Remote Attestation with Constrained Disclosure

Michael Eckel
michael.eckel@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Germany

Dominik Roy George
d.r.george@tue.nl
Eindhoven University of Technology
Eindhoven, Netherlands

Björn Grohmann
bjoern.grohmann@gematik.de
gematik GmbH
Berlin, Germany

Christoph Krauß
christoph.krauss@h-da.de
Darmstadt University of Applied Sciences
Darmstadt, Germany

ABSTRACT

Trusted Platform Modules (TPMs) are used for remote attestation to ensure the authenticity and integrity of software running on a computer system. However, measuring software executed as containers or virtual machines can be challenging as it is measured concurrently, resulting in a jumbled measurement log that is difficult to disentangle. Moreover, disclosing the entire measurement log in traditional binary remote attestation raises privacy and intellectual property concerns. To address these issues, we propose a remote attestation method with constrained disclosure, allowing for selective disclosure of entries in the measurement log using a non-interactive zero-knowledge (NIZK) proof with Schnorr signatures. Our approach is evaluated for security and privacy and proven to be correct, sound, and satisfies the properties of a NIZK proof. Formal verification of our solution with ProVerif also supports our claims. Furthermore, the performance evaluation of our proof-of-concept implementation shows that our contribution is feasible, and the overhead introduced is negligible.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Cryptography**; **Privacy-preserving protocols**.

KEYWORDS

non-interactive zero-knowledge proof, remote attestation, trusted computing, privacy-enhancing technologies

ACM Reference Format:
Michael Eckel, Dominik Roy George, Björn Grohmann, and Christoph Krauß. 2023. Remote Attestation with Constrained Disclosure. In *Annual Computer Security Applications Conference (ACSAC '23), December 04–08, 2023, Austin, TX, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3627106.3627118

1 INTRODUCTION

Ensuring computer system trustworthiness is crucial for systems security. Trusted Computing, remote attestation, and measured boot are established technologies for verifying software integrity and authenticity. Measured boot computes deterministic hash values to measure the integrity of all loaded software components on a target system, the attester Measurements are stored in an event log in the kernel space of the operating system (OS), securely anchored in a Trusted Platform Module (TPM) [50] using a cryptographic folding hash. The Linux Integrity Measurement Architecture (IMA) extends measured boot to the OS and running applications. A verifier can use remote attestation to verify claimed software on the attester.

Most software systems today use software from different sources and vendors, and compartmentalization—e. g., containerization or virtualization—is a widely used security practice. This enables developers to bundle their software and all its dependencies into a singular package that can be deployed and executed on different computing environments. Furthermore, system administrators can utilize security policies to safeguard the host system and other software from potentially malfunctioning or malicious compartments.

However, traditional binary remote attestation faces several issues: (1) Measuring and reporting all events from all (sub-)components in the same place makes it hard for a remote verifier to disentangle individual (sub-)component log entries, especially with virtual machines (VMs) and containers. (2) Software vendors are unwilling to reveal file names or hash values of various binary versions and files included in a software package for privacy and intellectual property reasons. Passive attackers can infer running software versions based on hash values, enabling targeted attacks based on known vulnerabilities. Only vendors know which file names and hash values are valid and trusted, making them responsible for vouching for corresponding log entries' validity.

In this paper, we present a novel approach called Remote Attestation with Constrained Disclosure (RACD), which allows for traditional binary remote attestation to mask and selectively disclose event log entries. We achieve this by extending the Linux IMA concept and adding a non-interactive zero-knowledge (NIZK) proof based on Schnorr signatures. Our RACD approach complies with the third principle of remote attestation—constrained disclosure, which states that a target should be able to enforce policies governing which measurements are sent to each verifier. Our contributions in this paper are:

(1) Introducing our RACD approach, which enables constrained disclosure of event log entries for binary remote attestation.
(2) Providing a security and privacy analysis, including formal verification with ProVerif, to ensure that our solution does not compromise existing security properties.

ACSAC '23, December 04–08, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627118

(3) Developing a PoC implementation of our RACD approach, based on the Linux IMA concept, and making the source code publicly available.
(4) Demonstrating the feasibility of our approach through performance measurements based on our PoC implementation.

The remainder of this paper is organized as follows: In Section 2 provides an overview of Trusted Computing and remote attestation. In Section 3, we present our system model, use cases, and our threat model. We outline requirements for our proposed approach in Section 4. Section 5 details our RACD concept. To justify the efficacy of our solution in terms of security and privacy, we evaluate it in Section 7. Performance measurements and results of the formal verification with ProVerif are also discussed in this section. We analyze related work in Section 8. In Section 9, we conclude the paper and suggest potential future directions for our research.

2 BACKGROUND

Our RACD approach relies on a TPM and utilizes a modified measured boot process and remote attestation to ensure security. Measured boot, as defined by the Trusted Computing Group (TCG), enables a system to maintain an integrity-protected event log of all executed boot components, including the BIOS or UEFI, bootloader, and OS kernel [9, 32]. Each log entry identifies a component, e. g., "UEFI" or "Linux Kernel", along with a cryptographic hash digest known as the *measurement* of the software binary.

To protect the log entries, they are anchored in a tamper-evident manner in a TPM with multiple Platform Configuration Registers (PCRs). PCRs implement a cryptographic folding hash function called *extend*, and they only allow data to be read from or extended to the PCR. It is not possible to set a PCR to an arbitrary value.

In our RACD approach, the TPM is utilized due to its robust resistance against physical attacks, making tampering impractical for attackers. Being passive, every component must have the TPM measurement functionality integrated into its logic. Thus, (1) components execute their logic, (2) measure the next component in the boot order, (3) append the measurement to the boot event log and the TPM, and (4) pass control to the next component. It is crucial that components are measured before execution, to ensure potentially malicious components are recorded before they become active. At system power-on, the first component to activate is the Core Root of Trust for Measurement (CRTM), which is typically immutable and must be trusted implicitly.

In a remote attestation process, a target system, the attester, provides verifiable evidence to a remote verifier. Based on this evidence, the verifier appraises whether it is trustworthy, i. e., runs only authentic software. For this purpose, remote attestation relies on measured boot and the produced boot event log. When a communication partner, also known as a *relying party* in IETF Remote Attestation Procedures (RATS) terminology [7], needs to determine the trustworthiness of the attester, it requests an attestation from the verifier (Fig. 1, step 1). The verifier requests an attestation from the attester (step 2) by providing a random nonce that is used for freshness and to counter replay attacks. With a *TPM Quote* operation (step 3), the attester instructs the TPM to create a digital signature over its internal state, i. e., its PCRs, incorporating the nonce. The key that is used to sign the internal state of the TPM is only known to the TPM. The attester provides the TPM Quote and the event log as cryptographically signed evidence of its operational state to the verifier (step 4). The verifier assesses the validity of the digital signature (step 5) by using the corresponding public key or digital certificate. Then it compares the entries in the event log against a whitelist that contains *known good* component hashes. Finally (step 6), the verifier communicates the attestation result securely to the relying party, indicating whether the attester is trustworthy or not.

Linux IMA extends measured boot into the OS [52, 53]. Thereby, it measures and logs all native application binaries *before* they are loaded into memory for execution, irrevocably anchoring them in the TPM. Further, all files accessed by user *root* are measured [41, 42]. This way, IMA augments the remote attestation process with the IMA log, providing measurements of OS binaries and files.

3 SYSTEM MODEL

In this section, we use the terminology from the IETF RATS architecture [7] to describe the components in our system model. We focus on the roles, their interactions, and involved data (artifacts) to describe relevant processes. Involved roles are (1) the *attester*, that produces evidence to be appraised by the verifier, (2) the *verifier*, who appraises the validity of evidence from the attester and produces attestation results, (3) the *relying party*, that consumes attestation results from the verifier. Involved artifacts are (1) *evidence*, which are (digitally signed) claims about the platform configuration, including measurements of software binaries and files, such as TPM quotes and event logs, (2) the *attestation result*, that is produced by the verifier and that includes information about the trustworthiness of the attester, (3) the *reference values*, that the verifier uses as a whitelist of known good measurements of software binaries and files to appraise evidence.

Relying parties must ensure the integrity of a target system (attester) before entrusting it with the computation of sensitive or confidential data. For that purpose, relying parties consume attestation results from a verifier to determine the trustworthiness of an attester. We adapt the traditional remote attestation architecture from Fig. 1 and introduce the role of partial verifiers, which appraise masked event logs with only partially disclosed entries (cf. Fig. 2). In this way, we can partition the event log and disclose only log entries that are associated with a particular third party. To mask the event log and disclose only chosen entries, we use non-deterministic hashes based on Schnorr signatures. The original responsibility of the (main) verifier is distributed among several partial verifiers. The main verifier's new task is to assess if every entry in the event log has been verified by at least one partial verifier. This is because we assume that a system can only be considered trustworthy if all entries in the event log have been verified with a positive result, and no log entries remain unchecked.

