Lightweight Attestation and Secure Code Update for Multiple Separated Microkernel Tasks

Steffen Wagner(✉), Christoph Krauß, and Claudia Eckert

Fraunhofer Institute AISEC, Munich, Germany
{steffen.wagner,christoph.krauss,claudia.eckert}@aisec.fraunhofer.de

Abstract. By implementing all non-essential operating system services as user space tasks and strictly separating those tasks, a microkernel can effectively increase system security. However, the isolation of tasks does not necessarily imply their trustworthiness. In this paper, we propose a microkernel-based system architecture enhanced with a multi-context hardware security module (HSM) that enables an integrity verification, anomaly detection, and efficient lightweight attestation of multiple separated tasks. Our attestation protocol, which we formally verified using the automated reasoning tool *ProVerif*, implicitly proves the integrity of multiple tasks, efficiently communicates the result to a remote verifier, and enables a secure update protocol without the need for digital signatures that require computationally expensive operations.

Keywords: Lightweight attestation · Microkernel tasks · Multi-context hardware security module · Trusted platform module

1 Introduction

To increase and ensure safety and security, microkernel-based systems provide separation mechanisms to isolate individual tasks from the rest of the system. The system enforces this strict separation by partitioning resources, e.g., CPU time or physical memory, and by virtualizing address spaces and devices. In addition, a microkernel such as *L4* [7] is very small in terms of code size and less complex compared to monolithic kernels, hence considered more trustworthy. However, a strong separation of potentially complex software components by a trusted microkernel does not necessarily imply the trustworthiness of the isolated tasks, which is a desirable property in most security-critical systems.

One approach to verify the trustworthiness of software components takes advantage of a hardware security module (HSM), such as a Trusted Platform Module (TPM) [14]. A TPM provides a cryptographic context and mechanisms to securely store integrity measurements, create (a)symmetric keys, and perform certain cryptographic operations, such as encryption. For a remote attestation, for example, load-time integrity measurements are signed with a private key inside the TPM and sent to a remote verifier together with a stored measurement log (SML) in order to prove the integrity of a system. However, since those digital

Y. Desmedt (Ed.): ISC 2013, LNCS 7807, pp. 20–36, 2015.
DOI: 10.1007/978-3-319-27659-5_2

signatures are based on asymmetric cryptography, more precisely RSA (with at least 2048 bit keys), they are quite large and rather expensive[1], even though the TPM includes dedicated cryptographic engines for signature calculation. The reason is that the TPM was never intended to be and, in general, does not act as a cryptographic accelerator. In addition, the TPM was also not designed to handle run-time integrity values, such as events or behavior scores generated by an anomaly detection, which thus cannot be used for a remote attestation.

Furthermore, TPMs do not support virtualization natively, since they only provide one cryptographic context for system-wide load-time integrity measurements and keys. That is why most existing concepts rely on the virtual machine monitor, also known as hypervisor, to virtualize the TPM [1,3,12]. However, as a consequence of the implementation in software, cryptographic secrets, e.g., keys, or integrity measurement values are not always handled inside the TPM. As a result, recent efforts explored and showed the feasibility to realize and manage multiple TPM contexts in hardware [4]. With multiple individual cryptographic contexts, a TPM-based HSM can provide each task with its own security context, which can be used to securely store, for example, keys and integrity measurements on a per-task basis. However, as a consequence of the isolated contexts, the number of digital signatures (and SMLs) in a remote attestation as specified by the Trusted Computing Group (TCG) increases with the number of contexts (i.e., tasks), which makes classical attestation even more expensive and also inefficient, especially on resource-constrained devices, e.g., smartphones.

To overcome these challenges, we first propose a microkernel-based architecture with an integrity verification and anomaly detection component that is enhanced with a multi-context HSM. Within the HSM, the load-time integrity values and events collected by an anomaly detection during run-time are stored in distinct contexts, so that task-specific keys, for instance, can be cryptographically bound to these values. As our main contribution, we then propose and formally verify a lightweight attestation mechanism, which mainly relies on symmetric cryptography and, thus, is able to efficiently verify multiple tasks in a microkernel-based system. Additionally, our attestation protocol is designed to enable secure code updates based on the integrity of existing security-critical tasks while eliminating the need for digital signatures.

The rest of the paper is structured as follows. In Sect. 2, we discuss related work on remote attestation with a focus on hardware-based attestation. In Sect. 3, we explain our scenario and attacker model. Section 4 describes the system architecture for the attestation and code update presented in Sect. 5. In Sect. 6, we analyze the security of our protocols. Finally, Sect. 7 concludes the paper.

2 Related Work

For a *hash-based remote attestation* as specified by the TCG [14], static load-time hash values of the components comprising the relevant software stack are

[1] That is because of the exponentiation operations used in RSA's encryption and compared to symmetric cryptography.

calculated during *authenticated boot* starting from a Core Root of Trust for Measurement (CRTM). That means the current software component in the boot chain measures the next one before executing it. Later, these integrity measurement values stored inside the TPM are signed and sent together with a SML to the remote verifier. However, since this approach primarily focuses on load-time integrity measurements for software binaries only, other schemes, such as *property-based* [8], *semantic* [5], or *logic-based attestation* [11], try to extend and generalize the attestation mechanism. For instance, property-based attestation aims at proving certain security characteristics rather than the integrity of certain software binaries. However, most of these attestation protocols still rely on quite expensive cryptographic operations—more precisely, digital signatures—which make them less suitable for an attestation of multiple separated tasks in a virtualized embedded system with limited resources, such as smartphones.

As a result, more recent efforts [10] proposed to increase the performance and efficiency by implementing a proxy component on the prover's system, which verifies that system locally and, thus, reduces the time between the attestation and the verification of the result. However, this approach still relies on traditional attestation mechanisms based on digital signatures to verify the hypervisor, virtualized device drivers, and the proxy component. Our mechanism, in comparison, focuses on a lightweight attestation, which relies on symmetric cryptographic operations rather than digital signatures and only requires very small messages to prove the integrity of multiple tasks including the microkernel.

