OnBoard Credentials Platform Design and Implementation

Article · January 2008

CITATIONS
11

READS
463

6 authors, including:

Jan-Erik Ekberg
Nokia Research Center (NRC)
35 PUBLICATIONS **708** CITATIONS
SEE PROFILE

N. Asokan
University of Waterloo
252 PUBLICATIONS **8,536** CITATIONS
SEE PROFILE

Kari Kostiainen
ETH Zurich
49 PUBLICATIONS **990** CITATIONS
SEE PROFILE

Aarne Rantala
VTT Technical Research Centre of Finland
14 PUBLICATIONS **121** CITATIONS
SEE PROFILE

NOKIA

Research Center

NRC-TR-2008-001

OnBoard Credentials Platform Design and Implementation

Jan-Erik Ekberg, N. Asokan, Kari Kostiainen, Pasi Eronen
Nokia Research Center Helsinki, Finland
http://research.nokia.com

Aarne Rantala
Technical Research Center (VTT), Finland

Aishvarya Sharma
Helsinki University of Technology (TKK), Finland

January 29, 2008

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. On the other hand, dedicated hardware tokens provide high levels of security, but the logistics of manufacturing and provisioning smartcards are expensive, which makes it unattractive for most service providers. 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 report, we describe how we use general-purpose secure hardware to develop a platform for credentials which we call OnBoard Credentials (ObCs). ObCs combine the flexibility of virtual credentials with the higher levels of protection due to the use of secure hardware. Besides secure storage and execution the ObC architecture supports secure provisioning of both credential algorithms and secrets. The architecture is widely applicable and in this report we describe prototype implementations for three different platforms: an M-shield enabled mobile phone, a TPM-based Linux PC, and a mobile Linux tablet with para-virtualization.

Index Terms:

platform security
mobile phones
secure hardware
credentials

Contents

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 smartcards are expensive, which makes it unattractive for most service providers to issue their own smartcards. 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 issuers of smartcards usually retain ownership and control. A service provider who wants to use an existing installed base of multi-application smarcards has to obtain permission from the issuer of those cards in order to deploy new credentials to them. Such procedural obstacles, in turn, make it unattractive for service providers to share the hardware tokens issued by others.

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: e.g., Trusted Platform Modules (TPM) [24] and Mobile Trusted Modules [4] specified by the Trusted Computing Group [23] and other platforms like M-Shield [21, 20] and ARM TrustZone [1]. All these platforms enable, to different degrees, a strongly isolated secure environment, consisting of secure storage, and 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 high-end Nokia phones are based on hardware security features of the M-Shield platform.

In this report, we describe how we use such general-purpose secure hardware to develop a platform for credentials which we call "OnBoard Credentials" (ObCs). ObCs combine the flexibility of virtual credentials with the higher levels of protection due to the use of secure hardware. Our objective was to define a platform 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.

2 Requirements and assumptions

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 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 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).

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 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 [21, 20].

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 ObC system although our general architecture can, and has been, implemented on top of other secure environments as well. Reference platforms range from an off-the-shelf, TPM-enabled Linux PC (see section 7.2) to a virtualized environment on a Nokia N800 Internet Tablet using the Trango™ secure hypervisor, where the secure part of the ObC was isolated as a virtualized compartment (see section 7.3) in the spirit of [6]. Additionally, ARM TrustZone™ technology as well as the dynamic root of trust measurement (DRTM) technology for TPMs as implemented by Intel™ and AMD™ processors all seem to constitute valid security foundations for the ObC concept.

2.2 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 the ObC system as an inexpensive, open, and secure platform for credentials

by leveraging on-board secure environments. A credential consists of *secret data* such as keys, and an algorithm that operates on these data known as a *credential program*. In the context of the ObC system, we sometimes refer to credential programs as *ObC programs* and credential secret data as *ObC secrets*. We will explain other terminology as they are introduced, and will summarize them in the Glossary in Appendix A.

