NOKIA

Research Center
NRC-TR-2008-007

On-board Credentials with Open Provisioning

Jan-Erik Ekberg[1], N. Asokan[1], Kari Kostiainen[1], Aarne Rantala[2]

[1]Nokia Research Center, Helsinki
http://research.nokia.com
[2]Technical Research Center, Finland
http://www.vtt.fi

August 29, 2008

Abstract:

Abstract

Securely storing and using credentials is critical for ensuring the security of many modern distributed applications. Existing approaches to address this problem fall short. User memorizable passwords are flexible and cheap, but they suffer from bad usability and low security. On the other hand, dedicated hardware tokens provide high levels of security, but the logistics of manufacturing and provisioning such tokens are expensive, which makes them unattractive for most service providers. A new approach to address the problem has become possible due to the fact that several types of general-purpose secure hardware, like TPM and M-shield, are becoming widely deployed. These platforms enable, to different degrees, a strongly isolated secure environment. In this paper, we describe how we use general-purpose secure hardware to develop an architecture for credentials which we call On-board Credentials (ObCs). ObCs combine the flexibility of virtual credentials with the higher levels of protection due to the use of secure hardware. A distinguishing feature of the ObC architecture is that it is open: it allows anyone to design and deploy new credential algorithms to ObC-capable devices without approval from the device manufacturer or any other third party. The primary contribution of this paper is showing and solving the technical challenges in achieving openness while avoiding additional costs (by making use of already deployed secure hardware) and without compromising security (e.g., ensuring strong isolation) Our proposed architecture is designed with the constraints existing secure hardware in mind and has been prototyped on several different platforms including mobile devices based on M-Shield secure hardware.

Index Terms:

security and privacy
authentication
access control
secure execution environment
trusted computing
provisioning

On-board Credentials with Open Provisioning

Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, Aarne Rantala

Abstract

Securely storing and using credentials is critical for ensuring the security of many modern distributed applications. Existing approaches to address this problem fall short. User memorizable passwords are flexible and cheap, but they suffer from bad usability and low security. On the other hand, dedicated hardware tokens provide high levels of security, but the logistics of manufacturing and provisioning such tokens are expensive, which makes them unattractive for most service providers. A new approach to address the problem has become possible due to the fact that several types of general-purpose secure hardware, like TPM and M-shield, are becoming widely deployed. These platforms enable, to different degrees, a strongly isolated secure environment. In this paper, we describe how we use general-purpose secure hardware to develop an architecture for credentials which we call *On-board Credentials* (ObCs). ObCs combine the flexibility of virtual credentials with the higher levels of protection due to the use of secure hardware. A distinguishing feature of the ObC architecture is that it is *open*: it allows anyone to design and deploy new credential algorithms to ObC-capable devices without approval from the device manufacturer or any other third party. The primary contribution of this paper is showing and solving the technical challenges in achieving openness while avoiding additional costs (by making use of already deployed secure hardware) and without compromising security (e.g., ensuring strong isolation) Our proposed architecture is designed with the constraints existing secure hardware in mind and has been prototyped on several different platforms including mobile devices based on M-Shield secure hardware.

1 Introduction

Cryptographic protocols play an essential role in protecting distributed applications like access to enterprise networks, on-line banking, and access to other web-based services in general. These protocols make use of *credentials*, consisting of items of secret data, like keys, and associated algorithms that apply cryptographic transforms to the secret data. Securely storing and using credentials is critical for ensuring the security of the applications that rely on them.

Existing approaches to address this problem fall short. The most prevalent approach currently used for user authentication is based on user passwords and requires users to memorize passwords. This suffers from bad usability and is vulnerable to phishing. Although various identity management systems supporting Single Sign-On would minimize the number of passwords a user has to remember, it is unlikely that all the services a user wants to use would rely on the same trust domain. In other words, a user will need to be able to authenticate to many different trust domains. "Password managers", such as those found in popular web browsers, ease the usability problem somewhat, but are open to software attacks, like Trojans that steal passwords.

At the other extreme, dedicated hardware tokens provide high levels of security. The most widespread example of a hardware security token is the smartcard containing the subscriber identity module (SIM) used for authenticating access to Global System for Mobile Communications (GSM) cellular networks. However, the logistics of manufacturing and provisioning hardware tokens are expensive, which makes it unattractive for most service providers to issue their own hardware tokens. Although multi-application smartcards exist, and can support different credentials on the same card, they are not in widespread use with credentials from *multiple sources.* This is primarily because applications have to be authorized with respect to a finite set of trust domains (e.g., the smartcard issuer's domain) pre-loaded on the card. Although sometimes such restrictions are policy decisions, in some cases, they are crucial to the security of the system. An example of the latter is the case of javacards that do not have a bytecode verifier. The consequence of such restrictions, technical or otherwise, is that a service provider who wants to use an existing installed base of multi-application smarcards has to obtain permission from the card-issuer in order to deploy new credentials to them. Such procedural obstacles, in turn, makes it unattractive for service providers to share the hardware tokens issued by others. This has led to a situation that in practice, users end up having to carry several different hardware tokens to authenticate to different services.

Thus, on the one hand we have a cheap, flexible but not very secure software-only solutions like password managers, and on the other hand we have more secure, but expensive, inflexible, and usually dedicated solutions like hardware tokens.

In the last decade or so, several types of general-purpose secure hardware have been incorporated into end user devices and are starting to be widely deployed. These include Trusted Platform Modules (TPM) [19] and Mobile Trusted Modules [6] specified by the Trusted Computing Group and other platforms like M-Shield [17] and ARM TrustZone [1]. All these platforms enable, to different degrees, a strongly isolated secure environment, consisting of secure storage, and in some cases supporting secure execution where processing and memory are isolated from the rest of the system. TPMs are already available on many high-end personal computers. Several mid-range and high-end Nokia phones utlilize hardware security features based on the M-Shield platform.

In this paper, we first describe how we use such general-purpose secure hardware to develop an architecture for credentials which we call "On-board Credentials" (ObCs) and then focus on secure provisioning of ObCs. ObCs combine the flexibility of virtual credentials with the higher levels of protection due to the use of secure hardware. Our contribution in this paper is to define an architecture for credentials that is *simultaneously*

- **inexpensive** to deploy, by making use of existing general-purpose secure hardware rather than designing and provisioning new hardware tokens,
- **open**, so as to allow any service provider to provision new credential secrets as well as new credential algorithms to a user's device without having to co-ordinate with or obtain permission from any third party, and
- **secure** enough such that the credentials are protected from software and hardware attacks to the extent permitted by the underlying secure hardware.

We begin by describing our assumptions about the underlying secure environment and the requirements for an open credential architecture. We then give an overview of the ObC architecture followed by a more detailed description of the provisioning architecture which allows anyone to design and deploy credential algorithms without any third-party screening or approval, while still protecting malicious credential algorithms from stealing other credentials on the same device. After that, we briefly describe our current implementation. We conclude with a security analysis and a short review of related work.

