TokenWeaver: Privacy Preserving and Post-Compromise Secure Attestation

Cas Cremers[1], Gal Horowitz[3], Charlie Jacomme[2], and Eyal Ronen[3]

[1]CISPA Helmholtz Center for Information Security, Germany
[2]Inria Paris, France
[3]Computer Science Department, Tel Aviv University, Tel Aviv, Israel

October 15th, 2024 – v1.2*

Abstract

Modern attestation based on Trusted Execution Environments (TEEs) can significantly reduce the risk of secret compromise, allowing users to securely perform sensitive computations such as running cryptographic protocols for authentication across security critical services. However, this has made TEEs a high-value target, driving an arms race between novel compromise attacks and continuous TEEs updates.

Ideally, we want to achieve Post-Compromise Security (PCS): even after a TEE compromise, we can update it back into a secure state. However, at the same time, we would like to guarantee the privacy of users, in particular preventing providers (such as Intel, Google, or Samsung) or services from tracking users across services. This requires unlinkability, which seems incompatible with standard PCS healing mechanisms.

In this work, we develop TokenWeaver, the first privacy-preserving post-compromise secure attestation method with automated formal proofs for its core properties. Our construction weaves together two types of token chains, one of which is linkable and the other is unlinkable. We provide the formal models based on the Tamarin and DeepSec provers, including protocol, security properties, and proofs for reproducibility, as well as a proof-of-concept implementation in python that shows the simplicity and applicability of our solution.

1 Introduction

One of the most basic requirements for secure communication is the ability to authenticate the identity of remote parties. Proving one's identity usually involves proving the knowledge of some secret, such as a password or cryptographic key. However, the security of any authentication scheme is only as strong as the security of the secrets used for authentication. If an attacker is able to compromise the device that stores the secrets and extracts them, it can exploit them for impersonation. To mitigate such attacks, a Trusted Execution Environment (TEE) such as ARM's TrustZone [2], AMD's Secure Encrypted Virtualization (SEV) [26], and Intel's SGX [16] offers a secure and isolated environment at the hardware level that can be used to protect sensitive secrets. Remote attestation allows a device to prove that it is running the latest and most secure software version and that cryptographic keys were generated and used inside a client-side TEE.

In practice, remote attestation introduces two additional requirements. First, we want to protect users' privacy: no one should be able to track users as they access different services by exploiting the remote attestation mechanism, which can be expressed as an unlinkability property. For example, if the same device performs remote attestation to two different third-party servers, the servers shouldn't be able to learn it is the same device even if they both collude with the provider.

Second, we would like to achieve Post-Compromise Security (PCS) [15]. In the past few years, we have seen a large number of attacks that are able to extract cryptographic keys and bypass the attestation of even state-of-the-art TEEs such as TrustZone [39, 40, 43, 50], SEV [10, 29, 30, 31, 37, 48, 49], and SGX [7, 12, 18, 21, 22, 27, 28, 32, 38, 42, 44, 45, 46, 47]. Although subsequent software or microcode patches

*An extended abstract of this paper appears at IEEE S&P'25; this is the full version.

mitigate the attacks, all previous secrets stored in the TEE may have been compromised. This raises the question of how to detect if a vulnerability has been exploited in the wild and recover the trust in our attestation process even when cryptographic keys are leaked, i.e., whether we can achieve PCS.

To achieve PCS, even if an attacker obtains all cryptographic material of a user (e.g., by cloning a user) at some point and actively controls the network, it should still be possible to either lock out the attacker and heal the honest user, or at least to detect the compromise. However, the natural way to detect the compromise is to be able to detect that there are suddenly two agents, the honest user and the attacker, that in fact share the same TEE. But, crucially, this implies that the provider is then able to link the distinct uses of the same TEE that rely on the same set of secrets.

However, privacy explicitly requires unlinkability, i.e., to prevent any "tracking" of the usage of secrets. Due to the importance of this problem, both Google and Intel attempted to solve it (for TrustZone and SGX, respectively), but as we will show, only partial solutions were given. Thus, we ask ourselves the following question:

Is it possible to detect a compromise of a TEE and recover security while still preserving the privacy of users?

Contributions. In this paper, we present TokenWeaver, the first formal-analysis co-design solution for a privacy-preserving and post-compromise secure attestation with TEEs. To achieve this, we proceed in the following steps:

- We define and model the concrete security goals and requirements for a privacy-preserving and PCS secure remote attestation.
- We design both linkable and unlinkable one-time authorization mechanisms that are provably privacy-preserving and post-compromise secure.
- We leverage these mechanisms to build the fully-fledged TokenWeaver solution that a TEE provider can use to provision tokens that support both anonymous and identifiable (non-anonymous) attestation to third-party parties while achieving both PCS and privacy w.r.t to third parties and the TEE provider.
- Unlike existing solutions, we formally model and prove the core properties of TokenWeaver and provide reproducible results at [17]. Notably, our co-design allowed us to detect and fix an early design flaw.
- To advocate for the simplicity, efficiency, and scalability of our solution, we provide a python-based proof-of-concept and benchmarking results.

The design of our PCS solution allows the TEE provider to automatically detect when a TEE was compromised. Thus, our solution not only helps to protect individual users but also acts as a "canary in the coal mine." It allows the TEE provider to learn that a new vulnerability has been exploited and needs to be fixed. The fact that such attacks cannot remain stealthy and will be detected can be an added deterrence against exploiting or even developing them in the first place.

Outline. We give TEE's background and related work on remote attestation in Section 2. We then outline our security goals in Section 3, define the building blocks for our solution in Section 4, and present our full solution in Section 5. We formally analyze the security of our solution in Section 6, and describe our proof-of-concept implementation and benchmark in Section 7. We discuss limitations and future work in Section 8 and conclude in Section 9.

In the appendices, we provide pseudo-code for TokenWeaver, discuss secure channel requirements, and provide details on the privacy proof.

2 Background

We first describe here remote attestation in the TEEs setting, existing solutions and their limitations.

2.1 Remote attestation in the TEE setting

Context description. The scenario we are considering involves the following parties:

- A provider (e.g., Intel, Google, or Samsung);
- A user;
- A user device with a TEE provisioned by the provider; and
- A set of third-parties that the user interacts with through its device.

Each TEE is is provisioned with initial keying material by the provider. Such TEEs should be able to run the "Secure World" — a small and sensitive code base, securely and in isolation from the "Normal World" (that includes the operating system and regular applications). Additionally, they should be able to *attest* that they are valid TEEs to third parties, either using an Anonymous Certificate (AC) or using a Identifiable Certificate (IC). ACs are used to attest to being *some* valid TEE, and ICs are used to attest to being a *specific* valid TEE with a given Serial Number (SN). This attestation capability is the core functionality that we wish to design in this work. As such, the main functionalities that need to be considered from the TEE point of view are:

1. Attestation to a third party;
2. Provisioning of new certificates from the provider;

Point 1) is the core usage case of TEEs, and Point 2) is essentially the certificate provisioning and management. To ensure some security guarantees, additional mechanism are then added to those functionalities

Our goals. In this generic TEE setting, our main goals are:

1. **privacy**: A single user owning a TEE can use it with distinct ACs to attest to different parties. In such a case, even if those parties and the provider collude, they should not be able to detect that those different attestations in fact come from the same TEE.
2. **PCS**: if either the secrets of some certificate (or even the secrets used to performing the provisioning) are compromised, then it should be possible to either detect the compromise, or heal the corresponding TEE and lock the attacker out.

2.2 Existing solutions and limitations

Google. Google's certificate provisioning for Android is defined through their APIs, and some informal descriptions of the internal details are given a Google blog-post [6]. In this blog-post, PCS is explicitly stated as a goal of Android Attestation. Android Attestation will keep a list of known-compromised software and will not allow devices running this software to be provisioned. It is not clear which exact properties of PCS they claim to achieve. However, based on the user-facing API, it seems that it is possible for Google to track users using this mechanism. The informal description [6] states that tracking is prevented by a so-called *split-brain* solution: internally, Google strictly separates the verification of the device's public key from the processing of the attestation key. If done correctly, it means that the server verifying the device key (and thus knows which device) never learns the attestation keys that are handled by other servers, preventing the attestation keys from being linked to the device key. However, this cannot be externally verified, and depends on trusting Google to enforce this policy internally through some additional mechanism.

SGX. Intel provides a method for remote attestation for code running inside the SGX enclave based on Enhanced Privacy ID (EPID) [9]. It extends the previous Direct Anonymous Attestation (DAA) solution [8] with enhanced revocation capabilities. The code running inside the enclave is signed using an Intel-provisioned private key. The signature is then verified by Intel's Attestation Server (IAS). At a technical level, EPID uses bilinear pairing to support anonymous attestation. It also supports revocation by requiring clients to prove that they did not generate previous signatures that were flagged (using a zero-knowledge proof of discrete logarithm inequality). Note that the cost of verification grows linearly with the number of revoked signatures, so in practice, only a small number of revocations can be supported. The EPID keys can be updated using dedicated shared symmetric keys called the "Provisioning Key". Recently, Intel proposed a new attestation solution called SGX DCAP [41] that relies on ECDSA signatures which do not provide anonymity of the signer.

Limitations. Existing solutions are lacking in two respects:

- they do not provide PCS guarantees, but only offer basic healing mechanisms;
- privacy concerns are at best addressed informally.

Indeed, Google's solution only offers privacy as long as one trusts Google, which is a surprising setting for privacy. The only healing mechanism offered by Google's solution is a time based expiration for certificates. This means that if an attacker can compromise a certificate of a TEE, the compromised certificate will at some point be useless. However, for deeper compromises of the TEE, revealing for instance the cryptographic material used by the TEE to obtain fresh certificate, no healing would be possible. There is also no mechanism to detect such a compromise.

While privacy preserving, Intel's solutions are expensive because of their use of EPID. It offers the possibility to manually revoke a potentially compromised certificate through the blacklist system of EPID, and to also manually regenerate the EPID keys using the provisioning keys. However, these two

mechanisms need to be triggered manually, and there is no compromise detection mechanism that could inform such a decision. Moreover, if the provisioning keys are compromised, healing becomes impossible.

In contrast, we propose in the next sections a a first proposal for a **privacy preserving and post-compromise secure attestation mechanism for TEEs.**

2.3 Further related work

To the best of our knowledge, we are the first to consider the PCS issue in the context of certificate provisioning for TEEs, and as such, there does not exist any real-world or academic solution in the literature. More generally, w.r.t. the privacy concerns, our work can be seen as neighbor to the anonymous credentials research area. In this context, we previously mentioned how Direct Anonymous Attestation [8] was in fact the ancestor of EPID.