Whenever a relying party needs to know if the attester is *trusted*, it contacts the verifier (cf. Fig. 2, step 1). The verifier requests an attestation from the attester (step 2), which performs a *TPM Quote* operation (step 3). Up to this point there is no difference to the traditional remote attestation. However, instead of returning the TPM quote and the event log to the verifier, the attester now requests partial verifications from the partial verifiers (step 4). For this

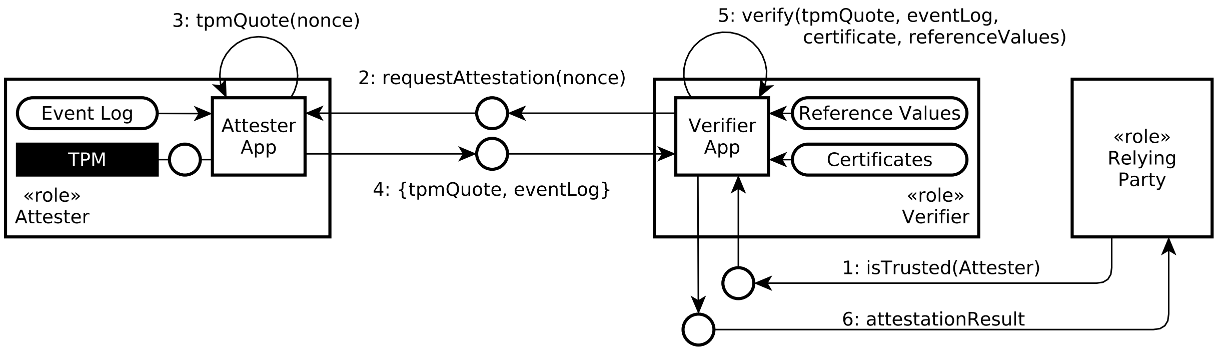

Figure 1: Traditional remote attestation process with the roles Attester, Verifier, and Relying Party (cf. IETF RATS [7]).

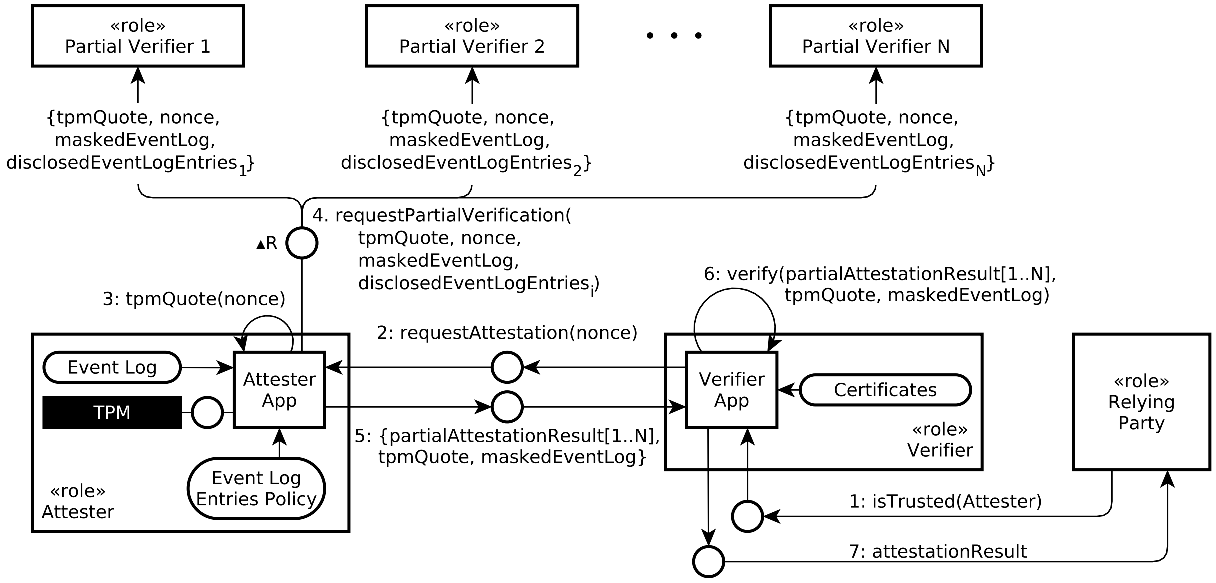
«role» Partial Verifier 1
«role» Partial Verifier 2
«role» Partial Verifier N
{tpmQuote, nonce, maskedEventLog, disclosedEventLogEntries$_1$}
{tpmQuote, nonce, maskedEventLog, disclosedEventLogEntries$_2$}
{tpmQuote, nonce, maskedEventLog, disclosedEventLogEntries$_N$}
▴R
4. requestPartialVerification(tpmQuote, nonce, maskedEventLog, disclosedEventLogEntries$_i$)
3: tpmQuote(nonce)
6: verify(partialAttestationResult[1..N], tpmQuote, maskedEventLog)
2: requestAttestation(nonce)
Event Log
Attester App
TPM
5: {partialAttestationResult[1..N], tpmQuote, maskedEventLog}
Event Log Entries Policy
«role» Attester
Verifier App
Certificates
«role» Verifier
«role» Relying Party
1: isTrusted(Attester)
7: attestationResult

Figure 2: Remote Attestation with Constrained Disclosure (RACD) process with the roles Attester, Verifier, Partial Verifier, and Relying Party (cf. IETF RATS [7]).

purpose, the attester sends the TPM quote, the nonce, the masked event log (as created by our adapted measurement process, using non-deterministic hashes), and the disclosed event log entries to the partial verifiers. Based on an *event log entries policy*, the attester knows which entries must be appraised by which partial verifier. The event log entries policy constitutes a lookup table, and may be implemented as one or more files, or a database. This lookup table can, for instance, be installed and maintained by the system operator, or distributed as part of the installed software packages, containers, or VMs.

Each partial verifier (1) verifies the TPM Quote signature, (2) appraises the integrity of the masked event log against the TPM quote, (3) appraises the disclosed event log entries based on our NIZK proof, and (4) matches the disclosed log entries to known-good reference values (whitelist). As a result, partial verifiers reply to the attester with a signed *partial attestation result*, containing the nonce and a map of matched entries (the corresponding non-deterministic hashes) from the masked event log, associated with an indicator whether the entry is authentic/trusted.

After receiving all partial attestation results, the attester forwards them to the main verifier, together with the TPM quote and the masked event log (step 5). In step 6, the main verifier (1) verifies the TPM Quote signature and the nonce, (2) validates the masked event log against the TPM quote to verify its integrity, (3) verifies the signature of all partial attestation results (using trusted certificates), and then (4) checks if each entry in the masked event log is hit by at least one (trusted) entry in the partial attestation results. As a response to the relying party (step 7), the main verifier sends a signed attestation result about the trustworthiness of the attester, exactly as in the traditional remote attestation process.

Note that in our RACD approach the main verifier no longer needs to maintain reference values. Further, note that the RACD approach is still based on roles, i. e., attester, verifier (split into partial verifiers and main verifier), and relying party. One entity may implement several roles, e. g., a system with a Trusted Execution Environment (TEE) can implement both a (co-located) verifier in a TEE and an attester in the Rich Execution Environment (REE).

With US President's Executive Order (EO) 14028 on "Improving the Nation's Cybersecurity" [19], the Executive Office of the President mandates software vendors to provide customers with Software Bill of Materialss (SBOMs) [43]. A SBOM is a manifest for a software package and contains information such as file paths, hashes, and metadata supplied as part of the software package in a machine-readable format. They are vendor-specific and can perfectly serve as lookup tables for the event log entries policy. Moreover, SBOMs can serve as reference values for a verifier in a remote attestation process. Microsoft [3, 18, 36] and Adobe [39], among others, ship their products with SBOMs.

3.1 Use Cases

Our RACD approach has several real-world applications, ranging from IoT and other embedded Linux-based systems to desktop PCs and even cloud scenarios. The integrity and authenticity of the software is verified by several provider-specific partial verifiers, each one responsible for verifying a set of software components, represented as entries in the measurement log. With our RACD approach, we are able to disclose only entries that belong to a partial verifier, blinding all others. For instance, a system with the OS Ubuntu (Canonical), Microsoft Teams, and JetBrains IntelliJ IDEA installed, would require partial verifiers from Canonical, Microsoft, and JetBrains. This motivates us to provide a scheme with capabilities of traditional remote attention while providing privacy with constrained disclosure.

As long as there is software from multiple sources or providers involved, including containers and VMs, RACD is applicable. This applies to more or less managed systems, such as home energy management systems and network equipment (routers, switches, etc.) as well as interactive systems where users are allowed to install apps, such as automotive head unit systems, Linux-based smartphones like Android, and cloud appliances. In the following, we consider three concrete use cases.

3.1.1 Use Case 1: Bank Transfer from a Laptop. A bank customer wants to make a transfer on his or her laptop and—in addition to the usual authentication with username and password—must confirm that his or her system is running the latest and uncorrupted version of the banking software. For this purpose, the bank server requests an attestation from the customer system. The customer system acts as an attester that triggers a TPM quote, and requests attestation results from partial verifiers for the installed software. In this case, the bank itself is also a partial verifier, as it is the provider of the customer-facing banking software, and consequently has the authority to appraise the relevant disclosed event log entries. The bank server fulfills the roles of the relying party, the main verifier, and (a) partial verifier.

3.1.2 Use Case 2: Android Smartphone and VPN. An employee wants to use his or her smartphone to access his or her workplace virtual private network (VPN). In addition to the 802.1X and IPsec authentication methods, the VPN gateway requires proof that only authentic software is running on the smartphone, so as not to risk espionage or targeted attacks by malicious apps on the smartphone. For this purpose, the VPN gateway requests an attestation from the smartphone. The smartphone implements the role of an attester, that triggers a TPM quote, and requests attestation results from partial verifiers, e. g., app stores and the smartphone vendor, for the installed apps and system software. The VPN gateway acts as verifier, and grants access to the VPN based on the trustworthiness of the attester, i. e., the smartphone.

Since TPMs are not available per se in mobile phones, the TCG provides the TPM 2.0 Mobile Reference Architecture Specification [45] and the TPM 2.0 Mobile Common Profile [46] that describe how to implement a TPM 2.0 in software in a TEE as available in mobile phones. Further, Chakraborty et al. [12] describe a TPM 2.0 based on Subscriber Identity Module (SIM) cards: "simTPM: User-centric TPM for Mobile Devices".

