ZEKRO: Zero-Knowledge Proof of Integrity Conformance

ABSTRACT

In the race towards next-generation systems of systems, the adoption of edge and cloud computing is escalating to deliver the underpinning end-to-end services. To safeguard the increasing attack landscape, remote attestation lets a verifier reason about the state of an untrusted remote prover. However, for most schemes, verifiability is only established under the *omniscient and trusted verifier* assumption where a verifier knows the prover's trusted states and the prover must reveal evidence about its current state. This assumption severely challenges upscaling, inherently limits eligible verifiers, and naturally prohibits adoption in public-facing security-critical networks. To meet current zero trust paradigms, we propose a general ZEro-Knowledge pRoof of cOnformance (ZEKRO) scheme, which considers mutually distrusting participants and enables a prover to convince an *untrusted* verifier about the correctness of its state in zero-knowledge by ensuring that the prover cannot cheat.

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Privacy-preserving protocols**; **Software security engineering**.

KEYWORDS

Zero-Knowledge Configuration Integrity Verification, Configuration Privacy, Trusted Computing, Secure Zero-Touch Configuration

ACM Reference Format:

. 2018. ZEKRO: Zero-Knowledge Proof of Integrity Conformance. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

To make cloud computing services more resilient to increasing integrity concerns and enable detection of malicious or vulnerable software (e.g., Apache Log4j [13]), several proposals [1, 10, 11, 17, 22] have advocated leveraging trusted computing technology. This technology relies on a Trusted Execution Environment (TEE) or more commonly a Trusted Platform Module (TPM) [20] deployed on each node which acts as a trust anchor to store and report integrity evidence about the node's configuration. When a node's software stack boots up, it chronologically measures (hashes) each software component and extends each measurement into a Platform Configuration Register (PCR) of its TPM. To report the node's configuration integrity, the TPM uses a unique key to sign the aggregated PCR value, which a remote verifier can then compare against a list of trusted reference values to determine whether it corresponds to a trusted state. For enhanced integrity guarantees, we can also have nodes continue measuring their software beyond the boot process, e.g., using the Integrity Measurement Architecture (IMA) [16] or Policy-reduced IMA (PRIMA) [7].

Despite their benefits, most existing remote attestation protocols suffer scalability issues since the complexity of the verifier grows with the complexity of the prover's platform configuration [12]. Specifically, the verifier must continuously maintain an extensive allowlist of trusted reference values for each prover, which becomes prohibitively impractical for large networks. Furthermore, besides inherent scalability issues, it is easy to prove that remote attestation protocols that require the prover to disclose its configuration state to verifiers fail under the honest-but-curious adversarial model [14] since any dishonest verifier can easily link and identify the software executing on the prover. The failure to respect a prover's configuration privacy is known to raise serious privacy issues, such as discrimination [15] and can even foster dedicated software attacks against a vulnerable prover [23]. While catering to a platform's configuration privacy is essential from a security perspective, omitting the exchange of such information can also help promote services to mix more seamlessly in multi-domain coordinated services, including multi-vendor environments [3], where contractual differences would otherwise potentially prohibit such collaboration.

To remediate the privacy issue and simultaneously reduce the verifier complexity in remote attestation, some proposals [1, 2, 5, 10, 11, 22, 23] offer different, more privacy-respecting mechanisms for a verifier to reason about a prover's correctness. However, these schemes tend to either require an intermediate trusted third party (TTP) between the prover and verifier to distill or hide the integrity report from the verifier [1, 22], which limits network design and overall responsiveness, assume that the prover's configuration is translatable into abstract properties which the verifier is knowledgeable enough to interpret [2, 5], restrict who can verify a particular prover [10, 23], incur high network overhead [9], or consider only the load time measurements of a prover's software stack [11].

This paper presents ZEKRO, a zero-knowledge proof of conformance scheme that uses trusted computing abstractions to overcome the barriers of configuration privacy and scalability. These abstractions provide another building block for constructing scalable services that seamlessly mix in multi-domain environments and are more resilient to integrity concerns. Our design includes two crucial main innovations to overcome the limitations of existing TPM-based privacy-respecting remote attestation protocols.

First, the ZEKRO scheme provides the trusted computing abstraction, called *policy-restricted attestation key*, that restricts a node's attestation key (secured in its TPM) to policies chosen by an authorizing entity (e.g., a domain orchestrator) and ensures that the node can only use the key to sign challenges if its configuration satisfies a policy. By predicating its ability to sign on its configuration correctness, we can verify its conformance using a simple challenge-response protocol that neither requires nor reveals any configuration information. Moreover, to update a node's "trusted

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

configuration state" during runtime (e.g., in response to patches), the authorizing entity can reactively or proactively authorize new policies restricting the node's attestation key to a new configuration state. Second, to control which of the already authorized policies a node can satisfy during attestation, we propose creating policies that additionally require explicit, time-limited authorization, called *leases*, to be satisfiable, which allows an authorizing entity to control which policy can temporarily be satisfied.

To demonstrate ZEKRO's performance, we evaluated a proof-of-concept implementation on real hardware. Further, to ensure reproducibility and verifiability of our results, we make our prototype publicly available at https://github.com/anonavailability/ZEKRO.

2 RELATED WORKS

When measuring a node's software during runtime, one idea, which IMA [16] employs, is to record measurements both into a TPM PCR and a Measurement Log (ML). Due to the unpredictable order in which components are loaded, the ML helps verifiers verify the reported aggregated PCR value and whether each loaded component is trusted based on reference values. However, such information disclosure raises concern, especially if measurements belong to different tenants sharing a platform. To make it more privacy-respecting, Container-IMA [10] proposed generating unique secrets to obscure ML entries related to each tenant's software, thus effectively restricting verification of a particular tenant's software to verifiers that know the secret, in addition to the trusted reference values.

Similarly, to restrict verifiability only to "correct" verifiers, authors of [23] propose obfuscating a PCR by recording random values into the PCR and ML, which ensures that only a legitimate verifier can perform the verification and others learn nothing. However, the problems of dishonest verifiers and verifier complexity remain.

To avoid disclosing the low-level *binary* measurements (digests) of the platform configurations and reduce the verifier complexity, Property-Based Attestation (PBA) [5, 15], maps platform configurations to more semantical security requirements, called "properties", which a prover can prove to fulfill without disclosing its concrete configuration. However, bridging the semantic gap between digests and properties, i.e., identifying what maps to which properties, is not trivial and must be agreed upon beforehand. The same applies to [2], where, differently to PBA, they propose grouping sets of software and hardware versions into single digests using cryptographic functions such as chameleon hashes [8], or group signatures [4].

To unload verifiers, authors of [1] propose having an intermediate third party between a prover and verifier who is trusted to verify the prover's measurements and vouch for the prover's integrity. Similarly, the work in [22] proposes an attestation proxy to mediate attestation requests between the prover and verifier and translate the prover's concrete configuration into properties that are returned to the verifier. However, while effective, such approaches limit network flexibility in practice and incur overhead.

Like [1] in that no information is disclosed, but eliminating the intermediate third party, the work in [11] proposed having a trusted third party only involved initially to instruct a node in sealing (i.e., encrypting) a secret signing key to its trusted configuration state using its TPM. This way, the node can only ever unseal (decrypt) the secret key if its configuration has not changed. Thus, verifiers, who know a node's public key, can verify its integrity simply by requesting it to sign a challenge using its secret key. However, there are several problems with the sealing operation. First, the unsealed secret key is exposed to software during attestation. Second, as pointed out in [15], any configuration update (e.g., patches) would effectively prohibit unsealing of the data (i.e., secret key).

