcal{A}$ *wins* $\mathtt{AuthSec}_{\lambda,\mathsf{F}}] \leq \varepsilon_{TLS\text{-}auth}$.

Proof. (sketch) Note that in order to win this game, the adversary must make a party accept a session with the server, such that no matching server instance exists. This is against the server authentication property we assume of the TLS protocol.

Theorem 24 *Our construction is* $\mathtt{AuthLink}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$ *secure if the underlying primitive* LinkedAtt *is* $\mathtt{LinkSec}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$ *and TLS is at least (s)ACCE secure.*

Proof. (sketch) A key observation for this game is that the adversary cannot impersonate a server or determine it to provide bad randomness. Instead, the

adversary can compromise components and run TLS sessions on their behalf with the server, or try to obtain input from honest components instead. We distinguish between two types of adversary behaviours.

Say $\mathcal{A}$ has never queried $\mathsf{OCompromise}$ for some party P. If the adversary prompts P to run a session, then $\mathcal{A}$ will not actually know anything about the messages (so it cannot misbehave on the quote, the nonce, or anything else). If $\mathcal{A}$ runs the TLS session instead of P, it will learn the channel key, but will not be able to prompt P for the quote (since P wants to run TLS and not the attestation protocol, and since $\mathcal{A}$ cannot impersonate the server).

Say $\mathcal{A}$ queries $\mathsf{OCompromise}$ for some party P. Then the adversary can run TLS sessions on behalf of that party and query $\mathsf{OUseKey}$ in an attempt to get information on the quotes. However, in that case, the attestation of that component fails, except again if we break linked authentication security.

This essentially means that the adversary has no way to maul honestly-generated input to suit its purposes.

Theorem 25 *Our construction is* $\mathtt{AuthInd}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$ *secure if the TLS channel provides (minimally) (s)ACCE security. Let* $q_{sessions}$ *be the number of sessions.*

$$\Pr[\mathcal{A} \textit{ wins } \mathtt{AuthInd}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}] \leq \frac{1}{q_{sessions}} \varepsilon_{TLS\text{-}sACCE}.$$

Proof. Like in the proof of authorization, the reduction here is immediate. The property of sACCE (which is already provided by TLS 1.2, whereas TLS 1.3 gives even stronger guarantees) implies that messages exchanged across the TLS channel are secure.

3 Implementation

We provide a proof of concept implementation of our authorized linked attestation scheme. The implementation consists of three parts, a client for the hypervisor, a client for the Virtual Machines, and an attestation server written in Python 3. We do not consider the underlying NFV or cloud infrastructure, since our scheme abstracts those environments and can be used in any kind deep-attestation scenario. Therefore, any computer equipped with a TPM 2.0 (which can also be emulated) and which has virtualization capacities suffices for the purposes of our implementation. We provide our code as well as a detailed tutorial on how to install and configure both the infrastructure [3].

The infrastructure. We summarize our testing architecture in Figure 8. Although only two VMs are represented in this diagram, some of our tests will use 3 or 4.

Our *hypervisor* is a laptop running Ubuntu 20.04.1 (kernel version 5.4.0-58) with an Intel i5-10210U CPU, 8GB RAM and a Nuvoton TPM NPCT75X. We used KVM to turn this laptop into a hypervisor.

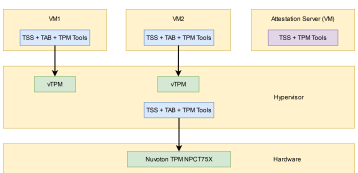

Fig. 8: Architecture for tests.

Table 1: Minimum, median, mean and maximum time in second for attestation of a hypervisor and a virtual machine for 100 trials.

	min	median	mean	max
Hypervisor	3.22	5.30	5.68	11.55
VM	0.66	0.97	1.03	1.41

In order to achieve a high attestation performance, we used a full *virtual TPM implementation*, using QEMU [1] with libtpms [7] version 0.7 and swtpm [15] version 0.5.

All *virtual machines* are QEMU virtual machines (version 4.2.1) with 2 cores and 4G RAM running Ubuntu 20.04.1.

The hypervisor, server, and VMs communicate through a virtual network. A virtual bridge on the hypervisor redirects packets on the appropriate virtual interface. Thus, connection time is not considered in our tests.

To communicate with the TPM we used tpm2-tss, tpm2-abrmd and tpm2-tools from the tpm2-software [2]. Note that the tpm2-tss project implements the TPM software stack (TSS), which is an API specified by the Trusted Computing Group to interact with a TPM. The tpm2-abrmd implements the access broker and resources to manage concurrent access to the TPM and manage memory of the TPM by swapping in and out of the memory as needed (hardware TPM have limited memory). tpm2-tools are a set of command-line tools based on TSS, which are used to send commands to the TPM. We used Python to wrap tpm2-tools commands.

The *attestation server* is also a virtual machine, with the same characteristics as those above. This allows us to test our implementation on a single machine. We establish a secure connection between the client and the server by using Python's SSL library and then sending protocol messages as encoded json strings directly into TLS socket.

Tests. We perform three types of experiments. The first is a comparison of hypervisor attestation time and VM attestation time. Although both those processes have some (very small) amount of noise, our values faithfully show the difference between attesting a component through the physical TPM – hypervisor attestation – and attesting it by using a virtual TPM – VM attestation.

We ran 100 attestations for the hypervisor and 100 attestations for a virtual machine. The results have high variance so Table 1 presents the minimum, the maximum, mean and the median value of those 100 trials. As expected, time for an attestation using a hardware TPM is much higher than using a Virtual TPM.

As our second and third experiments we wanted to see how the overall runtime of our scheme evolves with the number of virtual machines that need to be attested, when the attestation is sequential or parallelized for the VM attestations. In both cases, each experiment run first executed the attestation of the

hypervisor, and then (sequentially or in parallel) the attestations of a varying number of VMs (between 1 and 4). For both experiments we did 100 runs of the experiment. The median runtime was calculated in each experiment (sequential or parallel) for each number of VMs (1 through 4), and the results are plotted in Figure 9

We note that the runtime is not entirely linear. This is because in experiments 2 and 3 the initial attestation of the hypervisor (which only occurs once) takes a much larger time than the subsequent VM runtimes.

