Key Attestation from Trusted Execution Environments

Kari Kostiainen[1], Alexandra Dmitrienko[2⋆], Jan-Erik Ekberg[1], Ahmad-Reza Sadeghi[2], N. Asokan[1]

[1] Nokia Research Center, Helsinki, Finland
{kari.ti.kostiainen, jan-erik.ekberg, n.asokan}@nokia.com
[2] Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany
alexandra.dmitrienko@rub.de, ahmad.sadeghi@trust.rub.de

Abstract. Credential platforms implemented on top of Trusted Execution Environments[3] (TrEEs) allow users to store and use their credentials, e.g., cryptographic keys or user passwords, securely. One important requirement for a TrEE-based credential platform is the ability to *attest* that a credential has been created and is kept within the TrEE. Credential properties, such as usage permissions, should be also attested. Existing attestation mechanisms are limited to attesting which applications outside the TrEE are authorized to use the credential. In this paper we describe a novel key attestation mechanism that allows attestation of both TrEE internal and external key usage permissions. We have implemented this attestation mechanism for mobile phones with M-Shield TrEE.

1 Introduction

Cryptographic protocols use credentials to authenticate users to various security sensitive services, including on-line banking and corporate network access. Traditional credential solutions fall short. Software credentials, such as passwords, are vulnerable to on-line fraud [4] and software attacks [12]. Dedicated hardware tokens, such as SIM-cards used for authentication in cellular networks, provide higher level of security, but are expensive to manufacture and deploy, and a separated hardware token is typically needed for each service, which forces users to have multiple tokens.

Recently, hardware-based commodity general-purpose Trusted Execution Environments (TrEEs), such as Trusted Platform Module (TPM) [17], JavaCard [6], M-Shield [14] and ARM TrustZone [1], have started to become widely deployed. TPMs are already available on many high-end personal computers while several mobile phone models are based on TrEEs like M-Shield and TrustZone. Credential platforms implemented on top of these TrEEs, including On-Board

⋆ Supported by the Erasmus Mundus External Co-operation Window Programme of the European Union

[3] A trusted EE is a computing environment where execution takes place as expected

Credentials [7] and Trusted Execution Module [3], provide higher level of security compared to software credentials, and easier deployment and better usability compared to dedicated hardware tokens.

Credential platforms [7,3] allow third-parties to implement their own "credential programs" that are executed within the TrEE in a controlled manner. These credential programs may generate new asymmetric keys within the TrEE. One important requirement for a credential platform is the ability to *attest* that a key has been created and is kept within the TrEE. Additionally, the attestation should prove key properties, such as usage permissions. A straightforward approach would be to limit the usage permissions of such keys only to the credential program that generated the key. However, in some cases the developer of the credential program should be able to authorize other credential programs to use the key. Then the credential platform should be able to enforce specified by the developer key usage permissions and to provide an attestation of these permissions to an external verifier.

The following use case provides an example: IT department of a company creates a credential program that generates an asymmetric key within the TrEE and performs (possibly proprietary) corporate network authentication operation. The employees of the company may use this credential program to create themselves a corporate network authentication credential and enroll it to the authentication system of the company. Later, the same IT department wants to issue another credential program to their employees; this time for email signing. The email signing credential program should be allowed to operate on the same, already enrolled key, to save the employees from enrolling multiple keys (typically each enrollment operation requires some user interaction). At the same time credential programs developed by other companies should not be able to use this key. The credential platform should provide an attestation of these key properties to the enrollment server of the company, so that only compliant keys are enrolled to the authentication system of the company.

Contribution. In this paper we describe an extension to On-board Credentials platform [7] that enables credential program developers and applications to define which other entities both within the TrEE and externally are authorized to use the asymmetric keys they generate and for which operations these keys may be used. We also describe a key attestation mechanism that provides evidence on internal and external key usage permissions to a verifier. To the best of our knowledge, no other credential platform provides similar functionality. We have implemented the key attestation mechanism and matching key property enforcements for Symbian mobile phones with M-Shield TrEE.

2 Requirements and Assumptions

Requirements. The main objective is to design a framework for a credential platform that allows credential programs, written by third-parties, to generate new asymmetric keys within the TrEE and to prove certain properties of these

keys to any (correct) verifying entity. More concretely, the credential platform should support the following features:

R1: Key usage and usage permission definition. The *key creator*, i.e., the entity who generates a new key, should be able to define (i) key usage, i.e., allowed key operations (e.g., signing, decryption) and (ii) key usage permissions by defining entities, both internal and external to TrEE, which are authorized to use the key. In particular, the key creator should be able to authorize key usage for an entity whose exact identity is not known at the time of key generation (e.g., other credential programs written by the same credential developer in future).

R2: Key usage permissions update. The key creator should be able to update key usage permissions after the key has been generated. Such a possibility should be optional and be allowed or restricted at the time of key generation.

R3: Key usage enforcement. The credential platform should enforce key usage and key usage permissions defined by the key creator.

R4: Attestation coverage. The credential platform should provide an (externally) verifiable evidence/proof that the *subject key* was created and is accessible only within the TrEE. Additionally, the attestation should provide evidence on the following subject key properties: (i) key creator, (ii) key usage (signing, decryption), (iii) key usage permissions (entities which are authorized to use the key), and (iv) indication whether the key creator is allowed to update key usage permissions.