To prevent exposing the signing key to software, the authors of [9] proposed instead to create attestation keys, which is a special type of asymmetric keys created inside a TPM, where the secret part that is used for signing never leaves the protective shielding of the TPM. Then, similar to [11], these attestation keys are created under the supervision of a trusted third party with an authorization policy that constrains the use of the secret part to a particular configuration state. However, due to the brittleness of the authorization policy, each node must create a new attestation key with a new authorization policy whenever its configuration changes, resulting in a high computational overhead. Furthermore, the approach does not consider software adversaries that might attempt to block update requests to lock a node in a "trusted state".

To solve these problems, we propose instead creating attestation keys with "flexible" authorization policies that allow a named authority (e.g., a domain orchestrator) to approve new policies that restrict the same key's use to new configuration states. Before we explain our scheme, we first explain the necessary background on how this functionality works in Section 3.1 and also how we make nodes more resilient against software adversaries in Section 3.2.

3 TRUSTED COMPUTING CONCEPTS

This section presents the background necessary to understand the proposed approach by describing the leveraged functionalities.

3.1 Enhanced Authorization

The TPM can be used as a combination lock for securing access to TPM objects, such as cryptographic keys. To specify under which circumstances an object can be accessed (i.e., what should be fulfilled before access is granted) and what operations are permitted once access is granted, we must first create a policy statement that logically describes all of our conditions and the scope of the authorization. To ensure statement interpretability, we must use the available Enhanced Authorization (EA) TPM commands, which are documented in part 3 of TCG's specification [20]. For example, to allow the use of an object only if a PCR has a specific value, we can use the *PolicyPCR* command to reference which PCR should have what value. We then translate our policy statement into an "policy digest", which is computed by aggregating a digest over each EA command's Command Code (CC) and command-specific arguments (like an onion) as detailed in the documentation [20]. With a policy digest, we can ask the TPM to create an object (e.g., an attestation key) with this policy digest as its authorization policy.

To satisfy our object's authorization policy, we must start a "policy session" with the TPM, to which it will associate a policy digest internally. The rules are simple. To satisfy our object's authorization policy, we must invoke the correct combination of EA commands with the correct arguments to make the session's internal policy digest match our object's. Anytime we invoke the TPM to perform some action using our object, e.g., to sign a message if the object is

an attestation key, the TPM will first check if the current session's policy digest matches the authorization policy of the object. For some EA commands, e.g., those that restrict the use of an object only to a specific command and arguments, another session-specific digest called the command parameter hash *cpHash* is also used and checked by the TPM before allowing an action to be performed.

3.1.1 Flexible Policy. This paper aims to use the EA functionality to create an attestation key whose authorization policy is constrained to a node's correct configuration as measured into some PCR. Unfortunately, this is not possible with the *PolicyPCR* command since it only allows one state. While we could create a policy statement to permit several possible states by logically *or*ing several *PolicyPCR* commands, this requires knowing all future "good" states, making it inherently impractical. Fortunately, there is a workaround to this "brittleness" problem using the *PolicyAuthorize* command, which allows creating a "flexible" policy owned by a secret key whose public key is associated with the policy. In other words, whoever owns the corresponding secret key of the public key that we commit to in the policy digest has complete control to sign (approve) policies during runtime that, when satisfied in a TPM session, will also cause the object's authorization policy to be satisfied. Suppose we only have the *PolicyAuthorize* command as part of our policy statement. In that case, the resulting "flexible" policy digest pol is computed as $pol \leftarrow \mathsf{H}(\mathsf{H}(CC_{PolicyAuthorize} \parallel \mathsf{pk}) \parallel ref)$, where pk is the public key, and ref is an optional reference that restricts the authorization policy. For example, we use each node's unique identifier as the corresponding reference value to distinguish between the authorization policy of different nodes in a domain.

Once we have a flexible policy, we can have the node create an attestation key with a flexible authorization policy, thus allowing the respective orchestrator to approve different restrictions to use the key, e.g., depending on its currently accepted configuration.

To approve a policy, the orchestrator first creates the policy digest to approve $aPol$ and then signs an authorization hash $aHash \leftarrow \mathsf{H}(aPol \parallel ref)$ using its secret key, where ref references the node.

Finally, to use the approved policy to satisfy the attestation key's authorization policy, the node first satisfies $aPol$ in a policy session and then calls *PolicyAuthorize* with $aPol$, a public key pk, its identifier ref, and a proof that $\mathsf{H}(aPol \parallel ref)$ was signed by the owner of pk, which proves whether the current session's policy digest was approved. Then, if it holds, the TPM replaces the session's policy digest with $\mathsf{H}(\mathsf{H}(CC_{PolicyAuthorize} \parallel \mathsf{pk}) \parallel ref)$, which permits the use of the attestation key if it matches its authorization policy.

3.2 Trustworthy Runtime Measurements

While the TPM provides secure storage and reporting, the entity that stores measurements into the TPM should be trustworthy. For example, to create a chain of trust of the integrity of a platform's boot sequence, we could have each component, such as firmware and boot drivers, first measure the next component into a TPM PCR before passing control to that component. However, if we cannot trust the code that measures the first component, called the core root of trust for measurements, we cannot trust the PCR aggregate.

The most prominent method of extending such a chain of trust into the operating system is IMA [16]. When IMA is used, it hooks onto file-related system calls to remeasure a file (part of the trusted computing base) into an ML and a PCR whenever the file is accessed. To only remeasure a file when it is modified, the filesystem must support i_version and, if needed (e.g., EXT3, EXT4), be mounted with this option. When enforced, the filesystem updates the i_version field of the inode associated with a file when a file is modified.

As before, if we cannot trust IMA, we cannot trust its measurements. Similarly, we must also rely on a trusted entity to measure a node's configuration. This trusted entity must be isolated and immutable. Note, however, that the choice of isolation, e.g., OS-based process isolation, user/kernel-level isolation, or hypervisor-based approaches, will depend on use-case-specific requirements. We assume some Trusted Execution Environment (TEE) for this paper, like ARM TrustZone or Intel SGX. However, note that since we consider remote TEE invocation, it raises two problems. First, requests can be blocked by an adversary to evade detection. Second, the adversary can spoof measurements. To prevent both attacks, we must ensure that the measurements are authentic and approved policies are only temporarily satisfiable. One way to achieve the first requirement, which we utilize, is to create a TPM object inside the TPM's non-volatile (NV) memory space of type PCR, which allows us to associate an authorization policy to our PCR object as described in Section 3.1 that allows the node's TEE to authenticate the measurements. Our solution to the second requirement is to enforce a *leasing* mechanism, which we describe in Section 5.4.2.

3.3 Zero-Touch Enrollment

An attestation key (AK) is especially beneficial due to its inherent restriction. Whereas unrestricted keys, when created inside a TPM, can be used to sign any data, an AK, which is restricted, will not sign any externally provided data structure that appears to be valid and TPM produced but is not. Thus, if an AK is known to be protected by a TPM, it may be relied on to report that TPM's contents accurately.