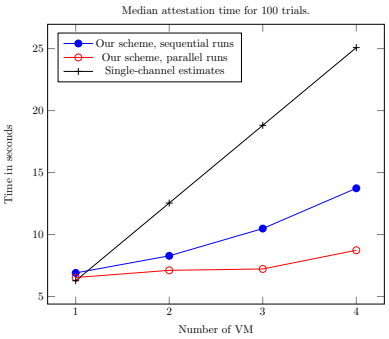

Fig. 9: Attestation time for 1, 2, 3 and 4 VMs.

Comparison to single-channel attestation. We did not implement single-channel attestation. However, since we have implemented hypervisor and VM attestations, we can theoretically estimate the runtime of single-channel attestation for a varying number of VMs – which we plot in Figure 9. Indeed, a single-channel attestation process for a single VM includes a VM attestation *and* a hypervisor attestation. If we want to run it for 2 VMs, then we need to perform 2 hypervisor attestations and 2 VM attestations. This cannot be easily parallelized either, because the same TPM has to run the attestations. This yields a much higher runtime, as depicted in Figure 9.

Comparison to multi-channel attestation.. Although our method follows basic multi-channel attestation approaches, we do add an extra computation (a hash function computation) compared to traditional multi-channel attestation. In addition, we require a little extra memory overhead for both the attestation server and for each platform, so that the additional attestation keys are stored for each VM. There is also a slight transmission overhead, since those keys are also sent upon attestation. However, the transmission overhead is negligible since it only appears for the hypervisor attestation (which occurs only once).

4 Conclusions and Future Work

We proposed a layer-binding in deep-attestation without running into the complexity of single-channel attestation. Our construction achieves the best of both worlds, with a complexity similar to that of multi-channel attestation, but with the strong linkage properties provided in single-channel attestation.

We accompany our construction by a proof-of-concept implementation that clearly shows the viability and scalability of our solution, especially if VM attestations are run in parallel.

In addition, we are the first to present a full, formal treatment of our new protocol, which we call *authorized linked attestation*. Our construction of authorized linked attestation is modular, built on primitives which have increasingly

stronger properties. Our underlying assumption is a primitive called basic attestation. We show that in order to be able to prove security, we need that attestations be able to reflect compromise of the component. In addition, we rely on a collision-resistant hash function, an EUF-CMA-secure signature scheme, and the sACCE security of a TLS protocol (having AKE properties would be even better).

However, our model (and scheme) does not immediately account for other features of virtual infrastructures, such as privacy CAs, migrating VMs, multiple hypervisors managing the same VM, or even replacing TPMs. These aspects are left as future work.

References

1. The Qemu machine emulator. `https://www.qemu.org/`
2. TPM2_tools. `https://github.com/tpm2-software`
3. Anonymized: Implementation. `https://github.com/AnonymousDeepAttestation/deep-attestation`
4. Armknecht, F., Sadeghi, A., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 1–12. ACM (2013)
5. Asokan, N., Brasser, F.F., Ibrahim, A., Sadeghi, A., Schunter, M., Tsudik, G., Wachsmann, C.: SEDA: scalable embedded device attestation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 964–975. ACM (2015)
6. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 773, pp. 232–249. Springer (1993)
7. Berger, S.: Library for TPM tools. `https://github.com/stefanberger/libtpms`
8. Brickell, E.F., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Atluri, V., Pfitzmann, B., McDaniel, P.D. (eds.) Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004. pp. 132–145. ACM (2004)
9. Buhren, R., Werling, C., Seifert, J.: Insecure until proven updated: Analyzing AMD sev's remote attestation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 1087–1099. ACM (2019)
10. Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Knudsen, L.R. (ed.) Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2332, pp. 337–351. Springer (2002)
11. ETSI: Network functions virtualisation(nfv); trust; report on attestation technologies and practices for secure deployments (10 2017)

12. Gopalan, K., Kugve, R., Bagdi, H., Hu, Y., Williams, D., Bila, N.: Multi-hypervisor virtual machines: Enabling an ecosystem of hypervisor-level services. Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC 2017 pp. 235–249 (2019)
13. Hussain, S.R., Echeverria, M., Karim, I., Chowdhury, O., Bertino, E.: 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 669–684. ACM (2019)
14. Lauer, H., Kuntze, N.: Hypervisor-based attestation of virtual environments. In: 2016 IEEE UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld. pp. 333–340 (2016)
15. Safford, D., Berger, S.: Software TPM emulator – swtpm. `https://github.com/stefanberger/swtpm`
16. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 pp. 1–33 (2006)

A Authenticated Attestation

We consider an environment parametrized by a security parameter λ, which will contain up to N parties. Parties keep track of two attributes: the compromise bit γ used also for basic attestation, and a set of public and secret keys $(\mathsf{pk}, \mathsf{sk})$: a public key pk (assumed to be unique per party, and known to all other parties including the adversary) and a private key sk known only to that party. We use $\mathsf{P}.\mathsf{pk}/\mathsf{P}.\mathsf{sk}$ to indicate the public/secret key of party P. We require that this primitive be *correct* in two ways: $\mathtt{aAuthVerif}(\cdot,\cdot,\cdot,\aleph) = 0$ for all possible input values in the first three parameters, and: for all $\mathsf{ppar} \leftarrow \mathtt{aAuthSetup}(1^\lambda)$, $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathtt{aAuthKGen}(\mathsf{P})$, $\mathsf{aux} \in \mathcal{AUX}$, if $\mathsf{authQuote} \rightarrow \mathtt{aAuthAttest}(\mathsf{ppar}, \mathsf{P}.\mathsf{sk}, \mathsf{aux})$ and $\mathsf{authQuote} \neq \aleph$, $\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{P}.\mathsf{pk}, \mathsf{aux}, \mathsf{authQuote}) = 1$.