Recently, Privacy-pass [19] was proposed to provide an anonymous user-authentication mechanism. They use a Verifiable Oblivious Pseudo-Random Function (VOPRF) to provision anonymous tokens, that can for instance be used to reduce the number of CAPTCHAs challenges specific users are requested to solve. Our work can also be linked to the anonymous blocklisting/allowlisting ideas from [23]: intuitively, only honest TEEs can obtain certificates.

In general, none of these works match our specific needs, nor provide PCS. While such techniques could be reused to emulate the blinded token part in the unlinkable chain we later define, for instance with a VOPRF, they would not allow for delivering blindly signed certificates that the TEE can then use to third parties. Since we already rely on blind signatures for the third-party functionality, we chose to also use them for the blinded tokens.

The simpler global attestation key variant of TokenWeaver (see Section 8) was suggested as a new attestation mode for FIDO2 in a subsequent work [5]. The authors mention that this will be the first FIDO2 attestation mode that will provide anonymity together with PCS and the ability to revoke compromised authenticators. They also provide a pen-and-paper proof showing that the simple TokenWeaver-based mode indeed provides authentication security, unlinkability, and PCS. Their proof complements our symbolic analysis, each covering slightly different aspects of the full solution, and yielding a different type of assurance.

3 Informal Security Goals

A first basic security goal of the certificate provisioning is **Authentication**: only valid TEEs can successfully obtain certificates from the provider. This will of course be covered by our TokenWeaver proposal. In this paper, we go beyond this, and try to achieve PCS guarantees in a privacy preserving way. We now provide a high-level description of our security goals (see Section 6 for the formal version).

3.1 PCS

The property we aim to achieve deals with the possibility of automatically recovering after a compromise, namely **Post-Compromise Security (PCS)** [15]. In many security protocols, compromising the state of some honest party implies a complete security loss (at least in regard to that party). However, for protocols that have a state-update mechanism, PCS specifies that it is possible to recover from compromise: after a compromise of some honest party's state, if the honest party performs one more step of the protocol, the honest party should be able to **heal**, and the attacker can be locked out again. In the context of TEEs, PCS is relevant as the TEE state is frequently updated by a recurring certificate provisioning mechanism. The PCS goal is then that, when an attacker compromises a TEE, if after the compromise this TEE did a new certificate provisioning round, then the TEE heals and the attacker is locked out again.

Compromise levels. We can define various flavors of PCS, depending on whether all secrets are compromised, or only a subset. For our TEE context, we consider two variants:

1. Certificate compromise, where only a signing key is stolen by the attacker;
2. Full compromise, where any and all secrets within the TEE are stolen by the attacker.

Distinguishing these levels make sense as the long-term secrets can be stored more securely and used only for a dedicated set of operations, while the certificates can be accessed (and misused) through the API by multiple applications as was exploited in [43].

Each type of compromise corresponds to a different level of possible PCS guarantees. We will set out to design in the next sections a solution that provides the following PCS guarantees:

- In case of a certificate compromise, the certificate will become useless at some point.

- If an honest TEE is fully compromised (the attacker learns all secrets), and it subsequently runs the certificate provisioning, the attacker becomes locked out again.

Notice that the first point is easily achieved, as we can simply put an expiry date on the certificates, similar to Google's approach. The second point is substantially more challenging: recall that SGX does not meet the second point, as the attacker can obtain the EPID key and go forever undetected, and only a manually triggered recovery mechanism can lock the attacker out again.

Additional goals beyond PCS. There are several reasons why stronger variants of PCS cannot be achieved. For example, a persistent compromise where the attacker continuously compromises fresh material from the TEE, allowing it to always defeat any healing mechanism.

In addition, in the case of a full compromise, there is nothing we can hope for if the honest TEE never runs the protocol again to obtain new certificates, as the attacker can simply perfectly impersonate the honest TEE forever; intuitively, this is why healing is required. As noted in [15], this is in general impossible to prevent, since the attacker cannot be distinguished from the honest TEE. A side effect is that if after a full compromise, the attacker runs the protocol itself first, the healing mechanism may lock out the honest user. Fortunately, such a situation can be detected [36], and we thus consider two additional goals:

- **Clone Detection**: if an attacker performs a full compromise ("clones" the honest TEE) and runs the healing mechanism to lock out the honest user, then this can be detected and the user will be notified.
- **Active revocation**: if the clone detection is triggered, an independent mechanism will allow the honest user to completely reset the protocol, locking out the attacker and restoring the trust for the honest user. Active revocation typically needs to rely on some Out-of-Band (OoB) channel to reset the state of the protocol.

3.2 Privacy

To propose a fully satisfying solution, we want to achieve PCS but in a **privacy-preserving** way. For example, if the provider and third-party servers collude, they might be able to track the usage of certain certificates across multiple servers, and even link them to a specific device or group of devices.

Our privacy goal is an **unlinkability** property, where an adversary cannot distinguish whether some actions were performed by the same TEE or not, which is stronger than anonymity. We aim for two joint privacy properties:

- **Certificate Provisioning Unlinkability**: the certificate provisioning step is unlinkable, even for colluding providers and third-parties. This implies that no one can know if a particular TEE even requested some certificates, or if two distinct provisioning map to the same TEE.
- **Attestation Unlinkability**: the attestation step is unlinkable, even for colluding providers and third-parties. This implies that no one can know if a particular TEE performed some attestation, or if two distinct attestations map to the same device.

This unlinkability requirement leads to an interesting problem. How can the provider detect a clone if it cannot tell if two actions originated from the same device or two different devices?

To resolve this apparent contradiction, the core idea is that while the provider should not know whether two updates link back to the same device, the provider can make sure that the same update from the same state is never performed twice. However, this seems to lead to a strong performance implication, a new update needs to prove that it is an update that was never done before, with respect to all previous updates of all TEEs. We remark that classical mechanisms such as counters for clone detection trivially fail to satisfy the privacy requirements, as observed for instance by the WebAuthn specification [33, Section 6.1.1].

4 Authorization Chains for PCS

We first describe here the two core mechanisms that we will use for the certificate provisioning:

- a **linkable authorization token chain**, which will gives us PCS in a simple way, but will not provide any privacy;
- a **unlinkable authorization token chain**, which will give us both PCS and privacy by relying on so-called blind signatures.[1]

[1]Note that as is done in [19], Verifiable Oblivious Pseudo-Random Function (VOPRF) can be used instead of blind signatures. However, as the full TokenWeaver solution requires anyway the provisioning of anonymous certificates for third-party servers, we use blind signatures that can be used for both use cases.

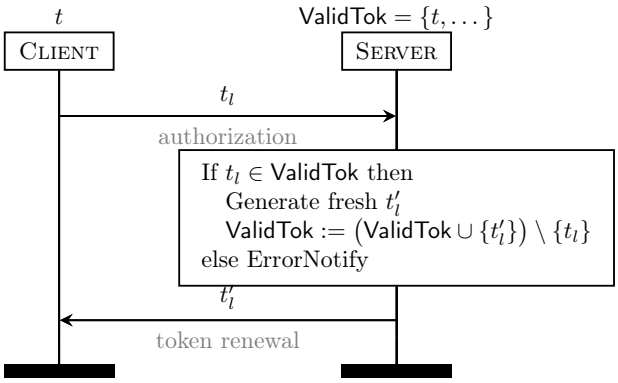

Figure 1: Linkable Authorization Token Chain

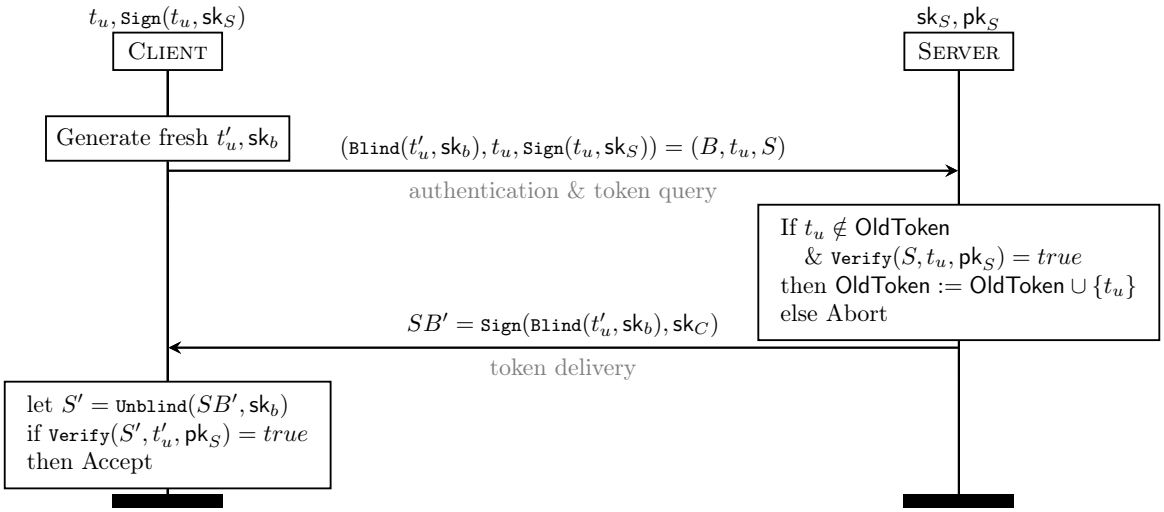

Figure 2: Unlinkable Authorization Token Chain

We first present those two mechanisms independently from the TEE context, as they are generic mechanisms for the following question: *how to perform a recurring authorization from a client to a server with PCS and privacy?* The context is then that we have a server S, and a set of authorized clients C. We assume that C can easily establish a one-sided authenticated channels to S, for instance using TLS. Then, each client C should be able to confirm it is one of the authorized clients, and may have to perform this operation regularly. In the TEE context, the clients will be the TEEs and the server the provider. The security goals will be similar to those in Section 3, namely authentication, PCS and unlinkability.

4.1 Linkable Authorization Token Chain

The simplest way to establish an authorization mechanism would be to deliver to each client an authentication key pair $(\text{pk}_C, \text{sk}_C)$ that each authorized client uses to authenticate to the server. However, after a client compromise, the attacker would obtain the corresponding secrets, enabling the attacker to keep logging in indefinitely.

To design an authorization mechanism that achieves PCS, one possibility is to have the server deliver one-time authorization tokens to clients. Clients should present a valid one-time tokens to perform a single authorization step, and every time the authorization is successful, the server delivers a freshly generated authorization token to the client. This first mechanism requires that the server maintains a set ValidTok of currently valid authorization tokens. Then, the following steps are followed:

- Whenever a new client is registered for the first time, the server generates a fresh token, and adds it to ValidTok and provides it to the client.

- To perform an authorization, a client with current token t_l establishes a TLS connection with the server and sends t_l over the connection.
- The server then verifies that the token is valid, i.e. $t_l \in \mathsf{ValidTok}$, in which case the authorization succeeds. The server then generates a new fresh token t'_l, removes t_l and adds t'_l to $\mathsf{ValidTok}$, and finally sends t'_l back to the client.