R5: Attestation unforgeability. The credential platform should only attest credentials it has generated itself and which are under its control. In other words, an attacker should not be able to fool the credential platform to attest keys generated by the attacker.

R6: Attestation freshness. In case the creator of the key is allowed to update key usage permissions (R2), an external verifier should not trust previous (old) attestations (the key creator might have changed key usage permissions after the old attestation was created). Thus, the key attestation mechanism should provide freshness guarantee.

Assumptions. We make some assumptions regarding underlying hardware and operating system level security:

A1: Trusted execution environment. We assume availability of a hardware-based TrEE that provides: (i) isolated code execution (by means of separation of processing and memory), (ii) secure storage (by ensuring integrity and confidentiality of persistent data), (iii) integrity protection of secure execution environment for credential programs.

A2: OS security. We assume existence of operating system level platform security framework with the following features: (i) availability of the secure storage for OS level applications/processes, (ii) access control on inter-process communication, (iii) integrity protection of security critical components, (iv) isolation of application/process execution, and (v) access control model that allows only trusted (e.g. signed) OS-level components to communicate with TrEE.

Note that these assumptions are reasonable in the context of our primary implementation platform: We utilize M-Shield [14] security hardware and Sym-

bian [10] operating system. M-Shield provides all required features for TrEE. First, it supports secure boot[4] which ensures integrity of TrEE. Second, M-Shield supports secure code execution in hardware by means of separation of processing and memory. Third, it provides the secure storage by means of sealing all data with a device-specific symmetric key which is protected by the TrEE. The Symbian OS provides the application-specific secure storage and process execution isolation, and enforces control on inter-process communication via capability mechanism[5]. Moreover, the integrity of security critical OS components is ensured with secure boot that utilizes M-Shield hardware.

Adversary Model. We assume the following adversary capabilities:

AC1: Communication channel attacks. The adversary has access to communication channel between the attesting device and the external verifier and is able to eavesdrop, reply, relay or alert any network traffic.

AC2: End-point software attacks. The adversary can launch software attacks targeting the ObC platform. The execution of the OS-level components cannot be affected and OS-level secure storage cannot be accessed **if** the adversary is not able to compromise OS platform security at runtime. Assuming inability to compromise OS security framework may not be realistic due to the large size of modern operating systems and in Section 6 we discuss the implications of OS security compromise to our proposal.

AC3: End-point hardware attacks. The adversary can launch limited subset of hardware attacks on a circuit board level. We assume that the adversary is not able to tamper with chips and launch side-channel attacks, but can eavesdrop on the conductor wires connecting components or try to modify data or program code stored on the device (e.g., via programming interface).

3 On-board Credentials Platform

In this section we give a brief overview of the On-board Credentials (ObC) platform. Figure 1 describes the parts of the ObC platform architecture that are relevant to key usage control and attestation. For more detailed description of the ObC platform see [7].

Interpreter. The core of the ObC platform is a trusted Interpreter that can be executed within the TrEE. The trust on the Interpreter can be based on code signing, i.e., only authorized code is allowed to be executed within the TrEE. Interpreter provides a virtualized environment where "credential programs", i.e., scripts developed by untrusted third-parties, can be executed. When a credential program is executed, the Interpreter isolates it from secrets that are stored within the TrEE and from the execution of other credential programs.

[4] Secure boot means a system terminates the boot process in case the integrity check of a component to be loaded fails [5]

[5] A capability is an access token that corresponds to access permissions [10]

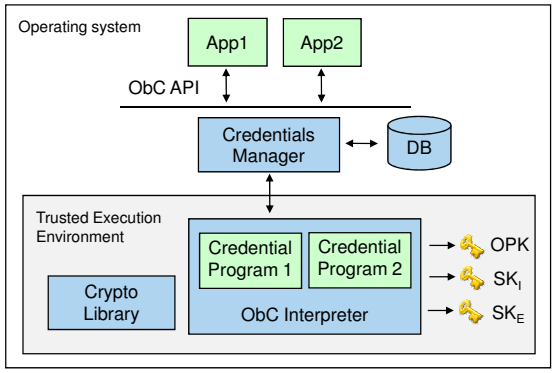

Fig. 1. On-board Credential architecture

The Interpreter provides a sealing[6]/unsealing function for ObC programs, which can be used to protect secret data stored persistently outside the TrEE. Additionally, the Interpreter provides common cryptographic primitives, such as encryption, decryption and hash functions, for credential program developers. The credential programs are written using (a subset of) Lua scripting language [8] or in assembler.

The ObC platform supports three types of credentials: (1) credential programs that operate on symmetric secrets provisioned by an external provisioner, (2) credentials programs that locally generate and operate on asymmetric keys, and (3) asymmetric keys locally generated by applications without involvement of credential programs. In this paper we focus on two latter credential types.

The ObC platform supports a concept of "credential families". A family is defined by a credential provisioner (full description of credential provisioning and families can be found in [7]). Credential programs belonging to the same family may share sealed and persistently stored data.

Credential Manager. The ObC platform includes a trusted operating system level component called Credentials Manager *CM*. The trust in *CM* can be provided, e.g., based on secure boot. *CM* provides an API for third-party developed applications. Using the API the applications can execute credential programs, and create and use new asymmetric keys. *CM* maintains a database, in which credentials and key properties are stored. *CM* also enforces that only authorized applications are allowed to use credentials.