Security. Formally, $\mathcal{A}$ will play the `AuthSec` game against a challenger $\mathcal{G}$, which begins the game by running `aAuthSetup` and outputting ppar to $\mathcal{A}$. The challenger also initializes N to 1. The adversary then has access to the following oracles:

- $\mathsf{OAuthReg}() \rightarrow (\mathsf{P_i}, \mathsf{P_i}.\mathsf{pk})$: if $i \leq N$, it runs $\mathtt{aAuthKGen}(\mathsf{P_i}) \rightarrow (\mathsf{P_i}.\mathsf{pk}, \mathsf{P_i}.\mathsf{sk})$. It outputs $\mathsf{P_i}.\mathsf{pk}$ to all parties and keeps $\mathsf{P_i}.\mathsf{sk}$ private, stored in the keys attribute of party $\mathsf{P_i}$. A handle (in practice the index) of this party is also returned to $\mathcal{A}$.
- $\mathsf{OAuthAttest}(\mathsf{P_i}, \mathsf{aux}) \rightarrow \mathsf{authQuote} \cup \aleph$: this oracle runs the `aAuthAttest` algorithm on ppar, $\mathsf{P_i}.\mathsf{sk}$ and input aux, and returns the output. On adversarially chosen input $\mathsf{P_i}$ and aux, the oracle updates a list $\mathcal{L}_{\mathsf{Att}} \leftarrow \mathcal{L}_{\mathsf{Att}} \cup (\mathsf{P_i}, \mathsf{aux}, \mathsf{authQuote})$.
- $\mathsf{OCompromise}(\mathsf{P_i}) \rightarrow \mathsf{OK}$: this oracle allows an adversary to compromise party $\mathsf{P_i}$, thus changing $\mathsf{P_i}.\gamma$ to 1.
- $\mathsf{OAuth}(\mathsf{P_i}, M) \rightarrow \sigma_M$: this oracle can only be queried for a party whose compromise bit is 1, and it outputs an EUF-CMA-secure signature keyed with $\mathsf{P_i}.\mathsf{sk}$ on a message M. We require that M be outside the range of any basic

attestation scheme. This last oracle reflects the fact that compromised parties can access a signing function within the TPM.

Finally the adversary outputs a tuple $(\mathsf{P}, \mathsf{aux}, \mathsf{authQuote})$. It is said to *win* if and only if the following condition holds: $\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{P}, \mathsf{aux}, \mathsf{authQuote}) = 1$ and there exist no tuples $(\mathsf{P}, \mathsf{aux}, \mathsf{authQuote})$ such that $(\mathsf{authQuote} \neq \aleph) \rightarrow \mathsf{OAuthAttest}(\mathsf{ppar}, \mathsf{P}, \mathsf{aux})$ for the current public parameters ppar (output by the challenger $\mathcal{G}$).

Proof of Theorem 21.

Proof. Let $\mathcal{A}$ be a probabilistic polynomial-time algorithm. The goal of $\mathcal{A}$ is to provide a signed quote with correct auxiliary value (a nonce) such that the quote and the signature are valid for a fresh nonce.

We propose a proof using a sequence of game hops as introduced in [16]. The initial game corresponds to the security game $\mathtt{AuthSec}$. The successive games are slight modification to its previous one to end up with a game corresponding to generic primitive game (such as EUF-CMA).

Game 0. This is the original security game

Game 1 (transition based on indistinguishability). This game is defined as the previous one except that the challenger aborts the game if a compromised component is able to generate a valid attestation quote. Suppose that $\mathcal{A}$ has a non-negligible advantage ϵ_B of winning the basic attestation game. This means that there exists a party P such that $\mathsf{P}.\gamma = 1$ (*i.e.*, P is compromised) but $\mathsf{quote} \in \mathsf{DB}$ with $\mathtt{aBVerif}(\mathsf{ppar}, \mathsf{quote}) = 1$ (note that $\mathsf{quote} = \aleph$ potentially). By difference lemma we have: $|\Pr[\mathcal{A} \text{ wins } G_0] - \Pr[\mathcal{A} \text{ wins } G_1]| \leq \varepsilon_B$

Game 2. This game is defined as the previous one except that the game aborts if $\mathcal{A}$ can generate a valid signature. We show that: $|\Pr[\mathcal{A} \text{ wins } G_1] - \Pr[\mathcal{A} \text{ wins } G_2]| = \frac{\varepsilon_{\text{EUF-CMA}}}{N}$ where $\varepsilon_{\text{EUF-CMA}}$ is the advantage of EUF-CMA security game. The proof is done by reduction. Assume that $\mathcal{A}$ can generate a valid $\mathsf{authQuote}^*$, *i.e.*, $\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{P}^*, \mathsf{aux}^*, \mathsf{authQuote}^*) = 1$ with $(\mathsf{P}^*, \mathsf{aux}^*, \mathsf{authQuote}^*) \notin \mathcal{L}_{\mathsf{Att}}$. We then show that there exists adversary $\mathcal{B}$ using $\mathcal{A}$ as a sub-routine with non-negligible advantage of winning the EUF-CMA security game.