In the context of zero-touch provisioning of a remote platform equipped with a TPM, we must first verify the authenticity of the TPM [18, 19, 21] and all other primitives that enable the subsequent attestation. The core in zero-touch provisioning is the correct creation of the attestation key to secure the integrity of the attestation process. For discrete TPMs, the manufacturer generally installs an Endorsement Key (EK) and an associated certificate inside the TPM, which allows verifying the EK's authenticity and can further be used to create a Local Attestation Key (LAK). However, the EK differs from an AK. An EK is a "storage key" used to protect (encrypt) the secret key of other keys to allow safe storage outside the TPM, and creating a LAK based on the EK requires some work [19].

Another method is to install an Initial Attestation Key (IAK) and associated certificate, which can be used to create the LAK. Specifically, by utilizing the inherent characteristics of attestation keys, we can, after verifying the IAK's certificate [19], verify that a LAK is created in the same TPM by certifying it using the IAK.

4 SYSTEM AND THREAT MODEL

Before we delve into details of the ZEKRO scheme, we present the considered setting and assumptions regarding protocol participants.

4.1 System Model and Security Assumptions

We consider a network setting with three types of entities:

(1) *Prover* is an untrusted node equipped with a secure TPM provisioned with a certified Initial Attestation Key (IAK) and a secure element for providing secure runtime measurements. We consider a TEE with a certified key pair for brevity for the rest of the paper. Finally, to detect file modifications, we assume a trusted filesystem that enforces i_version.
(2) *Verifier* is an untrusted node that knows the orchestrator of a prover node and wants to remotely check the correctness of the prover's configuration (though not limited to this role).
(3) *Orchestrator* is a trusted entity that: (i) onboards each node by verifying its initial key certificates, (ii) performs the zero-touch configuration of each node's LAK, and (iii) maintains and approves each node's acceptable configuration. We assume that the orchestrator has already done the onboarding (i), which lets us instead focus on more relevant challenges.

4.2 Threat Model

We consider a software adversary who, on some prover, exploits a software vulnerability that allows it to modify that node's critical configurations that are part of the Trusted Computing Base (TCB) as determined by the domain orchestrator. The adversary's goal is to remain undetected and may even attempt to disrupt the node's communication with the orchestrator to do so. Further, we assume that verifiers are dishonest and attempt to infer information about the prover's configuration. However, we assume that verifiers do not collude with the prover's adversary to obtain prover information.

4.3 Objectives

Our scheme's objectives are threefold: (i) any tampering of a prover node's TCB configurations or continued disobedience is detectable, (ii) verifiers require no knowledge except the trusted key certificate of a prover node's orchestrator to verify the prover, and (iii) verifiers learn no information from the verification process besides the correctness of the prover's configuration integrity.

5 THE ZEKRO SCHEME

We continue with the terminology used in our description of the ZEKRO scheme. Then we proceed to give an overview of the scheme before diving into its protocols with explicit reference to the required TPM commands found in part 3 of TCG's specification [20].

5.1 Notation

As an accompanying reference while reading the protocol diagrams, we consider the following symbols and simplified TPM terminology:

$\mathsf{H}(m)$	Compute m's digest using collision-resistant hash function H.
$\mathsf{Sign}(k, m)$	Compute a cryptographic signature over m using k.
$\mathfrak{T}(@conf)$	Tracer, which, given a path, returns a tuple $(c, ino, iver)$ with the contents c, inode number ino, and inode version $iver$.
$\mathsf{Vf}(expr)$	Verification of $expr$, which interrupts if the evaluation fails.
PCR	Platform Configuration Register, which is an extend-only structure internal to the TPM: $PCR \leftarrow \mathsf{H}(PCR \parallel someVal)$
$mPCR$	Mock PCR, which mimics a PCR and is used by the orchestrator and provers to maintain the expected value of the NV PCR that is used for recording measurements (needed for attestation).
CID	Configuration identifier associated with a $mPCR$ on the orchestrator to reference a prover's current configuration version.
ID	A unique (prover) node identifier.
$\mathfrak{H}$	Handle, which references an internally loaded TPM object.
$tmpl$	Template for a TPM object that describes its type and attributes.
CC_{cmd}	Command Code of the TPM command cmd.
exp	Expiration time in some unit of time.
ref	Policy reference used in policies to differentiate authorizations.
pol	Policy digest as described in Section 3.1, which, for our purposes, is a chain of computations: $pol \leftarrow \mathsf{H}(\mathsf{H}(pol \parallel CC_{cmd} \parallel name) \parallel ref)$, where $name$ denotes a TPM object's (e.g., a key) name, which is generally a digest of its public area.
$cpHash$	Command parameter hash, which is computed from the parameters of a TPM command as described in part 1 of TCG's specification [20] and, for our purposes, is computed as: $cpHash \leftarrow \mathsf{H}(CC_{cmd} \parallel params)$, where $params$ refers to the command-specific parameters. With a $cpHash$, an authorizing entity can restrict policies to a specific command and arguments.
$aHash$	Authorization hash as described in part 3 of TCG's specification [20], which for the: $PolicyAuthorize$ command has the form $aHash \leftarrow \mathsf{H}(aPol \parallel ref)$ to enable an authorizing entity to dynamically approve a policy $aPol$ to use the object, and for the $PolicySigned$ command has the form $aHash \leftarrow \mathsf{H}(n \parallel exp \parallel cpHash \parallel ref)$ to enable signed authorization for $expirable$ execution of $cpHash$ in a TPM session whose nonce is n.
tkt	Ticket, which the TPM computes for a specific command and its arguments that later proves to the same TPM that it has already performed the necessary (signature) verification.
{opk, osk}	The orchestrator's asymmetric keypair.
{tpk, tsk}	A TEE's certified asymmetric keypair.
AK	Attestation Key, a restricted signing key that can sign internal TPM structures and has a public part $AKpub$ and private part $AKpriv$, where $AKpriv$ never leaves a TPM unencrypted.
IAK	Initial AK, which is initially created and certified inside a TPM.
LAK	Local AK, which is remotely certified to reside inside a node's TPM by the local (domain) orchestrator.
SK	Storage Key, which is a restricted decryption key used as a parent to wrap (encrypt) the private part of descendant keys, thus ensuring child key secrecy when stored outside the TPM.

5.2 High-Level Overview

Fig. 1 shows the conceptual work-flow of the ZEKRO scheme considering the different entities described in Section 4.1. Our scheme aims to facilitate zero-touch enrollment and configuration of deployed nodes that execute privacy-critical services such that anyone can efficiently verify the correctness of the service in zero-knowledge, thus enabling stitching of privacy-respecting services. Note that, for generality, our scheme succeeds the initial onboarding of the nodes, where a domain orchestrator verifies a node's TPM and TEE certificates and (possibly) verifies that its filesystem enforces i_version. The scheme comprises three protocols: (i) prover enrollment, (ii) configuration update, and (iii) oblivious remote attestation. Let us start by clarifying the idea of each protocol while regarding Fig. 1.