2 Assumptions and Requirements

2.1 Assumptions

We assume the availability of a general-purpose *secure environment* with the following features:

- *Isolated secure execution environment:* It must be possible to execute trusted code in a strongly isolated fashion from untrusted code executing on the same device. Preferably the secure execution environment
  - is supported by the secure hardware itself so that it is isolated even from the general-purpose operating system on the device, and
  - can use on-chip runtime memory, because in contemporary computing platforms, and especially mobile ones, externally located memory and unprotected memory busses are a commonly used attack vector for breaking the isolation of programs and their data.
- *Secure storage:* It must be possible for trusted code to securely store persistent data so that their confidentiality and integrity can be assured. It is not necessary to store all sensitive data within the secure environment itself. Typically, if a unique, device-specific secret is available only in the

secure execution environment, it can be used to protect data which can be stored in untrusted external storage. Persistent data must also be protected against roll-back attacks. This can be achieved, for example, by using device-specific trusted counters or a secure clock reference.

- *Integrity of secure environment:* Secure storage naturally implies that there must be a way to ensure the integrity of the secure environment so that persistently stored data is accessible only by the secure environment. Additionally, a remote party may want to either send some confidential data to trusted software executing in the secure environment or may want a proof that a certain computation was actually carried out within the secure environment. Both of these require the means to ensure the integrity of the secure environment. This can be achieved using secure boot (only authorized software is allowed to be loaded during the secure environment boot process) or authenticated boot (any software can be loaded during the boot process, but a secure record of the loaded software is retained and can be used for access control or reporting).

2.2 Example Secure Environments

M-Shield: Texas Instruments' M-Shield is an example of a general-purpose secure environment that meets these assumptions. M-Shield is a security architecture available for the OMAP platform used in mobile devices. It has a secure environment consisting of a small amount of on-chip ROM and RAM, as well as one-time programmable memory where unique device key(s) can be maintained. All of these are only accessible in a secure execution environment implemented as a special "secure processor mode". The secure mode is isolated from ordinary software, including the device operating system. Trusted applications, called "Protected Applications" (PAs), can be run in the secure environment. M-Shield supports secure boot so that only authorized software can be run on the device, in particular in the secure environment. For more detailed information on M-Shield, see [17]. M-Shield-like secure environments can be built on top of the ARM TrustZone architecture as well.
TPM: TPMs [19] are usually separate hardware modules with their own processor. They enable secure storage (in the form of sealed storage that can be bound to a specific configuration) and authenticated boot. A TPMs only allows a set of pre-defined cryptographic algorithms to be executed within the TPM itself; it does not provide an execution environment for arbitrary code within the TPM. Thus a TPM-based secure environment has to rely on the operating system to provide secure execution. This provides a lower level of isolation than in the case of environments like M-Shield because the entire operating system kernel becomes part of the trusted computing base. Dynamic root of trust for measurement (DRTM) technology for TPMs as implemented by Intel and AMD processors can be used along the lines described in [13] to minimize the part of the operating system that needs to be trusted. Nevertheless the secure execution environment has to use the main memory as its run-time memory and is vulnerable to attacks on main memory [9], unlike in the case of M-Shield-like secure environments where secure execution can use on-chip memory.
Hypervisor: A hypervisor can also be used to provide an isolated secure execution environment along the lines described in [7]: the normal device operating system and other untrusted software will run as one guest of the hypervisor while the secure execution environment can run as a separate guest. The hypervisor can be combined with TPM-enabled authenticated boot. Again, the level of isolation of secure exection is lower than in M-Shield-like secure environments because of the use of main memory.

As we noted already, Nokia phone models using hardware security features of the M-Shield platform already exist. Hence, this has been the primary target environment for our implementation of the ObC architecture although it can, and has been, implemented on top of other secure environments such as an off-the-shelf, TPM-enabled Linux PC [16] and a virtualized environment on a Nokia N800 Internet Tablet using a commercial secure hypervisor [5].

2.3 Terminology

Before we go on to describe the requirements, let us fix some terminology. As we mentioned in Section 1, our objective is to design an inexpensive, open, and secure platform for credentials by leveraging on-board secure environments. A credential consists of *credential secrets* such as keys, and an algorithm that operates on these secrets known as a *credential program*. In the context of ObC architecture, we sometimes refer to credential programs as *ObC programs* and credential secrets as *ObC secrets*. We refer to a realization of the ObC architecture as an *ObC system*. We will explain other terminology as they are introduced.

2.4 Requirements

Our initial goal is to minimize the cost of implementing and deploying an ObC system. To achieve this, we *re-use existing secure environments* like M-Shield hardware security features rather than design a new one. The design should therefore take the constraints of the existing secure environments into account. For example, in secure environments with on-chip memory, the amount of memory available for an ObC system is very small: as little as ten(s) of kilobytes of RAM, and ROM sizes limited to hundreds of kilobytes at most. Thus, our first requirement is that an ObC system should have a minimal code and memory footprint. Although not every secure environment would have such stringent resource limitations, we still chose to consider the minimal footprint requirement rather than design different types of ObC architectures for different secure environments.

The second goal is to keep the system open: it should be possible for anyone to develop and deploy new ObC programs or provision secrets to existing ObC programs without having to obtain the permission of the device manufacturer or any other third party. Yet, such *openness must not compromise the third goal of a secure ObC system.* Recall that credential programs will execute in the secure environment. An ObC system must therefore be designed so that a malicious or errant credential program cannot harm or abuse the resources in the secure environment. This leads to two requirements: the design must ensure the protection of

- sensitive data of the secure environment, such as device-specific keys, should be isolated from credential programs, and
- resources, such as memory and CPU time, consumed by credential programs must be controlled.

Similarly, an entity relying on one credential program does not necessarily trust other credential programs. Thus, a further requirement is that credential programs must be isolated from one another both during run-time and in their access to persistent data.

By default, this last requirement implies that a credential program will not be able to access persistent data of another credential program. However, there are situations where such sharing of persistent data is essential. For example, when a new version of a credential program is installed, it should be able to have access to the same data as its predecessors (programs with lower version numbers). Also, the need to minimize the footprint of an ObC system imposes constraints on the size of credential programs or their data, implying that the intended credential functionality may need to be split between two or more programs. Because of such cases, the ObC architecture must provide a way to define a group of programs that can share access to confidential persistent data.

Finally, we have two requirements on provisioning. An issuer of credentials needs a way to provision credential secrets to a specific group of credential programs on a device. If the credential program itself is confidential, then a similar mechanism is needed to provision confidential program to a device.

