Group-Based Attestation: Enhancing Privacy and Management in Remote Attestation

Sami Alsouri, Özgür Dagdelen, and Stefan Katzenbeisser

Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt - CASED
Mornewegstraße 32, 64293 Darmstadt, Germany
{sami.alsouri,oezguer.dagdelen}@cased.de,
katzenbeisser@seceng.informatik.tu-darmstadt.de

Abstract. One of the central aims of Trusted Computing is to provide the ability to attest that a remote platform is in a certain trustworthy state. While in principle this functionality can be achieved by the remote attestation process as standardized by the Trusted Computing Group, privacy and scalability problems make it difficult to realize in practice: In particular, the use of the SHA-1 hash to measure system components requires maintenance of a large set of hashes of presumably trustworthy software; furthermore, during attestation, the full configuration of the platform is revealed. In this paper we show how chameleon hashes allow to mitigate of these two problems. By using a prototypical implementation we furthermore show that the approach is feasible in practice.

1 Introduction

One of the main functionalities of the Trusted Platform Module (TPM), as specified by the Trusted Computing Group (TCG), is the ability to attest a remote system, i.e., to verify whether the system is in a well-defined (trustworthy) state. The TCG specified a measurement process that uses the TPM as a *root of trust* and employs a *measure-then-load* approach: Whenever control is passed to a specific system component, its executable code is hashed and the hash is added to a tamper-resistant storage (the Platform Configuration Registers, PCRs) within the TPM in the form of a hash chain: the hash value of the program to be executed is concatenated with the current values in the PCR register, the resulting string is hashed and stored in the PCR. The content of the PCR registers therefore can be considered to reflect the current state of the system. In the process of *remote attestation*, this state is signed and transferred to a remote entity (called challenger), who can subsequently compare the provided measurements with a list of trusted measurements (Reference Measurement List, RML) and decide about the trustworthiness of the remote platform.

Research has identified several problems with the remote attestation process as specified by the TCG. These problems include privacy [1] and scalability issues [2, 3], problems with the sealing functionality [4] and high communication

A. Acquisti, S.W. Smith, and A.-R. Sadeghi (Eds.): TRUST 2010, LNCS 6101, pp. 63–77, 2010.

and management efforts [3]. In this paper we deal with these aforementioned problems. Remote attestation discloses full information about the software running on the attested platform, including details on the operating system and third-party software. This may be an unwanted privacy leak, as it allows for product discrimination (e.g., in a DRM context a party can force the use of a specific commercial software product before certain data is released, thereby limiting freedom of choice) or targeted attacks (e.g., if a party knows that someone runs a specifically vulnerable version of an operating system, dedicated attacks are possible). Thus, attestation methods are required that do not reveal the full configuration of the attested platform but nevertheless allow a challenger to gain confidence on its trustworthiness. The second major problem of TCG attestation is the scalability of Reference Measurement Lists [2]. The large number of software products and versions of operating systems makes maintenance of the lists cumbersome. For instance, [5] notes that a typical Windows installation loads about 200 drivers from a known set of more than 4 million, which is increasing continuously by more than 400 drivers a day. The large number of third-party applications aggravates the problem further. Scalability of the remote attestation process is sometimes seen as a major limiting factor for the success of Trusted Computing [3].

In this paper, we propose novel attestation and integrity measurement techniques which use chameleon hashes in addition to SHA-1 hash values or group signatures in the integrity measurement and attestation process. Even though this increases the computational complexity of the attestation process, we show that the presented mechanisms increase the scalability of remote attestation, while providing a fine-grained mechanism to protect privacy of the attested platform. One construction uses chameleon hashing [6], which allows grouping sets of software and hardware versions, representing them through one hash value. For instance, all products of a trusted software vendor or versions of the same software can be represented by one hash value. On the one hand, this reduces the management effort of maintaining RMLs, and on the other hand increases privacy, as the challenger is not able to see any more the exact configuration of the attested platform, but only the installed software groups. At the same time, the challenger system can be assured that all running software comes from trusted software groups. We show that the proposed system can easily be integrated into an architecture similar to the TCG, with only minor modifications. We have implemented the attestation process in a prototypical fashion and show that the approach is feasible in practice. Finally, we show that a very similar attestation technique can be implemented by group signatures instead of chameleon hashes as well.

This paper is organized as follows. In Section 2 we briefly review the mechanism provided by the TCG standards to measure system integrity and to perform remote attestation. In addition, we give background material about chameleon hashes and discuss its security. Furthermore, we discuss the problems with remote attestation and outline solutions proposed in related work. In Section 3 we outline our Chameleon Attestation approach to integrity measurement and

remote attestation and also propose an alternative using group signatures. Section 4 provides details on our implementation, and Section 5 discusses the advantages of Chameleon Attestation and details our experimental results. Finally, we conclude the paper in Section 6.