3.1.3 Use Case 3: Containerized Edge Node. Containerization in edge computing involves packaging applications and their dependencies into containers for deployment on edge devices. Containers provide a consistent environment, ensuring applications run uniformly across diverse edge infrastructures. They offer efficiency, portability, and scalability benefits. In edge scenarios, containers are often managed using orchestration tools, allowing for automated deployment, scaling, and operations of application containers across clusters of edge devices. This setup streamlines application updates, minimizes overhead, and tackles the unique challenges of edge computing, like intermittent connectivity and limited resources. Containers share the same underlying OS kernel, which lets Linux IMA within that kernel "see" and measure containerized applications. Unlike hypervisor-based virtualization that emulates an entire machine with its guest OS and kernel, the host OS kernel remains unaware of applications running in the guest OS.

Security is crucial in edge containerization, as vulnerabilities could compromise the host system or expose sensitive data, necessitating robust isolation and container management practices. Using our RACD approach, edge nodes can selectively verify the trustworthiness of containers by disclosing only log details specific to each container. RACD allows all containers to log into a single TPM PCR. With the event log entries policy—that holds information about containerized software—, RACD can disentangle and partition the Linux IMA event log, unmasking a separate set of entries for each of the containers without revealing entries from other containers or software.

In general, our use cases could benefit from TEE technologies such as Intel Software Guard Extensions (SGX) [27] or AMD Secure Encrypted Virtualization (SEV) [1]. However, these technologies are only available on Intel or AMD machines, not on mobile phones. Further, Intel discontinued SGX technology for Core processors [38, 44] in favor of Trust Domain Extensions (TDX) [28].

3.2 Threat Model

In traditional remote attestation, an attester reveals the entire list of running software and verifiers know about all software running on attester systems. A (partial) verifier being an *honest-but-curious (HbC)* adversary, can leverage this information and query the Common Vulnerabilities and Exposures (CVE) database to identify and exploit vulnerable software. The HbC verifier will follow the attestation protocol, without deviating from the protocol's procedure while trying to learn as much information as possible. Even though the attester sends only (deterministic) hashes of binaries along with the file path, an HbC verifier can use or produce a reverse lookup table to match up the hashes to corresponding binaries. Consequently, the HbC verifier has the ability of knowing if attesters run versions of software with unpatched vulnerabilities to compromise their systems by launching targeted attacks. In the real-world scenario, it would lead to a threat in network equipment that complies with the Bell-LaPadula integrity model, as addressed in TCG Trusted Attestation Protocol (TAP) specifications [47, 48].

The Internet Engineering Task Force (IETF) Supply Chain Integrity, Transparency, and Trust (SCITT) working group addresses EO 14028 requirements further and features transparency of evidence, including remote attestation data and event logs. According to the requirements, the traditional remote attestation does not provide user privacy or vendor privacy. Hence, based on our use case 1 (see Section 3.1.1), the bank knows about all installed programs on the target system (user privacy issue). In addition, the partial verifiers are capable of understanding the programs of other partial verifiers (vendor privacy issue).

4 REQUIREMENTS

Our goal is to introduce the technical means to enable constrained disclosure for TPM-based remote attestation. Based on our system model (cf. Section 3), and especially our threat model (cf. Section 3.2), we define the following security requirements:

- R_1^S Integrity and authenticity of the attester's operational state must be assured. A TPM-anchored event log of all loaded software must be maintained.
- R_2^S Confidential communication must be established between attester and (partial) verifiers in order to prevent *passive man-in-the-middle* attacks.
- R_3^S Mutual authentication must take place between attester and (main/partial) verifier to ensure the identity of each party.
- R_4^S A suitable elliptic curve must be used that guarantees no information is leaked to attackers. It must be efficient and have advantageous security properties for the integration with the RACD protocol based on a NIZK proof.
- R_5^S The attester must be able to (securely) disclose any subset of event log entries to several partial verifiers in a remote attestation process, while maintaining the existing (TPM-rooted) security properties integrity and authenticity (cf. [14]).
- R_6^S Partial verifiers must not collude with each other, nor must they know each other. Otherwise, they would have the power to merge constrained disclosures.

5 Remote Attestation with Constrained Disclosure

In this section, we introduce our RACD concept that enables binary remote attestation to selectively disclose event log entries from the attester to partial verifiers, as described in Section 3. Section 5.1 outlines our adapted measurement process. In Section 5.2, we describe our adapted remote attestation process. The RACD scheme was first presented in the master's thesis of D. R. George [22], and is extended in this paper. We use the notation depicted in Table 1.

Table 1: Notation Summary.

Notation	Description
$h_T(\cdot)$	Template hash function
$H(\cdot)$	Cryptographic hash function
$\varphi(\cdot)$	Injective function
$h_{eventhash}(\cdot)$	Generates an *event hash*
$\mathcal{X}$	Template hash domain
$\mathcal{Y}$	Integer domain
g	Generator of cyclic group G
x_i	i-th element of the template hash domain
r_i	Randomly chosen value for i-th element in $\mathcal{X}$
v_i	Randomly chosen value for x_i from the integer domain $\mathbb{Z}$
g_i	Generator of the i-th element
t_i	Computed scalar for the i-th element/entry
c_i	Computed challenge for the i-th element/entry
(c_i, s_i)	Computed scalars as pairs of the i-th element
t'_i	Computed scalar from the partial verifier for the i-th element/entry
c'_i	Computed challenge from the partial verifier for the i-th element/entry
L	Prime order group of Curve25519

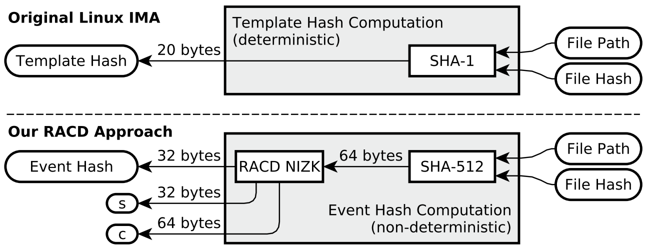

Figure 3: Computation of the *template hash* from Linux IMA and the *event hash* from our RACD approach.

5.1 Measurement Process

We adapt the IMA measurement process as well as the binary remote attestation process to allow selective disclosure of IMA log entries. For that purpose, we replace the deterministic hashes in the log with non-deterministic hashes based on a NIZK proof (Schnorr signature) per log entry. This allows us to disclose only corresponding entries to a partial verifier during remote attestation, while keeping intact the TPM-based integrity protection of the event log as a whole.

The measurement process of our RACD approach follows the same initial steps as the measured boot sequence. The chain of trust starts with the immutable Root of Trust for Measurement (RTM) and continues up to the OS. The difference in our approach lies in the Linux IMA measurement process. The original implementation of IMA measures software using a configurable deterministic hash algorithm, e. g., SHA-256, resulting in a file hash (cf. diagram at the top of Fig. 3). The file hash and the file path of the measured file are hashed together with SHA-1, resulting in the *template hash*, according to IMA terminology. This *template hash* is then used to *extend* the configured TPM PCR (by default PCR 10).

The deterministic *template hash* of a software binary is overly informative. An attacker can deduce details about a specific binary using just the *template hash*, given that software binaries are typically stored in known file paths and a limited number of versions exist with their corresponding hashes. To avoid this issue, we employ a non-deterministic hash, called *event hash*, in our RACD method. The term *event* aligns with TCG terminology. A diagram of the *event hash* computation is shown at the bottom of Fig. 3.

The event hash basically is the cryptographically blinded *template hash*, constructed by applying a NIZK signing process known as Schnorr signature [8, 25]. First, the *template hash* function is defined as $h_T : \mathcal{X} \to \mathcal{Y}$ (we use SHA-512). Let $\mathcal{X}$ be a set of *template hashes* and function h_T transforms them into the range $\mathcal{Y}$. In order to generate the *event hash*, a generator g of a (cyclic) group G needs to be selected (generator g in the case of this work is the base point of an appropriate elliptic curve over a finite field). Second, the function for the *event hash* is defined as $h_{eventhash} : \mathbb{Z} \times \mathcal{X} \to G$, which is computed as

$$h_{eventhash}(r, x) := g^{r\varphi(h_T(x))} \tag{1}$$

The first step of the computation is to blind the *template hash* by generating a random integer $r \in \mathbb{Z}$. x is an element of the *template hash* domain ($x \in \mathcal{X}$). The injective function $\varphi : \mathcal{Y} \to \mathbb{Z}$ maps $\mathcal{Y}$ into the integer domain $\mathbb{Z}$. The *event hash* (instead of the *template hash*) is stored in the IMA log and extended into the TPM PCR. Note: The scalar r is not saved in the IMA log.

5.1.1 Non-Interactive Zero-Knowledge Proof Generation. Once the *event hash* is created, the attester must produce a (non-interactive) "proof of knowledge" based on the well-known Schnorr signature and Fiat-Shamir heuristic, which is then presented to the partial verifier during the remote attestation process. The NIZK proof must be generated for each measured binary in the IMA log. The attester (IMA) computes for every measured entry i a generator $g_i := g^{\varphi(h_T(x_i))}$. Next, it chooses a random integer $v_i \in \mathbb{Z}$ and computes the value $t_i := g_i^{v_i}$. Afterwards, the challenge $c_i := H(g_i, t_i, h_{eventhash}(r_i, x_i))$ is generated, where H is a cryptographic hash function. Finally, the attester computes the scalar $s_i := v_i - c_i r_i \pmod{|G|}$. The pair (c_i, s_i) is the "extra data" (cf. Fig. 3) for each log entry that is necessary for the partial verifier to verify the *event hash* since it has zero knowledge of the key r_i. The computational steps are based on the standard Schnorr NIZK proof over an elliptic curve [25].