We show this simple flow in Fig. 1.

Security Intuition. The crucial observation is that each token can be used only once, and cannot be predicted. When a client is compromised, the attacker learns the current t_l. However, once the client performs a subsequent authorization, t_l is consumed on the server side, and the attacker is effectively locked-out again.

However, this design offers no strong privacy. In fact, each client is linkable by the server: the server can effectively remember the previous token delivered to some client, and then recognize that a new connection comes from the same client when this specific token is presented. This allows reconstructing the full chain of tokens corresponding to a given client.

4.2 Unlinkable Authorization Token Chain

We now consider a client C that wishes to perform the authorization process in a completely unlinkable way while retaining PCS. The core issue in the previous mechanism is that the server knows each authorization token owned by the clients. Our solution is then to make it so that the tokens are not generated by the server but rather by the client. The server will not even learn the token but will only *blindly* sign it with a dedicated signing key. This blind signature can later be verified to check the token validity.

Blind signatures. We use so-called blind signature schemes [11]. They are defined by four algorithms $(\mathtt{Sign}, \mathtt{Verify}, \mathtt{Blind}, \mathtt{Unblind})$, such that for a valid keypair $(\mathsf{sk}_S, \mathsf{pk}_S)$, $(\mathtt{Sign}, \mathtt{Verify})$ is a classical signature scheme, but with the additional feature that for a freshly sampled blinding key sk_b, $\mathtt{Blind}(m, \mathsf{sk}_b)$ reveals no information about m, and even a *malicious server* cannot link $\mathtt{Blind}(m, \mathsf{sk}_b)$ and the unblinded signature $\mathtt{Sign}(m, \mathsf{sk}_s)$, and of course we also have that $\mathtt{Unblind}(\mathtt{Sign}(\mathtt{Blind}(m, \mathsf{sk}_b), \mathsf{sk}_s), \mathsf{sk}_b) = \mathtt{Sign}(m, \mathsf{sk}_s)$.

Setup. We consider that the server S has a signature key pair $(\mathsf{sk}_S, \mathsf{pk}_S)$, and the client C already has a token t_u (a nonce) along with a proof of validity of the token $\mathtt{Sign}(t_u, \mathsf{sk}_S)$. S maintains a set of expired tokens $\mathsf{OldToken}$, initialized with the empty set.

Process description. Assume that C can establish an authenticated channel to S (e.g. via a TLS certificate for S). Then, the authentication and token renewal process is depicted in Fig. 2.

1. C generates a blinding key sk_b and a new secret token t'_u, and send the values $(\mathtt{Blind}(t'_u, \mathsf{sk}_b), t_u, \mathtt{Sign}(t_u, \mathsf{sk}_S))$ to S.
2. S receives (B, t_u, S), checks the validity of the given token by running $\mathtt{Verify}(S, t_u, \mathsf{pk}_S)$ and checking that $t_u \notin \mathsf{OldToken}$. If so, authentication succeeds, the server deprecates the token by updating $\mathsf{OldToken} := \mathsf{OldToken} \cup \{t_u\}$, and sends $\mathtt{Sign}(\mathtt{Blind}(t'_u, \mathsf{sk}_b), \mathsf{sk}_S)$ to the client. If the token was already used, the server notifies the user that it owns a deprecated token.
3. C computes $\mathtt{Unblind}(\mathtt{Sign}(\mathtt{Blind}(t'_u, \mathsf{sk}_b), \mathsf{sk}_S), \mathsf{sk}_b) = S'$, runs $\mathtt{Verify}(S', t'_u, \mathsf{pk}_S)$ and if it is valid, it stores the token t'_u along with its signature S'.

Security Intuition. We present the formal analysis later in Section 6, but intuitively, our solution meets the expected security properties as:

- **Authentication**: The unforgeability of the signature ensures that only people with a valid token can obtain a new token.
- **Privacy**: Thanks to the blinding, the server does not know which token it is delivering to which client. Note that it is *crucial* that the client does check it got a valid signature, as otherwise the server could maliciously deliver a broken signature to detect the later use of this particular token. This check was actually missing from our initial protocol draft, but the flaw and the attack were discovered automatically by our tools when we tried to formally prove it.
- **PCS**: If a client is compromised and the token is stolen, either the client first uses it and the token obtained by the attacker becomes useless, or if the attacker first uses it and then the client tries to use it, the update will fail and the compromise will be detected.

Crucially, we note that the detection of the compromise in this initial design has a significant limitation. If the attacker did use the honest user's token, the honest user update will fail and they could report it to the server. However, even if the compromise is detected, the server can only deliver a new token to the honest user, but it *cannot deprecate* the attacker's token, as due to the blind signatures and the resulting unlinkability, the server has no information about this token. We are thus missing the active revocation capability. In the next section, we show how to solve this remaining issue.

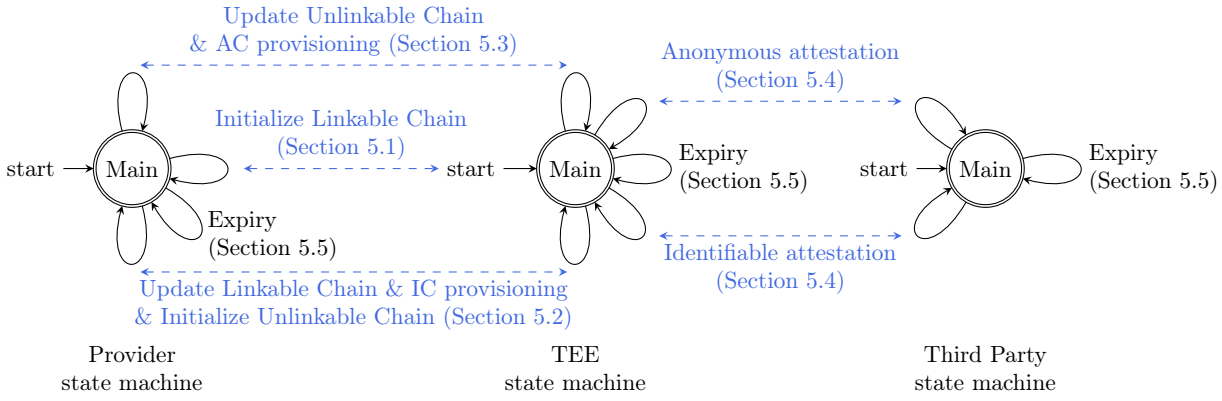

Figure 3: TokenWeaver: high-level state machines and connecting protocols indicated with dashed edges, and in blue. The Provider and the TEEs states evolve together either when 1) a new TEE is created (initialization of the linkable chain), 2) a TEE performs a linkable chain update or 3) a TEE performs an unlinkable chain update. A TEE and a third party state evolve when a TEE performs either an identifiable or anonymous attestation. The only operation where all agents are updated is when the global provider's certificate expires.

5 TokenWeaver

We now define our full solution, dubbed TokenWeaver, that combines the previously presented linkable and unlinkable token chains in order to achieve the full certificate provisioning solution. The only cryptographic dependency are classical and blind signature schemes.

Overview. The linkable chain is used between the provider and each TEE, to track which entities are valid TEEs. This part does not yield privacy, but enables efficient compromise detection and revocation. To then provide privacy, we sprout from each node of this linkable chain a fresh unlinkable chain. The TEE then uses this unlinkable chain between itself and the provider for on-demand provisioning of new ACs for third-party authentication. This ensures quick recovery from compromise as well as privacy.

We summarize the high-level state machines in Fig. 3 and provide the protocol details in the remainder of this section. Additionally, we provide a pseudo-code specification in Appendix A.

External dependencies. In the design of TokenWeaver, we assume that several things are classically deployed based on TEEs. We notably mention the following features:

- We require that each TEE is able to establish a secure channel to its provider, in order to send a single message and receive a single answer such that messages are confidential and integrity protected and the TEE is anonymous. Such a channel can easily be built, for instance by relying on the existing TLS certificates of the Provider. We provide additional details on how to define, build, and use this channel in Appendix B.
- *Optional* – In many scenarios, OoB channels are set-up between providers and end users (e.g., using SMS communications). When our design automatically detects a compromise, such a channel can be used to authenticate the honest user, revoke the current linkable token, and allow provisioning of a new set of secrets to the TEE to establish a new linkable chain. This is only optional, as in some application contexts, it is often enough to detect the compromise.

Cryptographic materials. To ease the presentation of the underlying protocols, we now highlight all materials owned by either the provider or the TEEs and their use, for a summary see Table 2 in Appendix A. We first consider that the provider owns two distinct key pairs:

- **attestation key pair** $\mathsf{sk}_A, \mathsf{pk}_A$: this is the public key trusted by third parties for the attestation aspect.
- **provisioning key pair** $\mathsf{sk}_P, \mathsf{pk}_P$: this is the key pair used by the provider for signing tokens in the unlinkable authorization chain.

Both key pairs should expire and be renewed frequently to ensure that attackers are eventually locked out. The expiration of sk_P and sk_A is our only way of locking out an attacker that compromised some unlinkable token.

The state of a TEE then depends on the following values:

- A SN - a public identifier of the TEE.
- Current valid AC and IC - multiple TEE owned key pairs signed by the current sk_A, that can be

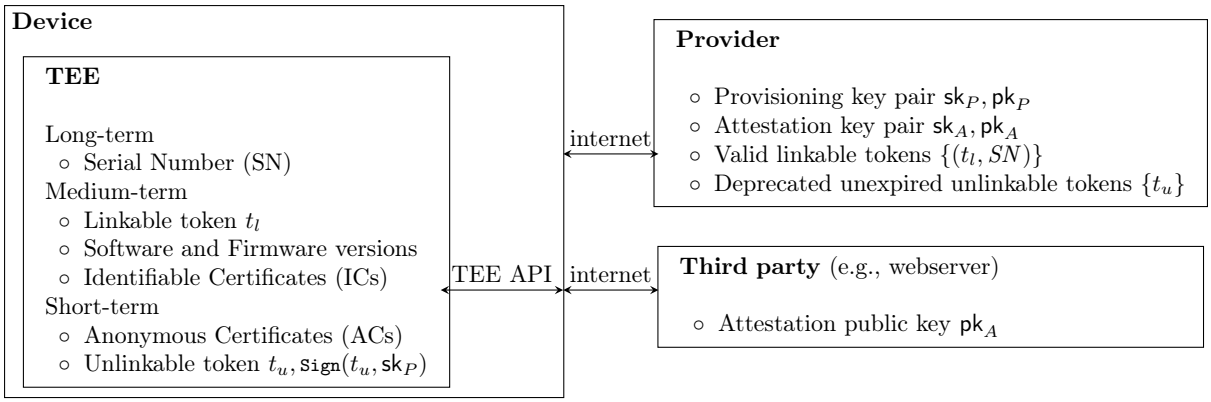

Figure 4: TokenWeaver's TEE setup

used to attest to third parties. AC is fully anonymous, while IC is tied to the SN. They are implicitly deprecated when sk_A is renewed.