Adversary $\mathcal{B}$ simulates the game of $\mathcal{A}$ thus acting as the challenger in the $\mathtt{AuthAtt}$ game. The behavior of $\mathcal{B}$ is defined as follows:

- receives pk from its own challenger of the EUF-CMA game.
- runs the $\mathtt{aAuthSetup}$ algorithm to get ppar (and also N).
- randomly selects $i^* \xleftarrow{\$} \{1, \ldots, N\}$. Two cases need to be studied depending on the i^{th}query of $\mathcal{A}$:
  case 1: $i \neq i^*$. When $\mathcal{A}$ calls oracle $\mathsf{OAuthReg}()$ then $\mathcal{B}$ runs $\mathtt{aAuthKGen}(\mathsf{P_i})$ to retrieve $(\mathsf{P_i.pk}, \mathsf{P_i.sk})$. $\mathcal{B}$ sends back $\mathsf{P_i.pk}$ to $\mathcal{A}$. When $\mathcal{A}$ calls oracle $\mathsf{OAuthAttest}()$ then $\mathcal{B}$ runs algorithm $\mathtt{aAuthAttest}$ (which is possible since $\mathcal{B}$ has the corresponding secret key). $\mathcal{B}$ sends back to $\mathcal{A}$ the output of the algorithm. Note that in this case, the simulation is the same as the original game since $\mathcal{B}$ uses the same algorithm of the oracles.

case 2: $i = i^*$. In this case, $\mathcal{B}$ will inject its own material to use it in its EUF-CMA game. When $\mathcal{A}$ calls oracle $\mathsf{OAuthReg}(\mathsf{P}_{i^*})$ then $\mathcal{B}$ simply returns pk. Note that in this case, $\mathcal{B}$ does not have access to sk. Thus when $\mathcal{A}$ calls oracle $\mathsf{OAuthAttest}()$, $\mathcal{B}$ cannot sign the quote. Instead, $\mathcal{B}$ runs $\mathtt{aBAttest}(\mathsf{ppar}_{i^*})$ and sends to its challenger $\mathsf{quote}||\mathsf{aux}$. In response, its challenger will send a signed value of it using sk. $\mathcal{B}$ then forward this to $\mathcal{A}$. The view of $\mathcal{A}$ that is different from the original game in this case is the output of the $\mathsf{OAuthAttest}()$ oracle. The latter runs algorithm $\mathtt{aAuthAttest}$ which generate a quote $\mathsf{quote} \leftarrow \mathtt{aBAttest}(\mathsf{ppar}_i)$ and a signature $\sigma \leftarrow \mathtt{aSigSign}(\mathsf{P_i.sk}, (\mathsf{quote}, \mathsf{aux}))$. In the simulation, $\mathcal{B}$ has access to algorithm $\mathtt{aBAttest}()$ but the signature scheme is different. Yet, both signature schemes are EUF-CMA thus their outputs are indistinguishable (meaning that $\mathcal{A}$ cannot decide from which schemes the output comes from with non-negligible probability) since the keys have the same probability distribution. Hence, the simulation of the game by $\mathcal{B}$ and the real game are indistinguishable.

- When $\mathcal{A}$ returns its forgery on query i, $\mathcal{B}$ parses $\mathsf{authQuote}_i$ as $\mathsf{authQuote}_i := (m^*||\mathsf{aux}^*, \sigma^*)$.
- Finally $\mathcal{B}$ returns $(m^*||\mathsf{aux}^*, \sigma^*)$ to its challenger and wins if $\mathcal{A}$ forges the i^*query (meaning that $i = i^*$).

By combining the results, we have: $\Pr[\mathcal{A} \text{ wins } G_0] \leq \varepsilon_B + \frac{\varepsilon_{\text{EUF-CMA}}}{N}$ which is negligible.

B Linked attestation

We formalize a linked-attestation scheme $\mathtt{LinkedAtt}$ as a tuple of algorithms $\mathtt{LinkedAtt} = (\mathtt{aLSetup}, \mathtt{aLReg}, \mathtt{aLAttest}, \mathtt{aLVerif}, \mathtt{aLLink})$, defined for some auxiliary set $\mathcal{AUX}$.

- $\mathtt{aLSetup}(1^\lambda) \rightarrow \mathsf{ppar}$: on input the security parameter 1^λ (in unary), this algorithm outputs public parameters ppar. This security parameter includes the maximal number of allowed disjoint linkable sets, which we denote as L.
- $\mathtt{aLReg}(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_L) \rightarrow \{(\mathsf{PK}_1, \mathsf{SK}_1), \ldots (\mathsf{PK}_L, \mathsf{SK}_L)\}$: this algorithm keeps as state a number L of sets $\mathcal{S}_i$ originally set to $\emptyset$, and a vector of sets of public keys $\mathcal{PK}$ (also initialized to $\emptyset$). On input a number of subsets $\mathbf{s}_i$ $(i = 1, 2, \ldots, L)$, this algorithm first checks that $\forall i, j, \mathbf{s}_i \cap \mathcal{S}_j = \emptyset$ (else the algorithm outputs $\perp$). If the relation is true, then the algorithm generates for each party $\mathsf{P_j} \in \mathbf{s}_i$, (for all j, i) a tuple of public and private keys $\mathsf{P_j.pk}, \mathsf{P_j.sk}$, initializing $\mathsf{P_j}$ with those keys. We require the unicity of all the generated public keys. The subsets $\mathbf{s}_i$ are each added to greater sets $\mathcal{S}_i$. The algorithm groups the keys of all parties $\mathsf{P_j} \in \mathbf{s}_i$ in a pair of private/public key subsets: $(\mathsf{PK}_i, \mathsf{SK}_i)$, updating the i-th component $\mathcal{PK}[i]$ of $\mathcal{PK}$ as $\mathcal{PK}[i] \cup \mathsf{PK}_i$. All parties are given access to the public-key subsets (and more generally, to $\mathcal{PK}$).
- $\mathtt{aLAttest}(\mathsf{ppar}, \mathcal{PK}, \mathsf{P.sk}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux}) \rightarrow (\mathsf{linkedQuote} \cup \aleph, \mathsf{lkaux})$: on input public parameters ppar, the current set of public keys $\mathcal{PK}$, the private key $\mathsf{P.sk}$ of some party P, subsets $\mathbf{s}_i \in \mathcal{S}_i$, and an auxiliary value $\mathsf{aux} \in \mathcal{AUX}$, this algorithm outputs either a linked quote $\mathsf{linkedQuote}$ or a special failure

symbol $\aleph$, and a different value lkaux (this last entry could be used to store linkage-related information).