2.3 Requirements

Our initial goal is to minimize the cost of implementing and deploying the 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 the 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 the ObC system should have a minimal code and memory footprint. Although not every secure environment has such stringent resource limitations, we still chose to consider the minimal footprint requirement rather than use different types of ObC systems 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. The 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 the 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. In these cases, the ObC system must provide a way to define a *family* 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 secret data to a specific family 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 the following requirements for the ObC system:

- minimal code and memory footprint
- 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 secret data by a group of programs

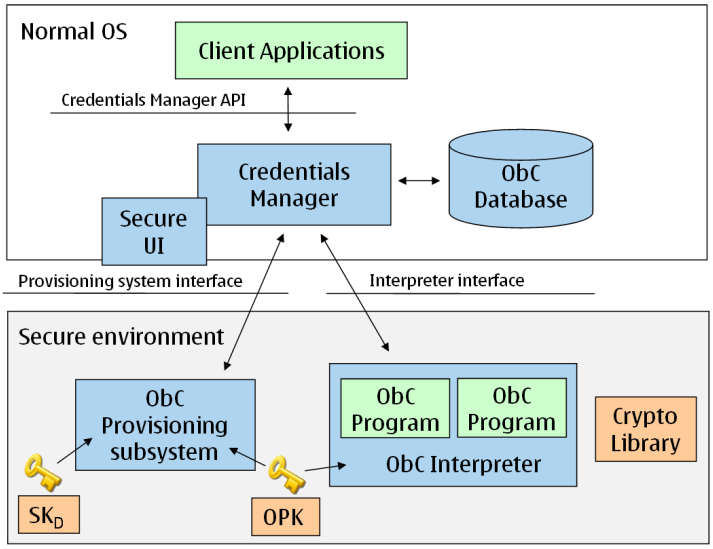

Figure 1: Architecture of the OnBoard Credentials Platform

- provisioning secret data to a group of credential programs on a device
- provisioning confidential credential programs to a device

3 Architecture

Figure 1 shows a high-level overview of the ObC system 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. This environment could be the process scope in a well-designed (and secured) operating system, a virtualized processor environment provided by a secure hypervisor, or hardware support providing either virtualization or a privilege level that might be reserved for the use of the ObC system.

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. However, a simple byte-code interpreter can be designed to run even within these constraints. This is not a new architectural choice - the same design is visible, for example, in JavaCards [9] or in the STIP architecture for mobile devices [7]. Thus we chose to use a bytecode interpreter as the primary component of the ObC platform. The interpreter runs in the secure environment. Credential programs are scripts that can execute on the interpreter.

In our primary target environment based on M-Shield, the memory constraints were such that an interpreter of around 6kB (compiled) could be fit on-chip with enough (1.5kB) additional memory to execute a small script and maintain some local data. We discarded the possibility of direct porting an existing interpreter for JavaCard or interpreted C as well as an ARM command set emulator because of these size constraints. We considered using a Forth interpreter, but decided against it because it is possibly not user-friendly enough for third-party development of credential programs. Still, we wanted to base our application development process on an existing programming language and bytecode 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 [15] 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. In Section 5, we describe the design of the ObC Interpreter in more detail.

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 master key called the ObC platform key (OPK). OPK is the only secret protected by the secure storage in the secure environment. How the OPK is initialized depends on the specific secure environment being used. In Section 7, we describe the specific implementations we have used.

The ObC interpreter provides a sealing/unsealing function for ObC programs. The programs can use it to protect secret data to be stored outside the secure environmnet. 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 program families and for provisioning secret data from external provisioning entities.

Credentials Manager: Client applications use ObCs via Credentials Manager (CM). CM has a simple "secure user interface" which the user can recognize by customizing its appearance. It also manages a credential database where sealed credential secrets and confidential programs can be stored persistently. We assume that only the CM is allowed to communicate with the ObC Interpreter. The actual means of enforcing this depends on the particular operating system in which the CM is running. In Section 7, we describe how this access control is implemented in specific environments. In Section 6, we describe the CM itself in more detail.