5.1.2 Adapted IMA Log for RACD. We introduce a new IMA log format where we replace the column *template hash* with *event hash*. The *IMA Template* column holds our new IMA template identifier *ima-cd* ("constrained disclosure"). Again, for each entry i in the IMA log, an element $r_i \in \mathbb{Z}$ is randomly chosen and used to compute the hash value $h_{eventhash}(r_i, x_i)$. x_i, an element of the domain $\mathcal{X}$ of *template hashes*, is a concatenation of *File Hash* and *File Path* for each entry i in the IMA log (the integer i is the index of a log entry). Besides, the columns c and s are known as *extra data* (cf. Table 2). The pair (c_i, s_i) is necessary for the partial verifier to verify the Schnorr signature (the "proof of knowledge").

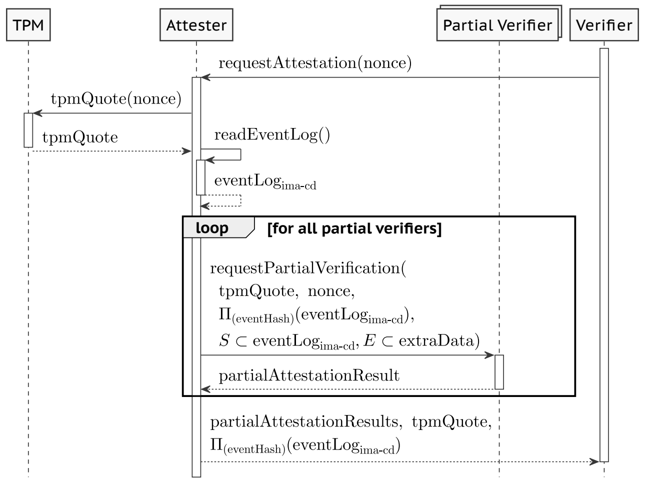
TPM
Attester
Partial Verifier
Verifier
requestAttestation(nonce)
tpmQuote(nonce)
tpmQuote
readEventLog()
eventLog$_{\text{ima-cd}}$
loop
[for all partial verifiers]
requestPartialVerification(
tpmQuote, nonce,
$\Pi_{(\text{eventHash})}(\text{eventLog}_{\text{ima-cd}})$,
$S \subset \text{eventLog}_{\text{ima-cd}}$, $E \subset$ extraData)
partialAttestationResult
partialAttestationResults, tpmQuote,
$\Pi_{(\text{eventHash})}(\text{eventLog}_{\text{ima-cd}})$

Figure 4: Remote Attestation with Constrained Disclosure process.

5.2 Remote Attestation Process

Figure 4 visualizes our RACD process with focus on the partial verifiers. The (main) verifier sends an attestation request to the attester, including a nonce for freshness and against replay attacks . The attester performs a TPM quote and reads the event log. Then, the attester requests partial verifications of the event log from all partial verifiers, passing the TPM quote, the nonce, the masked event log—i. e., the entire *Event Hash* column $\Pi_{(\text{eventHash})}(\text{eventLog}_{\text{ima-cd}})$—as well as the corresponding disclosed log entries $S \subset \text{eventLog}_{\text{ima-cd}}$ and $E \subset$ extraData. $E \subset$ extraData defines the subset of corresponding pairs (c_i, s_i) of the disclosed log entries. Based on the *event log entries policy*, the attester knows which event log entries must be verified by which partial verifier. Note that the *Event Hash* column contains only the non-deterministic event hashes. Partial verifiers run through the following verification process:

(1) *Verify the overall event log integrity* of the masked event log by simulating the TPM PCR *extend* operation over all (non-deterministic) *event hashes*, thus, recomputing the expected TPM PCR value. Then, the partial verifier compares the recomputed PCR value against the actual one from the TPM quote. Only if they match, the log is considered.
(2) *Verify the log entry integrity* of all disclosed entries by verifying the Schnorr signature. For this purpose, the partial verifier computes the generator g_i for each of the corresponding entries, which is the same as in the *measurement procedure*. Further, the scalar $t'_i := g_i^{s_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$ is computed, which is used in the cryptographic hash function H in order to compute $c'_i := H(g_i, t'_i, h_{eventhash}(r_i, x_i))$. Note that the corresponding *event hash*, c_i, and s_i can be directly read from the log entry. Finally, the partial verifier accepts the proof of knowledge, if and only if $c_i = c'_i$. This verification process allows the attester to prove to the partial verifier that the *event hash* is the *masked template hash*, without revealing the key. By applying this procedure, the partial verifier has zero knowledge of the integer r_i. Figure 5

PCR	Event Hash	IMA Template	File Hash	File Path	c	s
10	c4456...331	ima-cd	65727...12b	boot_aggregate	b4612...1ac	3cbd2...a47
10	73c9b...eff	ima-cd	153f3...c95	/lib64/ld-linux-x86-64.so.2	9ce45...2bf	feb14...476
10	9b0ed...c09	ima-cd	4ebaf...c9a	/lib/x86_64-linux-gnu/libc.so.6	2bb78...100	2199e...fd5
10	fef56...71b	ima-cd	64743...f69	/bin/dash	5234d...edd	06069...e68
10	ee599...667	ima-cd	fc66b...cef	/bin/mkdir	de567...4bd	7fc90...a2b
10	089c4...4bf	ima-cd	2f43e...699	/bin/ln	7d400...031	6b1a7...e90
10	78bd2...14c	ima-cd	e46fd...b48	/bin/mount	067d...93d	d7fc6...70c
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2: Constrained disclosure enhancements to the IMA log (eventlog$_{\text{ima-cd}}$)

illustrates in mathematical notation the adapted version of the NIZK proof [8, 25].

(3) *Appraise the content of log entries* by matching the *File Hash* and *File Path* of each log entry with known-good reference values from a whitelist.

Partial verifiers return a signed partial attestation result to the attester. This includes the nonce and a list of verified entries in the form $(eventhash, result)$. The *eventhash* corresponds to a disclosed log entry and the *result* is either *trusted* or *not trusted*. Finally, the attester replies to the verifier with all partial attestation results, the TPM quote, and the masked event log $\Pi_{(\text{eventHash})}(\text{eventLog}_{\text{ima-cd}})$.

The (main) verifier appraises all received partial attestation results. Only if attestation results for *all* entries in the masked event log are present and *trusted*, the entire attester system is considered trusted. The attestation result is then sent to the relying party.

There is a chance that entries are verified by more than one partial verifier. Shared libraries, such as the *libc*, are used by several applications or containers, and consequently may be verified by multiple partial verifiers. The following is a more formal presentation of the partial verification process: Let E be the set of all entries of an event log, and $e \in E$ be a single entry. Further, let $d_i \subset E$ be disclosed entries from an attester to a partial verifier i, and n be the number of partial verifiers needed to verify all entries at least once. Then, the (main) verifier needs $\bigcup_{i=1}^{n} d_i \geq E$ verified entries to ensure verifications for all entries ($\forall e \in E$) are present.

6 IMPLEMENTATION

We conduct performance measurements using a Raspberry Pi 3 Model B V1.2 running Raspberry Pi OS Lite in version *bullseye* from February 21, 2023. A LetsTrust TPM with an Infineon Optiga™ SLB 9670 TPM 2.0 is attached to the GPIO ports. The proof-of-concept (PoC) implementation is based on CHARRA [20] and D. R. George's master's thesis [22]. It is written in portable C99 and the source code is available on GitHub at https://github.com/DominikRoy/RACD. Our concept builds upon Linux IMA and remote attestation, but without modifying the kernel source code. Instead, we develop userspace applications based on the IMA principle and log format, implementing the RACD protocol. To obtain more meaningful results, we evaluate the performance by running both attester and verifier applications on separate Raspberry Pis, enabling a direct comparison of attestation and verification processes on the same hardware.

We implement the Schnorr NIZK proof using *Ed25519* [25], an elliptic curve that with a maximum size of 32 bytes, as restricted by the hardware TPM on our target platform. It supports SHA-256 as the strongest hash algorithm for event data in a *TPM PCR Extend* operation. We realize our NIZK logic with Libsodium [17] that employs the Ristretto technique. We establish mutually-authenticated encrypted TLS 1.2 channels between the attester and the verifier, as well as between the attester and all partial verifiers, using mbed TLS [2]. For wire-encoding, we use Concise Binary Object Representation (CBOR) with the QCBOR library [34]. To communicate with the TPM, we use the TPM2-TSS library [21, 49].

To generate the Schnorr signature, we use Libsodium's function `crypto_core_ristretto255_scalar_reduce()` to uniformizes input values for arithmetic operations. The input consists of 512 bits (64 bytes), while the output is 256 bits (32 bytes). We use SHA-512 to hash the file hash and path, which produces a 64-byte digest. This digest serves as the input for the Libsodium function (cf. Algorithms 1 and 2). The resulting 32-byte output is our *event hash*.

Algorithm 1 provides pseudocode for Schnorr NIZK signing. The input is an *event*, i. e., a file path and file hash. The algorithm generates the *template hash* on line 2 and reduces it into a scalar on line 4 to fall within the range of 0 to L, where $L = 2^{252} + 27742317777372353535851937790883648493$. Ristretto scalars are designed to stay within the range of 0 and L for each arithmetic operation. Line 7, 9, and 12 check the validity to ensure that the resulting value is still a valid point on the elliptic curve after scalar multiplication, and uses mathematical steps similar to those in Fig. 5. Line 8 creates a new generator that is accepted if it is a valid point on the curve, and can be used in place of the basepoint if the output is a valid point on the curve. Lines 4, 5, 14, 15, and 16 use the modulo operation to keep scalars in the valid range. Eventually, line 17 returns the *event hash*, and the scalars c and s.

Our NIZK verification (Algorithm 2) takes an *event record* (c, s, and event hash) as input, and outputs *true* if the verification succeeds, *false* otherwise. Lines 3–6 create the generator g_i. Line 12 adds the two valid curve points to obtain $t_{i'}$, which determines if the proof of knowledge is successful. Success is achieved only if c' equals $eventrecord.c$ (see Section 5).

