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

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

[1] Orange Labs, France
[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. 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 virtual configuration. 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 NFVs – 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 [13], "*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). 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). Functionalities of RTS and RTR can 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. First, each component must attest in isolation; then we must attest the *layer-binding* between VMs running on the same hypervisor. This is known as *deep attestation* (DA). There are two typical ways of achieving deep attestation (as described by ETSI standardization documents [13]): single- and multi-channel VM-Based Deep Attestation.
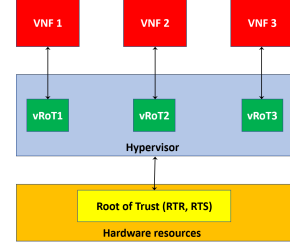

Fig. 1: The setup for DA.

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.
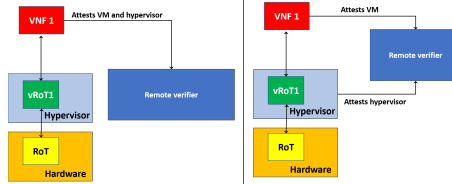

Fig. 2: Single vs multi-channel DA

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. 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. Our goal is to ultimately obtain *authorized linked attestation* (ALA): a primitive which allows components to individually attest (to an authorized entity), and to have their attestations linked. This primitive solves the problem outlined in the introduction, since VMs sharing the same hypervisor will attest in isolation and together with their hypervisor.

ALA schemes 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* (an attestation server can detect if two components are not linked)

We choose to formalize AKA security as the last of a sequence of primitives, each potentially of independent interest and providing gradually stronger properties. This approach has two virtues: first, we are able to use weaker primitives as black-box components in stronger primitives; and second, the individual proof steps are shorter and smoother.

At the basis of our construction is a yea-or-nay *basic attestation scheme*, which is "secure" by assumption. Its functionality is simple: the basic attestation scheme outputs a faulty attestation whenever a component is compromised, and a correct one for honest components. 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 verifies prior to corruption, but fails to verify as soon as a compromise occurs. We can think of authorized attestation as the minimum provided (and required) by multi-channel attestation. 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.

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. In reality, this primitive is based on the Platform Configuration Registers (PCRs) of a TPM. A PCR can store hash digests into a register of the length of the hash function output. Typically a TPM will have multiple banks corresponding to various hash functions (*e.g.*, a sha1 bank and a sha256 bank) with 24 registers for each bank. PCR are reset at each boot and are only updateable through an extension operation $PCR_1 \leftarrow H(PCR_1 \mid H(measurement))$. We assume the attacker has no physical access to the component and thus cannot tamper with TPM measurements by using hardware attacks. In practice, this is somewhat limiting since we do not account for runtime corruption; thus, the primitive is vulnerable to Time of Check Time of Use (TOCTOU) attacks. Several proposed mechanisms were introduced to monitor runtime integrity, *e.g.*, LKIM [19] or DynIMA [12]; moreover, in recent years several advancements were made towards verifying runtime integrity for IoT devices [15, 17]. Yet, these solutions are not as widely spread at the present day as TPM-based attestation at startup.

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 [13].

Another limitation of our scheme lies in our model of linked-attestation component. We consider classes of components which can be linked. At 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 [14].

Related Work. Many attacks have been recently reported on remote attestation mechanism [10] or 5G standards [16]. 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 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 [9] 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 [7, 11]. This is precisely our goal 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 [6], 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 [5] since it cannot rely on cryptographic secrets to authenticate the prover device. The first to have taken into account deep attestation are Lauer and Kuntze in [18] but their solution misses 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. Each of these intermediate steps plays a double role: on the one hand, it formalizes security guarantees that are of independent interest for attestation (if, for instance, layer-linking is not required); on the other hand, it provides an intuition of the guarantees which specific cryptographic primitives can help achieve.

The first, and basic-most step in our architecture is *basic attestation*. 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. Importantly, basic attestation does not employ cryptography to achieve this feature, but rather, the attestation of registers at startup, using a RoT[5].