Device keys. The ObC platform uses three device specific keys (which are only accessible within the TrEE) for key generation and attestation:

[6] Protecting an object so that only a certain set of OS-level or TrEE-level entities can access or use it

- **ObC platform key** (OPK) is a symmetric device key. The Interpreter uses OPK for sealing/unsealing function.
- **Internal device key** (PK_I, SK_I) is an asymmetric device key. The public part of this key is certified as an "internal device key" by the device manufacturer. The Interpreter uses this key only to sign data that originates from within the TrEE, or data whose semantics or structure it can verify.
- **External device key** (PK_E, SK_E) is an asymmetric device key. The public part of this key is certified as an "external device key" by the device manufacturer. The Interpreter uses this key to sign data that originates from outside the TrEE. Using secure boot and OS-level security framework, we limit the use of the external device key to CM only.

4 Key Attestation Design

Key attestation protocols involve the following entities: (i) attestor A, i.e., ObC platform which attests to properties of the locally generated *subject* key, (ii) the platform manufacturer M which certifies device specific keys of A, (iii) a server S which aims to get assurance regarding subject key properties, and (iv) certification authority CA which may issue subject key certificate.

We utilize the following notations: A signature scheme consists of algorithms $(\mathsf{GenKey}(), \mathsf{Sign}(), \mathsf{Verify}())$. Here $(SK, PK) \leftarrow \mathsf{GenKey}()$ is the key generation algorithm that outputs signing key (private key) SK, and the corresponding verification key (public key) PK. $\sigma \leftarrow \mathsf{Sign}(SK, m)$ is the signature algorithm on message m which outputs a signature σ, and $ind \leftarrow \mathsf{Verify}(PK, \sigma, m)$ is the signature verification algorithm with $ind \in \{0, 1\}$.

An authenticated encryption[7] scheme consist of algorithms $(\mathsf{Enc}(), \mathsf{Dec}())$. Here $c \leftarrow \mathsf{Enc}(K, m)$ is the encryption algorithm on a message m using K as the symmetric key which outputs an encrypted message c, and $(ind, m) \leftarrow \mathsf{Dec}(K, c)$ is the decryption algorithm on c using K as the symmetric key with $ind \in \{0, 1\}$ indicating integrity of c.

A hash algorithm is denoted by $\mathsf{H}()$.

4.1 Key Generation by Credential Programs

Key generation by a credential program is illustrated in Figure 2. We describe the main steps in the following:

Step 1: A credential program requests the Interpreter to generate the subject key. It may authorize other credential programs to use this key in two ways: (i) **Credential program identifiers** are used to define zero or more identifiers of credential programs that are authorized to use the generated key. In ObC platform, credential programs are identified by the hash of the program code; (ii) **Family identifiers** are used to define zero or more identifiers of credential

[7] We use authenticated encryption AES-EAX for various needs including sealing/unsealing operations to keep code and memory footprint minimal

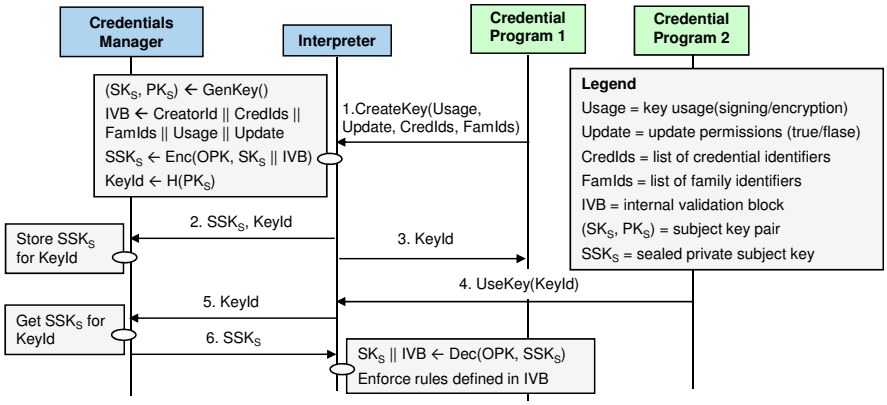

Fig. 2. Key generation by a credential program

families that are authorized to use the generated key. Credential families are identified by the hash of family key[8]. The credential program also defines key usage and whether key usage permissions are allowed to be updated.

Step 2: The Interpreter generates a new subject key (SK_S, PK_S) and creates a structure called *internal validation block* (*IVB*). *IVB* contains (i) the identifier *CreatorId* of the credential program that created the key, (ii) a list of credential program identifiers *CredIds*; (iii) a list of family identifiers *FamIds*, (iv) an indication whether credential program identifiers and family identifiers may be updated by the key creator *Update* and (v) the allowed key operations *Usage*. The Interpreter seals the private part of the subject key SK_S and *IVB* using platform key *OPK*. Then it derives the key identifier *KeyId* by hashing the public key PK_S. The resulting sealed key SSK_S and *KeyId* are stored on the operating system side by *CM*.

Step 3: The key identifier *KeyId* is returned to the credential program that may, e.g., export it to the application that triggered the credential program execution, so that the same key can be used later (from the same or another credential program).

Steps 4-6: The next time the key is used, the Interpreter requests, and obtains the sealed key from *CM* on operating system side based on *KeyId*, and unseals it using *OPK*. The Interpreter unseals SSK_S and verifies *IVB* components and performs the requested key operation only if the key usage is allowed, and the calling credential program is either the creator of the key or its identifier, or family is listed as authorized to use the key.