- $\mathtt{aLVerif}(\mathsf{ppar}, \mathsf{P.pk}, (\mathsf{linkedQuote} \cup \aleph), \mathsf{aux}) \rightarrow 0 \cup 1$: On input the public parameters ppar, a public key $\mathsf{P.pk}$, a linked quote (or a failure symbol $\aleph$), and an auxiliary value aux, this algorithm outputs a verification bit. By convention, 0 means failure and 1 means acceptance of the attestation.
- $\mathtt{aLLink}(\mathsf{ppar}, \mathcal{PK}, \Pi_1, \ldots, \Pi_L) \rightarrow 0 \cup 1$: on input the public parameters ppar, the set of public keys $\mathcal{PK}$, and subsets Π_i containing elements of the form $(\mathsf{P_j.pk}, \mathsf{aux}, (\mathsf{linkedQuote} \cup \aleph), \mathsf{lkaux})$, this algorithm outputs 1 if the quotes in all the indicated subsets can all be linked (thus also indicating the parties are linked) or 0 otherwise.

By convention, we allow the use of $\emptyset$ to indicate that any of the input or output (sub)sets to also be empty.

Formalization. The security of linked attestation informally states that an adversary, which has Person-in-the-Middle capabilities and can compromise devices at will, cannot make it appear that two devices are linked when they are not, in fact, so.

A significant limitation on the adversary's capabilities is that compromising a device will not leak its private keys (which are assumed to be held by a TPM). However, the adversary will gain a limited oracle access to those keys upon compromising the device. The limitations to those queries follow rules of access to an actual TPM.

More formally, we define the security of Linked Attestation as a game $\mathtt{LinkSec}_{\lambda,\mathsf{F}}$ played by an adversary $\mathcal{A}$ against its challenger $\mathcal{G}$. The game is parametrized by a security parameter λ and a set of functions F, which we call the *permitted key-access functions*. The challenger begins by running $\mathtt{aLSetup}(1^\lambda)$, returning ppar to the adversary, and then it instantiates two lists: a list of parties $\mathcal{L}_{\mathsf{Reg}} = \emptyset$ and a list of linkable attestations $\mathcal{L}_{\mathsf{Att}} = \emptyset$. The adversary then plays its game by using the following oracles adaptively:

- $\mathsf{OLReg}(n_1, \ldots n_L) \rightarrow (\mathsf{PK}_1, \ldots \mathsf{PK}_L)$: the linked user-registration oracle creates a linked platform consisting of n_i components of the type indicated by $\mathcal{S}_i$. The challenger first instantiates a counter $N_i = 0$ for all i; it also instantiates subsets $\mathbf{s}_i$ as a tuple of n_i handles $\mathsf{P}_{i,j}$, with $N_i + 1 \leq j \leq N_i + n_i$ and then runs the algorithm $\mathtt{aLReg}(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_L)$, instantiating the parties with their keys and outputting the public keysets to the adversary. The subset consisting of the list of subsets is added to $\mathcal{L}_{\mathsf{Reg}}$. We note that this way of registering parties ensures by construction that no party finds itself in multiple sets, nor on multiple platforms.
- $\mathsf{OLHAttest}(\mathsf{P}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux}) \rightarrow (\mathsf{linkedQuote} \cup \aleph, \mathsf{aux}^*)$: this oracle first verifies that $\mathsf{P}.\gamma = 1$. If the condition is false (P is compromised), then this oracle outputs an error symbol $\perp$ (compromised parties must use the oracle $\mathsf{OLCAttest}$ described below). If the condition is true, then this algorithm runs $\mathtt{aLAttest}(\mathsf{ppar}, \mathcal{PK}, \mathsf{P.sk}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux})$, and returns the output to the adversary. The tuple $(\mathsf{P}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*)$ is stored in $\mathcal{L}_{\mathsf{Att}}$.

- $\mathsf{OCompromise}(\mathsf{P}) \rightarrow \mathsf{OK}$: this oracle allows an adversary to compromise party P, thus changing $\mathsf{P}.\gamma$ to 1.
- $\mathsf{OLCAttest}(\mathsf{P}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux}, f) \rightarrow (\aleph, \mathsf{aux}^*)$: this oracle first checks that $\mathsf{P}.\gamma = 1$ (else, $\perp$ is returned as an output). If the condition holds, then this oracle first checks that $f \in F$ and if so, it runs f on $\mathsf{P.sk}$ and input aux to output aux^*. Then it runs $\mathsf{OLHAttest}(\mathsf{P}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux})$ to obtain $\mathsf{linkedQuote}$ (the second output is discarded). Note that by the security of the linked attestation primitive, we will have that $\mathsf{linkedQuote} = \aleph$. The tuple $(\mathsf{P}, (\mathbf{s}_1, \ldots, \mathbf{s}_L), \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*)$ is added to $\mathcal{L}_{\mathsf{Att}}$ and $(\mathsf{linkedQuote}, \mathsf{aux}^*)$ is returned to $\mathcal{A}$.

At the end of its interaction, $\mathcal{A}$ outputs a party P and a tuple of subsets $(\tilde{\mathbf{s}}_1, \ldots, \tilde{\mathbf{s}}_L)$ with an index $i^\star$ such that $\forall i \neq i^\star$, $\mathbf{s}_i := \tilde{\mathbf{s}}_i$ and $\mathbf{s}_{i^\star} := \tilde{\mathbf{s}}_{i^\star} \cup \{\mathsf{P}\}$. In addition the adversary outputs for every party $\mathsf{P} \in \mathbf{s}_1 \cup \cdots \cup \mathbf{s}_L$ (parties being indexed as $\mathsf{P}_{i,j}$) a tuple $(\mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*)$ such that $(\cdot, \cdot, \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*) \in \mathcal{L}_{\mathsf{Att}}$.