---

[5] To ease notation, we assume that all the registers are attested, 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. Here, thus, we enhanced basic attestation with a cryptographic component, which is in fact sufficient to guarantee the basic functionality required by multi-channel attestation. 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. While this primitive has no immediate parallel in real-world attestation, we use it as a handy way of dividing the security proof of our ultimate result into two: linked-attestation will focus on proving the fact that two attestations can be securely linked; whereas authorized linked attestation models attestation as a protocol, using fresh randomness and a secure channel using an honest attestation server.

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`):

– `aBSetup`$(1^\lambda) \to$ `ppar`: on input the security 1^λ (in unary), this algorithm outputs some public parameters `ppar`.

- $\mathsf{aBAttest}(\mathsf{ppar}) \to \mathsf{quote}$: 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 $\aleph$.
- $\mathsf{aBVerif}(\mathsf{ppar}, (\mathsf{quote} \cup \aleph)) \to 0 \cup 1$: 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: $\mathsf{aBVerif}(\cdot, \cdot, \aleph) = 0$.

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

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 $\mathsf{aBSetup}$ to output ppar which is given to $\mathcal{A}$. 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 $\mathsf{aBAttest}$ and $\mathsf{aBVerif}$ algorithms. In addition, it has access to the $\mathsf{OBAttest}$ oracle: $\mathsf{OBAttest}() \to (\mathsf{quote} \cup \aleph)$. This oracle calls the $\mathsf{aBAttest}()$ algorithm for the (corrupted) party P and returns 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 $\mathsf{aBVerif}(\mathsf{ppar}, \mathsf{quote}) = 1$. Note that by construction our basic attestation primitive is secure, since once the compromise bit is set, the output is $\aleph$, which always yields $\mathsf{aBVerif}(\mathsf{ppar}, \aleph) = 0$.

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

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

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

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.

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. If we also assume authentication, it would go against the principle of using minimal assumptions.

Correctness. The correctness of our construction depends on the detection of a compromised component. There are three cases to consider. Assume first that the component is compromised. In that case, the output attestation is $\aleph$. The component can try to authenticate this quote, but the verification will fail. In the second case, the component (VM or hypervisor) is not compromised, and so will receive a valid attestation quote, authenticated by the TPM. This authenticated quote will verify. Finally, in the third case, the component is not compromised, and receives a valid authentication quote. At this point, the adversary might try to forward the authenticated quote and pass it off as someone else's attestation, but this will fail as long as the authentication primitive is EUF-CMA secure.

Formalization. The precise formalization of this primitive is in the full version of our paper [4]. 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 $\mathtt{AuthAtt}$ scheme is as follows:

- $\mathtt{aAuthSetup}(1^{\lambda}) \to \mathsf{ppar}$: 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

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).

$\mathtt{BasicAtt}$ run for those parameters). It sets $\mathsf{ppar} := (\mathsf{ppar}_1, \mathsf{ppar}_2, \ldots, \mathsf{ppar}_N, N)$, and outputs this value.

- $\mathtt{aAuthKGen}(\mathsf{P_i}) \to (\mathsf{P_i.pk}, \mathsf{P_i.sk})$: 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 $\mathtt{aAuthKGen}$ runs $\mathtt{aSigKGen}$ 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.

- $\mathtt{aAuthAttest}(\mathsf{ppar}, \mathsf{P.sk}, R) \to \mathsf{authQuote} \cup \aleph$: 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 $\mathsf{authQuote} = \aleph$.

- $\mathtt{aAuthVerif}(\mathsf{ppar}, \mathsf{P.pk}, R, (\mathsf{authQuote} \cup \aleph)) \to 0 \cup 1$: 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 $b \wedge d$. Notably, 1 is output if, and only if, signature and basic attestation verify concomitantly.

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

The proof is given in the full version of our paper [4]

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 various VMs hosted on that plateform).