[8] Family key is used in credential provisioning

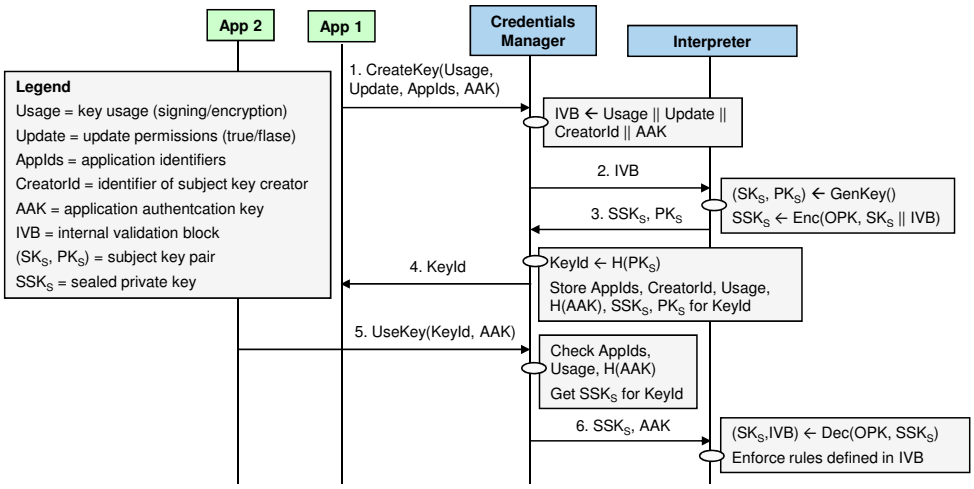

Fig. 3. Key generation by an application

4.2 Key Generation by Applications

The Credentials Manager *CM* provides an API for creating and using asymmetric keys directly from applications. Figure 3 illustrates key generation by an application.

Step 1: An application calls the key creation function over the API provided by *CM*. The application may authorize other applications to use the generated key in two ways: (i) to define zero or more **application identifiers**. The listed applications are permitted to use the key. This method requires that the underlying OS can provide reliable information about the identity of the calling application to *CM*[9]; (ii) to define that an authorization token called **application authentication key** (AAK) is required to use the key. In such a case the generated key may be only used if the correct AAK is provided by the application to *CM*. AAK may be shared among several applications.

When an application creates a new key, *CM* constructs IVB. Application identifiers *AppIds* are not included in IVB, since those cannot be reliably verified within the TrEE. If AAK is used, it is included in IVB together with the key usage *Usage*, the identity of the application that generated the key *CreatorId* and a flag *Update* that defines whether key usage permissions can be updated.

Step 2: *CM* loads IVB to the TrEE, in which the Interpreter generates the subject key (SK_S, PK_S), and seals the private part together with IVB to SSK_S.

Steps 3-4: PK_S and SSK_S are returned to *CM*. *CM* stores them together with hash of AAK, the list of application identifiers *AppIds*, the key creator *CreatorId*, and the key usage *Usage*. A key identifier *KeyId* is returned to the application.

[9] For example, in Symbian OS each process has a unique identifier which can be verified for each inter-process function call

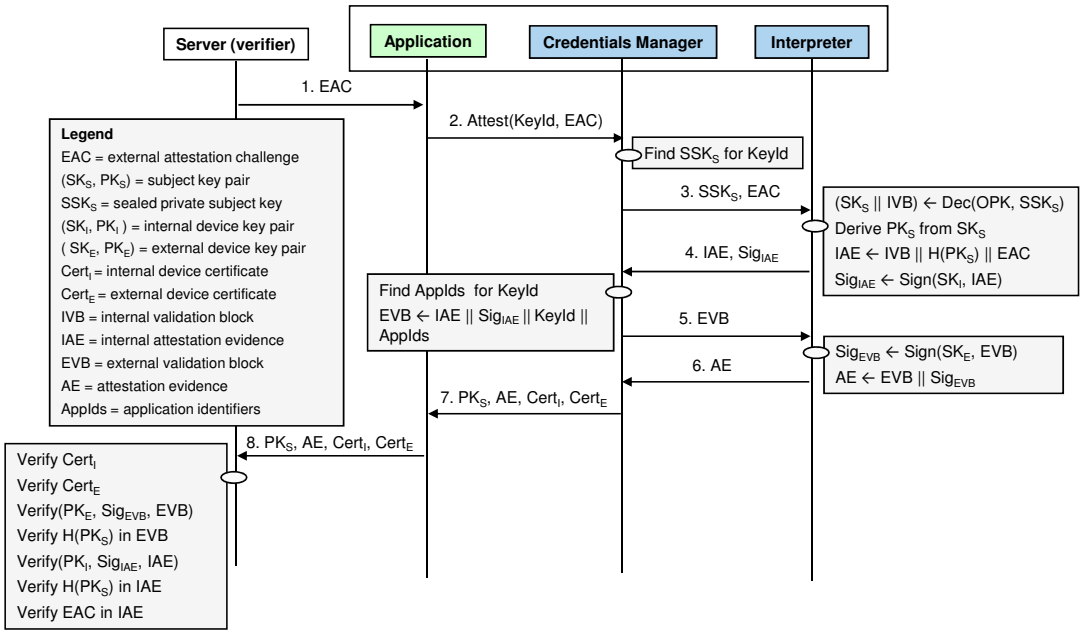

Fig. 4. Interactive key attestation

Steps 5-6: When the same or another application requests to use the subject key, *CM* verifies the identifier of the calling application with respect to locally stored *CreatorId* and *AppIds*. *CM* also checks key usage *Usage* and hash of AAK if needed. If these checks pass, *CM* loads the sealed private key SSK_S and possible AAK to the TrEE in which the Interpreter unseals the key and checks that AAK matches the one defined in IVB (if used), and that key usage is allowed before performing the private key operation.