To summarize, we have identified that the code- and memory footprint is crucial requirement for an ObC system in order to meet the objective of reusing existing secure environments. In addition, we have identified the following security requirements for an ObC system:

- **isolation of credential programs**:
  - isolation of secure enviroment resources from credential programs
  - control of resource consumption by credential programs
  - isolation of credential programs from one another, both at runtime as well as in access to persistently stored data
- **authorized sharing of credential secrets** by a group of programs
- **security of provisioning**:
  - provisioning credential secrets to a group of credential programs on a device
  - provisioning confidential credential programs to a device

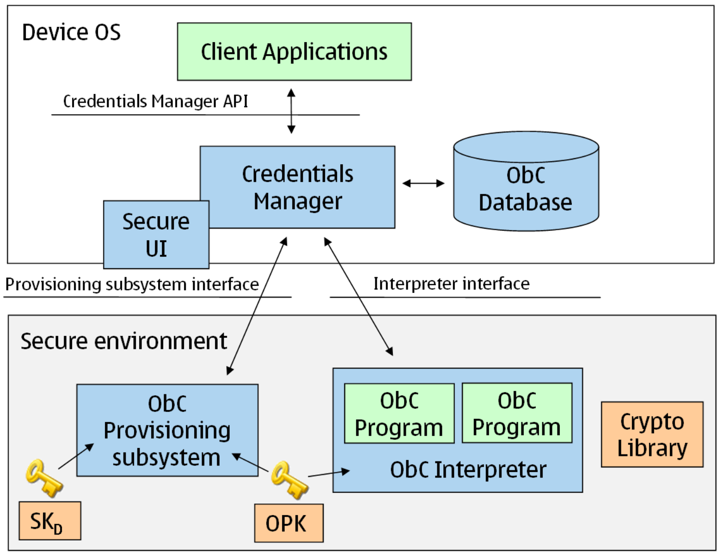

Figure 1: ObC architecture

3 ObC Architecture

Figure 1 shows a high-level overview of our proposed ObC architecture. We describe the components by stating our main architectural design decisions and explaining the rationales behind them.

ObC interpreter: Isolating credential programs from the secure environment resources has to be achieved by providing a virtualized environment where the programs can be run.

Our minimal footprint requirement aiming at very limited RAM usage, rules out the use of a general-purpose virtualized execution enviroment because they cannot be made to fit into the space available. For example, in our primary target environment based on M-Shield, only tens of kilobytes of runtime memory is available within the secure environment. However, a simple bytecode interpreter can be designed to run even within these constraints. Thus we chose to use a bytecode interpreter as the primary component of the ObC architecture. The footprint restrictions rule out popular bytecode interpreters like JavaCard [10]. Still, we wanted to base the credential program development process for our ObC system on an existing programming language so that third-party developers could use familiar development tools. Therefore, we decided to use a slimmed down version of the Lua (v2.4) language [12] for which we wrote a clean-slate interpreter. In addition to the language constructs, our interpreter also provides an interface for commonly used cryptographic primitives.

The ObC interpreter runs in the secure environment. Credential programs are scripts that can execute on the interpreter.

ObC Platform Key: Only one credential program is allowed to execute on the ObC interpreter at any given time. Therefore, the primary issue in isolating credential programs from one another is with respect to their ability to access persistently stored data.

The ObC interpreter has exclusive access to a device specific master key called the *ObC platform key* (OPK). OPK is one of the two secrets protected by the secure storage in the secure environment (the other secret is private part of the device key pair SK_D which is used in provisioning and explained in Section 4). How the OPK is initialized depends on the specific secure environment being used. For example, it can be derived from a one-time programmable (E-Fuse) persistent on-chip key [17] as in the case of M-Shield or a physically uncloneable function as in the AEGIS secure processor [18].

The ObC interpreter provides a sealing/unsealing function for ObC programs. The programs can use it to protect secret data to be stored persistently outside the secure environment. The key used for sealing/unsealing is derived by applying a key-derivation function to OPK and a digest of the code of the credential program which invokes sealing/unsealing, thereby inherently isolating persistently stored data of one credential program from another. In Section 4, we describe how this basic sealing/unsealing functionality is extended to support data sharing among a group of co-operating programs and for

provisioning secret data from external provisioning entities.
Credentials Manager: Client applications use ObCs via the Credentials Manager. The Credentials Manager has a simple "secure user interface" which the user can recognize by customizing its appearance. It also manages the ObC database where sealed credential secrets and credential programs can be stored persistently. We assume that only the Credentials Manager is allowed to communicate with the ObC interpreter. The actual means of enforcing this depends on the particular operating system in which the Credentials Manager is running. For example, in Symbian Series 60 devices, we make use of Symbian OS platform security. Applications can claim a "vendor ID" if they have been verifiably signed. Only those applications with the manufacturer's vendor ID are allowed to communicate with protected applications in the secure environment. We present a more detailed description of the design and implementation of the ObC interpreter and the Credentials Manager elsewhere [5]. In the rest of this paper, we focus on the provisioning of ObCs.

We chose to separate provisioning functionality from the interpreter for two reasons. First, this separation increases reusability. The provisioning scheme could be used with different kind of interpreter and vice versa. Secondly, this approach better suits the limited memory available in the secure environment. Because provisioning and interpreter are separate components they need not be running concurrently within the secure environment. This reduces the footprint of the interpreter and thereby allowing more space for ObC programs.

4 Provisioning

We designed the ObC provisioning system with our "openness" goal in mind: namely we want to allow *any* entity to provision secret data to a group of credential programs on a device. A necessary sub-goal is a mechanism to allow authorized sharing of credential secrets (provisioned or locally created) by a family of programs.

We assume the availability of a unique device-specific keypair. The private part of this key (SK_D) is available only inside the secure environment. The public part (PK_D) should be certified by a trusted authority as a keypair belonging to a compliant ObC system. Typically the device manufacturer will carry out this certification during the device manufacturing process.[1]

Recall that the requirements for provisioning calls for a system that allows any provisioner the ability to

- provision credential secrets to a group of credential programs on a device, and
- provision confidential credential programs to a device.

A trivial solution would be to just encrypt the data to be provisioned using the device public key PK_D. However, this obvious approach has two drawbacks. First, the provisioner could not control which programs would have access to certain provisioned secrets. We need to provide a means by which the provisioner can specify the programs that can access the provisioned secrets. Second, this approach would imply that every piece of provisioned data must be packaged separately for every individual device. This is unoptimal, and unacceptable, in cases where the data being provisioned is actually shared by a group of devices. For example a content broadcast service needs to provision the same content decryption key to a large number of devices. A similar scenario is when encrypted programs are mass-produced (e.g., by creating an identical software image to be loaded on a large number of devices). In general, the network structure (broadcast), application structure, or the business model may necessitate the sharing of keys by multiple devices. Therefore, we adapted a hybrid approach as follows.

4.1 Provisioning to families

We define a *family* as the group of programs and the secret data they share. This secret data can either be credential secrets generated externally and provisioned to the family, or it can be data locally generated by the programs during execution on the ObC system. A provisioner can create a new family by creating a family *root key* (RK). RK is a symmetric key and can be provisioned to devices by encrypting RK with

[1] Considering the variety of device manufacturers, the number of different types of ObC systems, and the late binding between users and devices, certification may also be done by trust intermediaries.

a device public key PK_D. The resulting message is the ObC provisioning initialization message denoted as **ObCP/Init**. To begin provisioning, the provisioner (typically a provisioning server) acquires PK_D of the target device in a trustworthy manner. For example, the provisioner can obtain PK_D by receiving it from the target device itself along with a device certificate issued by the device manufacturer or by retrieving PK_D via an authenticated channel from a database maintained by the device manufacturer.