2 Background and Related Work

2.1 Integrity Measurement and Remote Attestation

One of the main goals of Trusted Computing is to assure the integrity of a platform. This is done by measuring every entity (such as BIOS, OS kernel and application software) using the SHA-1 hash before its execution. All measurements are securely stored by extending values in a particular PCR register by a hash chain. To allow the challenger to recompute the hash values, information on the measured entities is stored in form of a Measurement Log (ML). To prevent malicious software behavior, the TPM chip only allows to extend the PCR registers, so that PCRs can not be reset as long as the system is running (the only way to reset the registers is to reboot).

A practical attestation framework called IMA, an extension of the Linux kernel, was developed by IBM research [2]. IMA measures user-level executables, dynamically loaded libraries, kernel modules and shell scripts. The individual measurements are collected in a *Measurement List* (ML) that represents the integrity history of the platform. Measurements are initiated by so-called *Measurement Agents*, which induce a measurement of a file, store the measurement in an ordered list into ML, and report the extension of ML to the TPM. Any measurement taken is also aggregated into the TPM PCR number 10. Thus, any measured software can not repudiate its existence.

Signed measurements can be released to third parties during the process of "remote attestation". For this purpose, the challenger creates a 160-bit *nonce* and sends it to the attested platform. The attestation service running on that host forwards the received nonce and the PCR number requested by the challenger to the TPM chip, which signs the data using the *TPM_Quote* function. After signing, the results are sent back to the attestation service. To protect identity privacy, only the *Attestation Identity Keys* (AIKs) can be used for the signing operation. The attestation service sends the signed data together with the ML back to the challenger. Using the corresponding public key AIK_{pub}, the challenger verifies the signature and the nonce, and re-computes the hash chain using the ML. If the re-computed hash value equals the signed PCR value, then ML is untampered. Finally, the challenger determines whether all measurements in ML can be found in the trusted Reference Measurement List (RML); in this case the attested platform is considered as trusted.

2.2 Chameleon Hashing

Chameleon hashing was introduced by Krawczyk and Rabin [6]. Unlike standard hash functions, chameleon hashes utilize a pair of public and private keys.

Every party who knows the public key is able to compute the hash value on a given message. The possession of the private key enables collisions to be created. However, chameleon hash functions still provide collision-resistance against users who have no knowledge of the private key.

A chameleon hash function is defined by a set of efficient (polynomial time) algorithms [7]:

Key Generation. The probabilistic key generation algorithm $\mathbf{Kg} : 1^\kappa \rightarrow (\mathbf{pk}, \mathbf{sk})$ takes as input a security parameter κ in unary form and outputs a pair of a public key **pk** and a private key (trapdoor) **sk**.

Hash. The deterministic hash algorithm $\mathbf{CH} : (\mathbf{pk}, m, r) \rightarrow h \in \{0,1\}^\tau$ takes as input a public key **pk**, a message m and an auxiliary random value r and outputs a hash h of length τ.

Forge. The deterministic forge algorithm $\mathbf{Forge} : (\mathbf{sk}, m, r) \rightarrow (m', r')$ takes as input the trapdoor **sk** corresponding to the public key **pk**, a message m and auxiliary parameter r. **Forge** computes a message m' and auxiliary parameter r' such that $(m, r) \neq (m', r')$ and $\mathbf{CH}(\mathbf{pk}, m, r) = h = \mathbf{CH}(\mathbf{pk}, m', r')$.

In contrast to standard hash functions, chameleon hashes are provided with the **Forge** algorithm. By this algorithm only the owner of the trapdoor (**sk**) can generate a different input message such that both inputs map to the same hash value. In some chameleon hashes the owner of the private information can even choose himself a new message m' and compute the auxiliary parameter r' to find a collision $\mathbf{CH}(\mathbf{pk}, m, r) = h = \mathbf{CH}(\mathbf{pk}, m', r')$. This is a powerful feature since anyone who knows the private information can map arbitrary messages to the same hash value.

We desire the following security properties to be fulfilled by a chameleon hash function (besides the standard property of collision resistance):

Semantic Security. For all message pairs m, m', the hash values $\mathbf{CH}(\mathbf{pk}, m, r)$ and $\mathbf{CH}(\mathbf{pk}, m', r)$ are indistinguishable, i.e., $\mathbf{CH}(\mathbf{pk}, m, r)$ hides any information on m.

Key Exposure Freeness. Key Exposure Freeness indicates that there exists no efficient algorithm able to retrieve the trapdoor from a given collision, even if it has access to a **Forge** oracle and is allowed polynomially many queries on inputs (m_i, r_i) of his choice.