4.3 Interactive Key Attestation

The attestation process must be interactive in case the key creator is authorized to update key usage permissions during key life time (as required by the objective (R2)). In interactive scenario, the attestation evidence must be verified by the server S. Figure 4 illustrates this attestation protocol.

Steps 1-2: S picks a random nonce called *external attestation challenge* (EAC) and sends it to an application on the target device. The application identifies the subject key to attest (typically based on information originating from S) and triggers the attestation. *CM* retrieves the sealed subject key SSK_S from local storage.

Steps 3-4: *CM* loads SSK_S and EAC to TrEE. Inside the TrEE the Interpreter first unseals SSK_S, derives PK_S from SK_S[10] and then creates an *internal attestation evidence* (IAE). IAE is a concatenation of IVB, hash $H(PK_S)$ of a subject public key and EAC. Then IAE is signed using the internal device key SK_I. The Interpreter returns IAE and signature Sig_{IAE} to *CM*.

Steps 5-6: *CM* constructs an *external validation block EVB*. *EVB* is a concatenation of IAE, Sig_{IAE}, *KeyId* (hash of public subject key $H(PK_S)$), and a list of application identifiers *AppIds*. *CM* loads *EVB* to TrEE in which the Interpreter signs it using the external device key SK_E. The resulting *attestation evidence* AE is sent back to *CM*.

Steps 7-8: *CM* returns AE together with the subject public key PK_S and certificates of both internal and external device keys ($Cert_I$, $Cert_E$) to the application. The application forwards this data to the server S which verifies the following: (i) AE has been signed with a key that has been certified as an external device key by a trusted authority, and (ii) the public key hash in EVB matches the received subject public key PK_S. If these two conditions hold, S can parse the external key usage permissions and based on that determine which OS level key usage permissions are enforced by *CM*.

To verify the internal attestation, S checks that (i) IAE contains signature made with a key that has been certified as internal device key, (ii) the public key hash inside IAE matches the received subject public key, and (iii) EAC inside IAE matches the one picked by S earlier. If these three conditions hold, S can determine from IVB the key usage permissions enforced by the Interpreter within the TrEE.

4.4 Non-interactive Key Attestation

Non-interactive key attestation can be used when key usage permissions are not allowed to be updated and freshness guarantee is not needed. Figure 5 depicts non-interactive attestation. In this scenario a certification authority *CA* validates the attestation evidence and issues a subject key certificate that other servers can verify. The main steps of the protocol are described below:

Steps 1-2: The credential platform triggers attestation with fixed challenge (e.g., $EAC = 0$). *CM* and Interpreter create the attestation evidence as in interactive key attestation. *CM* generates also a certificate request containing the public part of subject public key, subject identity and proof-of-possession of the subject key[11]. The certificate request, attestation evidence and internal and external device key certificates are submitted to *CA*. *CA* validates AE using the fixed challenge (verification is performed in the same way as in interactive scenario described in Section 4.3). Additionally, *CA* verifies that the key usage permissions are not allowed to be updated, i.e., the field *Update* in IVB structure is set to false. Finally, *CA* issues a subject certificate $Cert_S$ and returns it to the ObC platform.

[10] PK_S can be derived from SK_S efficiently in our implementation

[11] E.g., a signature created using the subject key within the TrEE

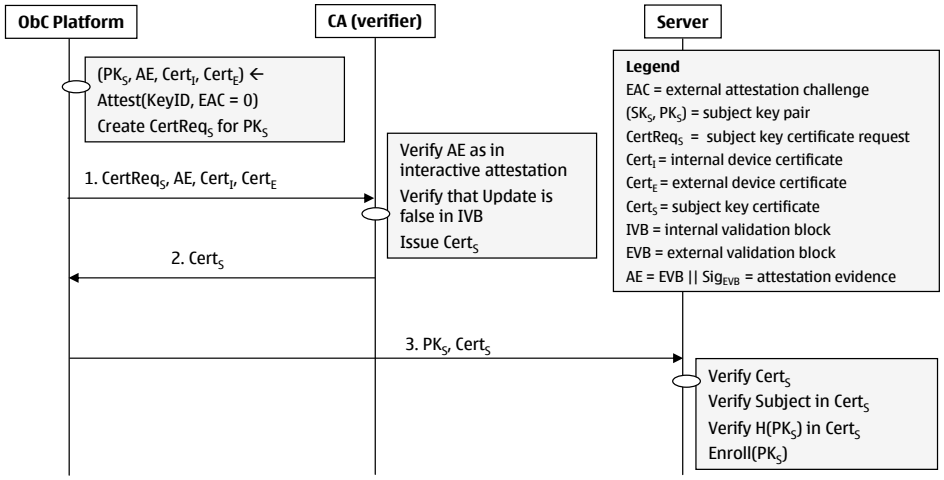

Fig. 5. Non-interactive key attestation

Step 3: The ObC platform submits the key enrollment request to S. The request includes PK_S and $Cert_S$. S validates $Cert_S$, and if it is correct, enrolls the subject key. In this scenario, S relies on CA to verify the attestation evidence. However, since X.509 certificates do not have standard place to indicate if the attestation evidence has been validated by CA, S must have out of band knowledge that the particular CA always validates the attestation evidence before the public key certificate is issued.