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 `LinkedAtt` as a tuple of algorithms `LinkedAtt` = (`aLSetup`, `aLReg`, `aLAttest`, `aLVerif`, `aLLink`), defined for some auxiliary set $\mathcal{AUX}$. The detailed formalization is given in the full version [4].

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 linkedQuote and an auxiliary linking value lkaux. Finally, the verification algorithm checks the attestation in each individual 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.
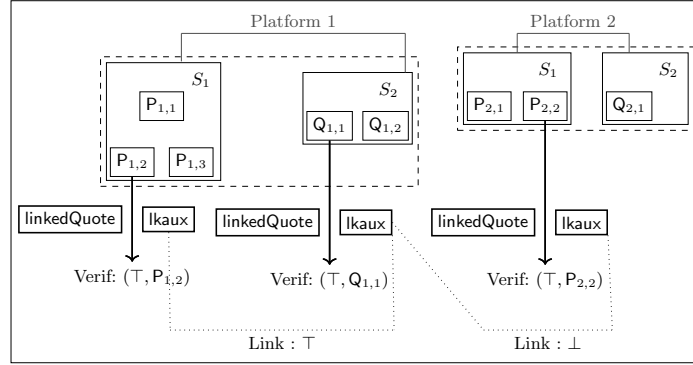

Fig. 5: Linked attestation primitive. Dashed lines indicate platform under the same registration. Here, both platforms are composed of two subsets (S_1, S_2). There are a total of three quote verifications ($\mathsf{P}_{1,2}, \mathsf{Q}_{1,1}, \mathsf{P}_{2,2}$). The link verification outputs true for devices registered on the same platform and false otherwise.

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 $\texttt{LinkSec}_{\lambda,\mathsf{F}}$ 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 attestations 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 ($\texttt{aAuthSetup}$, $\texttt{aAuthKGen}, \texttt{aAuthAttest}, \texttt{aAuthVerif}$) as a black box. The basic construction is depicted in Figure 6. During setup, our linked-attestation scheme first runs $\texttt{aAuthSetup}$ and outputs ppar and $L = 2$. Note that by construction $\texttt{aAuthSetup}$ 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 $\texttt{aAuthKGen}$ of the underlying authenticated attestation scheme for each party, 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

$\underline{\texttt{aLSetup}(1^\lambda)}$:
$\texttt{ppar}' \leftarrow \texttt{aAuthSetup}(1^\lambda)$
Return $\texttt{ppar} \leftarrow \texttt{ppar}'$

// Registers a platform with set $\mathbf{s}_1$ of VMs and the hypervisor in $\mathbf{s}_2$
$\underline{\texttt{aLReg}(\mathbf{s}_1, \mathbf{s}_2)}$: For each $i \in \{1, 2\}$:
  For each $j \in \mathbf{s}_i$:
   $(\mathsf{P}_j.\mathsf{pk}, \mathsf{P}_j.\mathsf{sk}) \leftarrow \texttt{aAuthKGen}(\mathsf{P}_j)$
  Group all $\mathsf{P}_j.\mathsf{pk}$ into PK_i and all $\mathsf{P}_j.\mathsf{sk}$ into SK_i
 Return$\{(\mathsf{PK}_1, \mathsf{SK}_1), (\mathsf{PK}_2, \mathsf{SK}_2)\}$

// Attesting VM P on platform $(\mathbf{s}_1, \mathbf{s}_2)$ for nonce aux
$\underline{\texttt{aLAttest}(\texttt{ppar}, \mathcal{PK}, \mathsf{P}.\mathsf{sk}, (\mathbf{s}_1, \mathbf{s}_2), \mathsf{aux})}$:
Get $\mathsf{P}.\mathsf{pk}$ matching $\mathsf{P}.\mathsf{sk}$ from $\mathcal{PK}$
$\mathsf{lkaux} \leftarrow \mathsf{P}.\mathsf{pk}$ *// The linking information is P's public key*
$\mathsf{aux}^* \leftarrow H(\mathsf{aux}||\mathsf{lkaux})$ *// Embed lkaux into attestation nonce*
$\texttt{authQuote} \leftarrow \texttt{aAuthAttest}(\texttt{ppar}', \mathsf{P}.\mathsf{sk}, \mathsf{aux}^*)$
$\mathsf{linkedQuote} \leftarrow \texttt{authQuote}$
Return $(\mathsf{linkedQuote}, \mathsf{lkaux})$