Device keypair: 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. We assume that the device manufacturer will carry out this certification in during the device manufacturing process[1].

Provisioning subsystem: The ObC architecture supports secure provisioning. The provisioning subsystem processes encrypted and integrity protected provisioned credential secrets and programs using device key SK_D and converts them into locally sealed data structures (that ObC interpreter can handle) using platform key OPK. The provisioning is described in more detail in Section 4.

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 saves limited memory available in the secure environment. Because provisioning functions 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 our 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 secret data (provisioned or locally created) by a family of programs.

Since we assume the availability of a certified unique device keypair in every ObC system such that the private part SK_D of the keypair is available only within the secure environment, a trivial solution would be to just encrypt the data to be provisioned using the device public key PK_D. However, this obvious approach would have two drawbacks. First, the provisioning entity could not control which programs would have access to certain provisioned secrets. We need to provide a means by which the provisioning entity 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

[1] If the ObC system is implemented and deployed as a stand-alone secure token or a self-contained chip, the chip vendor may be the certifying authority. However, considering the variety of device manufacturers, the number of different types of ObC platforms, and the late binding between users and devices, certification may also be done by trust intermediaries.

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 Key hierarchy and provisioning messages

We define a *family* as the group of programs and the secret data they share. This secret data can either be generated externally and provisioned to the family, or it can be locally generated by the programs during execution on the ObC platform. 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 a device public key PK_D. The resulting message is an initialization message denoted as ObCP/Init.

From RK we derive two other symmetric keys. One is called tje *integrity key* (IK). The other is called the *confidentiality key* (CK). CK is used to protect secret data so that they 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 the hash of the program text. [2]

Our family concept is outlined in Figure 2. The family is defined to include all data (secrets and/or programs) protected under a common root key RK. 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 is in our architecture completely determined by the provisioning entity. The family root key RK defines the provisioning keying for either credential programs or secrets, optionally both at the same time. The root key additionally 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. 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 provisioning entity 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.

This design meets our goal of openness: any provisioning entity, be it hobbyists, small organizations, user groups, or large corporations, can define and implement secure services based on the ObC platform independently without having to obtain permission or enter into contractual obligations with the device manufacturer or network operator or any other third party.

The mechanism to transfer secret data can be used to provision confidential ObC programs. Naturally this transferred data is not endorsed to any particular program. When a ObCP/Init is used to provision

[2] In our current implementation, IK is also used to protect the integrity of payload of ObCP/Xfer messages as well. 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.

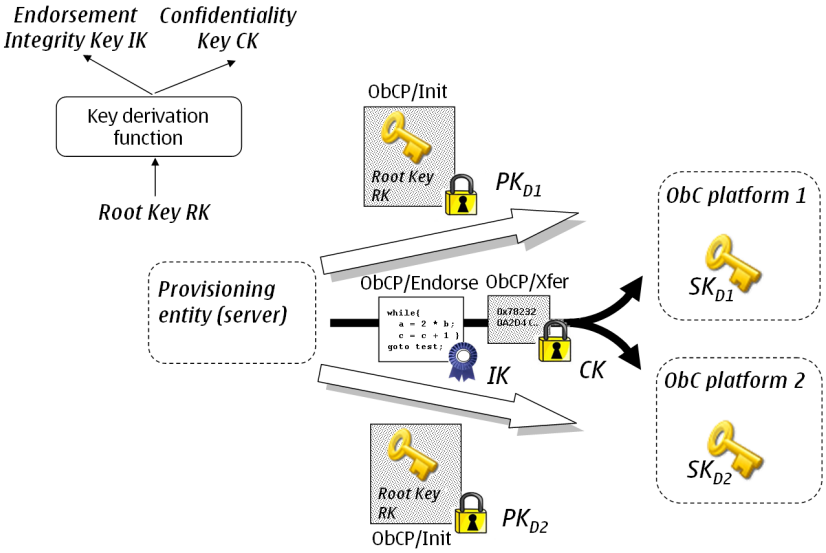

Figure 2: ObC Provisioning: Key hierarchy and provisioning messages

secret programs we use the notation RK_P (and similarly use RK_S when it is used to provision secret data). Using different RKs for programs and secrets is often motivated by the business model. E.g. for access control and authentication mechanisms, the program decryption keys are most likely managed by the supplier of the access control system, whereas secret data is provisioned by the entity owning the resource requiring controlled access.