- A current unlinkable one-time authorization token $t, \mathtt{Sign}(t_u, \mathsf{sk}_P)$ - when presented to the provider, the token is consumed. In return, the TEE can generate a new token t'_u and ask the provider to blindly sign it with sk_P. In addition, the provider also blindly signs a new key pair for AC generated by the TEE using the latest sk_A.
- A linkable one-time authorization token t_l - This token is used to renew the unlinkable one-time authorization token (e.g., when sk_P is renewed). When t_l is presented to the provider, it is consumed, the provider delivers a new token t'_l, and blindly signs a new token t' generated by the TEE. The provider also signs a new IC key-pair with sk_A.

The full states of the multiple parties can be summarized as in Fig. 4. In practice, t_l is used less often and should be better protected if possible in concrete deployments.

5.1 TEE initialization

Instantiating a new TEE is straightforward as it only requires to initialize the linkable token chain. Note that to be fully functional and obtain valid certificates, the TEE will then need to perform a linkable chain update followed by an unlinkable chain update as described next.

Setup. At the factory, the provider generates and provides a fresh token t_l to the TEE identified by SN, and stores the pair (t_l, SN) inside the set $\mathsf{LinkedToken}$.

5.2 Update Linkable Chain

We describe here the full update of a linkable chain, which leverages the mechanism from Section 4.1 to initialize or re-initialize the TEE's unlinkable chain and additionally deliver a new IC. This step must be performed every time sk_P is deprecated.

Protocol description. To (re-)initialize the unlinkable chain and receive a new IC the following steps are followed:

- the TEE establishes a one-way authenticated encrypted channel to the provider (e.g., a TLS channel based on the provider's PKI certificate);
- the TEE generates a new secret token t'_u, a blinding key sk_b, and a fresh key pair $(\mathsf{sk}_{IC}, \mathsf{pk}_{IC})$. He then sends the values $(\mathtt{Blind}(t'_u, \mathsf{sk}_b), t_l, \mathsf{pk}_{IC})$ to the provider.
- the provider checks that there exists SN such that $(t_l, SN) \in \mathsf{LinkedToken}$, in which case it drops it from the list, generates a fresh t'_l, adds (t'_l, SN) to $\mathsf{LinkedToken}$, and then sends back the values $(\mathtt{Sign}(\mathtt{Blind}(t', \mathsf{sk}_b), \mathsf{sk}_P), t'_l, \mathtt{Sign}((SN, \mathsf{pk}_{IC}), \mathsf{sk}_A))$.
- the TEE computes

$$\mathtt{Unblind}(\mathtt{Sign}(\mathtt{Blind}(t'_u, \mathsf{sk}_b), \mathsf{sk}_P), \mathsf{sk}_b) = S' \ ,$$

and runs $\mathtt{Verify}(S', t'_u, \mathsf{pk}_P)$. If the verification is successful, it stores t'_l as its new linkable token, (t'_u, S') as its current unlinkable token, and $(\mathsf{pk}_{IC}, \mathsf{sk}_{IC}, \mathtt{Sign}((SN, \mathsf{pk}_{IC}), \mathsf{sk}_A)$ as its IC.

If at any time a honest user is locked out because an attacker already used the token t_l, the honest user must use some OoB authentication and send a compromise report to the provider containing its SN. In this case, the provider erases any entry linked to SN inside $\mathsf{LinkedToken}$, thus deprecating the current

linkable token actually used by the attacker, and then should communicate a fresh t_l to the TEE through a secure OoB communication channel to the user.

Note that if a TEE updates its linkable chain multiple times in the lifespan of a single sk_P, it would obtain multiple valid blinded tokens and be able to initiate multiple parallel unlinkable chains. However, they would all still be deprecated simultaneously when the corresponding sk_P expires.

5.3 Update Unlinkable Chain

We now leverage the unlinkable chain mechanism from Section 4.2 to perform the AC provisioning. We assume here that the TEE has a valid unlinkable token $(t_u, \mathtt{Sign}(t_u, \mathsf{sk}_P))$ for the current provisioning key of the provider. Otherwise, it must first run the previous mechanism.

To achieve anonymous attestation, we run the AC provisioning in parallel to the unlinkable chain update. The TEE generate their own attestation key pairs and ask the provider to blindly sign it with the latest sk_A, which yields an AC. As this process can be run multiple times, the TEE can accumulate multiple distinct ACs. Using a different AC for attestation to each third-party server (or each account) preserves the unlinkability of the attestation process.

Protocol description. The AC provisioning is performed in parallel to a valid unlinkable chain update as follows:

- At any time, a TEE may choose to generate a new attestation key pair $(\mathsf{sk}_T, \mathsf{pk}_T)$. It then establishes a secure session with the provider (e.g., over TLS with a certificate for the provider), and then performs a one-time authorization, including the value $\mathtt{Blind}(\mathsf{pk}_T, \mathsf{sk}'_b)$ (with sk'_b a fresh blinding key) in the first message.
- The provider processes the one-time authorization, and if it is valid, includes in its answer $BSA' = \mathtt{Sign}(\mathtt{Blind}(\mathsf{pk}_T, \mathsf{sk}'_b), \mathsf{sk}_A)$.
- The TEE processes the answer and stores the new token. It additionally stores $SA' = \mathtt{Unblind}(BSA', \mathsf{sk}'_b)$, after successfully verifying that SA' is a valid signature.

Every time a TEE runs such a process, it has:

- A one-time anonymous authentication token $(t_u, \mathtt{Sign}(t_u, \mathsf{sk}_P))$ from completing the one-time unlinkable authorization process;
- A new attestation key pair $(\mathsf{sk}_T, \mathsf{pk}_T)$ along with a valid certificate of the public key $\mathtt{Sign}(\mathsf{pk}_T, \mathsf{sk}_A)$.

Due to the blind signature, the provider learns no information about the attestation key pair that it provided a certificate for. Furthermore, the TEE can now attest itself to any third party by using sk_T to sign a challenge and send this signature along with the certificate $\mathtt{Sign}(\mathsf{pk}_T, \mathsf{sk}_A)$.

5.4 Attestation: Certificate Verification

Assuming that a TEE is up to date and has performed a full AC provisioning valid for the current attestation public key pk_A, it should currently own:

- at least two signing secret keys, sk_T and sk_{IC};
- at least two certificates, an anonymous one $\mathtt{Sign}(\mathsf{pk}_T, \mathsf{sk}_A)$ and an identifiable one $\mathtt{Sign}((SN, \mathsf{pk}_{IC}), \mathsf{sk}_A)$.

Such a TEE can then use either sk_T or sk_{IC} to sign any desired data, and send this signature along with the corresponding certificate to a third party. The third party then simply has to validate the certificate chain, verifying the validity of the current public attestation key of the provider pk_A.

A TEE can perform multiple AC provisionings to obtain additional signing keys sk_T. When attesting to services, distinct ACs must be used to ensure unlinkability.

Security Intuition. We provide a formal analysis in Section 6, but on an intuitive level, we inherit the PCS guarantees from the one-time authorization token. However, the concrete privacy we achieve depends on the implementation details of the third party attestation process. Our scheme prevents the provider and third party servers from linking together a particular TEE with any of its ACs or linking together different ACs used by the same TEE. However, in some use cases, the attestation process itself may leak some metadata that can be exploited to link together distinct attestations even when using different ACs (e.g., the device's IP address). As this is application dependent, we consider it to be outside the scope of our work.

5.5 Expiring public keys

The provider needs to manage the renewal of the two public keys pk_A and pk_P. The expiration should be time-based to ensure PCS. Here, any classical technique used to manage public keys can be plugged in, in

a similar fashion to TLS certificate management. For example, a third longer term key could be published by the provider and used to sign the two sub keys along with the expiry date.

6 Formal Analysis

In this section, we describe in-depth the formal analysis of the core elements of TokenWeaver, providing stronger assurance guarantees than any other solution in this domain. The analysis was performed in parallel to the design, its results allowing us to update the design whenever an issue was reported. Our ideal proof goals are ambitious: a complete coverage of all the goals of the full design is currently beyond the reach of any single automated analysis tool. To obtain our strong guarantees, we therefore use a combination of state-of-the-art protocol verification tools.

In particular, we perform three distinct analyses:

- We first prove that in isolation, the unlinkable token chain is indeed unlinkable.
- We then show that the unlinkable token chain in isolation does meet the expected PCS property. We also illustrate how the formal analysis reported an attack on a previous design, and lead us to improve it.
- Finally, for the full AC provisioning mechanism, including both linkable and unlinkable chains, we prove that the linkable chain does meet PCS, and that linkable chain does lock out an attacker from the unlinkable chains.

We were not able to study the unlinkability of the full solution due to what we believe to be limitations of the state-of-the-art tools. The main difficulty is that tools enabling unbounded verification tend to verify a stronger privacy property (diff-equivalence) than what TokenWeaver actually requires and provides.

Reproducibility. All automated formal analyses carried out in this section where performed on a laptop with 16Gb of RAM and a quad-core Intel(R) Core(TM) i7-10510U CPU at 1.80GHz and can easily be inspected or reproduced. We provide the models at [17] along with instructions to download a docker image allowing readers to inspect and reproduce the analyses and proofs.

6.1 Choosing appropriate proof tools

The main mechanisms of our solution occur at the logical level: TokenWeaver can be seen as a complex stateful security protocol that keeps several different layers of state, but also depends on cryptographic primitives. This combination suggests that state-of-the-art symbolic protocol analysis tools may be appropriate, such as Tamarin [35] or Proverif, which can provide proofs for unboundedly many sessions. Because of the type of state machines in the protocol, and because we expected the complexity to be beyond the scope of fully automated proofs, we chose Tamarin.

However, during our analysis, it became clear that while Tamarin was appropriate for proving PCS, neither Tamarin nor Proverif currently provide for the privacy properties we wanted to prove for TokenWeaver. While Tamarin and other tools allow to verify a specific class of privacy properties, they only allow proving a privacy property that is too strong for our setting: for those tools, updating the state of one TEE and not another one lead to a false attack. As we require a fine grained notion of privacy, we thus turn to the DeepSec prover [13]: it has the downside of forcing us to consider only a fixed number of TEEs and possible updates, but it can verify a privacy notion that is fine-grained enough for our setting.

Thus, using this combination of tools, we can prove PCS properties for a complex model with an unbounded of TEEs and updates, and also fine-grained privacy properties for a fixed number.