$\underline{\texttt{aLVerif}(\texttt{ppar}, \mathsf{P}.\mathsf{pk}, \mathsf{linkedQuote}, \mathsf{aux}, \mathsf{lkaux})}$:
// Verify attestation quote of party P
$\mathsf{aux}^* \leftarrow H(\mathsf{aux}||\mathsf{lkaux})$;
$\texttt{authQuote} \leftarrow \mathsf{linkedQuote}$
Return $\texttt{aAuthVerif}(\texttt{ppar}', \mathsf{P}.\mathsf{pk}, \texttt{authQuote}, \mathsf{aux}^*)$

// Attest hypervisor P on platform $(\mathbf{s}_1, \mathbf{s}_2)$ with nonce aux
$\underline{\texttt{aLAttest}(\texttt{ppar}, \mathcal{PK}, \mathsf{P}.\mathsf{sk}, (\mathbf{s}_1, \mathbf{s}_2), \mathsf{aux})}$:
Parse $\mathcal{PK}$ as $\mathcal{PK}[1], \mathcal{PK}[2]$ *// $\mathcal{PK}[1]$ is the set of all VM pks*
Parse $\mathcal{PK}[1]$ as $\mathsf{PK}^1, \mathsf{PK}^2...\mathsf{PK}^{|\mathcal{S}_1|}$ *// PK^i contains the keys of all VMs on platform i*
Set $\mathsf{lkaux} \leftarrow \mathsf{PK}^k$ with k the index of $\mathbf{s}_1$ in $\mathcal{S}_1$
// lkaux is now the list of all VM keys on that platform
$\mathsf{aux}^* \leftarrow H(\mathsf{aux}||\mathsf{lkaux})$
// Embed lkaux into a new attestation nonce
$\texttt{authQuote} \leftarrow \texttt{aAuthAttest}(\texttt{ppar}', \mathsf{P}.\mathsf{sk}, \mathsf{aux}^*)$
$\mathsf{linkedQuote} \leftarrow \texttt{authQuote}$
Return $(\mathsf{linkedQuote}, \mathsf{lkaux})$

// Link VM quotes from Π_1 and the hypervisor quote from Π_2
$\underline{\texttt{aLLink}(\texttt{ppar}, \mathcal{PK}, \Pi_1, \Pi_2)}$:
Initialize $\mathsf{AUX}_{\mathsf{vm}} \leftarrow \emptyset$
For each $(\mathsf{P}_j.\mathsf{pk}, \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{lkaux}) \in \Pi_1$:
 Return 0 if $\texttt{aLVerif}(\texttt{ppar}, \mathsf{P}_j.\mathsf{pk}, \mathsf{linkedQuote}, \mathsf{aux}, \mathsf{lkaux})$ returns 0
 Return 0 if $\mathsf{lkaux} \neq \mathsf{P}_j.\mathsf{pk}$
 // Linking fails if quotes fail to verify or authenticate each VM
 Add lkaux to $\mathsf{AUX}_{\mathsf{vm}}$ *// Each lkaux here is a VM public key.*
Parse Π_2 as $(\mathsf{P}_j.\mathsf{pk}, \mathsf{aux}, \mathsf{linkedQuote}, \mathsf{lkaux})$
If $\texttt{aLVerif}(\texttt{ppar}, \mathsf{P}_j.\mathsf{pk}, \mathsf{linkedQuote}, \mathsf{lkaux})$ returns 0
$\mathsf{AUX}_{\mathsf{hym}} \leftarrow \mathsf{lkaux}$ *// This lkaux is a list of VM public keys.*
Return 0 if $\mathsf{AUX}_{\mathsf{vm}}$ is not a subset of $\mathsf{AUX}_{\mathsf{hym}}$
// Linking fails if the hypervisor's list of PKs does not include all VM keys.
Return 1

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 $\texttt{aLAttest}$ algorithm, the main difference between them being that the hypervisor embeds a list of public keys in its nonce.

then runs $\texttt{aAuthAttest}$ 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 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 $\texttt{aAuthVerif}$ 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.