3 Attestation Scenario and Attacker Model

In this section, we describe the scenario, which captures the settings for our attestation and secure code update protocol. We also specify the attacker model.

3.1 Scenario for the Attestation of Multiple Tasks

For our attestation and secure code update protocol, we define a ($\mathcal{P}$) and a ($\mathcal{V}$). The prover is a microkernel-based embedded system with different security- or safety-critical applications, such as a smartphone or a vehicle. $\mathcal{V}$, on the other hand, is a remote verifier, which is considered honest and trustworthy.

Without any loss of generality, we assume that the prover is a smartphone with a microkernel-based system architecture, which can execute various tasks in isolated virtualized environments. Typical tasks are the *baseband stack* for communications with a mobile network, *virtualized device drivers*, native tasks for *security-critical applications*, such as a secure email or VPN client, and *regular user applications* running on a rich *operating system*, e.g., a Linux-based Android. Since those tasks have different levels of criticality, they are strictly isolated by the separation mechanisms of the microkernel. However, an attacker might still be able to compromise tasks, e.g., by fuzzing their interfaces. That is why the prover has to provide verifiable evidence for the integrity of relevant security-critical tasks before the verifier grants access to restricted resources, such as emails, confidential documents, or updates for business applications.

For example, in our scenario, the smartphone might regularly connect to a company network via VPN, whenever the user needs access to company-internal resources. To establish a secure connection, the verifier in the company network first requires proof for the integrity of security-critical tasks, such as certain device drivers and the VPN client. Based on the attestation result, access to the company network is granted or denied. In case access is denied, e.g., because the VPN client was compromised, the verifier should be able to provide a code update based on the integrity of only the most basic security-critical tasks, such as the microkernel and the baseband stack. That way the prover is able to recover.

3.2 Attacker Model

In our scenario, the ($\mathcal{A}$) can read and manipulate messages between the prover and the verifier as long as they are not encrypted or otherwise protected. More precisely, an attacker cannot decrypt an encrypted message or forge a correct message authentication code (MAC) for a modified message without the correct key. The attacker is also not able to invert cryptographic hash functions.

Additionally, as specified for most remote attestation protocols, we assume that hardware attacks are not possible. In particular, the security mechanisms of the HSM cannot be compromised by an attacker. That means we assume that the implementation of hardware-based security features, e.g., cryptographic engines, and firmware implemented in software are correct.

4 Microkernel-Based System Architecture with a Multi-context HSM

In security-critical systems such as described in the scenario, microkernel-based virtualization provides the necessary means to safely execute multiple tasks with different levels of criticality on the same hardware. Nevertheless, most critical systems require hardware-based security mechanisms, since they need to securely store cryptographic secrets, such as private keys, or integrity measurements. That is why those systems are often equipped with a HSM, which acts as a hardware-based security anchor and usually provides an internal context for the system-specific cryptographic information. However, most HSMs with a system-wide cryptographic context, such as a TPM, are not designed to separate and isolate security-sensitive information of individual tasks.

For this purpose, we propose a microkernel-based system architecture enhanced with an HSM that supports virtualization by providing multiple task-specific cryptographic contexts. As depicted in Fig. 1, which shows the system architecture that we designed and implemented using the *L4*-based *PikeOS* [13], safety- and security-critical processes are realized as *native tasks*, whereas other non-critical components might be *POSIX tasks* or a *virtualized Linux instance*. All these individual tasks are isolated by the separation mechanisms of the kernel. However, the tasks are still able to communicate with other tasks in a controlled way, i.e., via kernel-based inter process communication (IPC) and individually

distinct shared memory pages. The HSM proxy, for instance, receives commands via IPC and shared memory. It identifies the origin of the command, forwards it to the HSM, and receives the result, which in turn is communicated back to the respective source. Since all IPC channels and shared memory pages are isolated by the kernel, other tasks cannot read the commands or the result messages.

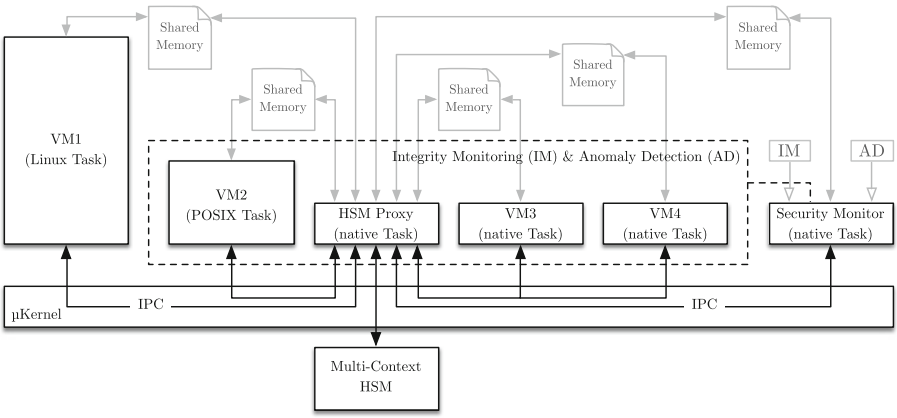

Fig. 1. Microkernel-based system architecture with multi-context HSM

There is, however, one exception: the *security monitor*. This component is one of the early tasks, which can start other tasks and is able to *measure the integrity* of a selection of tasks as indicated by the dashed box in Fig. 1. Since the security monitor, which is a critical task and hence realized as a native microkernel task, holds advanced capabilities, e.g., the right to directly access the memory of other tasks, it can effectively monitor their behavior and *detect anomalies*. As a result, the security monitor can store the integrity values measured at load-time in the task-specific context as well as keep a log of detected anomalies inside the HSM.

The design of our multi-context HSM, which is schematically depicted in Fig. 2, implements the functionality of a TPM and includes hardware-based security features, such as protected memory, a true random number generator (TRNG), and cryptographic engines for hash functions, MACs, and encryption algorithms like RSA or elliptic curve cryptography (ECC).