5.2.1 Enrollment. To enable a deployed node to become a "publicly" verifiable prover, it must first be securely enrolled by a certified orchestrator that will then be responsible for continuously determining the node's correct state, whose truthfulness is asserted to verifiers in zero-knowledge. For this, the orchestrator configures the node's TPM with two objects: (i) a local attestation key (LAK) created with a flexible authorization policy with the orchestrator as its authorizing entity (step 1), and (ii) a non-volatile (NV)-based PCR created with an authorization policy that binds the node's TEE

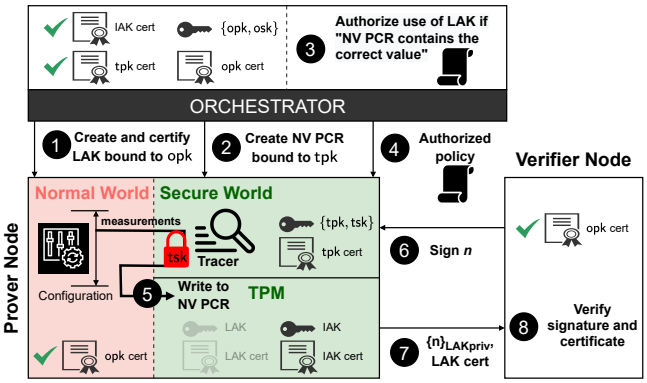

Figure 1: System model and conceptual work-flow after the orchestrator has verified a prover's TPM and TEE keys.

as its authorizing entity (step 2). The former assures that only the orchestrator may approve policies that permit the use of the LAK. Similarly, the latter ensures that only the TEE may approve policies that permit modification of the NV PCR. Together, the TPM objects enable the orchestrator to continuously and securely predicate the use of the LAK to the node's currently trusted configuration state by approving policies that require the NV PCR to contain the currently expected (trusted) aggregate value, which is securely maintained by the node's TEE. Finally, although omitted from Fig. 1, the node must initially report the unique number and current version of the inodes assigned to its configuration files to allow the orchestrator to include inode information in approved policies to ensure detection of unauthorized file modifications. Note that if the initially reported information is incorrect, the prover cannot satisfy approved policies since the TEE's measurements would cause the NV PCR to have a different value since the TEE always uses inode information currently associated with the files.

5.2.2 Configuration Update. Once a prover is successfully enrolled, its orchestrator's responsibility is to approve policies that restrict the prover's use of its LAK, which can occur either routinely or upon demand, e.g., due to a newly released patch. To approve policies for a specific prover node, the orchestrator keeps a trusted reference copy of that node's currently correct configuration, including the associated inode information, and a mock PCR, which it uses to deterministically compute the *expected* value of the TEE's NV PCR once it has remeasured the node's configuration (step 3).

Upon receiving a new approved policy for its LAK (step 4), the prover invokes a trusted application running inside the Secure World of its TEE, which: (i) securely measures the requested configuration, (ii) authorizes a one-use policy for extending the measurement into its NV PCR, and (iii) returns the measurement and corresponding authorized policy to the prover's Normal World, where the prover uses the authorization to extend the measurement into the NV PCR on behalf of the TEE (step 5). Note that we elaborate on the motivation behind outsourcing the extension of the NV PCR to the prover's Normal World in Section 5.3.2. Furthermore, to protect against an adversary blocking the orchestrator's request to measure the node's configuration from reaching the node's TEE in an attempt to evade detection, we describe in Section 5.4.4 the utilization of an accompanying leasing mechanism that enables the orchestrator to grant the node only temporary ability to satisfy a selected approved policy.

5.2.3 Attestation. The final, oblivious remote attestation protocol is executed solely between a prover and a verifier, where the verifier represents anyone that, trusting the orchestrator, wishes to determine the correctness of the prover. Like any remote attestation protocol, the verifier initiates the execution of the protocol by challenging the prover with a fresh nonce (step 6). Then, due to the nature of the LAK object and its strong dependency on the TEE's NV PCR, the verifier knows that if the prover can correctly present a signature over the nonce using a LAK that was certified by the orchestrator (steps 7 and 8), then this serves as irrefutable evidence that it satisfies whichever policy that the orchestrator approved. Thus, without knowing any of the prover's configuration details or what is executing on the prover, and without requiring the prover to disclose any information, the verifier is convinced, in zero-knowledge, that the prover's configuration is correct. Conversely, if the prover cannot supply such a signature, then the verifier can reasonably assume that the prover cannot satisfy the orchestrator's policy. Note, however, that the freshness of the prover's assertion is directly correlated to the orchestrator's frequency of approving new policies, i.e., a higher update frequency leads to faster detection.

5.3 Prover Enrollment

While the creation of a LAK and an NV PCR are both subsumed under the initial enrollment protocol as described in Section 5.2.1, we here separate them for clarity since the creation of each object requires different operations that are unique to that specific object.

5.3.1 Secure Local Attestation Key Creation. Fig. 2 shows how the orchestrator verifies that a node has correctly created a LAK in the same TPM as the pre-provisioned IAK that was verified during the node's initial onboarding, as follows. First, to prepare an authorization policy for the LAK which ensures that the orchestrator is the object's only authorizing entity, the orchestrator composes a flexible policy digest which: (i) binds the orchestrator as the object's authorizing entity, and (ii) includes a reference to the specific node's unique identifier (*ID*) which allows the orchestrator, who potentially orchestrates several nodes, to distinguish between authorizations. Then, to inform the node's TPM about how it should create the LAK, including the LAK's attributes and authorization policy, the orchestrator sends a generic LAK template together with the prepared authorization policy to the prover, who passes these values as arguments in a call to its TPM to create the LAK. However, because the TPM's storage capacity is severely scarce, it cannot be used to store several keys persistently. Thus, to protect the LAK's private part when it is stored external to the TPM, it is created as a child of some storage key *SK*, where the purpose of *SK* is to wrap (encrypt) the LAK's private part for safe storage outside the TPM.

To prove that the LAK was created correctly, the prover loads it back into its TPM, where the same *SK* is used internally to decrypt its private part, and a handle referencing the loaded key is returned that enables cryptographic operations targeting the LAK. Then, to prove that the LAK resides in the same TPM as the IAK, the IAK is used to sign a TPM-generated certificate that includes all creation

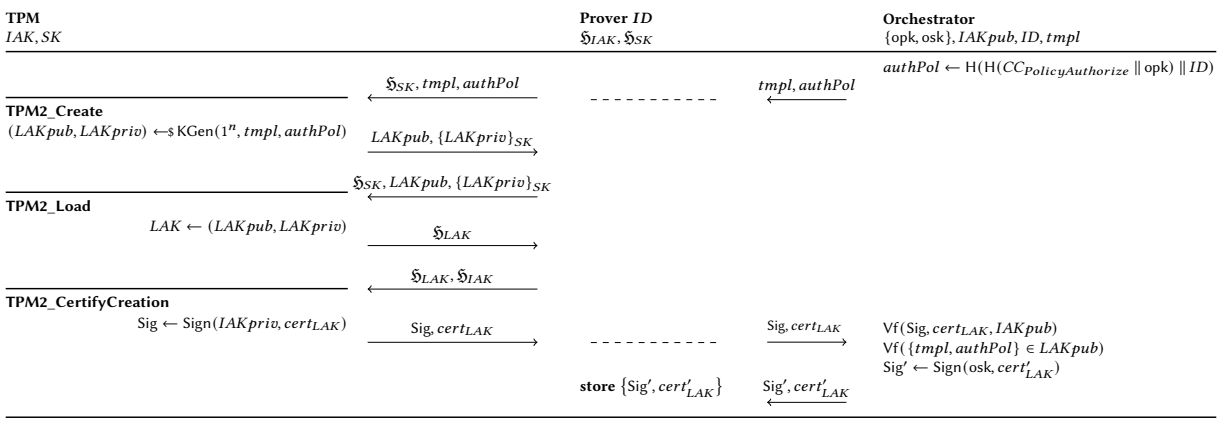

Figure 2: Creation of a LAK with a flexible authorization policy based on an IAK.

details of the LAK. The certificate and signature are then sent back to the orchestrator for validation and assurance that the key has the correct characteristics. In particular, the orchestrator verifies: that the LAK was created following the correct template and was generated using a solid and white-listed cryptographic algorithm, that the proper cryptographic method is used to secure its private part, that it is attributed as a restricted and non-migratable signing key, and that it has a flexible authorization policy bound to the orchestrator's public key. Finally, if everything holds, the orchestrator signs a certificate for the LAK using its secret key, which it gives to the prover node to show to verifiers to prove its LAK's validity.