Attester **Partial Verifier**

$$h_{eventhash}(r_i, x_i) = g^{r_i \cdot \varphi(h_T(x_i))}$$
$$v_i \leftarrow\!\$\, \mathbb{Z}$$
$$t_i \leftarrow g_i^{v_i}$$
$$c_i \leftarrow H(g_i, t_i, h_{eventhash}(r_i, x_i))$$
$$s_i \leftarrow v_i - c_i \cdot r_i \mod |L|$$

$$\xrightarrow{(c_i, s_i),\, h_{eventhash}(r_i, x_i)}$$

$$t'_i \leftarrow g_i^{s_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$$
$$= g_i^{v_i - c_i \cdot r_i} \cdot h_{eventhash}(r_i, x_i)^{c_i}$$
$$c'_i \leftarrow H(g_i, t'_i, h_{eventhash}(r_i, x_i))$$
if $c_i = c'_i$ **return** *true*

Figure 5: Adapted Schnorr Non-Interactive Zero-Knowledge Proof.

Algorithm 1: NIZK Signing

```
Input: $event$
Output: $event_{hash}$, $c$, $s$
Function nizksign($event$):
    $templatehash \longleftarrow h_T(event)$;
    $r \longleftarrow random()$;
    $reduced_{digest} \longleftarrow templatehash \pmod{L}$;
    $r_h \longleftarrow r \cdot reduced_{digest} \pmod{L}$;
    $event_{hash} \longleftarrow g^{r_h}$;
    if $event_{hash}$ is not a valid point then return error;
    $g_i \longleftarrow g^{reduced_{digest}}$;
    if $g_i$ is not a valid point then return error;
    $v \longleftarrow random()$;
    $t_i \longleftarrow g_i^{v}$;
    if $t_i$ is not a valid point then return error;
    $c \longleftarrow H(g_i \| t_i \| event_{hash})$;
    $reduced_c \longleftarrow c \pmod{L}$;
    $r_c \longleftarrow r \cdot reduced_c \pmod{L}$;
    $s \longleftarrow v - r_c \pmod{L}$;
    return $event_{hash}, c, s$;
End Function
```

Algorithm 2: NIZK Verification

```
Input: $eventrecord$
Output: $valid$
Function nizkverify($eventrecord$):
    $valid \longleftarrow$ false;
    $templatehash \longleftarrow h_T(eventrecord.event)$;
    $reduced_{digest} \longleftarrow templatehash \pmod{L}$;
    $g_i \longleftarrow g^{reduced_{digest}}$;
    if $g_i$ is not a valid point then return error;
    $gi_s \longleftarrow g_i^{eventrecord.s}$;
    if $gi_s$ is not a valid point then return error;
    $reduced_c \longleftarrow eventrecord.c \pmod{L}$;
    $eventhash_c \longleftarrow eventrecord.event_{hash}^{reduced_c}$;
    if $eventhash_c$ is not a valid point then return error;
    $t_i\prime \longleftarrow gi_s \cdot eventhash_c$
    $c\prime \longleftarrow H(g_i \| t_i\prime \| event_{hash})$;
    if $c\prime == eventrecord.c$ then $valid \longleftarrow$ true;
    return $valid$;
End Function
```

7 EVALUATION

In this section, we evaluate our RACD approach. In Section 7.1, we provide a security and privacy analysis. We demonstrate the feasibility of our approach with a performance evaluation in Section 7.2. Section 7.3 discusses several aspects of our approach.

7.1 Security Analysis

Traditional binary remote attestation provides integrity and authenticity of event logs, protected in hardware by the TPM. The *Template Hash* is anchored in the TPM using the folding hash function *PCR Extend*, providing integrity protection for the entire event log. Further, attestation involves sending the log and a digital signature over the TPM's internal state (TPM quote) to a verifier to ensure both authenticity and integrity. This fulfills requirement R_1^S.

Our RACD method improves binary remote attestation by adding the security property of selective confidentiality regarding running software. This allows an attester system to disclose log entries while preserving the unlinkability between non-deterministic event hashes and the actual software binary, which satisfies requirement R_5^S. The fully masked and signed event log, as well as the signed partial attestation results, are provided to the main verifier, which can then verify their integrity and authenticity to assess the attester's trustworthiness, without knowing the actual running software.

We can disclose all entries to one verifier just like with traditional remote attestation, but this is in contrast with our use case and does not require our RACD approach at all. Our approach only allows communication between the attester and the main verifier, as well as between the attester and partial verifiers. Partial verifiers aren't allowed to communicate with each other as per requirement R_6^S.

7.1.1 Threat Defense. We counteract threats identified in our threat model (cf. Section 3.2). To prevent passive man-in-the-middle attacks, we use encrypted communication (TLS) between attester and all verifiers (requirement R_2^S). This encrypted channel thwarts adversaries from reading the event log and combining subsets of disclosed data. Before starting the attestation process, we require mutual authentication between the attester and all verifiers (requirement R_3^S). RACD uses NIZK as a sub-protocol, so we utilize the existing NIZK proof system (with adaptations) to achieve constrained disclosure (and privacy) while preserving the overall integrity and authenticity of the event log. We rely on the pre-existing properties of NIZK and informally demonstrate their validity in the following.

Completeness (Correctness). With the proof of knowledge, the attester convinces partial verifiers that the event hash is the blinded template hash. In other words, if the verification is valid ($c_i = c'_i$), the partial verifier accepts the proof of knowledge over the relevant event hash for the provided values $h_{eventhash}(r_i, x_i)$, c_i, and s_i.

Soundness. The attester needs the secret to blind the template hash and to convince partial verifiers of the correctness of the statement. If the attester fails to provide the secret, the partial verifier cannot use the "extra data" for non-associated event hashes, making the proof of knowledge verification invalid. This means the attester cannot convince the partial verifier of a false statement.

Our approach involves providing the partial verifier with the subset of entries of the event log and the proof of knowledge over the associated template hashes. We generate a random value (secret) r_i for each measurement and proof of knowledge v_i. We assume the use of a secure random number generator and the recomputation of event hashes on each boot cycle. If the same r_i and v_i are used for each NIZK generation, the partial verifier can retrieve the secret. Thus, we use secure random generation and avoid using the same r_i and v_i during the measurement process. If the secret can be extracted, the event hash would no longer blind the template hash. If given two valid tuples with the same secret, the verifier can compute the secret r_1:

$$\frac{s_1 - s_2}{c_2 - c_1} = \frac{v_1 - r_1 \cdot c_1 - v_1 + r_1 \cdot c_2}{c_2 - c_1} = \frac{r_1 \cdot (c_2 - c_1)}{(c_2 - c_1)} = r_1 \quad (2)$$

Zero-Knowledge. To prove knowledge without revealing the secret used to blind the template hash, we pass only the extra data (c_i, s_i) to the associated event hash $h_{eventhash}(r_i, x_i)$, allowing partial verifiers to confirm the statement's accuracy using a NIZK proof.

$$\begin{aligned} t'_i &= g_i^{s_i} \cdot h_{eventhash}(r_i, x_i) \\ &= g^{\varphi(h_T(x_i))\cdot(v_i - c_i \cdot r_i)} \cdot g^{r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\ &= g^{\varphi(h_T(x_i))\cdot v_i - \varphi(h_T(x_i))\cdot c_i \cdot r_i} \cdot g^{r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\ &= g^{\varphi(h_T(x_i))\cdot v_i - \varphi(h_T(x_i))\cdot c_i \cdot r_i + r_i \cdot \varphi(h_T(x_i))\cdot c_i} \\ &= g^{\varphi(h_T(x_i))\cdot v_i} = g_i^{v_i} = t_i \end{aligned} \quad (3)$$

Since we rely on scalar multiplication, it is not possible for adversaries or verifiers to infer the template hash from Eq. (1). The Elliptic Curve Discrete Log Problem (ECDLP) makes it impractical to retrieve the secret used in the computation from publicly known values. Our method assumes a properly chosen elliptic curve and a collision-resistant cryptographic hash function used to compute the challenge c. The security of the NIZK depends on the ECDLP, the properties of zero-knowledge proofs, a secure random number generator, and a collision-resistant cryptographic hash function H.

7.1.2 Elliptic Curve Analysis. Curve25519 meets our requirements for size and security and supports the ladder technique for fast scalar multiplication, making it a suitable elliptic curve for our purposes. We based our selection on the elliptic curve analysis by Tanja Lange and D. J. Bernstein, who maintain a website listing secure elliptic curves [6]. However, for our implementation of the Schnorr signature scheme, we use the twisted Edwards curve Ed25519, which provides faster scalar arithmetic and equivalent security to Curve25519 for signature schemes [4, 31]. Additionally, Ed25519 is already used for signature schemes in existing literature [5], making it advantageous for our approach.

Curve25519 and Ed25519 have an order of $8L$, where L is the large prime number $2^{252} + 27742317777372353535851937790883648493$. However, the Schnorr signature scheme requires establishing prime order groups [10], which is more complicated with non-prime order groups such as Curve25519 and Ed25519 due to their complex group structure. To resolve this issue, we employ the Ristretto technique [23, 24], which constructs prime order elliptic curve groups and eliminates the cofactor. Ristretto extends existing systems using Ed25519 signatures and ensures no further cryptographic assumptions are required. We need this for our work and thus map Edward points to Ristretto points and validate their canonical encoding.

7.1.3 Protocol Verification with ProVerif. Our protocol selectively discloses log entries while maintaining integrity and authenticity. We use ProVerif to guarantee the secrecy of non-relevant log entries in a single round of our scheme. The cryptographic primitives underlying our protocol are assumed to be robust, such as ECDLP. ProVerif assumes that the adversary understands the cryptographic algorithms and public values. Based on these assumptions, ProVerif formally verifies our protocol against the Dolev-Yao attacker model. This model describes an adversary's ability to modify, access, delete, and forge new messages during the communication session. Furthermore, ProVerif identifies any attack while verifying the protocol and provides the necessary steps to break it.