Any chameleon hash function fulfilling the above definitions and security requirements can be used in our approach; our particular choice of a chameleon hash is detailed in [7].

2.3 Group Signatures

Group signatures were introduced by Chaum and van Heyst [8] and allow a member of a group to anonymously sign a message on behalf of the group. A group has a single group manager and can have several group members. Unlike standard digital signatures, signers of a group are issued individual signing keys

$\mathbf{gsk}[i]$, while all members share a common group public key **gpk** such that their signatures can be verified without revealing which member of the group created the signature. This provides anonymity. However, the group manager is assigned with a group manager secret key **gmsk** and is able to discover the signer (traceability).

Basically, a group signature scheme $\mathcal{GS} = (\mathbf{GKg}, \mathbf{GSig}, \mathbf{GVf}, \mathbf{Open})$ is defined by a set of efficient algorithms (for more details, we refer to [8] and [9]):

Group Key Generation. The probabilistic group key generation algorithm $\mathbf{GKg} : (1^\kappa, 1^n) \rightarrow (\mathbf{gpk}, \mathbf{gmsk}, \mathbf{gsk})$ takes as input the security parameter κ and the group size parameter n in unary form and outputs a tuple $(\mathbf{gpk}, \mathbf{gmsk}, \mathbf{gsk})$, where **gpk** is the group public key, **gmsk** is the group manager's secret key, and **gsk** is an vector of n secret signing keys. The group member $i \in \{1, \ldots, n\}$ is assigned the secret signing key $\mathbf{gsk}[i]$.

Group Signing. The probabilistic signing algorithm $\mathbf{GSig} : (\mathbf{gsk}[i], m) \rightarrow \sigma_i(m)$ takes as input a secret signing key $\mathbf{gsk}[i]$ and a message m and outputs a signature $\sigma_i(m)$ of m under $\mathbf{gsk}[i]$.

Group Signature Verification. The deterministic group signature verification algorithm $\mathbf{GVf} : (\mathbf{gpk}, m, \sigma) \rightarrow \{0, 1\}$ takes as input the group public key **gpk**, a message m and a signature σ and outputs 1 if and only if the signature σ is valid and was created by one of the group members. Otherwise, the algorithm returns 0.

Opening. The deterministic opening algorithm $\mathbf{Open} : (\mathbf{gmsk}, m, \sigma) \rightarrow \{i, \perp\}$, which takes as input a group manager secret key **gmsk**, a message m and a signature σ of m. It outputs an identity $i \in \{1, \ldots, n\}$ or the symbol $\perp$ for failure.

Join. A two-party protocol **Join** between the group manager and a user let the user become a new group member. The user's output is a membership certificate $cert_i$ and a membership secret $\mathbf{gsk}[i]$. After an successful execution of **Join** the signing secret $\mathbf{gsk}[i]$ is added to the vector of secret keys **gsk**.

In order to allow revocation of users, we require an additional property:

Revocability. A signature produced using **GSig** by a revoked member must be rejected using **GVf**. Still, a signature produced by a valid group member must be accepted by the verification algorithm.

2.4 Attestation Problems and Related Work

Integrity measurement according to the TCG specification seems to be a promising way to check the trustworthiness of systems. However, the suggested remote attestation process has several shortcomings:

- *Privacy.* We can distinguish between identity privacy (IP) and configuration privacy (CP). IP focuses on providing anonymity for the attested platform. This problem can be solved by Direct Anonymous Attestation (DAA) [1, 10, 11]. On the other hand, CP is concerned with keeping configuration details of an

attested platform secret, since disclosure may lead to privacy violations. Still, the challenger system must be assured that the attested platform indeed is in a trustworthy state. In this paper we focus on providing CP. (However, since CP and IP are orthogonal problems, our solution can be used in conjunction with mechanisms that guarantee IP).

- *Discrimination and targeted attacks.* By using remote attestation, product discrimination may be possible. For example, in the context of DRM environments, large operating system vendors and content providers could collaborate and force usage of specific proprietary software, which restricts the freedom of choice. Furthermore, an adversary could leverage the precise configuration of the attested platform and perform a specific targeted attack [12].
- *Scalability.* A further drawback lies in the scalability of Reference Measurement Lists [2]. The TCG attestation requires the challenger to maintain a Reference Measurement List, which contains hashes of all trustworthy software, to validate the received measurements. Consequently, software updates or patches require distribution of new hash values. For this reason, the management overhead increases to a point where attestation becomes impractical. Consequently, keeping these RML lists up-to-date involves high management and communication efforts.
- *Sealing.* Besides remote attestation, TCG offers the ability to seal data to the configuration of a specific platform. Again, any software update or configuration change can lead to a completely new platform configuration state and consequently hinder unsealing [4].