5.3.2 Secure NV PCR Creation. Like the creation of the LAK, Fig. 3 shows the creation of the NV PCR, where, instead of the orchestrator, the node's TEE is now appointed as the object's authorizing entity to ensure authentic measurements. The orchestrator begins by preparing an object template that instructs the node's TPM when defining the NV object's characteristics, e.g., that it should behave like a PCR. This object template, together with an authorization policy requiring signed authorization from the node's TEE to modify (extend or delete) the object, an index to reference the created NV object, and an initial value *iv* that should be initially extended into the NV PCR, are then sent to the prover node. Given these values, the prover first calls its TPM to create the NV PCR. Then, to extend the initial value into the created NV PCR, which requires authorization from the TEE, the prover starts a TPM session and then passes that session's nonce together with the initial value and NV PCR index to a trusted application executing inside the Secure World of its TEE. To authorize the prover to extend the NV PCR only once and only with the correct value, the trusted application composes a command parameter hash $cpHash$ to restrict its authorization to a single TPM command and arguments, namely the NV_extend command with the initial value and NV PCR index as arguments. Then, to provide signed authorization for the $cpHash$, the trusted application signs an authorization hash $aHash$ over the $cpHash$ and the prover's session nonce using its secret key that restricts the authorization only to the currently active session.

Note that if the trusted application could communicate directly with the TPM (or we had chosen another isolation mechanism), we would not necessarily need the command parameter hash. However, it is impractical to communicate with the TPM directly from within a trusted application due to their inherently small codebase and limited APIs, and it would require a large chunk of the TPM software stack (TSS) to manage an entire TPM session. Therefore, we instead have the TEE authorize permission to extend the measurement into the NV PCR, which can be outsourced to the Normal World since it is nonreusable and cannot be used to extend incorrect values.

Given signed authorization from the TEE, the prover runs the *PolicySigned* command with the TEE's signature, its active session nonce, *cpHash*, and a handle to the TEE's public key. Assuming the authorization was correctly signed, the *PolicySigned* command updates the session's policy digest and sets the session's command parameter hash to the authorized *cpHash*, thus restricting which command the prover can execute. Then, to extend the TEE's NV PCR, the prover runs *NV_Extend*, where, assuming the session's policy digest matches the NV PCR's authorization policy and the command arguments match the session's command parameter hash, the TPM will update the NV PCR with the provided initial value. Afterward, to keep track of the value of the NV PCR, the prover records the extension in its local mock PCR. Finally, similar to the LAK creation process described in Section 5.3.1, the prover proves that the NV PCR was created correctly by certifying it using its IAK, which the orchestrator verifies by inspecting the signed certificate.

5.4 Configuration Update

Securely equipped with a LAK and an NV PCR, Fig. 4 shows how the orchestrator can reliably approve policies that permit the prover to use its LAK only if its current configuration is correct by predicating policies on the authentic contents of the TEE's NV PCR, as follows.

5.4.1 Calculating the Golden Hash. Let us assume that the orchestrator wants to allow the prover to use its LAK only if its current configuration is correct. To simplify the discussion, let us narrow the scope of the considered configuration to a single configuration file, whose unique filesystem location on the prover is $@conf$. To create such a policy, the orchestrator must first compute the expected (trusted) value of the prover TEE's NV PCR after it has been extended with the correct configuration measurement. To compute the expected value, the orchestrator computes a hash over

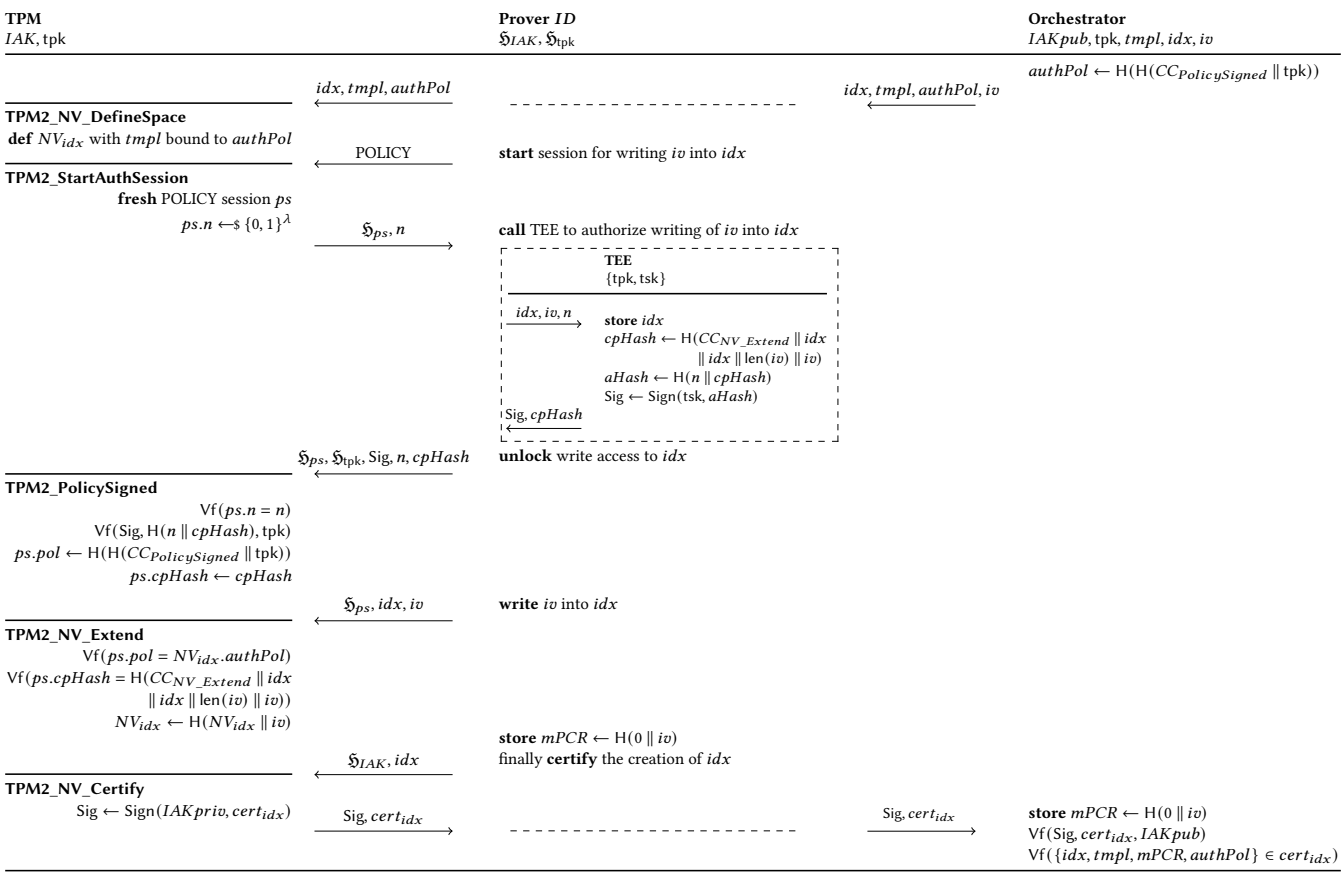

Figure 3: Creation of a NV PCR controlled by the prover TEE's public key.

the contents of the currently correct version of the prover's configuration file, denoted $conf$, including its known location $@conf$, inode number ino, and current inode version $iver$, which it extends into its local mock PCR. Note that by including (committing to) the file location, we effectively prevent any adversary from feeding the prover's TEE with a spoofed path since the TEE's measurement would differ. Similarly, the inclusion of the inode number and version ensure that any unexpected deletion or modification of the configuration file is detected. Finally, the orchestrator computes a unique configuration identifier CID as a hash over the current value of the mock PCR and the node's identifier (ID).