From RK we derive two other symmetric keys. One is called *endorsement integrity key* (IK). The other is called the *confidentiality key* (CK). CK is used to protect secret data so that it can be securely transferred to target devices. The resulting secure data transfer message denoted as **ObCP/Xfer**. Once a family is provisioned with an ObCP/Init message, any number of pieces of data can be added to the family, possibly over time, by sending only ObCP/Xfer messages.

When the provisioner wants to authorize a particular ObC program to have access to the "family secrets", he has to issue an *endorsement* of the program. He can do this by constructing a message authentication code over the program identifier using IK. The resulting endorsement message is denoted **ObCP/Endorse**. Programs are identified (statistically) uniquely by referring to a cryptographic hash of the program text.[2] Using a cryptographic hash of the program text as the program identifier also implies that there is no need for an authority to manage and maintain the namespace for program identifiers. This is in keeping with the openness requirement that anyone should be able to develop and deploy new credential programs without needing permission or approval from any third party.

Our family concept is outlined in Figure 2. The immediate advantage of using families is shown in the picture – only the family key needs to be protected by the device-specific public key. The rest of the provisioning messages can be protected by keys that are cryptographically bound to RK. This makes it possible to separate the provisioning function into several components – a service that identifies devices to be ObC-compliant, and if so, provisions a root key for the family by sending a unique ObCP/Init message to each unique recipient device. Thereafter ObC program endorsements and encrypted ObC secrets can be retreived from a publicly available service. The credential programs could even be distributed in a peer-to-peer fashion.

The family root key RK defines the scope of data sharing – all secrets provisioned under a common RK can be used by all credential programs *endorsed* by the RK in question, provided that the version indicators of the endorsements are consistent with program and secret versions (family versioning is explained in Section 4.4). Within a family version updates of code can be implemented, and all data belonging to a family can be read by all programs of that family. It is completely at the control of the provisioner to define the extent of these families with respect to ObC programs, ObC secrets and devices, i.e. the concept can be used to meet a variety of different security and provisioning needs. As an example, a secret may be provisioned for a single device, for a group of devices that later can exchange this data or even for all devices in the system.

Most ObC programs are not likely to be confidential. Their integrity is indirectly assured, since all their persistent critical data should be sealed and a modified ObC program would not able to unseal data sealed to the original one unless the modified version has a corresponding ObCP/Endorse. However, the ObC provisioning system can easily protect the confidentiality of the ObC program when needed because confidential programs can be provisioned in an encrypted format in an ObCP/Xfer message like credential secrets. ObCP/Xfer messages containing credential programs are identified by means of a special tag in the payload. Unlike encrypted credential secrets, encrypted credential programs are intended for the ObC system as a whole and not to any specific credential programs. Therefore, there is no need for any corresponding ObCP/Endorse message.

When a ObCP/Init is used to provision confidential programs we use the notation RK_P (and similarly use RK_S when it is used to provision credential secrets). Using different root keys for programs and secrets is often motivated by the business model. E.g. for access control and authentication mechanisms, the keys needed to decrypt the confidential programs are most likely managed by the supplier of the access control system, whereas keys needed for authentication are provisioned by the owner of the service that is being access controled.

If the same RK is used for multiple devices, a side effect of using a symmetric key (IK) to endorse programs is that if an attacker compromises one device and learns RK, he can create and endorse

[2] In our current implementation, IK is also used to protect the integrity of payload of ObCP/Xfer messages. Similarly, CK is used to encrypt the ObCP/Endorse message as well so that program identifiers are not exposed in the clear. This is important in the case where programs themselves are not public. We are currently revising the provisioning protocol so that only one cryptographic primitive, authenticated encryption, is used to protect both ObCP/Xfer and ObCP/Endorse.

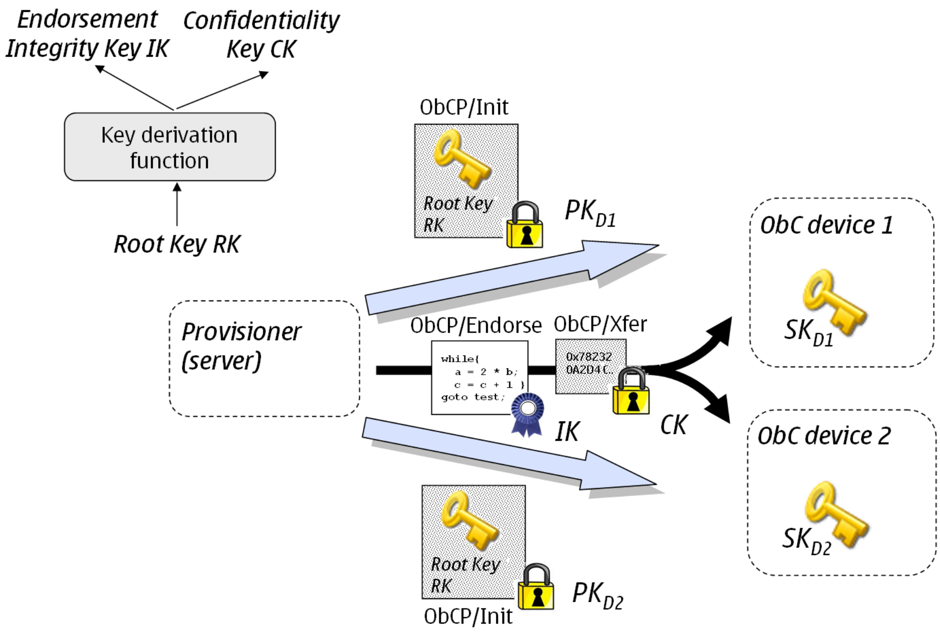

Figure 2: ObC Provisioning - key hierarchy and provisioning messages

new programs that can potentially compromise other devices. Endorsement using digital signatures as described in Section 6 avoids this risk.

4.2 Provisioning message formats

Now we describe the formats of the provisioning messages in more detail.

ObCP/Init message is intended to securely transport the family root key RK to the secure environment. It is of the form:

$$Init = PK_D(RK|PID)$$

where $PK_D()$ refers to a secure public key encryption scheme using PK_D as the encryption key, and PID is "provisioning identifier" that can be used to set up disjoint families from the same root key, to distinguish different generations of the ObC system.