Sadeghi and Stüble [4] approached the above mentioned problems by the introduction of Property-based Attestation (PBA). By applying PBA, the attested platform proves that it fulfills certain semantic security requirements, called "properties". This way, the concrete configuration of a platform does not need to be disclosed. However, PBA requires an extension of TPM or alternatively a Trusted Third Party along with a Trusted Attestation Service, which is responsible for translations between properties and software. Semantic attestation [13] verifies that the behavior of a platform fulfills given particular high-level properties. WS-Attestation proposed by Yoshihama et al. [14] employs PCR obfuscation to hide software versions; however, scalability remains a problem [15].

3 Group-Based Attestation

In this section we propose three novel attestation techniques, which are based on either chameleon hashes or group signatures. The first and second technique allow balancing configuration privacy with the control precision of the attestation process and substantially decrease the overhead for maintaining RMLs, while the third one provides more flexibility for the challenger in control precision but offers no privacy advantage when compared with the TCG attestation.

3.1 Chameleon Attestation I

In this section we describe a novel remote attestation approach, which makes it possible for the challenger to decide on the trustworthiness of the attested platform, without knowing its detailed configuration. The assumptions listed in [2] about the attacker model are also the basis of our approach. In particular we assume that once a measurement is stored in an RML, the corresponding software is considered trusted; additional security mechanisms must be in place to secure the integrity of the RML (this is out of scope of this work).

To reduce the management overhead, we propose the concept of *software groups*; according to the precise scenario, these groups may e.g. contain all software products of the same vendor, compatible software products or all versions of one specific software. We design the attestation process in such a way that we assign the same hash value to all members of a software group. To achieve this, we make use of a chameleon hash function. As mentioned in Section 2.2, any party who knows the public key **pk** is able to compute the hash value for a given message. In contrast, only the trusted instance holding the private key **sk** can create collisions. Based on the idea of software groups sharing the same hash value, we describe in the following a novel remote attestation we call *Chameleon Attestation I*.

Setup phase: For each group, a trusted instance (such as a software vendor) runs the key generation algorithm **Kg** to obtain a public/private key pair $(\mathbf{pk}, \mathbf{sk})$. When establishing a new software group, the software vendor picks for the first product contained in the new software group a random r and makes it available to the attested platform by delivering it with the software. Furthermore, he hashes the code m of the software with the chameleon hash to obtain $h = \mathbf{CH}(\mathbf{pk}, m, r)$; for performance reasons the SHA-1 hash value of the software is taken as m. The obtained chameleon hash is made public in a trusted RML. Subsequently, to add a new software m' to the same software group, he uses the algorithm **Forge** to find a new r' so that $\mathbf{CH}(\mathbf{pk}, m', r') = h$ and distributes the new r' alongside the software. Step 1 in Figure 1 (a) shows the parameters distributed to the attested platform by a software vendor.

Integrity measurement: On the attested platform, the operation proceeds in a similar way as in the original integrity measurement process, see Figure 1 (a). In particular, the software is first hashed using SHA-1 (step 2). Subsequently, the attested platform computes in step 3 the chameleon hash value h of the software using the public key **pk** and the random value r distributed alongside the software. Since the PCRs in the TPM accept only a 160-bit message to be extended to a particular register, the chameleon hash value is hashed again using SHA-1 in step 4 and the corresponding information is stored in the ML in step 5. The resulting value is finally extended to a PCR register (step 6).

Remote attestation: The attestation process of Chameleon Attestation I is very similar to the standard TCG attestation process. In step 1 in Figure 1 (b) the challenger sends a *nonce* and the PCR numbers whose content has

to be signed by the TPM. In step 2 the Attestation Service forwards the request to the TPM, and in step 3 the TPM signs the desired PCRs values and the *nonce*, and sends them back to the Attestation Service. In step 4, the attested platform sends the ML containing the chameleon hash values instead of SHA-1 values. In steps 5-7 the challenger verifies the signature, validates the PCRs values against ML, and checks the trustworthiness of the sent measurements. Only if ML contains trustworthy measurements the attested platform is considered trusted.