We provide the ProVerif code in Appendix A and on GitHub at https://github.com/DominikRoy/RACD. We verify the secrecy of x_i with ProVerif and ensure it cannot be retrieved by the verifier or extracted from the randomly generated variables r_i, v_i. ProVerif outputs *not attacker(element[]) is true* to signify that *element* cannot be retrieved by the adversary, and *not attacker(element[]) is false* if the adversary can retrieve it. Similarly, *weak secret(element[]) is true* signifies that the adversary cannot guess or brute-force *element*, while *weak secret(element[]) is false* means it is vulnerable to these attacks. ProVerif verifies message authenticity by defining event sequences such as (event(e_1)==>event(e_0)) to confirm the occurrence of e_0 before e_1. In our scheme, we define three events: (1) *verifiedAttestationResult* confirms the partial verifier has verified the disclosed events, (2) *sendAttestationResult* allows the attester to collect results from all partial verifiers and send them to the verifier, and (3) *trustable* indicates the attester device is trustworthy if all partial verifier attestation results are valid.

Listing 1: Verification results of ProVerif on RACD.

```
Weak secret x_i is true.
Query not attacker(x_i[]) is true.
Query not attacker(r_i[]) is true.
Query not attacker(v_i[]) is true.
Query event(trustable) ==> (event ↙
    (sendAttestationResult ↙
    (tpmQuote_signed,partialAttestationresults)) ↙
    ==> event(verifiedAttestationResult ↙
    (n,c_i',event_hash,result))) is true.
```

Listing 1 shows ProVerif's formal verification outcome of our protocol. The values of x_i, r_i, and v_i are not retrievable by the attacker (lines 1–4). Only the vendor knows the associated value x_i. The formal verification guarantees that the non-vendor event hashes do not reveal information to non-associated vendors (partial verifiers). ProVerif ensures that the event *verifiedAttestationResult* occurs before *sendAttestationResult* for the nonce, *eventhash*, and result, confirming the authenticity of *partialAttestationResults*. The verifier confirms the trustworthiness of the attester system only if all *partialAttestationResults* are valid. The order of events is crucial to ensure ProVerif's confirmation of the authenticity of the results and extend the trust base with the attester's device.

7.2 Performance Evaluation

We conducted experiments comparing our RACD approach with traditional remote attestation by running both attester and verifier applications on two Raspberry Pis (see Section 6 for details). Our aim was to measure the execution time of the attestation and verification processes on the same hardware platform. We focused exclusively on the incremental impact of RACD verification compared to traditional remote attestation process, as our main contribution and the most expensive operation is the Schnorr NIZK proof verification, which is performed for each entry in an event log.

Our experiments involved running 50 iterations with an event log size of 50 entries. The attestation process for both schemes was similar, with RACD taking on average 224 ms and traditional remote attestation taking 218 ms. The slight increase in RACD execution time was due to additional processing of the event log, such as creating the masked event log and identifying disclosed entries. However, the RACD verification process took much longer than traditional remote attestation, on average 178 ms compared to 13 ms. This was because the verifier had to verify the NIZK proof for each entry, which required significantly more computational steps.

Although TPM-based remote attestation with constrained disclosure comes at a cost, our approach enables scalability by allowing verifications to be distributed and performed simultaneously by multiple partial verifiers. As the number of partial verifiers increases, the time required for an entire verification process can decrease significantly. We did not consider communication costs in our experiments, as we did not observe a significant difference. Our results are depicted in Fig. 6, where the x-axis contains the respective iteration of the execution and the y-axis indicates the execution time in milliseconds.

7.3 Discussion

While initially using a nonce (as a salt or freshness key) might seem simpler and more elegant compared to our NIZK method, it comes with limitations. For instance, all verifiers must be online during the (measured) boot process to supply the nonce. Additionally, if the nonce is not selected carefully, it becomes vulnerable to rainbow table attacks. On the other hand, our NIZK approach offers the advantage of transferability, eliminating the need for verifiers to be online during boot-up. NIZK also securely proves the validity of the data without disclosing the underlying secret and allows for the verification of the obfuscated hash.

The computational burden associated with using NIZK for information security and limited disclosure is acknowledged. We also concur that increasing the volume of log entries would proportionally elevate the processing time. However, our analyses demonstrate that when parallelized across 1 to 50 partial verifiers—each handling 50 unique log entries—the computational time remains largely consistent. For example, Fig. 6 on the x-axis depicts a scenario involving 50 partial verifiers, each managing 50 individual log entries, culminating in a total of 2500 log entries. This is comparable to a basic Ubuntu system, which has between 600 and 800 files and binaries measured by Linux's IMA, not counting additional vendor applications. Jäger et al. [29] designed a software defined networking (SDN) node for Industrial Internet of Things (IIoT) applications. Their node features approximately 600 entries for Xen's Dom0 and about 1700 entries for Xen's DomU in the Linux IMA logs. These figures support the validity of our study, showing that our 2500 log entry evaluation closely aligns with or even replicates real-world configurations.

8 RELATED WORK

Debes et al. [15] present an approach called Oblivious Remote Attestation (ORA) to achieve zero-knowledge proof of integrity conformance. The authors emphasize the importance of privacy of the prover system's configuration and scalability in a multi-domain setting. To achieve this, they propose two additional protocols for provisioning and configuration updates, which are essential to ORA functioning. They also mandate the use of a TEE, an TPM NV Extend Index, and TPM enhanced authorization (EA) policies, as they state these measures are necessary to ensure a privacy-friendly attestation. Specifically, they limit access to an attestation key with TPM EA policies. In contrast, our RACD approach does not expand the Trusted Computing Base (TCB) by mandating the use of a TEE, TPM NV Extend Indexes, or TPM EA policies. Our protocol attains an equivalent level of security, without requiring the use of these supplementary measures, while extending its applicability to a broader range of devices. Furthermore, our protocol executes approximately 50 ms faster than theirs.

Sadeghi et al. [40] introduce property-based attestation (PBA) to address the privacy deficiency of binary attestation. The idea is to use a Trusted Third Party (TTP) to map platform configurations—i. e., binary measurements—to (security) properties, such as "booted up correctly" or "IDS is running". In contrast to our approach, a TTP is required. Chen et al. [13] extend the PBA protocol with a detailed design in theory that does not require a TTP. They apply ring signatures and commitments. Compared to our work, they rely

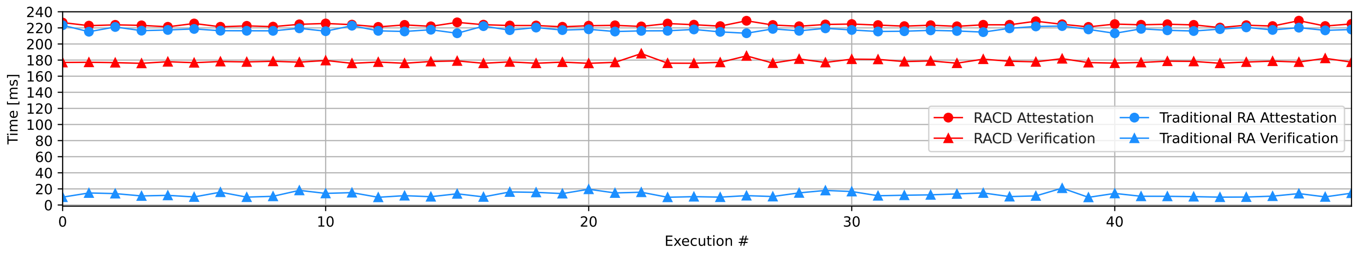

Figure 6: Execution time of attestation and verification processes for our RACD approach and traditional remote attestation approach, using 50 iterations and 50 log entries.

on a commitment scheme and require more computational steps. Further, their concept has two privacy limitations: (1) If the set of configurations is small, the verifier can guess configurations, and (2) multiple executions of the protocol with different configurations allows to find an intersection. Both [40] and [13] suggest designs but do not implement them. However, we design a practical approach, implement a PoC, and evaluate its security and performance.

Luo et al. [35] designed a privacy-preserving integrity measurement approach for containers. For each container, they measure all processes and generate a secret for each measurement (*PCR.secret*), and the secret is stored in the event logs. Each container log entry is *XOR'ed* with a corresponding *PCR.secret*. All *XOR'ed* values are folded into a folding hash that is extended into a specific PCR in the TPM. During remote attestation, the verifier receives the *PCR.secret*s for the corresponding container and the accumulated hash of a PCR signed by the TPM. The verifier applies the same procedure described above and compares the actual value with the received one. The state of the other containers and the host system is in the accumulated hash signed by the TPM. However, the verifier can only check the value of the corresponding container. Jie et. al [30] have a similar approach. Instead of the *XOR* operation, they concatenate the *template hash* with a random factor (*shielded factor*) that is transmitted in plain to the verifier. Both methods transfer their secrets to the verifier. If one binary measurement is sent to multiple verifiers, the uniqueness of the secret is not given. However, our work does not reveal the secret but proves that we have knowledge about it. Undeniably, approaches that use symmetric primitives offer performance benefits, but they also come with aforementioned downsides, which we account for in our approach.