Based on the hardware components, the HSM firmware realizes *scheduling*, *multiplexing*, and *prioritizing of separate contexts*, which can handle cryptographic keys, store integrity measurements in shared and individual platform configuration registers (PCRs), and log atypical run-time events in per-context anomaly detection records (ADRs). That way each task can have its individual key hierarchy, anomaly detection status, and its very own set of hardware-based integrity measurement registers. However, the HSM also allows to (physically) share certain PCRs, e.g., to store common integrity measurements for the boot loader or the microkernel, in order to make the HSM design more efficient.

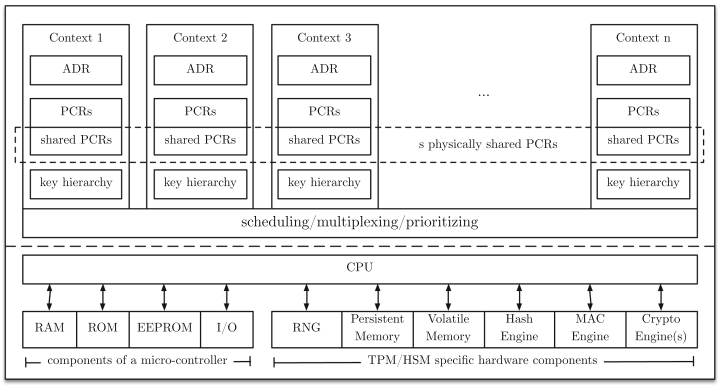

Fig. 2. Design of a multi-context HSM

5 Integrity Verification of Multiple Microkernel Tasks as Basis for a Secure Code Update

Based on the system architecture, we present our approach to implicitly verify the integrity of multiple separated microkernel tasks and communicate the result to a remote verifier without the need for expensive cryptographic operations. The attestation mechanism enables a secure code update of tasks based on the integrity of the system, in particular the microkernel and existing tasks.

The main idea of the attestation mechanism is to verify the integrity of a number of tasks locally rather than sending digitally signed integrity values to a remote verifier, which then has to check the signatures and evaluate the integrity values. Our attestation protocol, which we previously proposed for non-virtualized systems in [15], instead verifies the trustworthiness of tasks by loading task-specific keys into the key slots inside the HSM. This load operation is only possible if the specified tasks have not been tampered with, because the keys have been cryptographically bound to the correct integrity measurements of the tasks and their typical behavior, which is monitored by the anomaly detection component of the security task.

5.1 Notation

A *message authentication code* (*MAC*) is a cryptographic value, which is based on a shared symmetric key and can be used to verify the authenticity and integrity of a message. Formally, a MAC algorithm is a function that calculates a message digest dig with fixed length l for a secret key K and a given input m with virtually arbitrary size as $MAC(K, m) = dig = \{0,1\}^l$.

One method to construct a MAC algorithm is based on cryptographic hash functions, which are one-way functions with collision and pre-image resistance. Essentially, a hash function H compresses arbitrary-length input to an output with length l, that is $H : \{0,1\}^* \rightarrow \{0,1\}^l$. As an example for a hash-based

MAC, a *HMAC* generates a message authentication digest for data m based on key K as $HMAC(K, m) = H((K \oplus opad) \,||\, H((K \oplus ipad) \,||\, m))$, where $||$ denotes a concatenation, $\oplus$ the exclusive or, *opad* the outer and *ipad* the inner padding.

For our attestation protocol, we presume that the load-time integrity of a microkernel task can be adequately described by a set of measurement values, which are securely stored in the PCRs of our multi-context HSM. Thus, the PCRs values, which cryptographically represent the task c, are referred to as *platform configuration* $P_c := (PCR_c[i_1], \ldots, PCR_c[i_k])$, where $i \in \{0 \ldots r\}$ and r is the number of physically available PCRs. To store an integrity measurement value μ in a PCR with index i that belongs to task/context c, the current value is combined with the new measurement value using $PCRExtend(PCR_c[i], \mu)$, which is specified as $PCR_c[i] \leftarrow SHA1(PCR_c[i] \,||\, \mu)$ [14].

In addition to the load-time integrity measurements, the security task also monitors the run-time behavior of critical tasks. Based on machine learning algorithms [17,18], the anomaly detection component of the security task monitors, for instance, the order of system calls and assesses the probability for an attack. In case of an anomaly, an event $e = (m, p)$ is recorded by the security task using $ADRAdd(A_c, e)$, which securely stores a log message m in the ADR of task c and increases the *probability for an attack* stored in A_c with the value p. If the value in A_c exceeds a threshold T_c, the task must be considered compromised. In order to compensate for false-positives, the probability decreases over time very slightly, which might, however, be a security weakness and needs further research. As a consequence, we need to exclude false positives for now.

To cryptographically bind a key to a particular system state, which is also known as *wrapping*, the HSM links the key to the specified platform configuration and encrypts it with a public key (pk). A wrapped key, which is bound to a specific platform configuration P and encrypted with pk, is denoted as $\{K\}^{P}_{pk}$. This key can only be decrypted with the secret key (sk) and used by the HSM, if and only if the correct authentication value for the public wrapping key is provided and the current platform configuration P' equals the configuration P, which was specified when the key was wrapped. We extend this definition by also binding the wrapped key to an anomaly detection probability threshold T_c, that is $\{K\}^{P_c, T_c}_{pk}$. To load this wrapped key, the security monitor must also verify that the current probability stored in A_c is below T_c, which allows for more dynamic run-time verifications.