5.4.2 Approving the Policy. To approve a policy for the node's LAK that is unique to the current configuration identifier CID and requires that the TEE's NV PCR contains the currently expected value, the orchestrator creates an approval policy $aPol$ that has two conditions, namely that the node presents: (a) signed authorization from the orchestrator with explicit mention of CID and (b) that the TEE's NV PCR contains the expected value. To authorize the approved policy, the orchestrator signs an authorization hash $aHash$ over the approved policy and a reference to the node's ID, such that the approved policy will work (match) only with the authorization policy of the specific node's LAK. Note here that the purpose of the first part (a) of the approved policy is to allow the orchestrator to enforce a leasing mechanism on its approved policies. For example, to permit the node to satisfy an approved policy $aPol$ whose part (a) references some CID, the orchestrator *must* sign an authorization hash that references CID. Furthermore, by including an expiration in the authorization hash, the node's TPM will revoke the authorization after a set time, thus requiring the node to request a new "lease" from the orchestrator to continue satisfying the approved policy. Finally, because the orchestrator controls which CID it authorizes and presumably chooses the most recent, we effectively castrate any adversary attempting to lock any prover in a "good" state.

5.4.3 Configuration Remeasurement. When a new policy has been approved, the orchestrator sends it together with the location of the measured configuration file(s) and the approved policy's signed authorization hash to the prover node. The node then uses its TPM to verify the signature with a handle to the orchestrator's public key. Since signature verification is expensive, and the node must prove the approved policy's authenticity every time it attempts to satisfy it to use its LAK, the TPM outputs a ticket if the signature verification was successful, where the ticket is supplied as evidence to the TPM that it has already verified the approved policy's authenticity.

Then, to remeasure its configuration (or part thereof), the prover starts a new TPM session and passes the session's nonce together with the configuration location in a call to the trusted application executing inside the Secure World of its TEE. Then, using its accurate tracer mechanism, the trusted application: (i) retrieves the

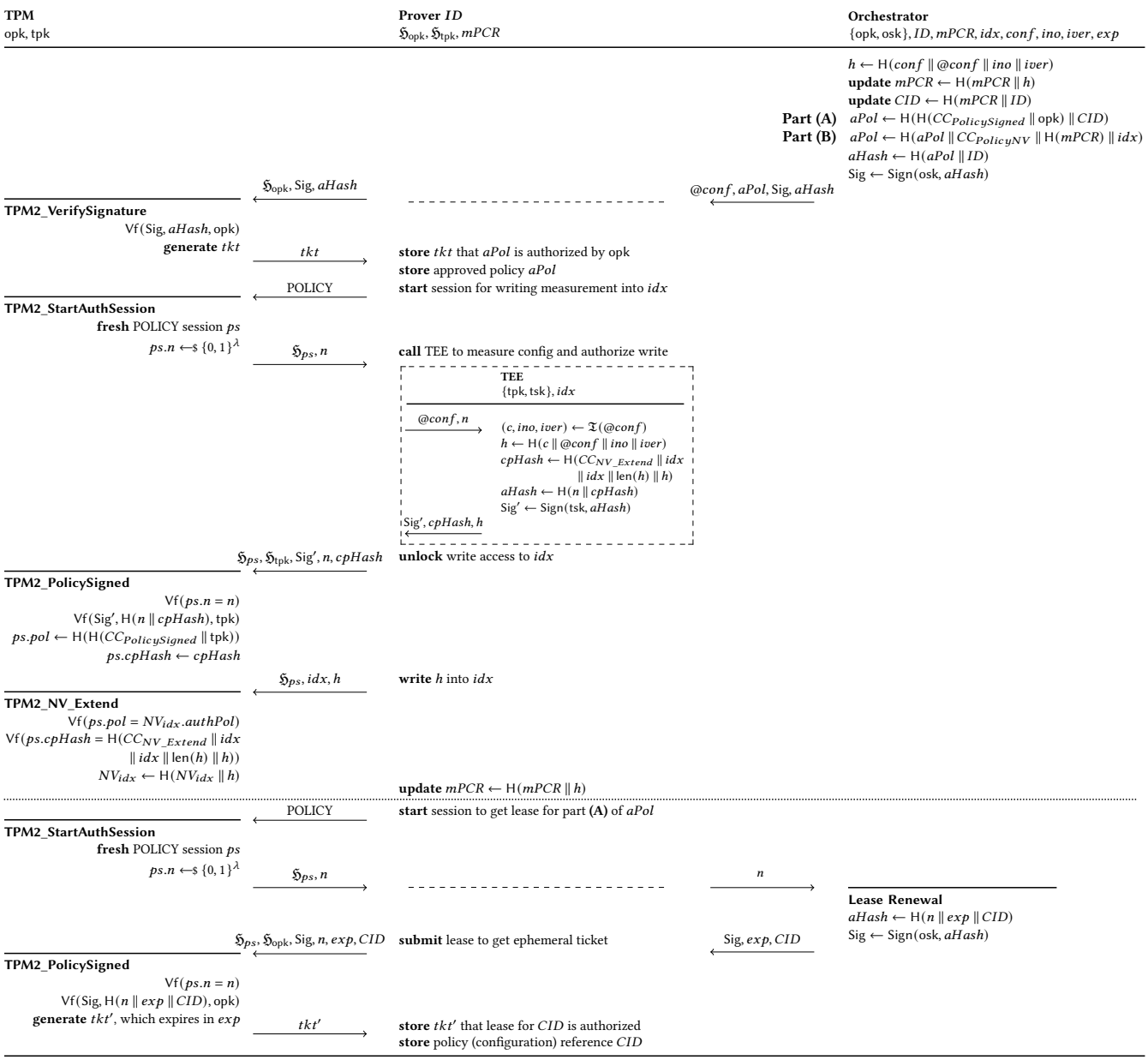

Figure 4: To enforce a new configuration state on a prover, the orchestrator approves a policy for the prover's LAK that is (i) restricted to the new state and (ii) requires time-limited authorization to use.

content and inode information about the requested configuration file and (ii) computes a hash (measurement) over the file's contents, its filesystem location, and the associated inode number and version. Finally, similarly to how it authorized extending an initial value into its NV PCR as described in Section 5.3.2, the trusted application creates an authorization hash to authorize a one-time policy to extend the measurement into its NV PCR, which the prover subsequently uses to update the NV PCR and also its mock PCR.