6.2 PCS Analysis of the Unlinkable Chain

We used Tamarin to prove that the unlinkable chain provides PCS against an attacker that can compromise the tokens of the chain.

Model. We modeled the core unlinkable mechanism as described in Fig. 2. Our model covers an unbounded number of TEEs (client) interacting with the provider (server), and contains the following possible actions:

- **Initialize** - Initialization of a new TEE, which sends a blinded token to the provider and receives the corresponding signature. We assume the communication is done over a trusted channel where the provider can authenticate the TEE, e.g., initialization done at the factory.
- **TEE Query** - A TEE sends its current token as well as a fresh blinded one to the Provider over a secure channel.

- **Provider Answer** - The provider checks the validity of the token, deprecates it, and returned the blind signature on the new one.
- **TEE Process** - The TEE checks the validity of the provided signature, and stores the new token.

There are two possible attacker actions:

- **Compromise** - The attacker can compromise a given TEE and obtain its current token.
- **Attacker Query** - The attacker can contact the provider and try to perform a step of the process with any message it can compute based on its knowledge.

Property. To model security properties such as authentication of PCS, we rely on so called events: whenever a possible action is executed, we add to the execution trace the corresponding event. We can then express security properties over those events inside a temporal logic.

For instance, in our models, we raise the event $\mathsf{Accept}(t_l)$ whenever the provider accepts a token t_l, $\mathsf{Query}(t_l)$ when a TEE sends the token t_l in the query step. For a TEE, processing a new token corresponding to healing, so we raise $\mathsf{Heal}(SN)$ whenever a TEE identified by SN performs the TEE process action and should then have healed, and $\mathsf{Compromise}(SN)$ whenever the attacker compromises the TEE corresponding to SN. Then, the PCS property can intuitively be expressed as follows: If a provider accepts a token at timepoint i then:

- either there exists a TEE that sent it as a query at a previous timepoint $j < i$;
- or the attacker compromised a TEE at a previous timepoint $j < i$, and between i and j the TEE never healed.

This property essentially captures the only possible ways for a token to be accepted. The first case corresponds to a valid token processing, and the second one to an attacker token processing. Note that if the attacker compromise a TEE that heal afterward, then only honest token processing can happen, and the attacker is locked out and cannot interfere with the process anymore. Formally, it then translates to:

$$\begin{array}{l} \forall\ t_l, i.\ \mathsf{Accept}(t_l)@i \Rightarrow \\ \quad (\exists\ j.\ \mathsf{Query}(t_l)@j\ \&\ j < i) \\ \quad \| \left(\begin{array}{l} \exists\ SN, j.\ \mathsf{Compromise}(SN)@j\ \&\ j < i \\ \quad \&\ \neg(\exists\ k.\ \mathsf{Heal}(SN)@k\ \&\ j < k\ \&\ k < i \end{array} \right) \end{array}$$

Proof. To prove such a property, Tamarin works in a so called backward fashion: for our PCS property, it starts from the Accept state, and will try to prove that all possible ways to get to a state that violates the property are impossible. As we consider an unbounded number of updates, the problem is in general undecidable. Tamarin then relies on heuristics to choose an optimal next proof rule out of its set of possible proof rules. In our case, Tamarin's built-in heuristics are not able to automatically find a proof. However, Tamarin has an interactive proof mode, which we can use to guide the proof search, either by adding helper lemmas, or by performing ourselves the proof in the interactive mode.

We formalized such lemmas in Tamarin, and were able to prove the PCS property with a total of 9 lemmas, 6 of them automatically proven by Tamarin and 3 proven by hand for a total of 174 proof steps. Tamarin verifies the corresponding proof files in under 10 seconds.

6.3 PCS Analysis of the AC provisioning

Model. We model the two linkable and unlinkable chains in Tamarin intertwined as described in Section 5. The model specifies the following possible actions:

- **Renew keys** - renew the provider keys sk_P and sk_A.
- **Initialize** - create a fresh TEE instantiated with a valid linkable token.
- **Linkable chain query** - a TEE establishes a secure channels and sends its current linkable token along with a blinded token and a fresh blinded public key pk_T;
- **Linkable chain answer** - the provider checks the validity of the linkable token, in which case it signs the blinded token with the current sk_P and the blinded pk_T, and send it along with a new fresh linkable token.
- **Linkable chain process** - the TEE receives and stores its new linkable and unlinkable tokens after unblinding and verifying the received signatures.
- **Unlinkable chain Query/Answer/Process** - the three possible actions from the previous model (Section 6.2) are then possible, where as long as the TEE has an unlinkable token signed with the current sk_P, it can then obtain a new AC certificate signed with the current sk_A.

The attacker can **Compromise** a TEE, in which case it gets the linkable and unlinkable tokens that it can use to obtain valid certificates for the compromised TEE. The attacker can also send its own unlinkable or linkable queries to the provider and obtain valid certificates for its own TEEs.

Properties. We have previously proved that each unlinkable chain gives us PCS. When combined with the linkable chain, we have to prove that:

- the linkable chain itself provides PCS: if the attacker compromise a linkable token, it can either use it and lock the user out, or the user will use it and the attacker is locked out.
- renewing the keys of the provider then ensure the global PCS: after a key is renewed, the only way for the attacker to continue obtaining ACs is to use a linkable token it compromised.

The first property is similar to the previous one for the unlinkable case. We can in fact be slightly more precise, as the provider knows which TEE it is authenticating. We now raise the event $\mathsf{AcceptL}(t_l, SN)$ whenever the provider accepts a linkable token t_l corresponding to TEE SN, $\mathsf{QueryL}(t_l, SN)$ when the TEE sends the token t_l in the query step, and $\mathsf{HealL}(SN)$ whenever the TEE finishes the final linkable process action and should then be healed. We still raise $\mathsf{Compromise}(SN)$ whenever the attacker compromise the TEE corresponding to SN, where the compromise is common to both the linkable and unlinkable processes.

$$\begin{aligned}\forall\, t_l, SN, i.\ \mathsf{AcceptL}(t_l, SN)@i \Rightarrow \\ (\exists\, j.\ \mathsf{Query}(t_l, SN)@j \ \&\ j < i) \\ \| \left(\begin{array}{l} \exists\, j.\ \mathsf{Compromise}(SN)@j \ \&\ j < i \\ \quad \&\ \neg(\exists\, k.\ \mathsf{HealL}(SN)@k \ \&\ j < k \ \&\ k < i \end{array} \right)\end{aligned}$$

We can show that if the attacker compromises a TEE and then the keys are renewed, the attacker needs to use the linkable token it obtained to keep getting new ACs. The only way for the attacker to keep compromising the system is thus to lock out the honest user, which enables clone detection. If we raise the event Renew whenever the provider keys are renewed, raise the event AcceptUAtt when the attack succeed in performing an unlinkable authorization, and raise the event AcceptLAtt when it succeeds in performing a linkable authorization, we can then formally express the property as:

$$\begin{aligned}\forall\, i, j, k.\ \mathsf{Compromise}(SN)@i \ \&\ \mathsf{Renew}@j \ \& \\ \mathsf{AcceptUAtt}@k \ \&\ i < j < k \\ \Rightarrow \exists\, l.\ \mathsf{AcceptLAtt}@l \ \&\ j < l < k\end{aligned}$$

This property precisely expresses that if a compromise happened at a timepoint i and the keys were renewed afterwards at $j > i$, then if the attacker can still perform an unlinkable step later at $k > j$, it must have in fact performed a linkable step at timepoint l in between j and k. The converse of this property then tells us that if the attacker did not perform the linkable step, it is locked out of the AC provisioning.

Proofs. The intuition of the proof is close to the previous one. On the one hand it is simpler, because there is no cryptography involved inside the linkable process, but on the other hand it is more complex, because the full model is bigger. Overall, we needed 19 lemmas, 2 of them being the target properties, 13 of them proved automatically and 6 of them proved manually in 425 steps. The full model verifies in about 15 seconds.

Encoding assumptions. An implicit assumption in our Tamarin analysis is that the encodings of IC $\mathtt{Sign}((SN, \mathsf{pk}_{IC}), \mathsf{sk}_A)$ and AC $\mathtt{Sign}(\mathsf{pk}_{AC}, \mathsf{sk}_A)$ are disjoint. The underlying reason is that in the natural symbolic encoding of public keys, a pk_{AC} can never be equal to a pair of elements (SN, pk_{IC}). In most implementation, this separation is naturally achieved by the formatting and encoding of public keys and tuples. For implementations that do not ensure this separation, one could either include a label or also use two distinct keys instead of a single sk_A to achieve domain separation.

6.4 Privacy Analysis of the Unlinkable Chain

Our goal here is to verify that TokenWeaver does provide the expected unlinkability of certificate provisioning and attestation actions. While many unlinkability notions have been formalized and equipped with dedicated proof techniques (see e.g. [1, 3, 24]), a core issue is that they all define an unlinkability notion in the context of protocols where a device will be fully unlinkable in all the possible protocol steps. In our context, within an execution of TokenWeaver, a given TEE will be sometimes trivially and as expected linkable (it is of course trivial to link a device during the linkable update, but it is also trivial and expected that one can distinguish two devices that are not in the same epoch), while it should be unlinkable at other times. Essentially, we consider an unlinkability property where all TEEs that are in the same epoch must all be between themselves unlinkable for any attestation or certificate provisioning step.

As such, we are forced to come up with a tailored definition of unlinkability for this use case. In this section, we only provide a high level intuition without formally defining the underlying security properties. We provide further details in Appendix C. To analyze the privacy properties, we use DeepSec, which can

verify that two scenarios are indistinguishable by the attacker. We describe now the two scenarios we compared.

Our DeepSec model. We model two TEEs that, after initialization, can both perform unlinkable and linkable updates. After performing a linkable update, a TEE state should contain a valid unlinkable token pair $(t_u, \mathtt{Sign}(t, \mathsf{sk}_P)$. We then model the fact that it can make a one step authorization, where it authenticates to the provider using t_u and renew its unlinkable token obtaining a new valid t'_u as well as obtaining a new AC.

We consider a threat model where the attacker controls all external elements to the TEE. Crucially, we thus consider that the Provider is malicious and attacker-controlled.

Note that whenever a TEE did not perform the latest linkable update after sk_P expired, it does not have a valid token. It is then easy to distinguish an unlinkable update from a TEE that did the latest linkable update compared to one that did not. We therefore aim to prove that when two TEEs have both made the same number of linkable updates, then a scenario where only a single TEE performs unlinkable updates is indistinguishable from the scenario where the two different TEEs are performing updates.

Using DeepSec, we verified that this property holds, which corresponds to proving that an attacker can not tell if two updates are linked, whenever the two TEEs are both either up to date or outdated with respect to sk_P.