5.2 Cryptographic Keys

For each context c, an *integrity key* pair $K^{\text{int}}_{\text{c}} = (pk^{\text{int}}_c, sk^{\text{int}}_c)$ is defined that needs to be loaded into the HSM for a successful attestation of the corresponding task. As shown in Fig. 3, the integrity keys are encrypted with the public portion of a shared non-migratable *wrapping key* $K^{\text{wrap}} = (pk^{\text{wrap}}, sk^{\text{wrap}})$ and cryptographically bound to a trusted platform configuration P_c and an anomaly detection probability threshold T_c, which is denoted as $\{K^{\text{int}}_{\text{c}}\}^{P_c, T_c}_{pk^{\text{wrap}}}$. The necessary authentication value, $Auth^{\text{wrap}}$, for loading the wrapping key is only known to the HSM and the remote verifier. The verifier also has access to the

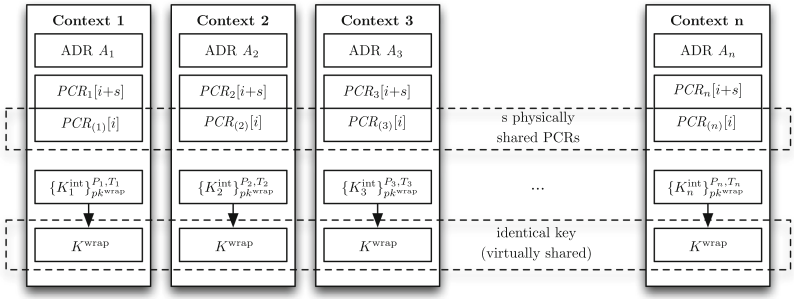

Fig. 3. Cryptographic keys, PCRs, and ADRs for multiple separated contexts

hash values of the wrapped integrity keys, which are combined with an *LoadKey* ordinal (*load*) and are denoted as $SHA1(load \,||\, \{K_{\text{c}}^{\text{int}}\}_{pk^{\text{wrap}}}^{P_c,T_c})$.

5.3 Integrity Verification and Attestation of Multiple Tasks

To verify the integrity of one or more microkernel tasks, a remote verifier $\mathcal{V}$ sends an *attestation request (req)* to the prover $\mathcal{P}$ as depicted in Fig. 4. The attestation request specifies k out of n microkernel tasks, which should be included into the attestation procedure, i.e., $req = \{c_i\}$ with $c, i \in \{1, \ldots, n\}$ and $|req| = k$.

Based on the request, $\mathcal{P}$ first transmits a set of random numbers $nonce_{\mathcal{P}_c}$, which are specifically calculated by the HSM for the selected tasks with context c. The random numbers are required to generate the authentication values $Auth_c$ and prevent replay attacks. $\mathcal{V}$ calculates the set of authentication values as

$$Auth_c = HMAC\big(SHA1(load \,||\, \{K_{\text{c}}^{\text{int}}\}_{pk^{\text{wrap}}}^{P_c,T_c}) \,||\, nonce_{\mathcal{P}_c} \,||\, nonce_{\mathcal{V}}, Auth^{\text{wrap}}\big), \quad (1)$$

where *load* is the ordinal for the *LoadKey* operation and $nonce_{\mathcal{V}}$ is a random number selected by the verifier (Fig. 4, step 1). This calculation of $Auth_c$ follows the authentication for TPM commands as specified by the TCG, more precisely for the load command [14, p. 72]. The authentication values $Auth_c$ and $nonce_{\mathcal{V}}$ are then sent to $\mathcal{P}$, which is now able to generate the command to load the task-specific integrity keys $K_{\text{c}}^{\text{int}}$. It is important to note that $\mathcal{P}$ has neither knowledge about nor access to $Auth^{\text{wrap}}$, which is only known to $\mathcal{V}$ and the HSM. As a consequence, $\mathcal{P}$ is not able to calculate the authentication HMACs without $\mathcal{V}$.

To load the keys $K_{\text{c}}^{\text{int}}$ into the HSM and, thereby, generate the implicit proof for the trustworthiness of the corresponding tasks, $\mathcal{P}$ simply needs to generate a *LoadKey* command per context as shown in Fig. 4. When the HSM receives a command for context c, it verifies the authentication value $Auth_c$ and compares the current platform configuration P_c' with P_c, which has been specified when the integrity key was wrapped (Fig. 4, step 2). It also checks the anomaly detection record for any log entries (more precisely, whether the current probability value in A_c is above the probability threshold T_c specified during the wrapping step), which might indicate that the task was compromised during run-time. If the

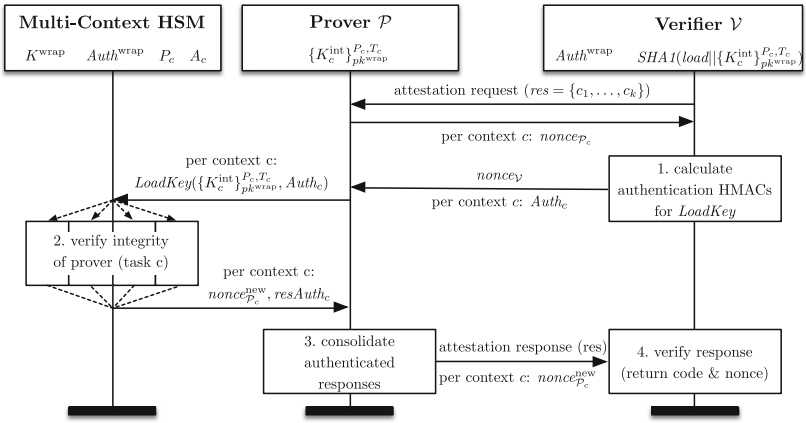

Fig. 4. Attestation protocol for multiple separated tasks

verification is successful and no anomalies were detected, the key is decrypted and loaded. However, if the key is already loaded into a key slot, an efficient HSM only verifies the wrapping conditions and omits the decryption.