5.4.4 Leasing. Finally, depending on the update frequency and the considered expiration time for leases, the prover must repeatedly request new leases from the orchestrator to continue satisfying part (a) of the currently approved policy. To get a new lease, the prover must start a new TPM session and send the session's nonce to the orchestrator, who then signs an authorization hash that includes the session nonce, some expiration time, and the current configuration identifier *CID*. These values together with the signed authorization hash are then returned to the prover who, in the same TPM session, must pass them in a call to *PolicySigned*, where, if everything holds, a self-expiring ticket will be returned that proves to the TPM that the orchestrator signed these specific arguments, which can be used to temporarily satisfy part (a) of the currently approved policy.

5.5 Oblivious Remote Attestation

When challenging a prover with a nonce, Fig. 5 shows how the prover attempts to satisfy an approved policy *aPol* to use its LAK to sign the nonce. To satisfy *aPol*, the prover starts a new policy session, over which it runs (i) *PolicyTicket* with a ticket proving that it has signed authorization from the orchestrator, and (ii) *PolicyNV* with the index of its TEE's NV PCR and the expected value, which it maintains in its mock PCR. If the ticket is correct and has not expired, the TPM updates the session's policy digest with a value corresponding to part (a) of *aPol*. Similarly, if the expected value matches the contents in (ii), the TPM updates the session's policy digest with a value corresponding to part (b) of *aPol*. Finally, assuming that the session's policy digest matches *aPol*, the prover can run *PolicyAuthorize* with its *ID*, *aPol*, the orchestrator's public key, and a ticket proving *aPol* was signed by the orchestrator. If everything holds, the TPM replaces the session's policy digest with the "flexible policy digest", which specifies the orchestrator's public key as the authorizing authority and references the node's *ID*. Assuming the session's policy digest matches the LAK's authorization policy, the prover can sign the nonce using its LAK and subsequently send the signature and its LAK certificate back to the verifier in a single pass, where, if everything holds, the verifier is convinced that the prover is currently in a correct state.

6 EMPIRICAL PERFORMANCE EVALUATION

6.1 Implementation and Experimental Setup

We implemented our protocols described in Section 5 in C++ with IBM's TPM Software Stack (TSS) v1.6.0 [6] and OpenSSL v1.1.1i, compiled using the GNU GCC compiler. We considered only elliptic curve (EC) keys and used SHA256 exclusively as our hashing function H. To determine our scheme's performance when considering nodes equipped with either a discrete (hardware) or software TPM, we executed all of our protocols on two platforms: (**P1**) a platform with a 3.6 GHz AMD Ryzen 7 3700X processor running the IBM's software TPM v1637 [6], and (**P2**) a Raspberry Pi 4 Model B platform with a 1.5 GHz ARM Cortex-A72 processor running the Raspbian (buster) operating system equipped with a TPM 2.0 compliant OPTIGA hardware TPM SLB9670.

6.2 Performance Benchmarks

Our performance results are summarized in Table 1, where, for each subprotocol, we show: how long it takes to complete the protocol (first row) and how much time it takes for the completion of each of the TPM commands when executed against either a hardware or software TPM (next rows). Note that the timings are produced using C++11's chrono library's system clock, and each timing statistic also includes the time spent by the TSS to perform the necessary processing of our commands and its internal session management. Furthermore, in the case of the hardware TPM, each timing statistic also includes any Low Pin Count (LPC) bus delay.

Note that the orchestrator's verification of the created Local Attestation Key (LAK) and the created NV PCR, and the verifier's verification of the signed nonce are all omitted since these verifications do not necessarily require interaction with the TPM and are near-instant operations, i.e., taking an average of $\approx$ 0.5 ms on the first platform and 2.4 ms on average on the second platform, respectively. While it is clear from the empirical results that a hardware TPM, whose focus is solely on security, is a bottleneck for efficiency, note that it can provide security guarantees against stronger adversaries than a software TPM.

Table 1: Performance of the protocols over 50 iterations. The table shows the average time (and standard deviation, σ) to run each of the subprotocols at the prover when considering a software TPM (P1) and a hardware TPM (P2).

Subprotocol	Avg. ms (P1)	Avg. ms (P2)
LAK creation	6.86 ($\sigma \approx 0.39$)	405.93 ($\sigma \approx 1.74$)
TPM2_Create	2.76 ($\sigma \approx 0.43$)	202.97 ($\sigma \approx 0.81$)
TPM2_Load	2.98 ($\sigma \approx 0.42$)	56.61 ($\sigma \approx 1.79$)
TPM2_CertifyCreation	1.12 ($\sigma \approx 0.33$)	146.35 ($\sigma \approx 2.29$)
Attaching a NV PCR	10.96 ($\sigma \approx 0.49$)	379.68 ($\sigma \approx 0.84$)
TPM2_NV_DefineSpace	2.52 ($\sigma \approx 0.50$)	26.67 ($\sigma \approx 0.81$)
TPM2_StartAuthSession	1.52 ($\sigma \approx 0.52$)	31.65 ($\sigma \approx 0.63$)
TPM2_PolicySigned	0.90 ($\sigma \approx 0.30$)	163.50 ($\sigma \approx 0.82$)
TPM2_NV_Extend	4.82 ($\sigma \approx 0.65$)	82.68 ($\sigma \approx 1.21$)
TPM2_NV_Certify	1.20 ($\sigma \approx 0.40$)	75.18 ($\sigma \approx 0.61$)
Measurement update	10.59 ($\sigma \approx 0.45$)	589.10 ($\sigma \approx 0.83$)
TPM2_VerifySignature	0.93 ($\sigma \approx 0.26$)	116.12 ($\sigma \approx 0.71$)
TPM2_StartAuthSession (X2)	3.04 ($\sigma \approx 0.52$)	63.30 ($\sigma \approx 0.63$)
TPM2_PolicySigned (X2)	1.80 ($\sigma \approx 0.30$)	327.00 ($\sigma \approx 0.82$)
TPM2_NV_Extend	4.82 ($\sigma \approx 0.65$)	82.68 ($\sigma \approx 1.21$)
ORA	9.35 ($\sigma \approx 3.05$)	335.00 ($\sigma \approx 0.85$)
TPM2_StartAuthSession	1.52 ($\sigma \approx 0.52$)	31.65 ($\sigma \approx 0.63$)
TPM2_PolicyTicket	1.63 ($\sigma \approx 0.46$)	43.40 ($\sigma \approx 0.73$)
TPM2_PolicyNV	0.24 ($\sigma \approx 0.43$)	61.96 ($\sigma \approx 0.63$)
TPM2_PolicyAuthorize	0.18 ($\sigma \approx 0.38$)	69.13 ($\sigma \approx 0.58$)
TPM2_Sign	5.78 ($\sigma \approx 6.77$)	129.86 ($\sigma \approx 1.39$)

Regarding our protocols, note that the first two protocols, i.e., the protocols for creating a LAK and the protocol for creating an NV PCR, need only be executed once for each node due to the flexibility of their authorization policies; thus, their performance is negligible. After that, the most time-consuming protocol is the configuration update protocol executed between a domain orchestrator and a node. However, note that the reported timings include the time for (i) verifying the orchestrator's newly approved policy, (ii) extending the node's measurements into the TEE's NV PCR, and (iii) using a lease from the orchestrator to get a ticket to satisfy the first part of the newly approved policy as shown in Fig. 4. If we only consider the time to get a new ticket for a new lease, which only requires one *StartAuthSession* command and one *PolicySigned* command, then the time is only $\approx$ 2.42 ms using the software TPM and $\approx$ 195.15 ms using the hardware TPM. Finally, the ORA protocol, which nodes execute among themselves, takes a prover < 0.4 seconds to complete on a hardware TPM and $\approx$ 10 ms with a software TPM.