ObCP/Xfer message contains a *tag* (which identifies the payload as either a confidential ObC program or ObC secret), the length of the payload, the payload itself, a version number defining the generation of the family that this secret payload belongs to, and a filler to make the data length multiple of 16 bytes for encryption. This data is encrypted using CK and integrity-protected using IK. In our current implementation, we use AES-CBC as the encryption algorithm $ENC()$ and HMAC-SHA1 as the integrity protection algorithm $MAC()$

$$data = tag|payload\_len|payload|version|filler$$

$$Xfer = ENC_{CK}(data)|MAC_{IK}(ENC_{CK}(data))$$

ObCP/Endorse message contains the program identifier, obtained by taking a cryptographic hash of the program code, encrypted and integrity protected using CK and IK respectively. The version number defines the generation of family secrets that this program is authorized to access. Our current implementation uses SHA1 as the cryptographic hash function.

$$data = H(program)|version|filler$$

$$Endorse = ENC_{CK}(data)|MAC_{IK}(ENC_{CK}(data))$$

At the time of writing, we are switching to using AES-EAX as the single cryptographic primitive for protecting both ObCP/Endorse and ObCP/Xfer. Such a uniformization helps in our quest for making the footprint of the provisioning subsystem smaller.

In remote-provisioning scenarios we assume that these message elements are provisioned to devices using some standardized (key) provisioning protocol like OMA-DM [15], CT-KIP [14] or equivalent, which define the transport mechanisms as well as specify how user authentication is to be done during provisioning. The message elements are still self-contained in terms of security, so in principle ObC provisioning is agnostic to the means of transmission.

4.3 Local data sealing formats

As shown in Figure 1, we chose to separate the provisioning subsystem and the interpreter into separate entities. At least on some platforms the provisioning subsystem has to be interleaved with the ObC interpreter in time, due to space constraints in the secure environments. For this purpose, the provisioning subsystem will need transform the provisioning messages into local device-specific secure packages for credential secrets as well as programs to be used later by the ObC interpreter. We now describe these data packages and their formats.

The sensitive data (both secrets and confidential programs), is stored outside the secure environment encrypted. The data is usable only in the local device as the keys used are device specific. We call the operation of converting data to the local storage format *sealing* and the complement operation *unsealing*.

The seal/unseal operations all use a single cryptographic primitive in the form of AES-EAX authenticated encryption using a template consisting of a fixed-length header and randomizer. The notation $AE_k()$ stands for AES-EAX authenticated encryption using this fixed template and a key k. A key derivation function $KDF_k()$ is used to derive new keys from a key k using a diversifier as input. $KDF()$ is also based on AES-EAX: by using k as the key and the diversifier as one input to the AES-EAX computation while fixing the rest of the inputs to AES-EAX to be constant bit strings, and taking the message integrity code output from AES-EAX as the output of $KDF()$.

The provisioning subsystem seals family secrets using a family-specific sealing key called *local family key* (LFK). LFK is derived from the root key for secrets RK_S, the provisioning identifier (PID) and the family version number using OPK.

$$LFK = KDF_{OPK}(RK_S|PID|version)$$

The provisioning subsystem encrypts LFK for each program endorsed for membership in the family by using a program-specific key called *local endorsement key* (LEK) which is derived from the program identifier:

$$LEK = KDF_{OPK}(H(program))$$

The LFK is encrypted using LEK resulting in an *endorsement token* (ET). A program can only access family secrets with proper endorsement token.

$$ET = AE_{LEK}(LFK)$$

When a program is executed and it needs to handle family specific secrets, it needs as input an endorsement token. The execution environment first derives LEK, uses it to decrypt ET in order to get LFK, and then retains it within the secure environment for subsequent sealing and unsealing family secrets. If a program does not get ET as input, then it uses LEK for sealing and unsealing (which means that the sealing/unsealing is program specific and not family specific).

Currently, confidential programs can only be provisioned through the family mechanism, although the in-device encryption of programs is not family specific – the relatively straightforward scheme is used to provide reasonable security with a modest code and data footprint. The format is the same as used when sealing secrets, the only difference being that programs tend to be much bigger than secrets.

The *local program key* (LPK) used to seal confidential programs is device specific only – it is the same for all programs and families in one device.

$$LPK = KDF_{OPK}(\text{“}SecretCod''})$$

Although it is possible to execute any confidential program with any family specific data, it is still not possible to use sealed data belonging to a certain family without the program being properly endorsed. In short, when a confidential program handles sealed data belonging to a family the course of events is as follows: the ObC interpreter is invoked with the sealed program, an endorsement token, and sealed data as input

1. A sealed program is detected – the device specific LPK is derived, and used to decrypt the program.
2. Hash of the decrypted program is calculated and the program specific LEK is derived from it.
3. The family specific LFK is acquired by decrypting endorsement token ET using LEK. LEK (or LPK) are no more needed.
4. LFK can now be used for sealing/unsealing data.

The local data sealing formats were designed with the assumption that credential programs are executed as a whole. In related work [4] we are investigating the possibility of piece-wise execution of credential programs. The scheduling architecture for piece-wise execution imposes certain changes to local data sealing formats, as described in [4].

4.4 Provisioning version control and naming

The version parameters in ObCP/Xfer and ObCP/Endorse are interpreted in a straight-forward manner. ObC secrets are assigned a *minimal* version identifier indicating the earliest family version that the secret is allowed to belong to. For ObCP/endorse messages, the version number indicates the *maximum* version, i.e. the latest family version that a given program can gain access to. In practice this means that a provisioning operation will be successful if version(ObCP/Xfer) $\leq$ version(ObCP/Endorse). Versioning may be extended for credential lifecycle management so that, for example, once an ObCP/Xfer message is accepted, the provisioning subsystem will refuse to accept older ObCP/Xfer messages for the same secret. However, this would require that secrets are named, and that the name and version information of the secrets are stored along with the secrets in local persistent storage. Neither of these is true for our current implementation.

As can be seen from the provisioning messages, there is no support for naming secrets or even linking messages that are to be used together. This is an intentional design choice in the interest of simplicity – credential naming and management metadata can be part of the provisioning protocol, or be known to participating parties in some other way.

4.5 Provisioning example

A simple example may help to clarify the provisioning concepts. Let us assume that a provisioner has a functionality that consists of two ObC programs prog1 and prog2 that need to share secret data (e.g, the programs might be pipelined to achieve a desired end result). Let us also assume that the provisioner wants to keep the algorithm in prog1 secret and therefore wishes to transmit prog1 in encrypted form. The programs operate on two pieces of secret data, data1 and data2.

The provisioner gets hold of a certificate containing the device public key PK_D. The provisioner will produce two ObCP/Init packages: one of these packets contains the root key for provisioning secrets (RK_S) and the other root key for provisioning confidential programs (RK_P). Both are 128-bit randomly generated AES keys that should be kept secret. Figure 3 shows the different ObC provisioning messages needed for this scenario and the relationships between them. For simplicity, we do now show confidentiality keys (CK) or endorsement integrity keys (IK) explicitly but always refer only to the corresponding root key.

The confidential program prog1 is transferred by constructing an ObCP/Xfer message based on the root key RK_P to encapsulate the encrypted bytecode for prog1. Prog2 is not confidential and can be transferred directly. Each program requires an ObCP/Endorse based on the root key RK_S. This will ensure that the secrets for the family defined by RK_S will be accessible by that program. For all secrets to be securely given to any of the programs now bound to the family, an ObCP/Xfer is constructed, using the key hierarchy originating from RK_S.