After a successful load operation, the HSM generates HMAC-protected result messages and a set of new random numbers $nonce_{\mathcal{P}_c}^{\text{new}}$ for each task c. The result messages mainly include a *return code (rc)*, which indicates the success of the load operation, the fixed command ordinal for the *LoadKey* operation (*load*), and the nonce selected by the verifier ($nonce_{\mathcal{V}}$). Both values are protected by an HMAC, which is denoted as $resAuth_c$ and calculated as

$$resAuth_c = HMAC\big(SHA1(rc \,||\, load) \,||\, nonce_{\mathcal{P}}^{new} \,||\, nonce_{\mathcal{V}}, Auth^{\text{wrap}}\big). \quad (2)$$

Again, it is important to note that the hash-based message authentication code (HMACs) are calculated based on $Auth^{\text{wrap}}$, which is the shared authentication value between the HSM and the verifier $\mathcal{V}$. Before the prover $\mathcal{P}$ sends the new random numbers and the HMACs $resAuth_c$, which carry implicit proof that the attested tasks are still trustworthy, to $\mathcal{V}$, the prover $\mathcal{P}$ can reduce the size of the *attestation response message (res)* by hashing the HMACs (Fig. 4, step 3), i.e.,

$$res = H(resAuth_c). \quad (3)$$

As a consequence, the efficiency of the transmitted attestation result can be increased without losing cryptographic information needed to verify the integrity of the selected tasks. However, if the attestation fail, the prover can also send the individual HMACs in a second attempt in order to allow for a recovery. In that case, the verifier only considers the most basic security-critical tasks and, in our scenario, grants access to a trusted version of the compromised software component as a failsafe. This recovery version can be provided using the secure code update protocol, which is described in more detail in the next section.

Finally, $\mathcal{V}$ verifies the response *res* by comparing it with a freshly generated hash res' (Fig. 4, step 4). First, $\mathcal{V}$ calculates a hash value $SHA1((rc'=0) \,||\, load)$, where $\mathcal{V}$ assumes that the load operation was successful, i.e., the return code, here rc', must be `SUCCESS`, which is defined as zero. Corresponding with Eq. 2, the hash value is then used to freshly calculate the task-specific HMACs based on the random numbers and the authentication value $Auth^{\text{wrap}}$ as

$$resAuth'_c = HMAC\big(SHA1((rc') \,||\, load) \,||\, nonce_{\mathcal{P}}^{new} \,||\, nonce_{\mathcal{V}}, Auth^{\text{wrap}}\big). \quad (4)$$

By hashing those values, i.e., $res' = H(resAuth'_c)\ \forall c \in req$ (cf. Equation 3), and comparing it to the attestation response, $\mathcal{V}$ can verify the implicit proof for the integrity of the selected tasks. The verifier can trust the individual attestation results, because they are protected by an HMAC and the shared secret $Auth^{\text{wrap}}$, which is only known to the HSM and $\mathcal{V}$. In addition, the messages also include random numbers, which protect against replay attacks by providing verifiable proof that the attestation response message is indeed fresh.

5.4 Updating a Task After Verifying the Integrity of Existing Tasks

Based on the attestation protocol for existing microkernel tasks, we now describe our code update protocol, which allows to *update a task*, *creates verifiable proof*, and *maintains the ability to attest* both, the tasks and the system.

The main idea of the update protocol is that the prover creates and loads a new integrity key, which is specific to the new task, i.e., wrapped to the integrity values of the code update and the corresponding request. To ensure that the key is actually wrapped to the correct integrity measurements, the verifier provides a specific cryptographic authorization value, which only the verifier can calculate. When the HSM receives this authorization, it checks whether the new key will be wrapped to the correct integrity values before creating and wrapping the key. In combination with an efficient attestation of the code update, the verifier can ensure the authenticity of the initial update request and has verifiable evidence for the transaction. The prover, on the other hand, can use the new integrity key to create proof of a successful load operation in order to obtain the code update.

For a code update (U), the prover $\mathcal{P}$ initiates the protocol by sending a request (req) to the verifier $\mathcal{V}$ as depicted in Fig. 5 (step 1). Apart from the requested software update identified by ID_U, the message includes the cryptographic values to generate the authorization value referred to as $pubAuth_U$, which is used to create the new wrapped integrity key $K_{\text{U}}^{\text{int}}$. In more detail, the prover encrypts the authentication value to *use* and *migrate* the new integrity key, which are denoted as *usageAuth* (*uA*) and *migrationAuth* (*mA*) following the TCG specification [14]. The request also includes a nonce, which we refer to as $nonce_{\mathcal{P}}$ or $n_{\mathcal{P}}$ for short.

When the verifier receives the request from the prover $\mathcal{P}$, we assume that $\mathcal{V}$ might require to verify the trustworthiness of existing tasks for safety and security reasons before allowing to update or install a new software component. That is why $\mathcal{V}$ first initiates an attestation, which checks the authenticity and integrity of tasks running on $\mathcal{P}$'s system as described in the previous section.

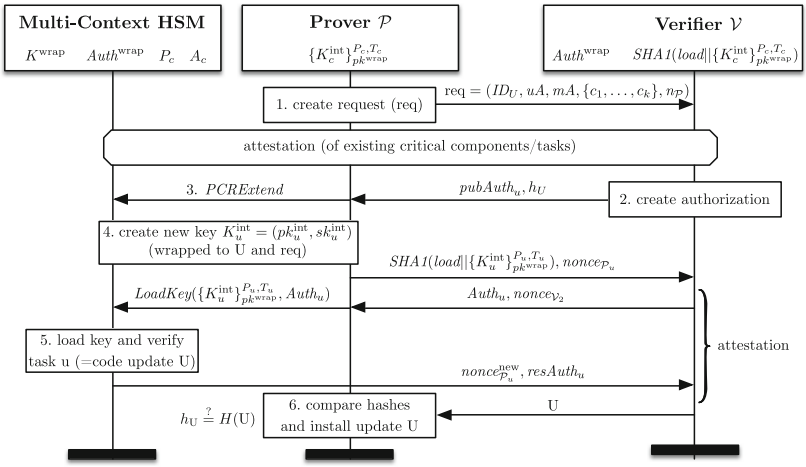

Fig. 5. Secure code update protocol

After a successful attestation of the relevant tasks, the verifier returns the encrypted authorization HMAC $pubAuth_U$ as well as a hash of the code update, denoted as h_U (Fig. 5, step 2). The HMAC authorizes the creation of the new integrity key and is calculated as

$$pubAuth_U = HMAC\big(sha1 \,||\, nonce_{\mathcal{P}} \,||\, nonce_{\mathcal{V}_1}, Auth^{\text{wrap}}\big). \tag{5}$$

In this equation the hash value *sha1* is generated as

$$sha1 = SHA1(wrap \,||\, usageAuth \,||\, migrationAuth \,||\, keyInfo), \tag{6}$$

where *wrap* is the fixed ordinal for the command, which creates a wrapped key. The structure *keyInfo* defines the cryptographic properties of the new key and follows the specification by the TCG [14, Part 2 Structures, p. 89]. In particular, the structure specifies the trusted platform configuration the key is wrapped to, which includes at least the integrity values of the *microkernel*, the *update* as well as the *request*. Additionally, the structure also specifies the threshold value T_c.