Lauer et al. introduce a formal model and logic for containerized systems and their interactions [33]. Based on these, they design and verify a secure integrity measurement system for containerized systems. Since Linux IMA is incapable of creating domain/container specific integrity reports, they extend IMA in theory to achieve this functionality and gain additional desirable properties, such as measurement log stability and constrained disclosure for multi-domain systems. However, their approach requires each container (or domain) to be measured into a separate TPM PCR. Since a TPM typically has 24 PCRs—from which only 2–12 are available, depending on the platform configuration—, the number of containers that can be run and measured separately is very limited. For example, measured boot uses PCRs 0–7 (and also 8–9, depending on the configuration), Linux IMA by default uses PCR 10, Intel Trusted Execution Technology (TXT) [26] occupies PCR 17 for its Dynamic Root of Trust for Measurement (DRTM), and PCRs 16 and 23 are debug PCRs, i. e., they are intended for development and can be reset during runtime. The work from Lauer et al. can greatly benefit from our work by measuring multiple containers into a single TPM PCR and achieve the same properties for measurement logs: log stability and constrained disclosure.

Our RACD approach targets the secrecy of *binary measurements* during remote attestation. There exists the Direct Anonymous Attestation (DAA) protocol [11] to keep the *identity* of a TPM secret. However, DAA, Enhanced Privacy ID (EPID), etc., are orthogonal to our approach and can be used in conjunction with it.

9 CONCLUSION AND FUTURE WORK

In this paper, we showed how to realize constrained disclosure for TPM-based binary remote attestation data by applying a non-interactive zero-knowledge (NIZK) system using Schnorr signatures. We thoroughly analyzed security properties informally as well as formally with ProVerif. Based on our PoC implementation, we evaluated the performance and demonstrated its feasibility. We compared our approach with existing work and highlighted its uniqueness. As future work, we intend to further research the application of our RACD approach in the context of VMs and software containers, and plan to integrate it into IMA in the Linux kernel.

ACKNOWLEDGMENTS

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE [37]. ATHENE is funded jointly by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research and the Arts. Further, this research is directly supported by the German Federal Ministry of Education and Research in the context of the projects VE-ASCOT [51] (ID 16ME0274) and TRUSTnet [16] (ID 16KIS1787). This work was partially supported by the INTERSECT project, Grant No.NWA.1160.18.301, funded by the Dutch Research Council (NWO).

REFERENCES

[1] Advanced Micro Devices, Inc. 2023. *AMD Secure Encrypted Virtualization (SEV).* Advanced Micro Devices, Inc. https://www.amd.com/en/developer/sev.html

[2] Arm Ltd. 2021. *mbed TLS.* Arm Ltd. https://tls.mbed.org/

[3] Danesh Kumar Badlani and Adrian Diglio. 2022. *Microsoft open sources its software bill of materials (SBOM) generation tool.* Microsoft Corporation. https://devblogs.microsoft.com/engineering-at-microsoft/microsoft-open-sources-software-bill-of-materials-sbom-generation-tool/

[4] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–228.

[5] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-Speed High-Security Signatures. In *CHES (Lecture Notes in Computer Science, Vol. 6917).* Springer, 124–142. https://doi.org/10.1007/978-3-642-23951-9_9

[6] Daniel J. Bernstein and Tanja Lange. 2017. SafeCurves: Choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to

[7] Henk Birkholz, David Thaler, Michael Richardson, and Wei Pan. 2023. *Remote ATtestation procedureS (RATS) Architecture.* RFC 9334. RFC Editor. https://www.rfc-editor.org/rfc/rfc9334

[8] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. 1991. Noninteractive Zero-Knowledge. *SIAM J. Comput.* 20, 6 (1991), 1084–1118. https://epubs.siam.org/doi/10.1137/0220068

[9] Felix Bohling, Tobias Mueller, Michael Eckel, and Jens Lindemann. 2020. Subverting Linux' Integrity Measurement Architecture. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20).* Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3407023.3407058

[10] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. 2020. The Provable Security of Ed25519: Theory and Practice. http://eprint.iacr.org/2020/823

[11] Ernie Brickell, Jan Camenisch, and Liqun Chen. 2004. Direct Anonymous Attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04).* Association for Computing Machinery, New York, NY, USA, 132–145. https://doi.org/10.1145/1030083.1030103

[12] Dhiman Chakraborty, Lucjan Hanzlik, and Sven Bugiel. 2019. simTPM: User-centric TPM for Mobile Devices. In *28th USENIX Security Symposium (USENIX Security 19).* USENIX Association, Santa Clara, CA, 533–550. https://www.usenix.org/conference/usenixsecurity19/presentation/chakraborty

[13] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. 2008. Property-Based Attestation without a Trusted Third Party. In *Proceedings of the 11th International Conference on Information Security* (Taipei, Taiwan) *(ISC '08).* Springer-Verlag, Berlin, Heidelberg, 31–46. https://doi.org/10.1007/978-3-540-85886-7_3

[14] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of Remote Attestation. *Int. J. Inf. Secur.* 10, 2 (June 2011), 63–81. https://doi.org/10.1007/s10207-011-0124-7

[15] Heini Bergsson Debes and Thanassis Giannetsos. 2022. ZEKRO: Zero-Knowledge Proof of Integrity Conformance. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22).* Association for Computing Machinery, New York, NY, USA, Article 35, 10 pages. https://doi.org/10.1145/3538969.3539004

[16] DECOIT GmbH & Co. KG. 2023. *TRUSTnet Project.* DECOIT GmbH & Co. KG. https://trustnet-project.de/

[17] Frank Denis. 2021. Libsodium documentation. https://libsodium.gitbook.io/doc/

[18] Adrian Diglio. 2021. *Generating Software Bills of Materials (SBOMs) with SPDX at Microsoft.* Microsoft Corporation. https://devblogs.microsoft.com/engineering-at-microsoft/generating-software-bills-of-materials-sboms-with-spdx-at-microsoft/

[19] Executive Office of the President. 2021. *EO 14028: Improving the Nation's Cybersecurity.* Executive Office of the President. https://www.federalregister.gov/executive-order/14028

[20] Fraunhofer SIT. 2019. *CHARRA: CHAllenge-Response based Remote Attestation with TPM 2.0.* Fraunhofer SIT. https://github.com/Fraunhofer-SIT/charra

[21] Andreas Fuchs and Tadeusz Struk. 2021. tpm2-software/tpm2-tss. https://github.com/tpm2-software/tpm2-tss original-date: 2015-06-30T16:21:57Z.

[22] Dominik Roy George. 2021. *Privacy-Preserving Remote Attestation Protocol.* Master's Thesis. TU Darmstadt/TU Wien. https://doi.org/10.34726/hss.2021.86825

[23] Mike Hamburg. 2015. Decaf: Eliminating cofactors through point compression. http://eprint.iacr.org/2015/673

[24] Mike Hamburg, Henry de Valence, Isis Lovecruft, and Tony Arcieri. 2021. Ristretto - The Ristretto Group. https://ristretto.group/ristretto.html

[25] Feng Hao. 2017. Schnorr Non-interactive Zero-Knowledge Proof. RFC 8235. https://doi.org/10.17487/RFC8235

[26] Intel Corporation. 2022. *Intel Trusted Execution Technology (Intel TXT) Overview.* Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/tool/intel-trusted-execution-technology.html

[27] Intel Corporation. 2023. *Intel Software Guard Extensions (Intel SGX).* Intel Corporation. https://software.intel.com/sgx

[28] Intel Corporation. 2023. *Intel Trust Domain Extensions (Intel TDX).* Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[29] Lukas Jäger, Dominik Lorych, and Michael Eckel. 2022. A Resilient Network Node for the Industrial Internet of Things. In *Proceedings of the 17th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '22).* Association for Computing Machinery, New York, NY, USA, Article 100, 10 pages. https://doi.org/10.1145/3538969.3538989

[30] Li Shang Jie and He Ye Ping. 2010. A Privacy-Preserving Integrity Measurement Architecture. In *2010 Third International Symposium on Electronic Commerce and Security.* IEEE Transactions on Information Forensics and Security, 242–246. https://doi.org/10.1109/ISECS.2010.60

[31] Simon Josefsson and Jim Schaad. 2018. Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure. RFC 8410. https://doi.org/10.17487/RFC8410

[32] Obaid Khalid, Carsten Rolfes, and Andreas Ibing. 2013. On Implementing Trusted Boot for Embedded Systems. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST).* IEEE, Austin, TX, USA, 75–80. https://doi.org/10.1109/HST.2013.6581569

[33] Hagen Lauer, Amin Sakzad, Carsten Rudolph, and Surya Nepal. 2019. A Logic for Secure Stratified Systems and its Application to Containerized Systems. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE).* 562–569. https://doi.org/10.1109/TrustCom/BigDataSE.2019.00081

[34] Laurence Lundblade. 2021. *QCBOR: an implementation of nearly everything in RFC8949.* https://github.com/laurencelundblade/QCBOR

[35] Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu. 2019. Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019).* USENIX Association, Chaoyang District, Beijing, 487–500. https://www.usenix.org/conference/raid2019/presentation/luo

[36] Microsoft Corp. 2022. *Microsoft SBOM Tool (Open Source on GitHub).* Microsoft Corp. https://github.com/microsoft/sbom-tool

[37] National Research Center for Applied Cybersecurity ATHENE. 2019. *ATHENE – National Research Center for Applied Cybersecurity.* National Research Center for Applied Cybersecurity ATHENE. https://athene-center.de/

[38] Anil Rao. 2022. *Rising to the Challenge – Data Security with Intel Confidential Computing.* Intel Corporation. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141

[39] Felix Ronin. 2023. *Being a Responsible User and Creator of Open Source.* Adobe Inc. https://blog.developer.adobe.com/being-a-responsible-user-and-creator-of-open-source-bbbcb79857fd

[40] Ahmad-Reza Sadeghi and Christian Stüble. 2004. Property-Based Attestation for Computing Platforms: Caring about Properties, Not Mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms* (Nova Scotia, Canada) *(NSPW '04).* Association for Computing Machinery, New York, NY, USA, 67–77. https://doi.org/10.1145/1065907.1066038

[41] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. 2004. Attestation-Based Policy Enforcement for Remote Access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security - CCS '04.* ACM Press, Washington DC, USA, 308. https://doi.org/10.1145/1030083.1030125

[42] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium (SSYM'04, Vol. 13).* USENIX Association, San Diego, CA, USA, 16. https://www.usenix.org/conference/13th-usenix-security-symposium/design-and-implementation-tcg-based-integrity-measurement

[43] The United States Department of Commerce. 2021. *The Minimum Elements For a Software Bill of Materials (SBOM).* The United States Department of Commerce. https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf (Pursuant to Executive Order 14028 on Improving the Nation's Cybersecurity).