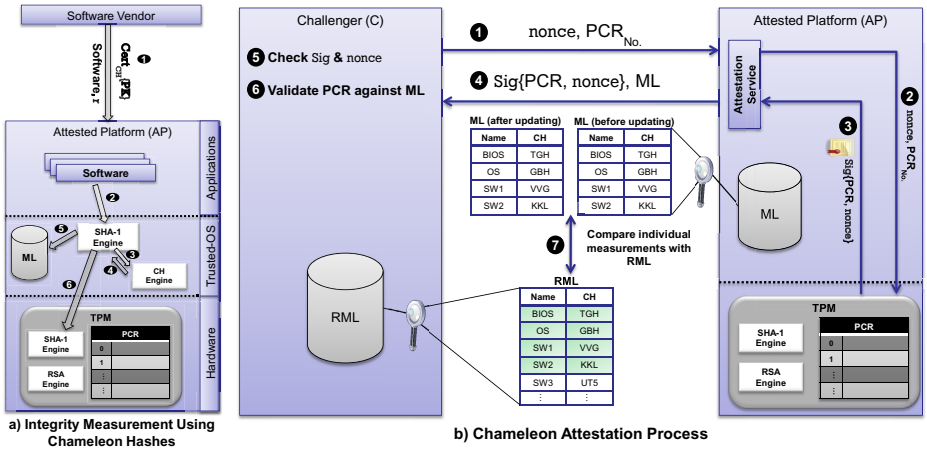

Fig. 1. Chameleon Integrity and Attestation

Chameleon Attestation I is flexible in the sense that the granularity of the software groups can easily be chosen to balance privacy and control precision: If more privacy is desired, then larger software groups may be formed; on the other hand, if distinction between different software versions is an issue, smaller groups can be maintained. Note that the decision of how granular a group is, can be made only by the software vendor. Without modifying the TPM, Chameleon Attestation I supports only the *static chain of trust*, since the TPM itself does not provide functionalities to calculate chameleon hashes.

3.2 Group Signatures Based Attestation

An alternative approach to improve the remote attestation process in terms of privacy and scalability is possible by applying digital signatures, in particular group signatures. This requires the following modifications to the integrity measurement architecture:

Setup phase: We again use the concept of software groups. This time, we use group signatures; each software in the software group has its own private signature key $\mathbf{gsk}[i]$, while all share a common verification key $\mathbf{gpk}$. Whenever a

new product or an update of software is published, the software is first hashed with SHA1 to obtain $h = \text{SHA-1}(SW)$, where SW is the code of the software. Then, the hash value h is signed by the private key $\mathbf{gsk}[i]$, i.e. $\sigma = \mathbf{GSig}(\mathbf{gsk}[i], h)$. The public verification key and the signature is distributed alongside the software. Furthermore, the public keys of all trusted software groups are stored in the RML.

Integrity measurement: Whenever a software is loaded, it is hashed with SHA-1 and its signature is checked with the included public key using the group signature verification algorithm **GVf**. If the signature is valid, the attesting platform hashes the public key and extends the particular PCR with the hash value of the public key of the verified software (instead of the hash value of the software). Afterwards, a corresponding information item containing the name of the software group and its public key **gpk** is stored in the Measurement Log (ML). If any failure occurs, similar to the process of IMA, the corresponding PCR is set to an invalid state.

Remote attestation: The remote attestation works exactly as described in Section 2.1 up to the point where the challenger receives the answer from the attested platform. Then, the challenger verifies the signed PCR and his chosen nonce, validates the hash chain of the PCR against the public keys contained in the ML and checks whether they are all listed in the trusted RML. If all checks succeed, the system is considered trustworthy.

Using group signatures instead of chameleon hashes provides some advantages. While in Chameleon Attestation I a revocation of chameleon hash value requires the revocation of all group members, using group signatures allows the revocation of specific members of the group without the need to revoke the whole group. A second advantage lies in the ability of fitting a group signature hierarchy to an organization structure. That is, every product realm or series could have its own private key, while verification is performed with one single public key.

On the other hand, Chameleon Attestation I outperforms group signature based attestation in terms of performance. While fast group signature schemes (like [16]) need about six exponentiations for signing and verification, chameleon hash functions require much less computations. For instance, our particular choice of a chameleon hash detailed in [7] performs only two exponentiations. To the best of our knowledge there exists no group signature which require less than three exponentiations.

3.3 Chameleon Attestation II

The remote attestation proposed above can be used to mitigate the privacy problem. However, there is a tradeoff between privacy and *control precision* of the approach: as the challenger is only able to see the software groups running on the attested system, the challenger cannot distinguish individual software versions any more: Assume a software vendor has developed a product $SW_{v.1}$ which is later updated to $SW_{v.2}$ because of disclosed security vulnerabilities. By applying the technique mentioned above, a challenger cannot distinguish platforms where $SW_{v.1}$ or $SW_{v.2}$ is run. When using Chameleon Attestation I we

lose the possibility to efficiently revoke certain members of a software group. A software vendor can only declare the old chameleon hash value for the group as invalid and publish a new one. However, this requires an update to the challenger's RML. That is, revocation in this context means revocation of the whole software group with all of its members and not revocation of a certain member or even a subgroup.