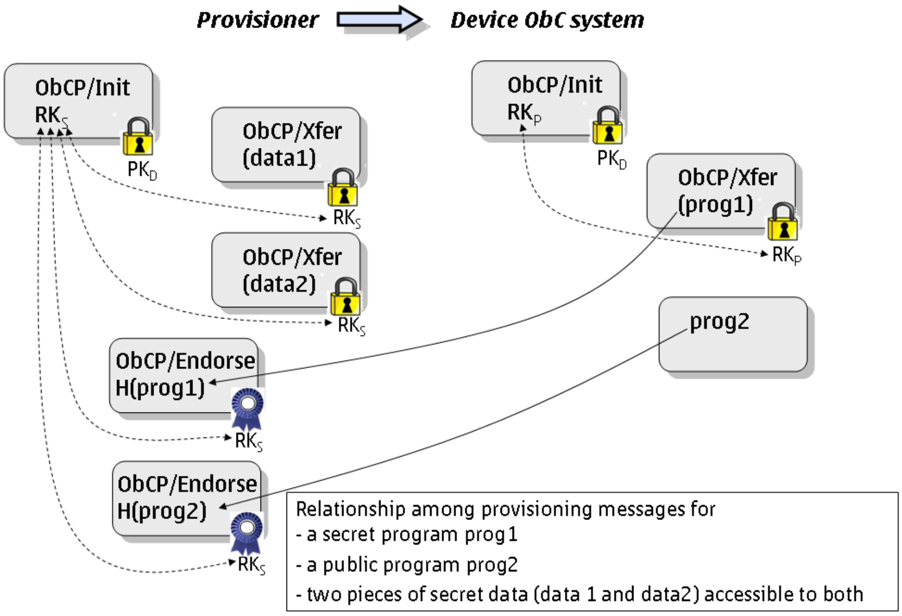

Figure 3: Provisioning messages - an example

There are three noteworthy issues. First, in this example there are two families – one for confidential programs (rooted on RK_P) and the other for credential secrets (rooted on RK_S). Both programs are endorsed to be members of the latter family so that they have shared access to the secret data automatically.

Second, the ObC system does not distinguish among programs that are endorsed to be members of a given family: any endorsed program in the family will be allowed to access credential secrets provisioned to the same family, provided the version constraints are satisfied. It is the responsibility of the client application (which uses ObCs via the Credentials Manager interface) to use the correct tuple of provisioning messages (ObCP/Init, ObCP/Xfer, ObCP/Endorse) where the parameters match would result in the credential secret being sealed so that it is accessible to the endorsed program. If this is an issue affecting the correctness of operation, a provisioner could optionally indicate which credential program is supposed to access/use which transferred credential secret as part of the transferred secret data itself. Any authorized program that successfully unseals the data can then check if the data was intended for its use.

Third, only indirectly related to provisioning, the sealing function will be compatible between programs in the same family that have the same version number, i.e. in the scenario outlined above, the programs should always be distributed as sets with the equivalent version numbers if sealed data from one program is to be read by another.

5 Implementation

As our main implementation target we selected Nokia N95 mobile phone which runs Symbian OS v9.2 operating system on 300 MHz OMAP 2420 platform and it is, like many other Nokia mid-range and high-end phones, based on hardware security features of M-shield.

We have implemented a complete ObC system based on the architecture described in Section 3. Both the interpreter and the provisioning subsystem were implemeted as M-Shield "Protected Applications" (PA) and written in C (in compiled format 5 kB in size each). The operating system level component Credentials Manager was implemented using C++ for Symbian OS. Credentials Manager uses typical Symbian client-server model and it has an SQL database for credential storage. All of these can be distributed to and installed on off-the-shelf phones in the form of standard Symbian OS software packages.

In this section, we describe our provisioning subsystem implementation, one example application built on top of the implemented ObC system, and our developer tools. For more details about other components and implementations for other target platforms see [5].

5.1 Provisioning subsystem implementation

The provisioning subsystem interface provides the following services: converting provisioned secrets and confidential programs into locally sealed data, endorsing programs to families, and transferring confidential data between programs.

Confidential program: Confidential ObC programs need to be provisioned encrypted. The provisioning subsystem converts the provisioned program into a locally sealed data structure. The needed inputs are: ObCP/Init (containing RK_P) and ObCP/Xfer (containing the encrypted program). The provisioning subsystem returns the program encrypted using local program key (LPK).[3]

Credential secret: Securely provisioned secrets must be processed by the provisioning subsystem as well. The needed input data for provisioning a secret are: ObCP/Init (containing RK_S) and ObCP/Xfer (containing the encrypted secret). The provisioning subsystem returns a secret encrypted with LFK.

Endorsing a program: The needed input data for endorsing an ObC program to access family secrets are: ObCP/Init (containing RK_S) and ObCP/Endorse (containing the encrypted hash of the program). The provisioning subsystem produces the endorsement token ET (LFK encrypted using LEK). Each ObC program needs its own ET in order to be able to access encrypted family secret, i.e. every program in a family needs to be separately endorsed.

Transferring confidential data between programs: Data sealed by an ObC program may need to be transferred to a (set of) ObC program(s) belonging to the same family which constitute(s) a newer version. The required inputs for this operation are: ObCP/Init (containing RK_S), ObCP/Endorse containing hash of the previous version ObC program, ObCP/Endorse containing hash of the new version ObC program, and sealed credential secret belonging to the previous version (encrypted using the previous version LFK). The provisioning subsystem produces sealed secret belonging to the new version (encrypted using the new LFK, since the version changed).

The subsystem checks that the new version number is the same or higher than the old one. This prevents data transfers to older, possibly vulnerable program versions. Regarding cases where there are several ObC programs belonging to each version, it is sufficient that each sealed data element is transferred to the next version, and that each ObC program authorized to handle the next version sealed data gets its own ObC program-specific ET.

5.2 One-time token ObC

As an example of an ObC we briefly describe a widely used one-time password (or "token code") scheme that has been implemented using our ObC system. The credential program, consisting of the actual token generation algorithm was implemented as a Lua script by a research partner. The token application was written in C++ for Symbian OS and it has two components.

The first component is a provisioning client. When the token application is started, the provisioning client checks if the phone already has an installed token. If not, the phone connects to a provisioning server (implemented by the research partner) and sends the certified device public key PK_D. The server replies with two sets of provisioning messages [4]. The first consists of ObCP/Init and ObCP/Xfer messages containing the encrypted credential program for token generation. The second consists of ObCP/Init, ObCP/Xfer and ObCP/Endorse that contain: 1) an encrypted token secret, and 2) an endorsement that grants the token generation program access to the secret. The token generation algorithm is proprietary. Therefore the token generation program is a confidential credential program and is provisioned to the device in encrypted form.

The second component of the application is a simple token UI that periodically calculates a new token code (short numeric string) using the provisioned program, provisioned secret and a PIN code which is requested from the user. The resulting token code is simply displayed to the user.