[44] Bill Toulas. 2022. *New Intel chips won't play Blu-ray disks due to SGX deprecation.* Bleeping Computer® LLC. https://www.bleepingcomputer.com/news/security/new-intel-chips-wont-play-blu-ray-disks-due-to-sgx-deprecation/

[45] Trusted Computing Group. 2014. *TPM 2.0 Mobile Reference Architecture Specification* (family 2.0, level 00, revision 142 ed.). Trusted Computing Group. https://trustedcomputinggroup.org/resource/tpm-2-0-mobile-reference-architecture-specification/

[46] Trusted Computing Group. 2015. *TPM 2.0 Mobile Common Profile* (family 2.0, level 00, revision 31 ed.). Trusted Computing Group. https://trustedcomputinggroup.org/resource/tcg-tpm-2-0-mobile-common-profile/

[47] Trusted Computing Group. 2019. *TCG Trusted Attestation Protocol (TAP) Information Model for TPM Families 1.2 and 2.0 and DICE Family 1.0* (version 1.0

revision 0.36 ed.). Trusted Computing Group. https://trustedcomputinggroup.org/resource/tcg-tap-information-model/
[48] Trusted Computing Group. 2019. *TCG Trusted Attestation Protocol (TAP) Use Cases for TPM Families 1.2 and 2.0 and DICE* (version 1.0 revision 0.35 ed.). Trusted Computing Group. https://trustedcomputinggroup.org/resource/tcg-trusted-attestation-protocol-tap-use-cases-for-tpm-families-1-2-and-2-0-and-dice/
[49] Trusted Computing Group. 2019. TCG TSS 2.0 Overview and Common Structures Specification.
[50] Trusted Computing Group 2019. *Trusted Platform Module Library – Part 1: Architecture* (family 2.0, level 00, revision 01.59 ed.). Trusted Computing Group. https://trustedcomputinggroup.org/resource/tpm-library-specification/
[51] WIBU-SYSTEMS AG. 2021. *VE-ASCOT—Advanced Security for Chains of Trust.* WIBU-SYSTEMS AG. http://ascot-trust.info/
[52] Mimi Zohar and Dmitry Kasatkin. 2018. Integrity Measurement Architecture (IMA). https://sourceforge.net/p/linux-ima/wiki/Home/
[53] Mimi Zohar, David Safford, and Reiner Sailer. 2009. Using IMA for Integrity Measurement and Attestation. https://blog.linuxplumbersconf.org/2009/slides/David-Stafford-IMA_LPC.pdf

A PROVERIF CODE

Listing 2 provides the ProVerif code of our RACD protocol.

Listing 2: RACD ProVerif code.

```
(*Dolev-Yao model Open Channel*)
free c:channel.
type list.
type tuple.
type none.
type nonce.
type skey.
type pkey.
type result.
type key.
free x_i: bitstring [private].
weaksecret x_i.


(* Randomness generated by Prover *)
free r_i: bitstring [private].
free v_i: bitstring [private].


(*noninterf r_i;
noninterf v_i;*)
(* Elliptic Curve *)
type G.
type L.


(*free nB: N [data].*)
(* Auxiliary Functions *)
fun templatehash(bitstring):bitstring.
fun mod(bitstring,L):bitstring.
fun mul(bitstring,bitstring):bitstring.
fun point_mul(G,G):G.
fun hash(G,G,G):bitstring.
fun map(bitstring):bitstring. (*secure function of 2H(x)+1*)
fun append(G,G,G):bitstring.
fun exp(G,bitstring):G.
fun sub(bitstring,bitstring):bitstring.
fun tpm_pcr_extend(bitstring,G):none.
fun ima_cd(bitstring,G,bitstring,bitstring):none.
fun ima_cd_event():G.
fun ima_cd_s():bitstring.
fun ima_cd_c():bitstring.
fun requestTPMQuote():bitstring.
fun hash_chain(G):bitstring.
fun retrieve_all():bitstring.
fun collect_results(G,bool):list.
fun retrieve_results(list):bitstring.
fun retrieve_event(list):G.
(* Public key Cryptography *)
fun pk(skey): pkey.


(* Signatures *)
fun ok () : result .
fun sign ( bitstring , skey ) : bitstring .
reduc forall m : bitstring, sk : skey ; getmess(sign(m, sk)) = m .
reduc forall m : bitstring, sk : skey ; checksign(sign(m, sk), pk(sk)) = ok() .


(*Events*)
event secureboot().
event requestAttestation(nonce).
event acceptAttestationRequest(nonce). (*attester*)
event sendAttestationResult(bitstring,list). (*attester*)
event requestpartialVerification(nonce,G, bitstring,bitstring,bitstring). (*attester*)
event verifiedAttestationResult(bitstring, G,bool). (*partialverifer*)
(*event failedAttestationResult(pkey, bitstring,bool).*)
event trustable(). (*verifier*)

(* A formal query, specifying the attacker can't ever be leaked the actual binary and the randomness during the protocol. *)
query attacker(x_i).
query attacker(r_i).
query attacker(v_i).


query pk:pkey, n:nonce, event_hash:G, c_i:bitstring,c_i':bitstring, tpmQuote_signed:bitstring, result:bool, partialAttestationresults:list;
event(trustable()) ==> (event(sendAttestationResult( tpmQuote_signed,partialAttestationresults)) ==> event(verifiedAttestationResult( tpmQuote_signed, event_hash,result))).


noninterf x_i among (r_i,v_i,   ima_cd_c(), ima_cd_s()).

let verifier(pk:pkey) =
    new n:nonce;
    (*event requestAttestation(n);*)
    out(c,n);

    in (c,(tpmQuote_signed:bitstring, ↙
        partialAttestationresults:list));
    let result' = ↙
        retrieve_results(partialAttestationresults) ↙
        in
    let event_hash = ↙
        retrieve_event(partialAttestationresults) ↙
        in
    let hash_chained = hash_chain(event_hash) in
    new valid:bitstring;
    let (hash_chained':bitstring, n':nonce) = ↙
        getmess(tpmQuote_signed)in
    if checksign(tpmQuote_signed,pk) = ok() then
        if n' =n && hash_chained' = ↙
            hash_chained && result' = valid ↙
            then (
            event trustable()).


let attester(index:bitstring, pk:pkey, sk:skey, ↙
    g:G,odr:L) =
    let event_hash = exp(g,mod(mul(r_i, ↙
        mod(templatehash(x_i),odr)),odr)) in
    let g_i = exp(g, ↙
        mod(templatehash(x_i),odr)) in
    let t_i = exp(g_i,v_i) in
    let c_i = hash(g_i,t_i,event_hash) in
    let s_i = mod(sub(v_i,mod(mul(r_i, ↙
        mod(c_i,odr)),odr)),odr) in
    (*event secureboot();*)
    let pcr = tpm_pcr_extend(index,event_hash)in
    let ima = ima_cd(index,event_hash,c_i,s_i)in


    in(c,n:nonce);
    (*event acceptAttestationRequest(n);*)
    let tpmQuote = requestTPMQuote() in
    let tpmQuote_signed = sign(tpmQuote,sk) in
    let event_hash_ima = ima_cd_event() in
    let s_i_ima = ima_cd_s() in
    let c_i_ima= ima_cd_c() in
    (*event requestpartialVerification(n, ↙
        event_hash_ima,c_i_ima,s_i_ima, ↙
        tpmQuote_signed);*)
    out (c,(event_hash_ima,c_i_ima,s_i_ima,n, ↙
        tpmQuote_signed));

    in (c,(tpmQuote_signed':bitstring, ↙
        event_hash':G,attresult:bool));
    let partialAttestationresults = ↙
        collect_results(event_hash',attresult) in
    event ↙
        sendAttestationResult(tpmQuote_signed', ↙
        partialAttestationresults);
    out(c,(tpmQuote_signed', ↙
        partialAttestationresults)).

let partialVerifier(pk:pkey, g:G, odr:L)=
    in(c, (event_hash_ima:G,c_i_ima:bitstring, ↙
        s_i_ima:bitstring,n:nonce, ↙
        tpmQuote_signed:bitstring));
    let g_i = exp(g, ↙
        mod(templatehash(x_i),odr)) in
    let g_i_s = exp(g_i,s_i_ima) in
    let event_hash_c = ↙
        exp(event_hash_ima,mod(c_i_ima,odr)) in
    let t_i' = point_mul(g_i_s,event_hash_c) in
    let c_i' = hash(g_i,t_i',event_hash_ima) in
    let hash_chained = ↙
        hash_chain(event_hash_ima) in
    let (hash_chained':bitstring, n':nonce) = ↙
        getmess(tpmQuote_signed)in
    if checksign(tpmQuote_signed,pk) = ok() then
        if c_i' = c_i_ima && n' = n && ↙
            hash_chained' = hash_chained then (
            event verifiedAttestationResult( ↙
                tpmQuote_signed,event_hash_ima, ↙
                true);
            out(c,(tpmQuote_signed, ↙
                event_hash_ima,true))).



process
    new sk:skey;
    let pkey = pk(sk) in
    new index:bitstring;
    new g: G;
    new odr:L;
    ((!verifier(pkey)) | ↙
        (!attester(index,pkey,sk,g,odr)) | ↙
        (!partialVerifier(pkey,g,odr)) )
```