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 new programs that can potentially compromise other devices. Endorsement using digital signatures as described in Section 10 avoids this risk.

4.2 Integration with provisioning protocols

The exact formats of the provisioning packages are explained in detail in Appendix C. In remote-provisioning scenarios we assume that these message elements are provisioned to devices using some standardized (key) provisioning protocol like OMA-DM, CT-KIP [17] 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 Relation to the interpreter

As shown in figure 1, we are faced with the fact that at least on some platforms the Obc provisioning subsystem has to be interleaved with the ObC interpreter in time, due to space constraints. For this purpose, the provisioning subsystem will on many platforms produce intermediary, device-specific secure packages for data as well as code to be consumed by the interpreter proper. For the M-shield implementation, these packages are described in the appendices, Section C.5. Although often a necessity, architecturally these intermediary packages are irrelevant from a provisioning perspective, and may vary based on underlying technology - i.e. code to be provisioned to a Java Card interpreter might need a signature to be compliant with specification in that specific domain.

5 Interpreter

As stated in the architecture section, our aspiration has been to provide as small a virtual machine as possible for the ObC concept. The evolution and optimization of this will certainly continue, but the interpretation of the Lua 2.4 bytecode has for now fit our needs. We have been able to keep the credential program development toolchain practically unmodified, i.e. a programmer can rely on the original documentation and use the original compiler, as long as a supported list of constraints with respect to the coding is followed.

Currently, we only support the unsigned 16-bit integer datatype. By not supporting subroutines, 2- or more dimensional tables, strings and a few other features we were able to match the Lua virtual machine written in C to our constraints, with room enough to accommodate extensible script interfaces to platform services as well as language "extensions" such as sealing. The extensibility achieved using the MPP [10] macro language pre-processor. Currently, the platform services include the SHA-1 hash function, AES, a random number source as well as parameter I/O and the sealing operations.

The provisioning subsystem makes localized versions of externally provisioned data that can be accessed by the interpreted programs by the unseal operation.

5.1 Interpreter security properties

The most significant security aspect of the interpreter is to provide isolation between the executed code and the underlying platform. On a byte-code level this implies special care regarding all kinds of overflows and cross-referencing (stack and heap), including the constraint that the program counter must remain inside the allocated byte code area. Another property to be constrained is how long the code may occupy the interpreter - the evaluation happens in a secured environment where externally aborting the execution may not be possible.

In addition to isolation, secure data storage, secure data sharing between ObC programs, confidential programs and for some use-cases proof-of-execution needs to be provided. On the interpreter level, we make use of the device-specific OPK, and diversify this secret based on the ObC program code. In this manner every unique program in every unique device will have a statistically unique value that can be used as a key. If we assume that the program has no back-door and that the interpreter works correctly (does not reveal either the device-specific OPK or the diversified key value) it is straight-forward to construct secure storage as an encrypted packet protecting script-generated data, e.g. using AES-CBC encryption with (truncated) hash for integrity protection and random salt. The diversified key value is easily re-generated every time by the interpreter during ObC program byte-code import.

The simplicity of the diversification scheme nicely solves the problem of complete openness - any untrusted script can have its own secure storage without danger of the scripts gaining access to each others' data. However there are many use cases where this is explicitely needed, like for script versioning, where we would want data to be available also to newer script versions, as well as if the functionality is divided between several scripts and we want to transfer data between them. For this purpose, an optional, additional key indirection, stored with the data, has been introduced, and is described in the appendices, section C.5. The key indirection for sealing is managed by the provisioning system as is the support for confidential ObC programs. The provisioning component locally modifies the encryption to be re-encrypted and integrity-protected with a diversified device-specific key, whereafter the final decryption is done on program loading.