In this section we show how chameleon hashes can be used to reduce the management overhead of maintaining large RMLs in scenarios where configuration privacy is not an issue. Instead of computing chameleon hashes on the attested platform, we can move this calculation to the challenger side. As in the system described in Section 3.1, the manufacturer picks one chameleon hash for each software group, publishes the hash value of each group in an RML, and sends alongside the software random values r required to compute the chameleon hash. On the attested system, the standard integrity measurement process is performed (in which SHA-1 hashes of loaded executables are stored into PCRs), except that the random values r required to compute the chameleon hashes and the SHA-1 hashes are both saved in the ML. The remote attestation process proceeds as in the standard TCG attestation, i.e., the challenger receives the signed PCR values. Subsequently, the challenger verifies the signed PCR and his chosen nonce and validates the contents of the PCR against the ML containing all SHA-1 values. Finally, for each entry in ML, the chameleon hash is computed to build software groups and validated against the RML.

Applying Chameleon Attestation II makes revocation of specific software group members easier. Unlike Chameleon Attestation I and group signatures based attestation, the challenger himself can refuse untrusted software versions by simply validating the SHA-1 values of these members against blacklists of revoked or untrusted group members. This leads to more flexibility for the challenger and gives him a tradeoff between scalability and control precision.

4 Implementation

In this section we describe the changes we made to the Linux system during the implementation of both variants of Chameleon Attestation as proposed in Sections 3.1 and 3.3.

In order to support a trusted boot we use the Grand Unified Bootloader (GRUB) version 0.97 with the TrustedGrub extension 1.13. All measurements taken are stored in the Intel iTPM. As Linux distribution, we used Fedora 10 with the kernel version 2.6.27.38. The kernel contains the Integrity Measurement Architecture (IMA), which measures all executables and stores the measurements in the Measurement Log (ML). For Trusted Computing support we use the Java based jTSS in version 0.4.1. Because jTSS supports only one measurement log, we modified it to also support reading the measurement log created by IMA. For the remote attestation process, we implemented a Java based server and client. jTSS is used by the server to access the functions of the TPM such as reading PCR registers, signing PCR content, etc. The client also uses the functionalities

provided by jTSS to verify signatures and recompute PCR contents. In addition, a MySQL database management system was used on the client side to store the Reference Measurement List (RML).

Implementation of Chameleon Attestation I. For the first variant described in Section 3.1, it is necessary to calculate our chosen chameleon hash function described in [7], denoted as **CH**, on the attested platform. For that reason, we extended IMA such that the **CH** value is calculated after measuring every executable. We assume that the parameters required to calculate **CH** are delivered with the executable and stored. We first created a special measurement list ML_{CH} which contains the chameleon hashes of measured executables. We also modified the standard ML to store the public **CH** parameters J, r, e and N. In particular, in order to store these parameters we extended the struct *ima_measure_entry*. Afterwards, to read these parameters again from ML, we implemented a new function in the file */security/ima/ima_main.c*, which is called from the functions that are responsible for measuring executables, namely *ima_do_measure_file* and *ima_do_measure_memory*. To calculate the **CH** value, we created a new function in the file */ima/ima_main.c*, which also stores the resulting **CH** value in ML_{CH} and the SHA-1 value in standard ML. Note that the standard ML is used only for internal purposes, whereas the ML_{CH} is sent to the challenger during the attestation process. For the implementation of **CH** we used a slightly changed version of the RSA patch for avr32linux.

Implementation of Chameleon Attestation II. In the second variant described in Section 3.3 we need to calculate the chameleon hash on the platform of the challenger. We thus modify the measurement process in a way that the parameters J, r, e and N are added to ML, as in Chameleon Attestation I. Furthermore, we extended the package *iaik.tc.tss.impl.java.tcs.evenmgr* of jTSS such that the new chameleon hash parameters can be read from ML in addition to SHA-1 values. To calculate the chameleon hash on the challenger side, we modified the server such that the SHA-1 values and the corresponding new parameters can be delivered to the challenger. We implemented the RSA based chameleon hash function using OpenSSL on the side of the challenger to enable it to calculate the hash value and verify it against the RML.

5 Experimental Results

In this section we show that Chameleon Attestation significantly reduces the number of the reference measurements required to decide the trustworthiness of the attested system. Subsequently, we discuss the performance of our approach.