As mentioned, DeepSec only works for a bounded number of protocol sessions, and the proof complexity increases exponentially with respect to the number of sessions. We can prove the unlinkability for 16 total updates (8 linkable and 8 unlinkable ones) in under a minute. Despite the exponential growth, we can push the numbers by using a server with more RAM and parallelization. On a 64 cores CPU at 2.60GHz and with 23Gb of RAM, we were able to go up to 24 total updates (12 linkable updates and 12 unlinkable updates), verified in about 20 hours.

Our formal analysis caught an early design error. An early version of our design did not include verification of the resulting blind signature by the TEE in the last step of the unlinkable chain update. When we tried to prove the unlinkability of this version with DeepSec, we uncovered a simple attack on the design that DeepSec reported in under a second. Without verification, a malicious provider can return an invalid signature. This can be exploited later on to track the TEE, as the provider can spot when it receives an invalid signature, thus violating unlinkability. We subsequently updated our design.

6.5 Summary of provable security guarantees

Based on our formal analysis, we can summarize the security guarantees of our solution as follows.

Privacy. We proved that the unlinkable chain combined with the linkable chain and the AC provisioning ensures unlinkability. There are cases not covered by our analysis that include naturally unavoidable breaches of privacy:

- a TEE uses an IC; or
- a TEE reuses the same AC for two distinct use cases, or use an AC while leaking non-anonymous metadata.

It would be interesting to be able to express a fine-grained unlinkability property over the full system, additionally capturing those behaviors and metadata. It would also be interesting to lift the limitation on the bound on the number of sessions. Both of those points however appear to require further advances in formal methods development.

PCS with compromise of unlinkable token or certificate. This is the most likely compromise, as the unlinkable token and the certificates are used frequently. In this case, our PCS proof of the unlinkable chain shows that if the honest user uses the unlinkable token, the attacker cannot obtain any new certificates. Additionally, the certificates obtained by the attacker will expire when the provider renews the attestation key sk_A. If the attacker first uses its clone of the unlinkable token, the user will detect it when trying to use it. Then, the user could renew its certificate using the linkable token and reveal its compromised IC to the provider. The provider could then, e.g., publish it in a deprecated certificate list.

PCS with compromise of linkable token. This is a deeper compromise, as the linkable token is only used to reset the unlinkable chain. In this case, our PCS proof of the linkable token chain shows that if the user uses the linkable token, it locks the attacker out of the linkable chain. The attacker can still run the unlinkable certificate provisioning as long as the provider keys are not renewed. However, our analysis of the AC provisioning shows that as soon as the provider's keys expire, the only way the attacker can keep getting valid certificates is by running the linkable update. If an attacker does so, it would lock the user out and would be detected. In such a case, the user sends a compromise report to the

TEEs	Unlinked Tokens	Token Buckets	Database Size (GiB)	Linked Operation Time (ms)			Unlinked Operation Time (ms)		
				TEE	Prov. Crypto	Prov. DB	TEE	Prov. Crypto	Prov. DB
10^5	$2 \cdot 10^6$	1	0.167	2	6	0.6	5	8	0.8
10^6	$2 \cdot 10^7$	4	1.684	2	6	0.6	5	8	0.8
10^7	$2 \cdot 10^8$	16	16.97	2	6	0.6	5	8	0.9
10^8	$2 \cdot 10^9$	256	169	2	6	0.6	5	8	1.1

Table 1: Experimental Results — For each number of TEEs n, the number of deprecated unlinked tokens in the database was set to $20n$, and the number of Token Buckets was adjusted accordingly. The database size is the total required for all databases and buckets. The run-times are averaged across 1000 repetitions. For the Provider, the measurements are provided for the cryptographic and database operations separately.

provider, who deprecates any linkable token corresponding to this user's TEE. Then, the provider can use a secure OoB channel to provide a fresh linkable token to the TEE.

7 Proof of Concept Implementation

To advocate for the simplicity and applicability of our solution, we implemented in Python the operations described in Section 5 for the full AC provisioning. We implemented two proof of concept variants — an un-optimized variant to show the simplicity of the solution and an optimized variant to show the efficiency and scalability of our solutions. Both variants are available at [17]. The un-optimized variant takes 42 lines of code for the TEE, 31 for the provider side, and 8 lines for the Third-Party. The core cryptographic dependency is a blind signature scheme, for which we use the reference implementation of the IETF standard for RSA blind signatures [20], which was proven secure in [34]. For the actual attestation, an extra signature is required, for which we use a standard ECDSA signature scheme. A linkable token update round requires one blind signature and verification and an additional standard signature for the IC, An attestation requires one standard signature from the TEE. On the third party side, it requires one blind signature verification and one standard signature verification.

Experimental Evaluation. To ascertain that a protocol implementation could efficiently support a large number of TEEs, we implemented an optimized provider that stores its state in an SQLite database [17]. To minimize overhead when scaling the number of tokens, the deprecated unlinkable tokens are distributed across multiple buckets, which are indexed based on the initial bits of the random token. This approach allows us to query only the relevant bucket when checking if an unlinked token is present in the database, as it cannot appear in the other buckets. We also explored the use of a Bloom Filter as an optimization for queries, but found that due to the high efficiency of the SQLite database the time difference was negligible.

We ran the experiments on an Intel Xeon E5-4669 v4 CPU with 504GB of RAM, running Ubuntu 18.04.3 LTS. We used Python 3.6.9 with PyCryptodome 3.18.0 and sqlite3 3.22.0. We varied the number of TEEs already stored in the database, storing 20 deprecated unlinked tokens for each TEE. The details are outlined in Table 1.

Across all experiments, the average TEE run-time was below 2 milliseconds for a linked operation and below 5 milliseconds for an unlinked operation. Correspondingly, the average provider run-time was below 4 milliseconds for a linked operation and below 9 milliseconds for an unlinked operation. The database queries constituted 10%–15% of the provider's run-time, the rest taken up by cryptographic operations.

As a point of comparison, we benchmarked the time required to perform the main cryptographic operations necessary for establishing a TLS connection from both the Server and Client sides. On the server side, we benchmarked two ECDH exponentiation operations and an RSA-PSS signature. Over 1000 iterations, the average run-time was 6 milliseconds, which means that the run-time complexity of our provider is comparable to a TLS server. For the client side, we benchmarked two ECDH exponentiation operations and an RSA-PSS verification. Over 1000 iterations, the average run-time was 3.4 milliseconds, which means that the run-time complexity of our TEE is actually smaller than a TLS client, comparable to a TLS server. Because TLS clients are commonly implemented and used inside modern TEEs, we claim that our protocol (which is faster and based on similar cryptographic building blocks) is efficient and practical for this application domain.

In terms of space, the provider needs to maintain a list containing a pair (t_l, SN) for each TEE, as well as a list containing all the deprecated unexpired unlinkable tokens, a list which is reset whenever the public key pk_P expires. Even for our largest experiment that stored $2 \cdot 10^9$ deprecated tokens, the total

space requirement is less than 170 GiB which is arguably very small for such applications.

8 Discussion

Out-of-Band Channel.

When TokenWeaver reports a compromise and locks out both the user and the attacker, it is possible to rely on some OoB channel to restore trust. Several banks have already set-up such mechanisms for online payments, with notably 3-D Secure, which either triggers a smartphone application confirmation, or sends a one-time code over SMS. Google also has an account recovery feature, with several questions asked, e.g. about older passwords, or relying on a recovery email account. Any such mechanisms can be fitted for TokenWeaver.

We note that even without the OoB channel, PCS and clone detection already provide additional guarantees, allowing to alert users of compromises. Further, TokenWeaver can motivate deployment of new OoB channels, as they are currently useless in many scenarios without a detection mechanism to trigger them.

Global Attestation Key Variant. In this work we presented a solution where each TEE can obtain a set of multiple ACs, and a single AC should be dedicated to a single use case, as otherwise colluding third party could link multiple attestations to the same TEE. This has the downside of requiring that each TEE manages and keeps track of which account is linked to each AC, and having the TEE request potentially many ACs from the provider.

To enable a fully anonymous attestation, the simplest solution is for the provider to just give sk_A to all TEEs, and each TEE simply uses this secret key to perform attestation. This ensures the privacy as all TEEs share the same attestation key.

In practice, we need to frequently renew the keys to enable healing. We thus tie this idea with the one-time authentication mechanism, where every time the provider renews the keypair, all TEEs perform an authorization upon which the provider sends them the new secret key. We thus consider the case where the provider owns a keypair $\mathsf{sk}_A, \mathsf{pk}_A$ for attestation, and a keypair $\mathsf{sk}_S, \mathsf{pk}_S$ for the authorization part. Then:

- All TEEs are initialized at the factory with a token $t_u, \mathtt{Sign}(t_u, \mathsf{sk}_S)$.
- The provider regularly renews the key pair $\mathsf{sk}_A, \mathsf{pk}_A$.
- When a TEEs requires a new valid key sk_A, it first establish a TLS connection with the provider, and then performs a one-time authorization to the provider as depicted in Fig. 2.
- Whenever the provider accepts a valid token, it additional appends to its reply the value sk_A.

While simpler than TokenWeaver, the global key does not allow to instantly revoke the current ACs of a TEE when a user detects a compromise. Furthermore, TokenWeaver also allows to detect or recover from compromises earlier, as it implies more frequent interactions between the TEE and the provider.

Mitigation of DoS Attack. Malicious devices might attempt a Denial-of-Service (DoS) attack by repeatedly requesting new tokens. However, as we show in Section 7, the computational cost of such a request is relatively low (similar to the cost of a TLS handshake), making it less useful for DoS attacks. Moreover, the provider can add a rate-limiting mechanism for token provisioning. It can insert a time delay between the time a token is provided and when it can be used to request a new one.

Linkable token. For the linkable token t_l, the provider is required to store for each TEE a pair (t_l, SN). Other solutions could be considered, e.g., counters, or chains based on repeated application of a pseudo-random function, combined with zero-knowledge proofs. We leave exploring alternative solutions based on such mechanisms as future work.

Third-Party Authorization.

A third-party can decide to set up their own Authorization Token process with a TEE (which is independent from the one with the provider). The mechanism can for instance be used by a third-party to provide one-time tokens to access a resource in a privacy preserving way, and in parallel ask for an attestation from the TEE.

Version-linked tokens. Tokens could be linked to the firmware, to allow to deprecate all tokens corresponding to some outdated insecure firmware. This would however be redundant with the recurrent expiration of sk_A.

9 Conclusions

We presented TokenWeaver, a solution for privacy-preserving and post-compromise secure attestation. TokenWeaver is based on a combination of both linkable and unlinkable token chains, which may be of independent interest.