Another approach for non-interactive attestation assumes that CA issues public key certificate omitting attestation evidence validation, then attestation evidence is incorporated into subject key certificate. The TCG SKAE [16] defines a X.509 certificate extension for this purpose. In this approach verification of the attestation evidence is left for the server.

Note, that in both scenarios communication between ObC platform and CA must be secured so that CA can associate the submitted public key with the correct authorizations allowed for the submitter.

5 Implementation

We have implemented the described attestation mechanism for Nokia N96 mobile phone with M-Shield TrEE. In M-Shield architecture trusted (signed) code can be executed within the TrEE isolated from the rest of the system. The trusted code is implemented as so called "protected applications" (PAs) in C. The maximum size of each PA is very limited (in terms of both implementation footprint and runtime memory) and for this reason we had to implement the Interpreter, key generation and attestation functionality as three separate PAs: (i) Interpreter PA, (ii) RSA PA and (iii) Attestation PA. Because in M-Shield

architecture the communication between different PA invocations must be mediated by an operating system level component (*CM* in our architecture), the data that is transfered from one PA to another one must be protected.

Interpreter PA is the component that handles credential program execution. When Interpreter PA encounters key creation macro it constructs *IVB*, creates a fresh session key and seals *IVB* and the current state of the program execution using the session key. The Interpreter PA saves the session key to volatile secure memory inside the TrEE and returns *IVB* and program state in sealed format.

CM on the OS side saves sealed program execution state temporarily and loads RSA PA to the TrEE together with the sealed *IVB*. RSA PA unseals *IVB*, generates a new RSA key and seals *IVB* and private part of the generated key using *OPK* for future use. RSA PA calculates key identifier (hash of public key) and returns this sealed with the session key. *CM* loads Interpreter PA to TrEE together with the sealed key identifier and sealed program execution state. The Interpreter PA can unseal the state and the key identifier using the session key and continue credential program execution.

Asymmetric key operations are handled in similar fashion. When Interpreter PA encounters key operation in credential program execution it seals key operation parameters and current state with the session key. *CM* triggers RSA PA which unseals parameters, performs the operation and seals the results for Interpreter PA. Key attestation and application triggered key operations are handled by Attestation PA which requires no communication with other PAs.

The operating system side *CM* component is implemented as a Symbian OS server in C++. Using Symbian OS platform security framework *CM* can check unique identifier of calling application for each function call. *CM* maintains a database in its private directory which is not accessible by other applications (except few trusted system components).

In our implementation, the device keys (internal and external) are generated when the credential platform is first taken into use. The keys are created within the TrEE and sealed using *OPK* for storage in *CM* database. When the device keys are created, a key type *tag* is included to the seal. With the tag, the key type can be determined when the key is later unsealed inside the TrEE.

In our implementation, the internal validation block (*IVB*) is a binary structure with fixed format, to keep the TrEE side implementation minimal. *IVB* can contain up to five identifiers which are used to define credential programs, families and *AAK*. A bitfield in the header defines the types of these identifiers. *IVB* header also defines the key creator, usage and whether usage permission can be updated. We have implemented the external validation block using ASN.1 formatting (similar to TCG SKAE [16]) to make external attestation flexible and easy to implement. For non-interactive attestation our *CM* implementation can generate standard X.509 certificate requests into which the attestation evidence is included as an extension.

We have not yet implemented a mechanism to update key usage permissions. Currently, key usage permissions are always defined as unchangeable at the time of key generation.

6 Security Analysis

Based on the assumptions on the underlying hardware platform (A1) and on the OS security framework (A2) (see Section 2), in the following we will give an informal security analysis of our proposal.

Our design and implementation provide key usage definition (R1) for keys generated both from within or outside the TrEE. The internal key usage permissions are defined in terms of credential program and family identifiers. The external key usage permissions are defined in terms of application identifiers and by means of applying the application authentication tokens. Allowed key operations are defined in *Usage*.

Key usage enforcement (R3) is provided in the following way: the allowed key operations *Usage* and internal usage permissions are enforced by the trusted Interpreter. Note that the Interpreter resides within TrEE. Moreover, the underlying hardware provides secure execution. Hence, the integrity of the Intepreter is ensured both statically and in run-time.

In case of external usage permissions, rules defined through application identifiers are enforced by *CM*. Note that *CM* is a trusted OS-level component, and hence its integrity is provided based on the assumptions regarding OS security framework, so that *CM* can enforce the usage permission rules specified for each credential., e.g., in EVB.

Key usage permissions update (R2) can be supported, because the key creator can be always identified via key creator identity *CreatorId* included into *IVB*. Also, the attestation evidence creation is not bound to time of the key generation, thus it can reflect changes made during key life time. Possible solutions for the key usage permissions update mechanism are discussed in Appendix B.

Attestation coverage (R4) is simply realized by including all required statements into the attestation evidence. Attestation unforgeability (R5) is ensured through the use of the device keys for attestation those are protected by the TrEE and their genuineness is certified by the trusted device manufacturer. Attestation freshness (R6) is guaranteed with inclusion of the challenge in the internal attestation evidence.