Scalability. To test the scalability of Chameleon Attestation, we first created an RML by measuring a fresh installation of Fedora 10 (kernel version 2.4.27.5), but neglecting the content two folders: the folder */var/* which contains variable data that can be changed or deleted at runtime, and the folder */usr/share/* which contains the architecture-independent data. Since it is difficult in retrospect to

Table 1. Reduction of measurements in RML

Packages	Measurement	Fresh installation	Update	Total	Statistics	
					% Fresh installation	% Update
kernel	TCG	1,820	1,816	3,636	50.1 %	49.9 %
	CA	1	0	1	100.0 %	0.0 %
samba-comomn	TCG	18	15	33	54.5 %	45.5 %
	CA	1	0	1	100.0 %	0.0 %
samba	TCG	24	26	50	48.0 %	52.0 %
	CA	1	0	1	100.0 %	0.0 %
httpd (Apache)	TCG	71	72	143	49.7 %	50.3 %
	CA	1	0	1	100.0 %	0.0 %
⋮	⋮	⋮	⋮	⋮	⋮	⋮
All	TCG	8,268	5,448	13,716	60.3 %	39.7 %
	CA	981	37	1.018	96.3 %	3.7 %
	ratio	8.5:1	147:1	13.5:1		10.7:1

group packages by manufacturer (because the package manager of Fedora does not store information about the author/manufacturer of a package), we grouped software products by packages and assigned each file in a package its appropriate random r. Table 1 shows how our approach reduces the number of entries in the RML. The table shows that we need 8,268 different entries in RML for the fresh installation when we employ classic TCG attestation (one for each file). In the contrast, we only need to store 981 measurements in the RML by applying our approach (one for each package in case of grouping by packages).

To test the management overhead when updating packages, we performed another experiment by updating the Linux distribution and its installed packages to newer versions. For instance, the kernel is updated from version 2.6.27.5 to 2.6.27.41, the package samba-common from 3.2.4 to 3.2.15, the package samba from 3.2.4 to 3.2.15, and the package httpd from 2.2.10 to 2.2.14. Table 1 shows that in case of using the classic TCG attestation 1,816 new SHA-1 measurements (49.9 % of the total measurements for the kernel) have to be distributed and published in RMLs. Conversely, by employing Chameleon Attestation no new measurements have to be distributed or published. For the overall distribution and its installed packages, we only need to update 37 chameleon hashes rather than 5,448. These hashes mainly account for newly added packages. Thus, the management and communication effort is significantly reduced.

Privacy. The configuration privacy of the attested platform is substantially enhanced by the use of Chameleon Attestation I: the challenger can decide on the trustworthiness of the attested platform without knowing the exact details of the configuration. Since there is a tradeoff between privacy and control precision, the scheme can be applied on different granularities: depending on the choice of the manufacturer, software groups may encompass different versions of individual files, packages, software systems or even software of a specific vendor (see

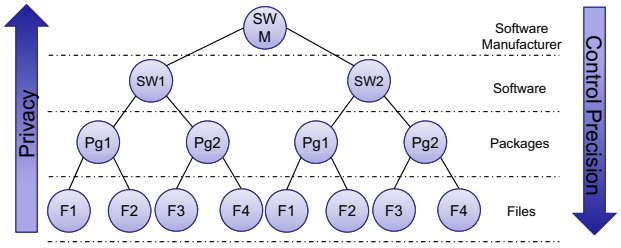

Fig. 2. Levels of privacy and control precision

Figure 2). The higher the level, the more privacy can be protected; on the downside, less information on the platform is available, i.e., the control precision is lower. Our approach can be easily combined with other identity privacy approaches, such as a Privacy CA and DAA.

Sealing. In a similar manner, the sealing problem can be avoided, since different versions of the same software will have the same chameleon hash value; consequently, data can be bound to this value without risking data unavailability when updating to the next version.

Performance evaluation. To evaluate the performance of Chameleon Attestation, we measure the timing difference compared to the standard TCG measurement process. Our experiments were performed on a Lenovo W500 with the following main components: Intel CPU Core 2 2.8 Ghz, 1066 Mhz FSB, a HD of 250 GB SATA 7200 rpm, 4 GB SDRAM, Fedora 10, and kernel version 2.6.27.41.

The calculation of **CH** in Chameleon Attestation I (see Section 3.1) is performed in the kernel space and requires 4,674 μs, while the calculation of **CH** in Chameleon Attestation II (see Section 3.3) is done in the user space and requires 896 μs, i.e., the fifth of the time needed for the first variant. The calculation of collisions takes 899 μs in the user space. All measurements were taken using the function *gettimeofday* in both the kernel space and the user space. Note that all measurements we present in this section aim at giving a gross overview on the overhead of applying public-key schemes in the attestation process. We expect that significant performance improvements can be obtained using highly optimized code also in kernel space.

We used bootchart[1] to determine the boot time of a standard kernel, a kernel with IMA, and a kernel with **CH**. While a standard kernel takes 30s to finish booting, a kernel with IMA takes 33s and a kernel with **CH** takes 44s.