Correctness. The $\texttt{LinkedAtt}$ scheme is built upon the $\texttt{AuthAtt}$ scheme. There are two types of component to consider, VM and hypervisor. When a component is registered on a plateform, its public key is appended in a list (PK_1 for VMs, and PK_2 for the hypervisor). The public key of a VM is appended to the quote in $\texttt{aLAttest}$ and can be retrieved by the hypervisor. The latter can link the attestation to a public key via algorithm $\texttt{aLLink}$. We consider two cases to verify the correctness (1) a VM (not compromised) is not registered on the plateform, and (2) a component (VM or hypervisor) is compromised. For (1) the attestation will be correct since the component is not compromised, but the linking process will abort since the public key does not belong to PK_1. For (2) if a VM (or the

hypervisor) is compromised then the attestation will fail since the authenticated attestation is supposed to be correct (the `aAuthAttest` algorithm is executed to generate the quote).

Security. We prove (see full version of our paper [4]) the security of our scheme with respect to a single permitted function, F_{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* `AuthAtt` *scheme is secure and 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* [18].

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, the server might have enough information in its state to decide if 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 which returns 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 the full version [4]. 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 $S.pk = pk$ and $S.sk = sk$ respectively, and a state $S.st$. 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 $ALA = (ASetup, AReg, AAttest, aALink)$. The first, second, and last of these are algorithms, while $AAttest$ 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 s_1 for the VMs on the platform and 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 s_1) and respectively the public key of the platform's hypervisor (the hypervisor in the corresponding s_2) are stored respectively in subsets PK_1, 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 instance 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 $verdict, S.st$: the verdict $verdict$ 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 $aALink$ algorithm, yielding either 1 (the components are linked) or 0 otherwise.

Construction. Our construction of the ALA primitive can be seen in the Figure 7. We consider the existence on an underlying $LinkedAtt$ scheme that we

$ASetup(1^\lambda)$: $ppar \leftarrow aLSetup(1^\lambda)$ Create S $(S.pk, S.sk) \leftarrow aSigKGen(1^\lambda)$ Return $(ppar, S.pk, S.sk, S)$	$AReg(s_1, s_2)$: $\{(PK_1, SK_1), (PK_2, SK_2)\} \leftarrow aLReg(s_1, s_2)$ Return $\{(PK_1, SK_1), (PK_2, SK_2)\}$	$aALink(ppar, \mathcal{PK}, S.st, s_1, s_2)$: Parse S.st as Π_1, Π_2 Return $aLLink(ppar, \mathcal{PK}, \Pi_1, \Pi_2)$

$$AAttest(ppar, \mathcal{PK}, \pi_P^i, \pi_Q^j):$$

Component		Server
	Establish TLS channel	
$(linkedQuote, lkaux) \leftarrow aLAttest(ppar, \mathcal{PK}, P.sk, (s_1, s_2), aux)$	$\xleftarrow{\quad aux \quad}$ AttestationRequest	$aux \xleftarrow{\$} \mathcal{AUX}$
	$\xrightarrow{(linkedQuote, lkaux)}$ AttestationResponse	$verdict \leftarrow aLVerif(ppar, P.pk_j, linkedQuote, aux, lkaux)$ Add $(P_j.pk, aux, linkedQuote, lkaux)$ to Π_i in S.st

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

use for the `aLSetup`, `aLReg` and `aLLink` in a straightforward manner. However, the `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 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 the full version [4].

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

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

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.*

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

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 (note that some of our tests use more than 2 VMs – up to 55).