7 SECURITY PROPERTIES

Unforgeable Configuration Integrity. Our scheme guarantees, under the assumptions outlined in Section 4.1, that any unauthorized modifications of a node's configuration files that are considered as part of its Trusted Computing Base (TCB) are detected immediately once the latest issued lease has expired. Specifically,

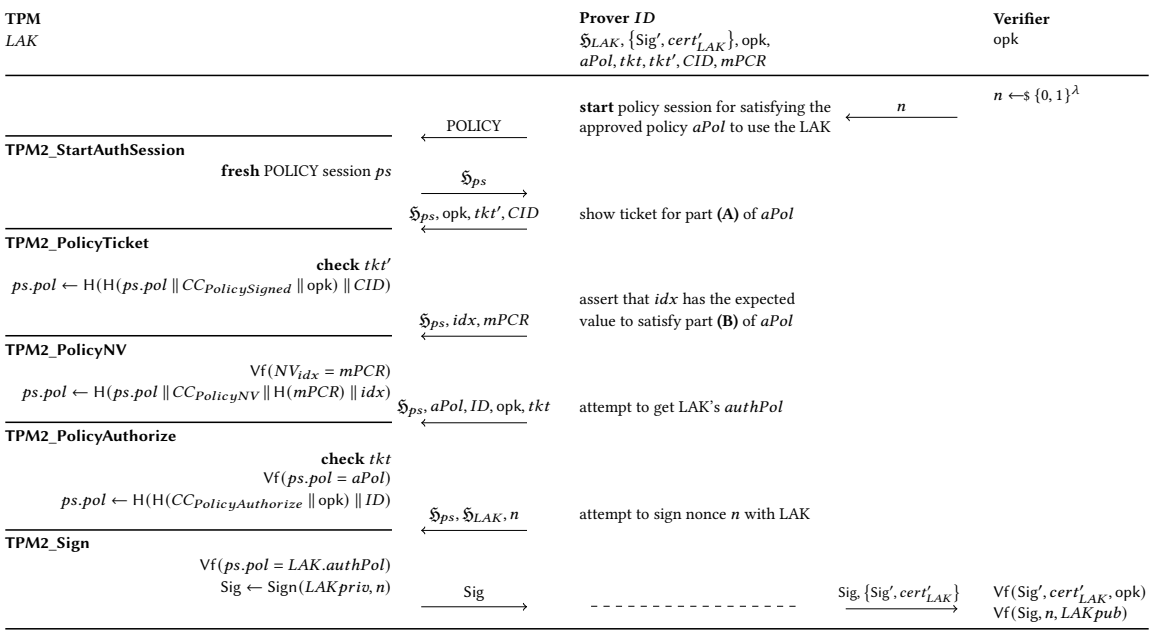

Figure 5: The Oblivious Remote Attestation protocol, where a verifier makes initial contact to a prover and challenges it to prove its configuration integrity by signing a nonce using a LAK certified by the prover's orchestrator.

since the TPM will revoke the signed authorization upon expiration, the node must request a new lease to use its LAK, and that lease will require the node's current configuration to be correct.

Secure Enrollment. Our scheme ensures controlled LAK enrollment, where a prover node's LAK is guaranteed to reside in the same, authentic TPM as the certified Initial Attestation Key (IAK).

Measurement Authenticity. Our scheme ensures that all configuration measurements recorded in a TEE's NV PCR are authentic.

Implicit Revocation. Our scheme ensures that any node has its ability to use its LAK implicitly revoked immediately after its latest lease expires unless its orchestrator decides to keep it alive.

Verification is zero-knowledge. Most importantly, our scheme ensures that a verifier, who trusts a prover node's orchestrator, can determine that node's correctness without any knowledge about the node's configuration, and our oblivious attestation protocol guarantees that absolutely no information can be inferred from the attestation process about the prover node's configuration.

8 CONCLUSIONS

We presented ZEKRO, a novel, scalable, efficient, and effective orchestration scheme that utilizes state-of-the-art trusted computing technologies to facilitate the secure orchestration of a multitude of nodes over multi-domain networks while allowing nodes to partake in privacy-preserving remote attestation activities to determine the configuration correctness of each other.

REFERENCES

[1] Tamleek Ali et al. 2010. Scalable, privacy-preserving remote attestation in and through federated identity management frameworks. In *ICISA*. IEEE, 1–8.

[2] Sami Alsouri, Özgür Dagdelen, and Stefan Katzenbeisser. 2010. Group-based attestation: Enhancing privacy and management in remote attestation. In *International Conference on Trust and Trustworthy Computing*. Springer, 63–77.

[3] Alcardo Alex Barakabitze, Arslan Ahmad, Rashid Mijumbi, and Andrew Hines. 2020. 5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges. *Computer Networks* 167 (2020), 106984.

[4] David Chaum and Eugène van Heyst. 1991. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 257–265.

[5] Liqun Chen et al. 2008. Property-based attestation without a trusted third party. In *International Conference on Information Security*. Springer, 31–46.

[6] Ken Goldman. 2022. IBM's Software TPM and TSS. Retrieved February 24, 2022 from sourceforge.net/projects/ibmswtpm2,sourceforge.net/projects/ibmtpm20tss

[7] Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT*. 19–28.

[8] Hugo Krawczyk and Tal Rabin. 1998. Chameleon hashing and signatures. (1998).

[9] Benjamin Larsen, Heini Bergsson Debes, and Thanassis Giannetsos. 2020. Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments. In *ESORICS*. Springer, 197–220.

[10] Wu Luo et al. 2019. Container-IMA: a privacy-preserving integrity measurement architecture for containers. In {*RAID*}. 487–500.

[11] Wu Luo, Wei Liu, Yang Luo, Anbang Ruan, Qingni Shen, and Zhonghai Wu. 2016. Partial attestation: towards cost-effective and privacy-preserving remote attestations. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 152–159.

[12] John Lyle and Andrew Martin. 2009. On the feasibility of remote attestation for web services. In *CSE*, Vol. 3. IEEE, 283–288.

[13] NVD. 2021. CVE-2021-44228.

[14] Andrew Paverd et al. 2014. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep* (2014).

[15] Ahmad-Reza Sadeghi and Christian Stüble. 2004. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW*. 67–77.

[16] Reiner Sailer et al. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture.. In *USENIX Security symposium*, Vol. 13. 223–238.

[17] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. 2012. {Policy-Sealed} Data: A New Abstraction for Building Trusted Cloud Services. In *21st USENIX Security Symposium (USENIX Security 12)*. 175–188.

[18] TCG 2018. *TCG Guidance for Securing Network Equipment.* TCG.

[19] TCG 2018. *TPM 2.0 Keys for Device Identity and Attestation.* TCG.

[20] TCG. 2022. TPM 2.0 Library - Trusted Computing Group. Retrieved February 24, 2022 from trustedcomputinggroup.org/resource/tpm-library-specification/

[21] K Watsen et al. 2019. Secure zero touch provisioning (SZTP). *Internet Requests for Comments, RFC Editor, IETF, Wilmington, DE, USA, Tech. Rep* 8572 (2019).

[22] Siyuan Xin, Yong Zhao, and Yu Li. 2011. Property-based remote attestation oriented to cloud computing. In *CIS*. IEEE, 1028–1032.

[23] Sachiko Yoshihama et al. 2005. WS-Attestation: Efficient and fine-grained remote attestation on web services. In *ICWS'05*. IEEE.