To create the new key, $\mathcal{P}$ first extends the PCRs of context U with the hash of the code update and the request (Fig. 5, step 3). After that, the prover creates the new integrity key $K_{\text{U}}^{\text{int}}$, which is encrypted with the wrapping key K^{wrap} and cryptographically bound to the trusted platform configuration, which is implicitly encoded into the HMAC $pubAuth_U$ (step 4).

After a successful generation of the wrapped integrity key, the prover $\mathcal{P}$ sends a hash of the key, $SHA1(load||\{K_{\text{u}}^{\text{int}}\}_{pk^{\text{wrap}}}^{P_u,T_u})$ and a random number $nonce_{\mathcal{P}_U}$ to the verifier, which initiates an attestation procedure for the code update (cf. Fig. 5). The verifier creates the authentication value $Auth_U$ and sends it to the prover together with a random number $nonce_{\mathcal{V}_2}$. $\mathcal{P}$ uses both values to load the new integrity key, which implicitly verifies the authenticity and integrity of

the code update, more precisely the hash h_U (Fig. 5, step 5). The attestation response, in particular $resAuth_U$, is then sent to $\mathcal{V}$, which can check the result. If the attestation was successful, the verifier sends the actual code update to the prover. $\mathcal{P}$ can check the integrity of the code update by freshly generating the hash value $H(U)$ and comparing it to the previously received hash h_U, which has already been implicitly verified. If both values match, the code update is trustworthy and the prover can install the code update.

6 Security Analysis

In this section, we analyze the security of the proposed protocols.

6.1 Analysis of the Attestation Protocol

We start our analysis from the premise that hardware attacks, such as the TPM *cold boot attack* [6] or manipulations of the HSM communication bus [16], are out of scope. As specified in Sect. 3.2, we will mainly focus on software attacks.

Furthermore, we assume that the platform configuration reflects the load-time integrity of the system at any time, although this would require a periodic or on-demand integrity measurement architecture, such as IBM's IMA [9]. We also assume that the trusted platform configuration is created during authenticated boot and cannot be easily forged by exploiting software vulnerabilities, such as buffer overflows. In addition, we exclude the *time of check to time of use* (TOCTOU) problem, which might affect the validity of the attestation result.

Based on the attacker model and these assumptions, we now analyze our attestation protocol. In the first attack scenario, the $\mathcal{A}$ might try to extract and obtain the authentication values or cryptographic keys in order to compromise the attestation. However, the wrapping key K^{wrap} is non-migratable as specified in Sect. 5.2, which means it is always securely stored inside the HSM. In addition, the authentication value for the wrapping key $Auth^{\text{wrap}}$, is only known to the verifier and never made public. Since this fact is very important for our attestation protocol, we will verify it formally in the next section.

$\mathcal{A}$ might also attempt to create and wrap a new integrity key to an insecure platform configuration, which does not include, for instance, any PCRs values, and replace the existing wrapped integrity key $\{K_c^{\text{int}}\}_{pk^{\text{wrap}}}^{P_c, T_c}$. However, this is not possible, because the wrapping key K^{wrap} is securely stored inside the HSM and the corresponding authentication value $Auth^{\text{wrap}}$, is only known to the verifier (cf. previous attacks scenario). So in the formal verification, we will also check that the wrapping key K^{wrap} does not leave the HSM during the attestation.

In a different attack scenario, the adversary tries to compromise the task by manipulating the implementation. However, since the task is measured before it is executed, the verifier would detect the manipulation, because the integrity measurements in the PCR would be incorrect, the key could not be loaded, and an attestation would fail. The failure is indicated by the return code in the result message from the HSM. If the attacker compromises the binary and replays an

old result message in order to convince the verifier that the load operation was successful, the verifier can detect the attack by checking the result, in particular the random number. Since this nonce is created by the verifier and does not match the old random number, the replay attack can be easily detected.

Finally, the attacker might try to compromise the behavior of a task during run-time. In this case, the security task detects an anomaly, e.g., in the number or order of certain system calls, by monitoring the behavior of the attacked task. As a result, the security adds an event (with a high probability, since it detected an attack) to the ADR, which is securely stored inside our HSM. So, if the prover tries to load the corresponding integrity key for the compromised task, the HSM prevents a successful load operation, since the current probability value is above the threshold T_c for any detected attack. As a consequence, the attestation fails.

6.2 Formal Verification of the Attestation Protocol

To substantiate our analysis, we have formally verified relevant security-critical properties of our attestation mechanism using *ProVerif* [2], an automated verifier for security protocols. We now discuss how our model implements certain aspects of the attestation protocol, which is also the core of the code update protocol.

The model for our remote attestation specifies a verifier, a prover, and an HSM, in this case with two contexts for simplicity (cf. Appendix A, Listing 1.1). To initiate the attestation protocol, the verifier creates a request and in turn receives the corresponding random numbers. The verifier then calculates the authentication values `Authc1` and `Authc2` (lines 9 and 11) based on the hash of the wrapped integrity keys, i.e., `hWrappedKint1` and `hWrappedKint2`, the random numbers `noncePc1` and `noncePc2` as well as `nonceV`. The key for the both HMACs is `AuthWrap`. The prover receives the HMACs and loads the wrapped keys, `wrappedKey1` and `wrappedKey2` (lines 22 and 24). The HSM verifies the nonces (lines 34 and 39), compares the current platform configuration `cpc` with the trusted one `tpc` (lines 35 and 40), and checks if the probability value `p` matches `T` (lines 36 and 41; simplified with $T = 0$, i.e., no anomalies tolerated). Finally, the HSM generates the result message. The prover forwards the result and the nonces `noncePc1New` and `noncePc2New` to the verifier, which then checks the attestation result. So, the verifier assumes a successful load operation (hence the hardcoded value `true` in lines 14 and 15) and calculates a fresh result hash `res` (line 16), which verifies `nonceV` and the trustworthiness of the selected tasks.