Fig. 8: Architecture for tests.

Our *hypervisor* is a laptop running Ubuntu 20.04.3 (kernel version 5.11.0-40) with an Intel i7-10875H CPU, 32GB RAM and a STMicroelectronics TPM. We used KVM to turn this laptop into a hypervisor. For high attestation performance, we used full *virtual TPM implementation*, using QEMU [1] with libtpms 0.7 [8] and swtpm 0.5 [20].

All *virtual machines* are QEMU virtual machines (version 4.2.1) with 1 core and 512 RAM running Fedora 35 Cloud. The VM as well as the virtual TPM instances are managed using Vagrant and Vagrant-Libvirt plugin.

The hypervisor, server, and VMs communicate through a private network created with Vagrant. 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).
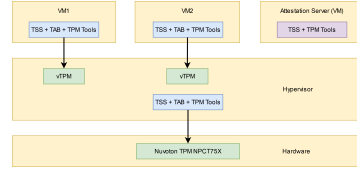
Table 1: Minimum, median, mean and maximum time in second for attestation of a hypervisor and a virtual machine for 100 trials.

	min	median	mean	max
Hypervisor	3.801	11.940	13.896	40.762
VM	0.337	0.416	0.422	0.572

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.

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 median value of those 100 trials. As expected, time for an attestation using a hardware TPM is much higher than using a vTPM.

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 first runs the attestation of the hypervisor, and then (sequentially or in parallel) the attestation of a varying number of VM (up to a maximum of 55). 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

Fig. 9: Attestation time.

(which only occurs once) takes larger time than the subsequent VM run-times.
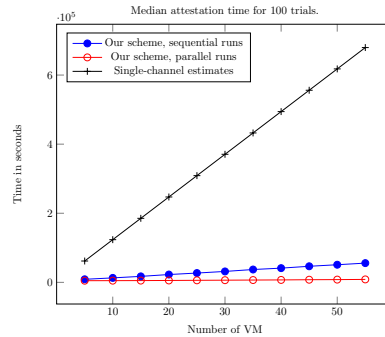
Comparison to single-channel attestation. We did not implement single-channel attestation. However, since we have implemented hypervisor and VM attestations, we can theoretically estimate the run-time of single-channel attestation for a varying number of VMs – which we plot in Figure 9. Indeed, a single-channel 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 run-time, 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 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. Arfaoui, G., Fouque, P.A., Jacques, T., Lafourcade, P., Nedelcu, A., Onete, C., Robert, L.: A cryptographic view of deep-attestation, or how to do provably-secure layer-linking. Cryptology ePrint Archive, Report 2021/1487 (2021), `https://ia.cr/2021/1487`
5. 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)

6. 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)

7. 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)

8. Berger, S.: Library for TPM tools. `https://github.com/stefanberger/libtpms`

9. 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)

10. 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)

11. 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)

12. Davi, L., Sadeghi, A., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the ACM STC. pp. 49–54. ACM (2009)

13. ETSI: Network functions virtualisation(nfv); trust; report on attestation technologies and practices for secure deployments (10 2017)

14. 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)

15. Hristozov, S., Heyszl, J., Wagner, S., Sigl, G.: Practical runtime attestation for tiny iot devices. In: Proceedings of NDSS DISS (01 2018)

16. 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)

17. Kuang, B., Fu, A., Zhou, L., Susilo, W., Zhang, Y.: DO-RA: data-oriented runtime attestation for iot devices. Comput. Secur. **97**, 101945 (2020)

18. Lauer, H., Kuntze, N.: Hypervisor-based attestation of virtual environments. In: 2016 IEEE UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld. pp. 333–340 (2016)

19. Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., McDonell, C.D.: Linux kernel integrity measurement using contextual inspection. In: Proceedings of ACM STC. pp. 21–29. ACM (2007)

20. Safford, D., Berger, S.: Software TPM emulator – swtpm. `https://github.com/stefanberger/swtpm`