We say the adversary *wins* if all the following conditions hold simultaneously:

- For each $\mathbf{s}_i$ the parties inside this set are all registered *i.e.*, they were output by OLReg. In addition P is registered;
- There exists at least one party $\mathsf{Q} \in \mathbf{s}_j$ such that P and Q were issued from different OLReg queries;
- By setting $\Pi_i := (\mathsf{P.pk}, \mathsf{aux}, (\mathsf{linkedQuote} \cup \aleph), \mathsf{aux}^*)$ and, for $k \neq i$, for all $\mathsf{P}_{k,j} \in \mathbf{s}_j$, $\Pi_k := (\mathsf{P_{k,j}.pk}, \mathsf{aux}, (\mathsf{linkedQuote}_{k,j} \cup \aleph), \mathsf{aux}^*_{k,j})$, it holds that $\mathtt{aLLink}(\mathsf{ppar}, \mathcal{PK}, \Pi_1, \ldots, \Pi_L) = 1$.

In other words, the adversary wins if it is able to make attestations stored in $\mathcal{L}_{\mathsf{Att}}$ for parties registered on different platforms (P and Q) link. Note that there are two ways that an attestation can end up in $\mathcal{L}_{\mathsf{Att}}$: either it is issued for an honest component (and then it should hold that $\mathsf{linkedQuote} \neq \aleph$), or it is issued for a compromised party, for an adversarially-chosen evaluation of a permitted function f on a secret key (in which case $\mathsf{linkedQuote} = \aleph$). In other words, at this point a compromised component cannot just bind the output aux^* from the function evaluation oracle with a different quote. The adversary will gain this ability at the next step (when the freshness aux will no longer be chosen by the adversary).

Proof of Theorem 2. We will now prove our construction is secure with respect to the $\mathtt{LinkSec}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$ experiment.

Proof. **Game 0.** The original security game $\mathtt{LinkSec}_{\lambda,\mathsf{F}_{\mathsf{Sign}}}$.

Game 1. We guess parties P, Q output by the adversary in the last part of its game. In other words, the challenger must draw at random two values between 1 and N, such that those values correspond to those chosen by the adversaries. We lose a factor $\frac{1}{N^2}$.

Game 2. We now rule out that $H(\cdot, \mathsf{lkaux}) = (\cdot, \mathsf{lkaux}')$ for any $\mathsf{lkaux} \neq \mathsf{lkaux}'$.

Trivially, if the converse were true,we could break the collision resistance of H with equal probability.

Note that now, since parties P and Q are registered on different platforms, since lkaux keys are unique, and since we have ruled out collisions, any honestly-generated attestations for P and Q will not link. The adversary's only hope is to forge an attestation for either one of those parties.

Game 3. At this point we rule out the fact that P's tuple $(\mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*)$ was in fact part of a tuple $(\mathsf{P}', \cdot, \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{aux}^*) \in \mathcal{L}_{\mathsf{Att}}$ (with $\mathsf{P} \neq \mathsf{P}'$). If that were so, we could construct an adversary against the authenticated attestation scheme (since P purports to be P'). In so doing an important oracle will be the signature oracle OAuth added artificially in the authenticated attestation primitive; the latter will allow us to simulate $\mathsf{OLCAttest}$ queries. We lose $\varepsilon_{\text{Auth-attest}}$.

Game 4. We repeat the previous game hop for party Q, and lose $\varepsilon_{\text{Auth-attest}}$.

At this point, the adversary can no longer win the game.

C Formalization of Authorized Attestation

We will consider parties of multiple categories as for the linked-attestation primitive. We also define a special server entity, denoted S, which stores the following attributes:

- $(\mathsf{pk}, \mathsf{sk})$: a tuple consisting of a public key pk (assumed to be unique and known to all other parties including the adversary) and a private key sk known only to the server. We use $\mathsf{S.pk}$ to indicate the public key of party S, and $\mathsf{S.sk}$ to indicate its private key.
- $\mathsf{S.st}$: a value called *state*, which stores tuples of linked attestations which are susceptible to be linkable to each other.

We will consider an environment in which parties interact with each other in *sessions*. The session is run by two party *instances*, one of the attesting party and the other, of the attestation server. For a party P we denote by π_{P}^{i} the i-th instance of party P.

Just as in the case of linked attestation, parties store a set of keys $(\mathsf{pk}, \mathsf{sk})$, as well as a compromise bit γ.

Party instances use the same keys and have the same compromise bit as the party itself, but in addition keep track of the following session-specific attributes:

- sid: a session identifier, which will be useful in understanding which two party instances converse together.
- pidpk: the public key belonging to this instance's intended communication partner.
- T: a transcript of messages exchanged throughout a protocol session, in plaintext. Even if encryption is used at some point, parties append messages to their transcripts only after decrypting.
- α: this bit is originally set to 0, but can be changed to 1 if this party instance has accepted its partner as a legitimate entity to run the authorized linked attestation with.

– lst: this local state variable stores instance- (and protocol-) specific values, such as encryption keys, randomness, etc.

In addition server instances π_{S}^{j} keep track of the following attribute, which is the output of the immediate attestation process taking place:
– $\mathsf{verdict}$: this attribute stores a bit, initially set to 0, which is flipped to 1 if the attestation server's instance has accepted the attestation received during that session.

We call two instances π_{P}^{i} and π_{Q}^{j} *partnered* if, and only if, the following conditions hold simultaneously: exactly one of P and Q is in fact the attestation server S; $\pi_{\mathsf{P}}^{i}.\mathsf{pidpk} = \mathsf{Q}.\mathsf{pk}$ and $\pi_{\mathsf{Q}}^{i}.\mathsf{pidpk} = \mathsf{P}.\mathsf{pk}$; and $\pi_{\mathsf{P}}^{i}.\mathsf{sid} = \pi_{\mathsf{Q}}^{j}.\mathsf{sid}$.