In *ProVerif*, our formal verification is automated with queries, which check if the authentication value `AuthWrap` (line 1) or the wrapping key `Kwrap` (line 2) are disclosed during the attestation. A third query additionally checks whether the attestation was successful (line 3), which is indicated by an event that is created only if the attestation response can be successfully verified. The results show that *ProVerif* cannot find an attack path for the `AuthWrap` or `Kwrap` and that the attestation is successful if all verification steps are successfully passed.

6.3 Analysis of the Code Update Protocol

To analyze the security of the code update protocol, we now discuss different attack scenarios, where an adversary might try to compromise the code update.

For the first attack, we presume that $\mathcal{A}$ tries to compromise the integrity of the code update by replacing it with a manipulated version. However, this is not possible, since $\mathcal{P}$ generates a new integrity key, which is wrapped to the untampered code update by $\mathcal{V}$. To receive the actual code update, $\mathcal{P}$ needs to successfully load the new integrity key in order to be able to send the correct attestation result to $\mathcal{V}$. To verify the integrity of the code update, $\mathcal{P}$ compares a fresh hash of the code update with the previously received hash h_U, which must have been part of the wrapped key and the trusted platform configuration. $\mathcal{P}$ also implicitly verifies the authenticity, because only $\mathcal{V}$ knows the authentication value $Auth^{\text{wrap}}$ for the wrapping key K^{wrap} and is able to generate the correct authorization value *pubAuth* for creating the new wrapped integrity key.

To ensure the authenticity of the code update request, the verifier authorizes the creation of a new integrity key, which is cryptographically bound not only to the trusted platform configuration of the system and the task, but also the code update request. That way, the prover has to extend the hash of the code update request to PCRs of the task in order to be able to load the newly created integrity key during the attestation procedure. If an adversary manipulates the code update request, the verifier creates an authorization value *pubAuth* for a false request and $\mathcal{P}$ cannot load the key, because $\mathcal{P}$ extended a different hash. As a result, the attestation fails and the verifier does not provides the actual code update in the final step of the protocol.

7 Conclusion

In this paper, we have presented a protocol for attesting the trustworthiness of multiple microkernel tasks. Compared to most existing attestation schemes, which mostly rely on expensive cryptographic operations, we show that our lightweight attestation mechanism can implicitly verify the integrity of multiple isolated microkernel tasks, securely communicate the result to a remote verifier, and enable secure code updates while eliminating the need for digital signatures.

As future work, we plan to evaluate our implementation in more detail. In comparison to existing attestation protocols, in particular classical remote attestation, we expect that our cryptographic integrity proof is more than ten times smaller, because our protocol mostly relies on symmetric cryptographic operations. When attesting more than one task with their own cryptographic contexts, this difference should become rather significant. Without even considering SMLs, we can already say that in this case our cryptographic integrity proof still only needs a constant size, whereas the number of digital signatures used in classical remote attestation increase with the number of tasks.

Acknowledgments. Parts of this work were funded by the *HIVE* project (GN: 01BY1200A) of the German Federal Ministry of Education and Research.

A ProVerif Code for the Attestation Mechanism

```
free AuthWrap:symKey [private]. query attacker(AuthWrap).
free Kwrap:sKey [private].      query attacker(Kwrap).
event successfulAttestation.    query event(successfulAttestation).

let Verifier(AuthWrap:symKey, hWrappedKint1:hash, hWrappedKint2:hash) =
    new req:attestationRequest; out(c2, req);
    in(c2, noncePc1:bitstring); in(c2, noncePc2:bitstring);
    new nonceV:bitstring;
    let Authc1= MAC(c(c(h2Bs(hWrappedKint1), noncePc1), nonceV), AuthWrap
        ) in
    out(c2, Authc1);
    let Authc2= MAC(c(c(h2Bs(hWrappedKint2), noncePc2), nonceV), AuthWrap
        ) in
    out(c2, Authc2); out(c2, nonceV); in(c2, res:hash);
    in(c2, noncePc1New:bitstring); in(c2, noncePc2New:bitstring);
    let resAuth1 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc1New),
         nonceV), AuthWrap)  in
    let resAuth2 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc2New),
         nonceV), AuthWrap)  in
    if res = SHA1(c(MAC2Bs(resAuth1), MAC2Bs(resAuth2))) then
        event successfulAttestation; 0.

let Prover(wrappedKint1:wKey, wrappedKint2:wKey, noncePc1:bitstring,
    noncePc2:bitstring) =
    in(c2, req:attestationRequest); out(c2, noncePc1); out(c2, noncePc2);
    in(c2, Authc1:mac); in(c2, Authc2:mac); in(c2, nonceV:bitstring);
    let cmd1 = createLoadKeyCmd(wrappedKint1, Authc1, noncePc1, nonceV)
        in
    out(c1, cmd1);
    let cmd2 = createLoadKeyCmd(wrappedKint2, Authc2, noncePc2, nonceV)
        in
    out(c1, cmd2);
    in(c1, res:hash);
    in(c1, noncePc1New:bitstring); in(c1, noncePc2New:bitstring);
    out(c2, res); out(c2, noncePc1New); out(c2, noncePc2New); 0.

let HSM(AuthWrap:symKey, noncePc1:bitstring, noncePc2:bitstring, cpc1:
    PConf, p1:ADR, cpc2:PConf, p2:ADR) =
    new noncePc1New:bitstring; new noncePc2New:bitstring;
    in(c1, cmd1:LoadKeyCommand); in(c1, cmd2:LoadKeyCommand);
    if getAuthc(cmd1) = MAC(c(c(h2Bs(SHA1(c(load, wKey2Bs(getWrappedKey(
        cmd1))))), noncePc1), getNonceV(cmd1)), AuthWrap) then
    if noncePc1 = getNoncePc(cmd1) then      (* comment: check noncePc1
        *)
    if cpc1 = pc(getWrappedKey(cmd1)) then   (* comment: check P_1 *)

    if p1 = ard(getWrappedKey(cmd1)) then    (* comment: check ADR_1 *)

    let resAuth1 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc1New),
         getNonceV(cmd1)), AuthWrap) in
    if getAuthc(cmd2) = MAC(c(c(h2Bs(SHA1(c(load, wKey2Bs(getWrappedKey(
        cmd2))))), noncePc2), getNonceV(cmd2)), AuthWrap) then
    if noncePc2 = getNoncePc(cmd2) then      (* comment: check noncePc1
        *)
    if cpc2 = pc(getWrappedKey(cmd2)) then   (* comment: check P_2 *)

    if p2 = ard(getWrappedKey(cmd2)) then    (* comment: check ADR_2 *)

    let resAuth2 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc2New),
        getNonceV(cmd2)), AuthWrap) in
    let res = SHA1(c(MAC2Bs(resAuth1), MAC2Bs(resAuth2))) in
    out(c1, res); out(c1, noncePc1New); out(c1, noncePc2New); 0.

process
    new noncePc1:bitstring; new noncePc2:bitstring;
```

```
48      new Kint1:intKey; new Kint2:intKey;
49      let wKint1 = wrapKey(Kint1, pk(Kwrap), tpc1, T1) in   (* cf. Sect.
            5.1 *)
50      let wKint2 = wrapKey(Kint2, pk(Kwrap), tpc2, T2) in   (* cf. Sect.
            5.1 *)
51      (!Verifier(AuthWrap, SHA1(c(load, wKey2Bs(wKint1))), SHA1(c(load,
            wKey2Bs(wKint2))))) | (!Prover(wKint1, wKint2, noncePc1, noncePc2
            )) | (!HSM(AuthWrap, noncePc1, noncePc2, tpc1, T1, tpc2, T2))
```