The times required to measure individual files give more insight into the performance. Table 2 illustrates the performance of **CH** in the measurement process. Obviously, the size of the measured files influences the required time significantly. For instance, the calculation of SHA-1 of a 1 KB file takes approx. 20.1 μs, while

[1] http://www.bootchart.org

Table 2. Performance of **CH** depending on SHA-1 and different file sizes

Measurement method	2 byte	1 KB	1 MB
SHA-1	2 μs	20 μs	18,312 μs
SHA-1 + **CH**	4,677 μs	4,694 μs	22,986 μs
CH fraction	99.8 %	99.6 %	20.3 %
SHA-1 + extend	9,972 μs	9,989 μs	28,281 μs
SHA-1 + **CH** + extend	14,646 μs	14,663 μs	32,955 μs
CH fraction	31.9 %	31.9 %	14.2 %

measuring a 1 MB file takes 18,312.3 μs $\approx$ 18.3 ms. Note that the time required to compute **CH** is constant, as it is only applied to a SHA-1 value. Table 2 also gives timing measurements for the whole process of computing the SHA-1 and chameleon hashes and extending the PCR register with the newly created hashes. The measurements show that for a file of 1 MB 14.2% of the total time required to extend a particular PCR is taken for computing the **CH** value. This percentage falls further when larger files are executed. Thus, we believe that Chameleon Attestation can be implemented in current Trusted Computing platforms with reasonable overhead.

6 Conclusion

In this paper we have considered the problem of privacy and scalability in remote attestation, as standardized by the Trusted Computing Group. In particular, the use of SHA-1 hashes to measure the integrity of programs and system components creates a large management overhead; in addition, remote attestation causes privacy problems, as the full state of the system is disclosed. To mitigate these problems we proposed Chameleon Attestation, where we can assign a single hash value to sets of trusted software. By a prototypical implementation we show that the performance overhead of using public-key operations in the attestation process is acceptable.

Acknowledgments. The authors would like to thank Carsten Büttner who helped in the implementation of our approaches. A special thank goes to Bertram Poettering who had early access to this paper and made valuable comments. The feedback and comments from all members of the SECENG group were much appreciated.

References

1. Brickell, E., Camenisch, J., Chen, L.: Direct Anonymous Attestation. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington DC, USA, pp. 132–145. ACM, New York (2004)
2. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: 13th USENIX Security Symposium, San Diego, CA, USA, August 2004, USENIX Association (2004)

3. Lyle, J., Martin, A.: On the feasibility of remote attestation for web services. In: 2009 International Conference on Computational Science and Engineering, Vancouver, BC, Canada, pp. 283–288 (2009)
4. Sadeghi, A., Stüble, C.: Property-based attestation for computing platforms: caring about properties, not mechanisms. In: Proceedings of the 2004 Workshop on New Security Paradigms, Nova Scotia, Canada, pp. 67–77. ACM, New York (2004)
5. England, P.: Practical techniques for operating system attestation. In: Lipp, P., Sadeghi, A.-R., Koch, K.-M. (eds.) Trust 2008. LNCS, vol. 4968, pp. 1–13. Springer, Heidelberg (2008)
6. Krawczyk, H., Rabin, T.: Chameleon hashing and signatures. In: Proceedings of the Network and Distributed System Security Symposium, pp. 143–154. The Internet Society, San Diego (2000)
7. Ateniese, G., de Medeiros, B.: On the key exposure problem in chameleon hashes. In: Blundo, C., Cimato, S. (eds.) SCN 2004. LNCS, vol. 3352, pp. 165–179. Springer, Heidelberg (2005)
8. Chaum, D., van Heyst, E.: Group signatures. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 257–265. Springer, Heidelberg (1991)
9. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, p. 644. Springer, Heidelberg (2003)
10. Brickell, E., Li, J.: Enhanced privacy ID: a direct anonymous attestation scheme with enhanced revocation capabilities. In: Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society, Alexandria, Virginia, USA, pp. 21–30. ACM, New York (2007)
11. Chen, X., Feng, D.: A new direct anonymous attestation scheme from bilinear maps. In: International Conference for Young Computer Scientists, pp. 2308–2313. IEEE Computer Society, Los Alamitos (2008)
12. Kühn, U., Selhorst, M., Stüble, C.: Realizing property-based attestation and sealing with commonly available hard- and software. In: STC 2007: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing, pp. 50–57. ACM, New York (2007)
13. 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, San Jose, California, vol. 3, p. 3. USENIX Association (2004)
14. Yoshihama, S., Ebringer, T., Nakamura, M., Munetoh, S., Maruyama, H.: WS-Attestation: efficient and Fine-Grained remote attestation on web services. In: Proceedings of the IEEE International Conference on Web Services, pp. 743–750. IEEE Computer Society, Los Alamitos (2005)
15. Alam, M., Nauman, M., Zhang, X., Ali, T., Hung, P.C.: Behavioral attestation for business processes. In: IEEE International Conference on Web Services, pp. 343–350. IEEE Computer Society, Los Alamitos (2009)
16. Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 41–55. Springer, Heidelberg (2004)