Authorized linking attestation is defined as the tuple of algorithms and protocols $\mathtt{ALA} = (\mathsf{ASetup}, \mathsf{AReg}, \mathsf{AAttest}, \mathtt{aALink})$ described as follows:
– $\mathsf{ASetup}(1^{\lambda}) \rightarrow (\mathsf{ppar}, \mathsf{S.pk}, \mathsf{S.sk}, \mathsf{S})$: on input the security parameter 1^{λ} (in unary), this algorithm outputs public parameters ppar, as well as the server handle S (such that S is equipped with newly generated keys pk, sk). The value ppar includes the maximal number of allowed disjoint linkable sets of parties, which we denote as L. The values $\mathsf{ppar}, \mathsf{S.pk}$, and S are public, $\mathsf{S.sk}$ remains private. The value $\mathsf{S.pk}$ is added as the first value in the set $\mathcal{PK}$.
– $\mathsf{AReg}(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_L) \rightarrow \{(\mathsf{PK}_1, \mathsf{SK}_1), \ldots, (\mathsf{PK}_L, \mathsf{SK}_L)\}$: this algorithm keeps as state a number L of sets $\mathcal{S}_i$ originally set to $\emptyset$, and a vector of sets of public keys $\mathcal{PK}$ (also initialized to $\emptyset$). On input a number of subsets $\mathbf{s}_i$ $(i = 1, 2, \ldots, L)$, this algorithm first checks that $\forall i, j, \mathbf{s}_i \cap \mathcal{S}_j = \emptyset$ (else the algorithm outputs $\perp$). If the relation is true, then the algorithm generates for each party $\mathsf{P_j} \in \mathbf{s}_i$, (for all j, i) a tuple of public and private keys $\mathsf{P_j.pk}, \mathsf{P_j.sk}$, initializing $\mathsf{P_j}$ with those keys. We require the unicity of all the generated public keys. The subsets $\mathbf{s}_i$ are each added to greater sets $\mathcal{S}_i$. The algorithm groups the keys of all parties $\mathsf{P_j} \in \mathbf{s}i$ in a pair of private/public key subsets: $(\mathsf{PK}_i, \mathsf{SK}_i)$, updating the i-th component $\mathcal{PK}[i]$ of $\mathcal{PK}$ as $\mathcal{PK}[i] \cup \mathsf{PK}_i$. All parties are given access to the public-key subsets (and more generally, to $\mathcal{PK}$).
– $\mathsf{AAttest}(\mathsf{ppar}, \mathcal{PK}, \pi_{\mathsf{P}}^{i}, \pi_{\mathsf{Q}}^{j}) \rightarrow (\mathsf{verdict}, \mathsf{S.st})$: this protocol is an interaction between two party oracles, such that exactly one of P, Q is S. The protocol yields a tuple of values $\mathsf{verdict}$ and $\mathsf{S.st}$ to the server (and no output for the other party). Both party oracles are assumed to update their attributes accordingly as the protocol unfolds.
– $\mathtt{aALink}(\mathsf{ppar}, \mathcal{PK}, \mathsf{S.S.st}, \mathbf{s}_1, \ldots, \mathbf{s}_L) : 0 \cup 1$: given the public parameters and public-key set, the server's current state, and a number of subsets of (purportedly-linked) parties, this algorithm outputs either 0 (the parties are not linked) or 1 (the parties are linked).

We require two types of *correctness* properties. First, we require that running the protocol between two honest parties yields a verdict of 1 (accept) on the side of the attestation server. Secondly, we require that components that are linked at registration will be viewed as linked by the $\mathtt{aALink}$ algorithm. More formally, we require that schemes $\mathtt{ALA} = (\mathsf{ASetup}, \mathsf{AReg}, \mathsf{AAttest}, \mathtt{aALink})$ be such that:
– For all $(\mathsf{ppar}, \mathsf{S.pk}, \mathsf{S.sk}, \mathsf{S}) \leftarrow \mathsf{ASetup}(1^{\lambda})$ and for all parties $\mathsf{P} \in \mathcal{S}_i$ for some $1 \leq i \leq L$, it holds that $(\mathsf{verdict}, \cdot) = \mathsf{AAttest}(\mathsf{ppar}, \mathcal{PK}, \pi_{\mathsf{P}}^{\cdot}, \pi_{\mathsf{S}}^{\cdot})$ (any legitimate party will successfully attest to the legitimate server);

– For all $(\mathsf{ppar}, \cdot, \cdot, \mathsf{S}) \leftarrow \mathsf{ASetup}(1^\lambda)$, for all subsets $\tilde{\mathbf{s}_1}, \tilde{\mathbf{s}_2}, \ldots, \tilde{\mathbf{s}_L}$ such that there exist sets $\mathbf{s}_i$ for $i = 1, 2, \ldots, L$ such that $\tilde{\mathbf{s}_i} \subset \mathbf{s}_i$ and $\mathsf{AReg}(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_L)$ was called and did not result in $\perp$, it holds that:
$\mathtt{aALink}(\mathsf{ppar}, \mathcal{PK}, \mathsf{S.S.st}, \tilde{\mathbf{s}_1}, \ldots, \tilde{\mathbf{s}_L}) = 1$ (parties that are registered together can be linked through the server's state).

D Security games for `AuthAtt`

The three security games we define are parametrized by a function space F and a security parameter λ. They start with the challenger running the setup algorithm and outputting ppar as well as the handle S and its public key $\mathsf{S.pk}$ to the adversary. Note that this will not give the adversary black-box access to S's attributes: it simply allows the adversary to later instantiate new attestation-protocol sessions for that server.