Compared to Google's split brain solution, we obtain true privacy, and do not have to trust the provider to uphold an internal split. This is a crucial distinction, since (a) users cannot verify that the split brain policy is actually implemented, and (b) even if implemented, the split may be breached over time. Compared to SGX, we have no need for EPID that are costly, and SGX makes no claims (let alone proofs) of PCS. In contrast to both of these solutions, we offer *provable* PCS and privacy properties. Additionally, we offer fine-grained clone detection, both at the linkable and unlinkable levels.

Acknowledgment

This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006 (PEPR Cybersecurity SVP). Eyal Ronen was supported in part by an ISF grant no. 1807/23, and the Len Blavatnik and the Blavatnik Family Foundation.

References

[1] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *2010 23rd IEEE computer security foundations symposium*. IEEE, 2010.

[2] Arm. Arm TrustZone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[3] David Baelde, Stéphanie Delaune, and Solène Moreau. A method for proving unlinkability of stateful protocols. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020.

[4] David Basin, Sasa Radomirovic, and Michael Schläpfer. A complete characterization of secure human-server communication. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 2015.

[5] Nina Bindel, Nicolas Gama, Sandra Guasch, and Eyal Ronen. To attest or not to attest, this is the question–provable attestation in fido2. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 297–328. Springer, 2023.

[6] Max Bires. Google Android developers blog - Upgrading Android Attestation: Remote Provisioning, March 2022. https://android-developers.googleblog.com/2022/03/upgrading-android-attestation-remote.html (Accessed Jan 2024).

[7] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*, 2022.

[8] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS*. ACM, 2004.

[9] Ernie Brickell and Jiangtao Li. Enhanced privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. *IEEE Transactions on Dependable and Secure Computing*, 2011.

[10] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. In *CCS*. ACM, 2021.

[11] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1983.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[13] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *IEEE S&P*, 2018.

[14] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. Exploiting symmetries when proving equivalence properties for security protocols. In *CCS*. ACM, 2019.

[15] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. Post-Compromise Security. Cryptology ePrint Archive, Paper 2016/221, 2016. https://eprint.iacr.org/2016/221.

[16] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086, 2016. https://eprint.iacr.org/2016/086.

[17] Cas Cremers, Gal Horowitz, Charlie Jacomme, and Eyal Ronen. Formal models for the Tamarin prover and DeepSec and Proof of Concept Python implementation of TokenWeaver, 2024. https://github.com/charlie-j/token-weaver.

[18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *CHES*, 2018.

[19] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3), 2018.

[20] Frank Denis, Frederic Jacobs, and Christopher A. Wood. RSA Blind Signatures. RFC 9474, October 2023.

[21] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[22] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018.

[23] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *IEEE S&P*, 2011.

[24] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for verifying privacy-type properties: the unbounded case. In *IEEE S&P*, 2016.

[25] ISO 15408-2: Common criteria for information technology security evaluation - part 2: Security functional components, July 2009.

[26] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[27] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[28] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[29] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "security-by-crash" based memory isolation in AMD SEV. In *CCS*. ACM, 2021.

[30] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CipherLeaks: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*, 2021.

[31] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. TLB poisoning attacks on AMD secure encrypted virtualization. In *ACSAC*, 2021.

[32] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *IEEE S&P*, 2021.

[33] Emil Lundberg, Akshay Kumar, J.C. Jones, Jeff Hodges, and Michael Jones. Web authentication: An API for accessing public key credentials - level 2. W3C recommendation, W3C, April 2021. https://www.w3.org/TR/2021/REC-webauthn-2-20210408/.

[34] Anna Lysyanskaya. Security Analysis of RSA-BSSA. 2022. https://eprint.iacr.org/2022/895.

[35] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *International conference on Computer Aided Verification (CAV)*. Springer, 2013.

[36] Kevin Milner, Cas Cremers, Jiangshan Yu, and Mark Ryan. Automatically Detecting the Misuse of Secrets: Foundations, Design Principles, and Applications. In *CSF*. IEEE Computer Society, 2017.

[37] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018.

[38] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *IEEE S&P*, 2020.

[39] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies. In *CCS*. ACM, 2019.

[40] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone. In *CCS*. ACM, 2019.

[41] Vinnie Scarlata, Vinnie Johnson, Vinnie Beaney, and Piotr Zmijewski. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*. Intel. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf (Accessed Jan 2024).

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*. ACM, 2019.

[43] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's trustzone keymaster design. In *USENIX Security*, 2022.

[44] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[45] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE S&P*, 2020.

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[47] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SGX.Fail: How secrets get eXtracted. https://sgx.fail, 2022.

[48] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *CCS*. ACM, 2019.

[49] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *IEEE S&P*, 2020.

[50] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. IACR Cryptology ePrint Archive 2016/980, 2016.

A Full Description

In this section, we provide the full description in pseudo code of our proposal. Table 2 provides a summary of the cryptographic material used in our proposal. For each agent, we provide the multiple operations that can be performed. Local variables are written as x, while long term storage is denoted with x. Comments are written in gray. The API provided by the cryptographic library for the signature is shown in Fig. 5, the code of a TEE is in Fig. 6, the one of the provider is in Fig. 7, and the (very small) code of a third party checking an attestation is in Fig. 8.

```
Signature library S:
   Blind part
   (sk,pk) <-$- S.gen()
   blinded, skB <-$- S.blind(pk,m)
   blinded_sig ← S.bsign(sk,blinded)
   sig ← S.unblind(skB,pk,blinded_sig,m)
     return Null if invalid blinded
   Classical part
   sig <-$- S.sig(sk,m)
   S.vrfy(pk,sig) return bool

```

Figure 5: Signature Interface

Name	Description
Serial Number (SN)	Serial number of a TEE
Identifiable Certificate (IC)	Attestation certificate e.g. tied to the SN, used by the TEE to attest itself to third party servers
Anonymous Certificate (AC)	Attestation certificate but fully anonymous
Authorization token	One-time authorization token, where upon each authorization a new token is delivered
Linkable Token	Authorization token owned by the TEE, linked to its SN and valid for the provider
Unlinkable Token	Authorization token owned by the TEE and blindly signed by the provider
Attestation key pair $(\mathsf{sk}_A, \mathsf{pk}_A)$	Key pair trusted by third parties to sign ACs and ICs
Provisioning key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$	Key pair used by the provider to sign Unlinkable Token

Table 2: Cryptographic Material Summary

B Communication Channel models

We here provide more details on how a secure channel suitable for our protocols can be established as well as how we model it in Tamarin. We assume here basic knowledge about KEM, encryption, TLS, and Tamarin models.

Formal Model. Our protocol and our proofs rely on a channel that is both unilaterally authenticated and confidentiality protected (any message sent over it will be delivered as is to the Provider), and unlinkable: two distinct sessions of the same clients are not related at all. We formally model this in Tamarin using classical channel modelings, similar e.g. to the one of [4]. Our confidential channels are however simplified, as there is a single receiver, the provider.

Without going into the details of the Tamarin notation, we can describe how our formal models capture the following possible actions for the communication channel between clients and a single authenticated server:

- A client can sample a fresh identifier id, and secretly transmit once to the Provider a pair (id, m). Such transmission can only be delayed by the attacker.
- The server, if it knows a particular id, can send a single reply to the client.

```
Long term storage:
    pkP: provisioning public key
    pkA: attestation public key
    SN: TEE serial number
    lt: linkable token
    ult: unlinkable token
    skT,pkT: Anonymous Certificate
    signed_pkT: pkT signed with pkA
    skIC,pkIC: Identifiable Cert
    signed_pkIC: pkIC signed with pkA

====================================
TEE Initialize(pkP,pkA,SN,lt)
------------------------------------
  Store all values in memory:
    (pkP,pkA,SN,lt) ← (pkP,pkA,SN,lt)

====================================
Linkable Chain
------------------------------------
 new unlinkable token
 t ←$ Token Space
 blinded, skB ←$ S.blind(pkP,t)
 new identifiable cert
 skIC,pkIC ←$ S.gen()
 send authenticated with linkable token
 send Provider (SN, blinded, lt, pkIC)
 receive (new_lt,bsig,sig_pkIC)
 token_sig ← S.unblind(skB,pkP,bsig,t)
 abort if  token_sig is Null
 abort if not S.vrfy(pkA,sig_pkIC)
 if success, store vars
 ut ← (t,token_sig)
 lt ← new_lt
 signed_pkIC ← sig_pkIC

====================================
Unlinkable Chain
------------------------------------
 new unlinkable token
 t ←$ Token Space
 bt, skB ←$ S.blind(pkP,t)
 new identifiable cert
 nskT,npkT ←$ S.gen()
 bT, skBT ←$ S.blind(pkA,pkT)
 send all authenticated with previous
 blind token
 send Provider (bt, bT, ut)
 receive (bisgt,bsigT)
 token_sig ← S.unblind(skB,pkP,bsigt,t)
 abort if  token_sig is Null
 npkT_sig ← S.unblind(skBT,pkA,bsigT,npkT)
 abort if  npkT_sig is Null
 if success of unblind, store vars
 ut ← (nt,ntoken_sig)
 skT,pkT ← nskT,npkT
 signed_pkT ← npkT_sig

====================================
Anonymous Attest(m)
 sig ← S.sign(skT,m)
 sent to TTP (m,sig,pkT, signed_pkT)

====================================
Identifiable Attest(m)
 sig ← S.sign(skIC,m)
 sent to TTP (m,sig,pkIC, signed_pkIC)

====================================
Refresh keys
 Upon expiry, fetch new pkP,pkA, e.g.,
 relying on provider's TLS cert.
 Drop all expired signatures and run
 linkable chain.
```

Figure 6: TEE specification

- The attacker can chose its own identifier id and also send a message to the server.
- The server answers to the attacker when an attacker identifier was received.

Realization. Such a channel can be for instance established by using the TLS protocol, to avoid managing an additional certificate on the provider side. Then, for each loop of Section 5.2, the TEE starts a fresh TLS handshake with the provider and use the TLS session as a channel. And for each loop of Section 5.3, another fresh TLS handshake is also used. Implementers must take care not to use any session resumption mechanism, which would otherwise compromise trivially unlinkability. If TLS is too heavyweight for the use-case constraints, a simple communication channel can be established as follows:

- the provider has a long term KEM key pair $(\mathsf{pk}, \mathsf{sk})$;
- any client that wants to send a message m to the provider first executes $\mathsf{sk}_t, ec_t \leftarrow \mathtt{Encap}(\mathsf{pk})$, and then sends on the network the pair $(ec_t, \mathtt{Enc}((1, m), \mathsf{sk}_t)$.
- the provider given (ec_t, em) obtains the key $\mathsf{sk}_t \leftarrow \mathtt{Decap}(ec_t, \mathsf{sk})$, and decrypts the message with $\mathtt{Dec}(em, \mathsf{sk}_t)$. It can then process it, and finally answer to the client with message m_2 by sending back $\mathtt{Enc}((2, m_2), \mathsf{sk}_t)$.