Listing 1.1. ProVerif Code for the Attestation Mechanism (excerpt)

References

1. Berger, S., Cáceres, R., Goldman, K.A., Perez, R., Sailer, R., Doorn, L.: vTPM: virtualizing the Trusted Platform Module. In: Proceedings of the 15th Conference on USENIX Security Symposium, vol. 15 (2006)
2. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, CSFW 2001. IEEE Computer Society, Washington, DC (2001)
3. England, P., Loeser, J.: Para-virtualized TPM sharing. In: Lipp, P., Sadeghi, A.-R., Koch, K.-M. (eds.) Trust 2008. LNCS, vol. 4968, pp. 119–132. Springer, Heidelberg (2008)
4. Feller, T., Malipatlolla, S., Kasper, M., Huss, S.A.: dctpm: a generic architecture for dynamic context management. In: Athanas, P.M., Becker, J., Cumplido, R. (eds.) ReConFig, pp. 211–216. IEEE Computer Society (2011)
5. Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation: a virtual machine directed approach to trusted computing. In: Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium, Berkeley, CA, USA (2004)
6. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)
7. Liedtke, J.: Microkernels must and can be small. In: Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, October 1996. http://l4ka.org/publications/
8. Sadeghi, A.R., Stüble, C.: Property-based attestation for computing platforms: caring about properties, not mechanisms. In: Proceedings of the 2004 Workshop on New Security Paradigms, NSPW 2004, pp. 67–77. ACM, New York (2004)
9. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th Conference on USENIX Security Symposium, vol. 13, Berkeley, CA, USA (2004)
10. Schiffman, J., Vijayakumar, H., Jaeger, T.: Verifying system integrity by proxy. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) Trust 2012. LNCS, vol. 7344, pp. 179–200. Springer, Heidelberg (2012)
11. Sirer, E.G., de Bruijn, W., Reynolds, P., Shieh, A., Walsh, K., Williams, D., Schneider, F.B.: Logical attestation: an authorization architecture for trustworthy computing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 249–264. ACM, New York (2011)
12. Stumpf, F., Eckert, C.: Enhancing trusted platform modules with hardware-based virtualization techniques. In: Emerging Security Information, Systems and Technologies, pp. 1–9 (2008)
13. SYSGO AG: PikeOS. http://www.sysgo.com/
14. Trusted Computing Group: TPM Main Specification Version 1.2 rev. 116 (2011). http://www.trustedcomputinggroup.org/resources/tpm_main_specification

15. Wagner, S., Wessel, S., Stumpf, F.: Attestation of mobile baseband stacks. In: Xu, L., Bertino, E., Mu, Y. (eds.) NSS 2012. LNCS, vol. 7645, pp. 29–43. Springer, Heidelberg (2012)
16. Winter, J., Dietrich, K.: A Hijacker's guide to the LPC bus. In: Petkova-Nikova, S., Pashalidis, A., Pernul, G. (eds.) EuroPKI 2011. LNCS, vol. 7163, pp. 176–193. Springer, Heidelberg (2012)
17. Xiao, H., Eckert, C.: Lazy Gaussian process committee for real-time online regression. In: 27th AAAI Conference on Artificial Intelligence, AAAI 2013. AAAI Press, Washington, July 2013
18. Xiao, H., Xiao, H., Eckert, C.: Learning from multiple observers with unknown expertise. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds.) PAKDD 2013, Part I. LNCS, vol. 7818, pp. 595–606. Springer, Heidelberg (2013)

http://www.springer.com/978-3-319-27658-8

Information Security
16th International Conference, ISC 2013, Dallas, Texas, November 13-15, 2013, Proceedings
Desmedt, Y. (Ed.)
2015, XIV, 418 p. 52 illus. in color., Softcover
ISBN: 978-3-319-27658-8