The adversary will then have access to some, or all of the following oracles:

– $\mathsf{OALAReg}(n_1, \ldots n_L) \rightarrow (\mathsf{PK}_1, \ldots \mathsf{PK}_L)$: the authorized linked user-registration oracle creates a linked platform consisting of n_i components of the type indicated by $\mathcal{S}_i$. The challenger first instantiates subsets $\mathbf{s}_i$ as a tuple of n_i handles $\mathsf{P}_{i,j}$, with $1 \leq j \leq n_i$ and then runs the algorithm $\mathsf{AReg}(\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_L)$, instantiating the parties with their keys and outputting the public keysets to the adversary. The subset consisting of the list of subsets is added to $\mathcal{L}_{\mathsf{Reg}}$.
– $\mathsf{ONewSession}(\mathsf{P}, \mathsf{Q}) \rightarrow \pi^i_{\mathsf{P}}$: on input the identity of a target party P and a partnering party Q, if both entities are correctly registered and exactly one of them is the server, then this oracle instantiates an instance of P whose partner will be instantiated as $\mathsf{pidpk}= \mathsf{Q.pk}$. Note that in order to observe an honest session between two parties, an adversary would have to create two partnered instances, one of P and the other of Q.
– $\mathsf{OSend}(M, \pi^i_{\mathsf{P}}) \rightarrow M'$: this oracle simulates sending a message M to an instance π^i_{P}, and outputs the response M' of the party instance. If the input message takes a special value $M = \mathtt{prompt}$ and P is the initiator of the protocol (*i.e.*, the first party to have to send a message), this will trigger π^i_{P} to output the first message in the protocol. We note that some messages, when sent, might trigger errors, leading to an output $M' = \perp$. Other messages might trigger the attributes of the party (or instance) to be modified.
– $\mathsf{ORevealState}(\pi^i_{\mathsf{P}}) \rightarrow \mathsf{lst}$: on input a valid party instance, this oracle returns the value stored by the attribute lst of that instance.
– $\mathsf{OUseKey}(\mathsf{P}, f, \mathsf{aux}) \rightarrow \mathsf{aux}'$: on input a compromised party P (not the server), a function $f \in F$, and an auxiliary input value aux, this oracle evaluates f on $\mathsf{P.sk}$ and aux.
– $\mathsf{OCompromise}(\mathsf{P}) \rightarrow \mathsf{OK} \cup \perp$: on input a registered party $\mathsf{P} \neq \mathsf{S}$, this oracle turns the party's compromise bit to 1 and returns OK. If $\mathsf{P} = \mathsf{S}$ or the party has not been registered, the output is an error symbol $\perp$.

We now proceed to describe each of the three security experiments we consider for our authorized linked attestation primitive.

In the *authorization* game $\mathtt{AuthSec}_{\lambda,\mathsf{F}}$, after the challenger runs the setup algorithm, the adversary $\mathcal{A}$ gets access to all the oracles described above. It ultimately stops with a `stop` message. We say $\mathcal{A}$ wins if, and only if, there exists an instance π^i_{P} such that $\mathsf{P} \neq \mathsf{S}$, for which the following conditions hold simultaneously:
– π^i_{P} ends in an accepting state, *i.e.*, $\pi^i_{\mathsf{P}}.\alpha = 1$;
– There exists no server instance π^j_{S} such that π^j_{S} is partnered with π^i_{P}.
In other words, the adversary wins if it can make a registered party believe it has talked to the server when this is not the case.

In the *linking* game $\mathtt{AuthLink}_{\lambda,\mathsf{F}}$, after the challenger runs the setup algorithm, the adversary $\mathcal{A}$ gets access to all the oracles above. It ends by outputting a tuple $(\mathsf{P}, \mathbf{s}_1, \ldots, \mathbf{s}_L)$: such that for all $1 \leq i \leq L$, $\mathbf{s}_i \subset \mathcal{S}_i$ and there exists a unique i^* such that $\mathsf{P} \in \mathcal{S}_{i^*}$ and $\mathbf{s}_{i^*} = \emptyset$. The challenger sets $\tilde{\mathbf{s}_i} := \mathbf{s}_i$ for all $i \neq i^*$, and $\tilde{\mathbf{s}_i} := \mathsf{P}$. Then the challenger evaluates: $b \leftarrow \mathtt{aALink}(\mathsf{ppar}, \mathcal{PK}, \mathsf{S.S.st}, \tilde{\mathbf{s}_1}, \ldots, \tilde{\mathbf{s}_L})$. The adversary is said to *win* if, and only if the following conditions hold simultaneously:
– $b = 1$;
– There exists a party Q and an index $j^* \neq i^*$ such that $\mathsf{Q} \in \tilde{\mathbf{s}_{j^*}}$ and P and Q were not output by the same $\mathsf{OALAReg}$ query.

In other words, for this second game, the adversary has to run several sessions between (potentially compromised) parties and the (honest) server, thus bringing the server's state to a point where linkage can be verified based on that state.

Finally, for the *indistinguishability* game $\mathtt{AuthInd}_{\lambda,\mathsf{F}}$, once the challenger has finished the setup, it also draws a bit b at random. The adversary gets once more access to the oracles described above. It finally outputs a tuple $(\pi^i_{\mathsf{P}}, m_0, m_1)$, consisting of a party instance and two messages, such that: $|m_0| = |m_1|$ and $\pi^i_{\mathsf{P}}.\alpha = 1$. The challenger uses its knowledge of π^i_{P}'s state on input m_b (which is m_0 or m_1 depending on the challenger's hidden bit) to simulate outputting a message M_b which corresponds to the next protocol message of π^i_{P} as that party would have sent it. Clearly if the protocol requires messages be sent in plaintext, $M_b = m_b$. The instance π^i_{P}, as well as any of its partnering instances, are closed and may no longer be used in any oracle. The adversary may subsequently continue to use oracles at will (except on the instances closed above) and eventually outputs a guess $d \in \{0, 1\}$. We say the adversary *wins* if, and only if, the following conditions hold simultaneously:
– $d = b$;
– No $\mathsf{ORevealState}$ query was made for either π^i_{P}, nor for any instance π^j_{S} of the server such that π^i_{P} and π^j_{S} are partnered.