[3]Note that the developer of the ObC program may decide to use the same RK_P for many devices. In this case, the actual encrypted ObC program in ObCP/Xfer may have been sent to the device ahead of time, e.g., as part of the system image, or a separately available installation package common to all devices.

[4]Each set could also come from a different provisioning server.

5.3 Developer tools

We have created tools to help third-party development of ObCs. First, we provide a Windows emulator of the secure environment. Essentially, the tool is a debugger, where credential program bytecode can be executed in a step-by-step fashion. Secondly, we have an ObC implementation that enables testing of credential provisioning and execution in both Symbian phones where M-Shield secure environment is not available and Symbian emulator on PCs. Both of these tools are available from the authors on request.

6 Analysis

In this section, we revisit the objectives for the ObC architecture identified in Sections 1 and 2, and informally reason how well the ObC system meets those objectives.

The first objective was that the system should be inexpensive to deploy. We achieve this by leveraging existing already available hardware security environments. Our prototype implementation can be distributed as a standard add-on software package and can be installed and used on already deployed devices.

The second objective was openness in provisioning. In traditional code-signing the target device is pre-configured with a finite number of trust domains. Our concept of families allows trust domains to be created dynamically. Hence, it meets the goal of openness in that any provisioner, be it hobbyists, small organizations, user groups, or large corporations, can define and implement secure services based on the ObC architecture independently without having to obtain permission or enter into contractual obligations with the device manufacturer, network operator or any other third party.

The third objective was security, which we elaborated further by identifying three classes of security requirements in Section 2.4. We now consider those requirements.

6.1 Isolation of credential programs

In our current design, only one credential program can execute in the secure environment at any given instance. A program in execution runs until it finishes execution or is terminated by the interpreter. No interleaved execution of credential programs is possible. Thus the primary concern in isolation is with respect to persistently stored credential secrets. Secrets are sealed before being stored in the Credentials Manager database. The sealing key is derived by applying a message authentication function to the program code with OPK as the key. A credential program cannot access sealed data of another credential program if the following hold true:

- OPK remains secret,
- the key derivation algorithm KDF used to derive the program-specific sealing keys (LEK) is one-way,
- the hash function $H()$ used to calculate statistically unique program identifiers is collision-free,
- the implementation of the interpreter is correct, and
- the authenticated encryption algorithm AE used to construct seals does not leak information about the plaintext.

We are currently extending the ObC interpreter to allow for on-demand paging and in-line subprogram calls for credential programs [4]. This is done in order to remove the constraint on the size of credential programs. However, piecewise execution mediated by the operating system will leak some information regarding program state. We intend to investigate ways of helping developers identify potential leakage as well as techniques to minimize the leakage.

6.2 Authorized sharing of credential secrets

The family concept allows authorized sharing of credential secrets. In order to access family secret, a program must be able to access the local family key LFK. A program can access LFK if there is a valid endorsement token ET for that program. If there is no ET for a program it cannot access family secrets as long as the conditions for the isolation of credential programs hold.

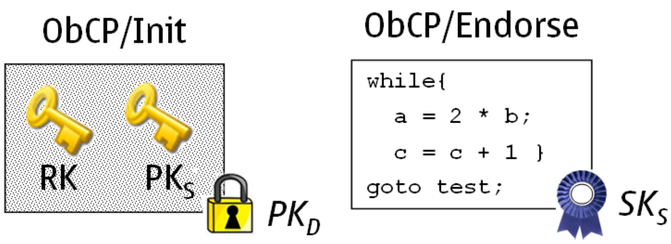

Figure 4: Endorsement using a digital signature

6.3 Security of provisioned data

The security of provisioned data depends on three factors: the data is provisioned to the device of the correct user, to a valid secure environment within that device, and is accessible to the correct set of programs executing within the secure environment.

- *correct user:* the ObC provisioning system does not concern itself with user authentication. However, the use of the device public key PK_D uniquely identifies the target device. Thus, the provisioning protocol used to provisioned the ObC provisioning messages can correctly bind user authentication to the right PK_D.
- *correct secure environment:* ObCP/Init is encrypted for PK_D. The root key protected by ObCP/Init and the keys derived from RK to protect provisioned data will remain within the secure environment if the following hold true:
  - the corresponding SK_D remains within the secure environment,
  - the process of certifying PK_D as a valid public key is correct,
  - the provisioner has securely obtained the necessary keys needed for verifying the certificates on PK_D (such as a manufacturer's signature verification key)
  - the implementation of the provisioning subsystem is correct, and
  - the encryption schemes used in ObCP/Init and ObCP/Xfer are correct.
- *correct set of programs:* ObCP/Endorse enables a program to access data provisioned to a family. If the conditions listed above for provisioning the data to the correct secure environment hold, then the endorsement integrity key IK will remain within the secure environment. In this case, a program that is not intended by the provisioner cannot access family secrets as long as the message authentication code $MAC()$ used for endorsements remains one-way.

Instead of using a shared symmetric key IK to endorse programs, it is also possible to use digital signatures. The basic principle of endorsement is that the endorsement key must be cryptographically bound to the encryption key used to provision credential secrets. Suppose a provisioner has a signing key pair PK_S/SK_S. He can use SK_S to digitally sign the programs to be endorsed. In order to do this, he must include PK_S in the ObCP/Init message so that it is cryptographically bound to RK. Figure 4 shows how ObCP/Init and ObCP/Endorse are modified when digital signatures are used for endorsement.

As explained in Section 4.4, we intentionally chose to avoid naming provisioned secrets in order to keep the ObC system minimal. This choice forces the developer to correctly include and check identification information in the parameter itself whenever a secure parameter is sensitive to e.g. parameter position in the set of input parameters or to the processing program in the set of credential programs for a given family. As this is a typical feature, explicit interpreter support for parameter naming could be added. Also, allowing provisioning messages to be linked via an opaque token may be a useful feature, e.g. to help in securely binding user authentication to provisioning.

7 Related Work

From an architectural viewpoint, ObC is close to the Small Terminal Interoperable Platform (STIP) by the GlobalPlatform consortium [8], in that both aim to provide an open, well specified platform complete with provisioning support to be used for security services in mobile devices. However STIP is built around smartcard technology whereas ObC is intended to be deployed without the need for additional hardware by making use of existing secure environments. A second difference is that STIP applications must be certified by the card issuer while the ObC provisioning system is designed for open provisioning.

McCune et al. [13] describe how the support for dynamic roots of trust in modern processors can be used in conjunction with a TPM to implement a secure execution environment as an isolated software module without having to trust the device operating system. Our architecture can be implemented using this approach: the isolated software module will consist of the ObC interpreter and the provisioning subsystem.

Gajek et al. [7] describe combining a TPM with a virtual machine monitor so that a "wallet" can be implemented as a trusted guest virtual machine. The ObC system is a generalization of the wallet and can be implemented in the same way. In [5], we describe an ObC system where isolation is based on virtualization.