Discussion on run-time compromise. As mentioned in Section 2, we cannot generally assume that the adversary cannot compromise OS-level security framework. In this section we discuss implications of OS compromise to our solution.

First, we consider credential program generated keys. As shown in Figure 2, OS-level components including *CM* do not have access to the key properties in unsealed form during key generation process. A compromised *CM* is able to forge an external attestation for credential program generated key with false application level usage permissions, but it does not allow *CM* to use the key since the Interpreter inside the TrEE will deny the key operation invoked by the *CM* for a key generated by a credential program. Internal attestation can be trusted, since it is performed internally by Interpreter within the TrEE. Also, a malicious *CM* is not able to invoke SK_I usage to sign forged *IVB* since the Interpreter will not use this key to sign data that originates outside the TrEE.

The external attestation evidence cannot impersonate the internal attestation evidence since they are signed with different keys, SK_E and SK_I respectively.

Next, we consider application created keys. Again, a compromised *CM* is able to forge external attestation evidence and specify usage permissions for false OS-level applications. If the key usage permissions are defined in terms of application identifiers, a compromised *CM* can allow key usage for unauthorized applications. If the key usage permissions are defined in terms of AAK a compromised *CM* cannot use the key without knowledge of valid AAK. However, one should note that if the adversary is able to compromise *CM* he most likely can read AAK from the storage of the authorized application as well, and thus use the key. Internal attestation can be trusted for application generated keys, only if *CM* has been compromised *after* the key was generated. If *CM* was compromised before key generation, even internal attestation cannot be trusted for application generated keys.

As a conclusion, our design and implementation can only partly address the problem of runtime compromise of OS-level security framework[12]. Thus in real life scenarios the verifier should take into account the discussed arguments and define the trust to the attestation created by the ObC platform according to its security policy.

7 Related Work

Trusted Computing Group (TCG) [15] has specified a mechanism called Subject Key Attestation Evidence (SKAE) [16] for attesting TPM generated asymmetric keys. In short, a SKAE attestation contains the public part of the attested subject key and the platform configuration (in terms of platform configuration register values) under which the subject key can be used, signed with a certified and device-specific attestation identity key. A typical use of SKAE is to include it as an extension to a certificate request; the SKAE extension proves to the certificate authority that the subject key was created and is kept within a TPM and specifies the application(s) that can use the key by defining the platform configuration.

The TCG SKAE is limited to attesting which applications *outside* the TrEE are allowed to use the attested subject key whereas our attestation mechanism provides evidence on TrEE-internal key usage permissions as well. Moreover, the TCG SKAE is a non-interactive mechanism, and thus not applicable to attesting keys which usage setting may be updated (R2).

The work closest to ours is "outbound authentication" (OA) architecture [13] for IBM 4758 programmable secure coprocessors. IBM 4758 is TrEE with layered security architecture: layers 0-2 boot up the coprocessor and run an operating

[12] It should be noted that handling runtime compromise is still an open research problem and the existing solutions such as Runtime Integrity Monitors either require extra hardware support (e.g., [9]) or utilize virtualization technology to run the system under inspection within a virtual machine (e.g., [2]) which is hard affordable for mobile devices due to the corresponding overhead

system. Applications originating from different (possibly mutually distrusting) sources can be loaded to the coprocessor and executed on layer 3. External parties should be able to verify which of the applications within the coprocessor performed certain operation. The OA architecture uses certificate chaining to achieve this. Layer 0 has a root key (certified by a trusted authority) which is used to certify higher layers. When an application is executed, the operating system layer creates a key for the application and certifies this key. The application may authenticate itself to an external verifier using its key.

Our attestation mechanism and OA architecture address essentially the same problem — providing evidence on which entity within a TrEE is allowed to access a certain key. However, our attestation mechanism supports certain features that fall outside the scope of OA. First, the ObC architecture supports sharing of keys between entities within the TrEE and our attestation mechanism provides evidence on this in terms of credential programs and family identifiers. Second, our attestation mechanism provides also evidence on TrEE external access.

KeyGen2 [11] is a proposal for provisioning of asymmetric keys to devices, such as mobile phones. In KeyGen2 asymmetric keys are created inside the TrEE of the client device. To enroll a key to a server, the client creates an attestation of the key by signing it with a device key. To distinguish this attestation signature from other signatures made with the *same* device key, special padding (reserved for this use only) is applied.

The key attestation in KeyGen2 does not include information about software that is authorized to use the key neither in terms of platform configuration (as it is done in the TCG SKAE), nor in form of TrEE internal key usage permissions (as in our proposal). The attestation only proves that the to-be-enrolled key was created and is kept within the TrEE.

8 Conclusion

In this paper we have described a key attestation mechanism that allows a platform to attest to a verifier key usage permissions and properties of both (internal) programs residing in a Trusted Execution Environment (TrEE) as well as OS-side applications outside the TrEE. We have implemented this key attestation mechanism and matching local enforcements as an extension to the existing on-board Credentials platforms for mobile phones based on M-Shield secure hardware. To the best of our knowledge, this is the first credential platform that efficiently provides such an enhanced attestation functionality.