Secure Handling of Communication issues and TEE reset. In our formal models and descriptions, we implicitly assume that the connection between the provider and a TEE cannot be lost but only delayed. In practice, we need to support scenarios where the connection with the TEE is lost after the provider sends the new blinded signature but *before* it was received by and stored by the client. If the provider doesn't accept the old token anymore, an honest TEE who didn't get the new blinded signature

```
Long term storage:
    skP,pkP: provisioning key pair
    skA,pkA: attestation key pair
    dlts: deprecated unlinkable tokens
    lts: valid linkable tokens

====================================
ProviderInitialize()
------------------------------------
 skP,pkP <-$- S.gen()
 skA,pkA <-$- S.gen()

====================================
Create TEE(SN)
------------------------------------
 lt <-$- Token Space
 Make TEE execute:
  TEE Initialize(pkP,pkA,SN,lt)
 lts <- lts + (SN,lt)

====================================
Linkable Chain
------------------------------------
 receive from non authenticated client
 receive  (SN, blinded, lt, pkIC)
 abort if (SN,lt) is not in lts
 new_lt <-$- Token Space
 lts <- lts - (SN,lt) + (SN,new_lt)
 bsig <- S.bsign(skP,blinded)
 sig_pkIC <- S.sign(skA,pkIC)
 send (new_lt,bsig,sig_pkIC)

====================================
Unlinkable Chain
------------------------------------
 receive from non authenticated client
 receive (bt, bT, (t,sigt))
 Check validity of blind token
 abort if not Sig.vrfy(pkP,sigt,t)
 Check t not in dlts
 dlts <- dlts + t
 bsigt <-$- S.bsign(skP,bt)
 bsigT <-$- S.bsign(skA,bT)
 send (bsigt,bsigT)

====================================
Key Refresh()
------------------------------------
 Upon expiry, run ProviderInitialize().
 Publish new public keys.
```

Figure 7: Provider specification

```
====================================
Check attest()
------------------------------------
 receive (m,sig,pkIC, signed_pkIC)
 Fetch current attestation
 key pkA from provider.
 abort if not
     S.vrfy(pkA,  signed_pkIC, pkIC)
 abort if not S.vrfy(pkIC, sig,m)
 Accept
```

Figure 8: Third Party

will be locked out. If it accepts the old token, a malicious client may get two or more valid tokens in exchange for one.

Suppose we use the previously proposed KEM based approach for the secure channel implementation. In that case, the solution is trivial as the clients and server just need to replay the message previously sent using the same encryption keys to avoid any message loss.

Another solution if TLS is used is to rely on the session resumption mechanism of TLS (but only within the same linkable or the same unlinkable update). When as the server gets a valid token, it marks the corresponding TLS session as valid and sends and stores its reply. If this TLS session is resumed from a client, it accepts to resend the stored reply.

C Formal details on the privacy proof

In this appendix, while we cannot re-introduce formally all the notions needed to fully explain to a new reader the formal details of our proofs, we present all the required notions and give precise references for them.

Unlinkability properties. A common privacy goal is the so-called unlinkability property, informally defined in [25] as: "Unlinkability aims at ensuring that a user may make multiple uses of a service or resource without others being able to link these uses together." It is a stronger goal than anonymity but

n TEEs	m_l linkable updates	m_u unlinkable updates	Result
2	8	8	ok in 54s, 68MB
2	12	12	ok in 19h29m, 23GB
2	12	24	OOM>500GB in 10m

Table 3: DeepSec analysis results for three parameter sets

difficult to prove and multiple works have tried to define it formally and enable proving it [1, 3, 24]. An important notion in the field is the *trace equivalence* [3, Def. 2] property, which allows to specify that two protocols have exactly the same set of possible executions, and that those executions are indistinguishable by any attacker. We denote $P \approx Q$ the fact that a protocol P is trace-equivalent to a protocol Q. Given a protocol $P(id)$ where id corresponds to the identifier of an agent involved in the protocol, if we denote $\|$ the parallel composition of session, the unlinkability of the agents in the protocol P is expressed as $P(id)\| \dots \|P(id) \approx P(id_1)\| \dots \|P(id_n)$. That is, an attacker cannot distinguish whether it was n times the same agent that was involved in the n sessions of the protocol or n distinct agents. This implies that the attacker cannot link together two sessions of the same user.

Unlinkability definition of TokenWeaver. Our goal is to verify the privacy of our solution, even for a malicious server trying to track the TEEs. As such, we will consider here that the server is fully attacker controlled, and the unlinkability property will only talk about the parts of the protocol executed by the TEEs. A difficulty of specifying the unlinkability property for our protocol is that it has both unlinkable and linkable components. Let us introduce some notations to clarify this. We split our protocol for the TEE side into three component:

- $\mathsf{Init}(SN)$ - A TEE with serial number SN is given a linkable token;
- $\mathsf{UUpdate}(SN)$ - The TEE with id SN tries to perform an unlinkable update, sending out its unlinkable token (if it has one) and a request for a new AC.
- $\mathsf{LUpdate}(SN)$ - The TEE SN sends its linkable token, and should obtain a new one along with a blinded token.

Those three protocol parts all share the common state of the TEE SN, and cannot be interwoven together. A first attempt at defining the unlinkability of our protocol could be, when only considering two TEEs:

$$\begin{gathered}\mathsf{Init}(SN_1)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_1)\\ \|\mathsf{Init}(SN_1)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_1)\\ \approx\\ \mathsf{Init}(SN_1)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_1)\|\\ \mathsf{Init}(SN_2)\|\mathsf{UUpdate}(SN_2)\|\mathsf{LUpdate}(SN_2)\end{gathered}$$

This property is however trivially false, as for instance the sequence $\mathsf{LUpdate}(SN_1).\mathsf{LUpdate}(SN_2)$ is of course distinguishable from the sequence of actions $\mathsf{LUpdate}(SN_1).\mathsf{LUpdate}(SN_1)$. This property does not correspond to a satisfying definition of unlinkability for our setting. The natural next option is to only consider the unlinkability for the parts of the protocol that are indeed unlinkable, as follows:

$$\begin{gathered}\mathsf{Init}(SN_1)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_1)\\ \|\mathsf{Init}(SN_2)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_2)\\ \approx\\ \mathsf{Init}(SN_1)\|\mathsf{UUpdate}(SN_1)\|\mathsf{LUpdate}(SN_1)\|\\ \mathsf{Init}(SN_2)\|\mathsf{UUpdate}(SN_2)\|\mathsf{LUpdate}(SN_2)\end{gathered}$$

While this would work for a stateless protocol, this is still not a valid definition of unlinkability in our setting as it is also trivially false. Indeed, the sequence $\mathsf{LUpdate}(SN_1).\mathsf{UUpdate}(SN_1).\mathsf{UUpdate}(SN_1)$ is possible on the left hand side, but the correspond one on the right hand side, which would be $\mathsf{LUpdate}(SN_1).\mathsf{UUpdate}(SN_1).\mathsf{UUpdate}(SN_2)$, is an impossible execution, as $\mathsf{UUpdate}(SN_2)$ cannot be triggered without a corresponding $\mathsf{LUpdate}(SN_2)$. This in fact matches the intuition that in the real world, only devices that did perform the latest linkable update are unlinkable. Unfortunately, the existing state of the art tools do not allow us to accurately express such a property.

We however go around the issue by restricting the scope of our analysis: we only express unlinkability over a system where we forces all TEEs to perform at the same time the linkable update.

We now denote $\mathsf{LUpdate}(SN_1, \ldots, SN_n)$ the protocol where all the TEEs perform one after another the linkable update, and none of those TEEs launch any other action during this global update. We can then express our unlinkability target, in the specific case of only two TEEs performing a single linkable and unlinkable update, as:

$$\begin{gathered}\mathsf{Init}(SN_1)\|\mathsf{Init}(SN_2)\|\mathsf{LUpdate}(SN_1, SN_2)\\ \|\mathsf{UUpdate}(SN_1)\|\mathsf{UUpdate}(SN_2) \approx\\ \mathsf{Init}(SN_1)\|\mathsf{Init}(SN_2)\|\mathsf{LUpdate}(SN_1, SN_2)\\ \|\mathsf{UUpdate}(SN_1)\|\mathsf{UUpdate}(SN_1)\end{gathered}$$

We then generalize this property to a number of TEEs n, a number of possible linkable updates m_l, and a number of possible unlinkable updates m_u by using the notation $\|_{1\leq i\leq m} P_i$ to denote $P_1\|\ldots\|P_m$ and writing it as:

$$\begin{gathered}\|_{1\leq i\leq n}\mathsf{Init}(SN_i)\|_{1\leq j\leq m_l}\mathsf{LUpdate}(SN_1, \ldots, SN_n)\\ \|_{1\leq k\leq m_i}\mathsf{UUpdate}(SN_k) \approx\\ \|_{1\leq i\leq n}\mathsf{Init}(SN_i)\|_{1\leq j\leq m_l}\mathsf{LUpdate}(SN_1, \ldots, SN_n)\\ \|_{1\leq k\leq m_i}\mathsf{UUpdate}(SN_1)\end{gathered}$$

Our unlinkability property then captures the fact that for n TEEs that all performed the same number of linkable updates, an attacker cannot distinguish whether it is always the same TEE or always a distinct one which is doing an unlinkable update.

Actual Verification. Because our unlinkability property is not as simple as the classical one and involves non unlinkable components, we cannot reuse solutions dedicated to a particular specification of unlinkability such as [3].

We thus have to resort to general tools that support the verification of trace equivalence. In this work, we rely on one of the latest development in this field, the DeepSec prover [13, 14], which allows verifying trace equivalence between two given protocols. While DeepSec is a state-of-the-art tool, it puts two restrictions on our analysis. First, it does not have built-in support for states, and we must use private communication channels to emulate them. However, while the content of messages sent over private channels is unknown to the attacker, the attacker can observe that a communication occurred over a channel. Thus, if we were to encode each TEE state as one private channel, the attacker would instantly distinguish $\mathsf{UUpdate}(SN_1)\|\mathsf{UUpdate}(SN_2)$ and $\mathsf{UUpdate}(SN_1)\|\mathsf{UUpdate}(SN_1)$, as in the first case a single private channel is used, and in the second two are used. We solve this by using a single private channel to encode the joint states of all TEEs as a tuple, and a given protocol will only touch the part of the tuple corresponding to the given TEE. This however makes it so that our protocol model is not at all modular in the number of different TEEs n in the proof, and we thus only restrict it to 2 instances. Second, DeepSec only enables verification for a bounded number of sessions, and the complexity of the verification grows exponentially with the number of sessions.

We summarize our results in Table 3, with some timeouts and out of memory occurrences to give an idea of the limits.