Our approach of using program-specific derived keys to isolate programs from one another is similar to the "secret sharing" approach taken in AEGIS [18]. In AEGIS, the response for a given challenge depends on the physically uncloneable function on the device as well as the particular software configuration (program requesting the response and the security kernel currently executing) on the device. The response is given to the program which can use it as a secret key. In our design, the sealing key of an ObC program depends on OPK and the program itself. However the program only gains the right to use the key for sealing/unsealing. The actual program-specific key is never given out to the program.

Lee et al. describe a hardware-assisted architecture for protecting "critical secrets" in microprocessors [11, 3]. "Critical secrets" in their terminology is similar to our notion of credential secrets. However, Lee et al. focus on designing new microprocessor features whereas our focus is on *re-using* existing general-purpose secure environments. They also do not support the notion of isolating credential programs from one another or facilitating families of co-operating programs – the only software allowed to operate on critical secrets are the "trusted software modules" which are authorized by the device owner or issuer.

The related work closest to ours is the Trusted Execution Module (TEM) [2]. The motivations behind TEM are identical to ours. The TEM architecture has several similarities to ObC architecture as well: for example, TEM also uses a bytecode interpreter executing within a secure environment and each TEM device has a unique device-specific keypair similar to PK_D/SK_D. The primary difference between TEM and ObC is in how the persistently stored data of credential programs are protected. This in turn results in different provisioning systems. TEM uses a persistent global store with a very large address space. A piece of mutable persistent data is assigned a random address at the time of compiling a TEM program (called "closure" in TEM terminology). The address of a variable also serves as the capability to access that variable and hence must be kept confidential. The address of a given variable in a TEM program is the same on every TEM where that program runs. When a TEM program is packaged (in the form of "bound SECPacks" in TEM terminology) for a target device the addresses of persistent data it needs to use are put in a binding table which is then encrypted using the device public key. Authenticated sharing of data among TEM programs can be achieved by including address of the variable that holds the data in the binding table of all the programs that need access to that data. Since the binding table is included at compilation time, the TEM architecture makes the implicit assumption that the same entity provisions the TEM code as well as any secret data used by that code. This has two implications. First, bandwidth and storage usage is not optimal because a device will have to receive and store multiple copies of commonly used algorithms (e.g., HTTP Digest authentication). Second, and more important, the assumption does not always hold. For example, the credential program in the example we described in Section 5.2 is a proprietary algorithm by a leading provider of one-time token systems. However the shared secret to authenticate a particular use to a service is chosen and provisioned by the service provider. Although the system provider could provision a bound SECPack to a device, he cannot choose the (global) address of the shared secret for the binding table because the secret is confidential to each service provider. The service provider cannot provision the algorithm because the algorithm is confidential to the system provider. The ObC provisioning architecture, as described in Section 5.2, naturally lends itself to the case where code and data come from different sources. A secondary difference between the two architectures is that unlike in ObC, there is no separate installation step in TEM. The price for not having a separate installation step is that asymmmetric cryptography (decryption using SK_D) is needed every time a TEM program is executed.

8 Conclusions

Although there has been significant research and development of multi-application cards or "white-cards", they have never been widely adopted to support credentials from multiple sources to co-exist in the same device. A likely reason is the high barrier for entry for new service providers to use cards that have been already deployed by some other issuer. As a result, the current situation is that either hardware security tokens are not used, or the user is compelled to carry separate hardware tokens for each different service provider who requires them.

Our On-board Credentials architecture addresses this issue in a manner that may stimulate larger-scale deployments of credentials. The architecture is designed so that it can be realized on secure environments that are already widely deployed for other purposes. The openness of provisioning will allow small-scale service providers to build their authentication and authorization mechanisms around ObCs for securing their services independently of device manufacturers or other stakeholders. However, the ObC architecture, solves only the first pieces of this puzzle. Several open issues remain. First, techniques for determining and describing the level of security in the secure environment on the target device are needed. Second, the provisioning server needs ways to specify policies on how the provisioned credentials are to be accessed and used locally on the target device. Third, both the security and the usability of ObC system need to be more rigorously validated.

References

[1] ARM. Trustzone-enabled processor. `http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf`.

[2] V. Costan, L. Sarmenta, M. van Dijk, and S. Devadas. The trusted execution module: Commodity general-purpose trusted computing. In *Eighth Smart Card Research and Advanced Application Conference*, August 2008. `http://people.csail.mit.edu/devadas/pubs/cardis08tem.pdf`.

[3] Jeffrey S. Dwoskin and Ruby. B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM Conference on Computer and Communication Security*, pages 389–400, October 2007.

[4] Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, and Aarne Rantala. Scheduling the execution of credentials in constrained secure environments. In *(to appear) in Proc. ACM Workshop on Scalable Trusted Computing*, Oct 2008.

[5] Jan-Erik Ekberg et al. Onboard credentials platform: Design and implementation. Technical Report NRC-TR-2008-001, Nokia Research Center, January 2008. `http://research.nokia.com/files/NRCTR2008001.pdf`.

[6] Jan-Erik Ekberg and Markku Kylänpää. Mobile trusted module. Technical Report NRC-TR-2007-015, Nokia Research Center, November 2007. `http://research.nokia.com/files/NRCTR2007015.pdf`.

[7] Sebastian Gajek, Ahmad-Reza Sadeghi, Christian Stueble, and Marcel Winandy. Compartmented security for browsers – or how to thwart a phisher with trusted computing. In *Proceedings of IEEE International Conference on Availability, Reliability and Security (ARES'07)*, April 2007.

[8] GlobalPlatform. Why the mobile industry is evolving towards security, August 2007. GlobalPlatform white paper. `http://www.globalplatform.org/uploads/STIP_WhitePaper.pdf`.

[9] J. Alex Halderman et al. Lest we remember: Cold boot attacks on encryption keys. In *Proc. Usenix Security Symposium*, 2008. `http://citp.princeton.edu/memory/`.

[10] JavaCard Technology. http://java.sun.com/products/javacard/.

[11] Ruby B. Lee, Peter Kwan, John McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA '05)*, pages 2–13, May 2005.

[12] The Programming Language Lua. http://www.lua.org/.

[13] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal TCB Code Execution (Extended Abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[14] Magnus Nyström. Cryptographic Token Key Initialization Protocol (CT-KIP). IETF RFC 4758, November 2006.

[15] Open Mobile Alliance - Device Management Working Group. http://www.openmobilealliance.org/Technical/DM.aspx.

[16] Aishvarya Sharma. On-board credentials: Hardware-assisted secure storage of credentials. Master's thesis, Helsinki University of Technology, January 2007. http://asokan.org/asokan/research/Aish-Thesis-final.pdf.

[17] Jay Srage and Jérôme Azema. M-Shield mobile security technology, 2005. TI White paper. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[18] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random function. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 25–36, May 2005.

[19] Trusted Platform Module (TPM) Specifications. https://www.trustedcomputinggroup.org/specs/TPM/.