References

1. ARM. Trustzone technology overview. http://www.arm.com/products/security/trustzone/index.html, 2009.
2. Fabrizio Baiardi, Diego Cilea, Daniele Sgandurra, and Francesco Ceccarelli. Measuring semantic integrity for remote attestation. volume 5471 of *Lecture Notes in Computer Science*, pages 81–100, 2009.
3. Victor Costan, Luis Sarmenta, Marten van Dijk, and Srinivas Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *Proc. Eighth Smart Card Research and Advanced Application Conference*, August 2008. http://people.csail.mit.edu/devadas/pubs/cardis08tem.pdf.
4. Internet Crime Complaint Center. Internet crime report. http://www.ic3.gov/media/annualreport/2008_IC3Report.pdf, 2008.
5. Naomaru Itoi, William A. Arbaugh, Samuela J. Pollack, and Daniel M. Reeves. Personal secure booting. In *ACISP '01: Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pages 130–144, Jul 2001.
6. JavaCard Technology. http://java.sun.com/products/javacard/.
7. Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *Proc. of ACM Symposium on Information, Computer & Communications Security (ASIACCS'09)*, 2009.
8. The Programming Language Lua. http://www.lua.org/.
9. Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194. USENIX, August 2004.
10. Nokia. Symbian OS platform security. www.forum.nokia.com/Technology_Topics/Device_Platforms/S60/Platform_Security/.
11. Anders Rundgren. Subject key attestation in keygen2. http://webpki.org/papers/keygen2/keygen2-key-attestation-1.pdf, 2009.
12. SANS Institute. SANS Top-20 2007 Security Risks. http://www.sans.org/top20/2007/top20.pdf, November 2008.
13. Sean W. Smith. Outbound authentication for programmable secure coprocessors. *International Journal of Information Security*, 3:28–41, 2004.
14. Jay Srage and Jerome Azema. M-Shield mobile security technology, 2005. TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.
15. Trusted Computing Group. https://www.trustedcomputinggroup.org/home.
16. TCG Infrastructure Workgroup. *Subject Key Attestation Evidence Extension", Specification Version 1.0 Revision 7*, June 2005. https://www.trustedcomputinggroup.org/specs/IWG/.
17. Trusted Platform Module (TPM) Specifications. Available at: https://www.trustedcomputinggroup.org/specs/TPM/.

Appendix A Device Key Alternatives

The key attestation mechanism described in the paper requires two device keys: Internal device key is used to sign internal attestation and external device key is used to sign external attestation. Both of them must be certified by a trusted authority, such as the device manufacturer. The device keys can be created either during device manufacturing or when the credential platform is first taken into use. In the latter case, the device key certification is an on-line protocol between the TrEE and the device manufacturer – we assume that the device manufacturer may authenticate its own TrEEs in reliable fashion.

Creating asymmetric keys is a time consuming process on TrEEs with limited resources. Thus, the need to have two certified device keys increases the device manufacturing time and cost, or alternatively decreases credential platform installation user experience. In this appendix we discuss two alternative approaches to device key creation and certification to address this problem.

Single device key. Instead of creating two separate device keys, a single device key could be used for signing both internal and external attestations. In such a case, signatures made over *IVB* and *EVB* should be distinguishable from each other to prevent *EVB* to be interpreted as *IVB* by the verifier. To distinguish different type of signatures made with the same key, one of the following two techniques could be applied.

First, the Interpreter could apply distinguishable formatting to *IVB* and *EVB* before signing them. The Interpreter could, e.g., concatenate a tag to these elements before signing. If such an approach were used, the device key should never sign anything else except an attestation evidence, otherwise the specific formatting can be forged. Thus, such a solution does not scale well since may require new device keys for other operations.

Second, the signatures can be made distinguishable by applying different padding schemes or hashing algorithm as proposed in [11]. For example a unique padding could be used for internal attestation signatures, another unique padding for external attestation signatures, and standard padding could be used for normal signatures. The disadvantage of this approach is that the external verifier is required to understand these non-standard padding schemes which can be an obstacle for wide scale deployment.

Device key chaining. Another alternative would be to use certificate chaining. In this approach two separate device keys would be used for signing the attestations, but the device manufacturer would have to certify only *one* device key which in turn could certify the second needed device key locally on the platform. The benefit of such an approach is that only one device key has to be created when the device is manufactured or when the credential platform is taken into use. The second device key can be generated and certified later, e.g., when the device is in idle state, but before the device is used for attestation. This approach would also scale better, if more than two device keys are needed.

Appendix B Key Usage Permissions Update

The task of updating key usage permissions can be seen as consisting of two subtasks: (i) to grant usage rights to new credential programs and applications; (ii) to revoke usage rights granted before.

One alternative would be to provide the key creator the possibility to do both, to grant and to revoke key usage permissions. In this way, lists of credential programs and applications authorized to use the key may be freely modified by the key creator.

Another alternative would be to provide the key creator the only possibility to revoke key usage permissions. In this way, identities of credential programs and applications may be excluded from the lists defined before, but new identities may not be added. In this situation, key usage permissions can be granted via utilization of already available mechanisms: Family paradigm can be used to grant usage permissions to additional credential programs, and application authentication token can be used to grant usage permissions to new applications.

The former design solution provides better flexibility, since family identifiers and application tokens can be added and updated by the key creator. The latter design solution is less flexible, but it does not require to ensure attestation freshness. Indeed, if the old attestation is satisfactory for the verifier, the new one would be also for sure accepted because it has reduced list of authorized entities compare to the old version. When freshness is not required, the attestation could be always performed in non-interactive manner, that is an advantage of this scheme.