Proof-of-execution is not provided by the architecture itself. The combination of the provisioning system and the secure storage makes it possible for any stakeholder to provision a secret to his program, and apply this secret on a challenge to provide a credential attesting to this fact.

The space reserved for bytecode can also be enlarged by securely paging the bytecode – loading only those parts of the bytecode that currently is needed for execution. This form of paging will reveal varying amounts of the application data flow to a cunning eavesdropper. If deployed, the constraints on byte code size can be lifted, at the expense of the required insight by the application developer, who now must take this property into account.

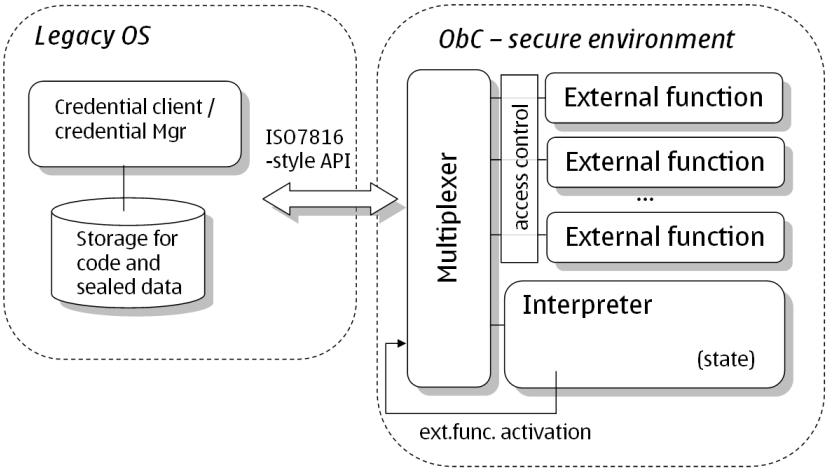

Figure 3: ObC Interpreter design

5.2 Interpreter design and extensibility

Including a macro language in the toolchain was already mentioned as a way to enable future extensions to the language interfaces provided by any one platform. It is also important that the same ease of extensibility is visible in the interpreter design – in wide deployments it is highly probable that an interpreter would be distributed e.g. in ROM code – consider e.g. Java smart cards where the interpreter engine is an integral part of the card itself. In our work, we have abstracted an "external API", encapsulating essentially all I/O and more complex supporting functionality in the following manner.

The calling paradigm of the interpreter (see Figure 3) is modelled after the ISO 7814 smart card interface - the code, as well as input parameters are represented as type-length-value (TLV) blocks. All platform services have the exact same calling format.

A multiplexer, logically situated in front of the interpreter, will decide between "external" functionalities supported by the platform - based on the type value in the call parameter. By consequence, all external functions can in principle also be called directly from the unprotected side. For an external call, the interpreter will construct a TLV *call* block, and channel this request reecursively through the above-mentioned multiplexer. An authenticator is provided as part of the TLV call block. This can contain an external credential, but will also be used to provide unconditional information regarding the origin of the call (internal, from the interpreter, v.s. external) for those external functions needing to separate this conditions.

Rather than providing APIs to the external functions, the interpreter natively only provides a few functions to construct and decompose TLV blocks as well as to invoke the multiplexer. The macro language is used, on the language side, to provide a higher-level abstraction for translating the composition of the TLV construction, the multiplexer invocation as well as return parameter retrieval for the programmer.

In this manner, the extensibility of the system is completely dependent on the multiplexer and the services of the platform. Depending on the platform, the multiplexing functionality itself may be extensible to support new platform functionality without needing interpreter modification. The same mechanism can with small additions also be used to model the calling of other scripts, to support a kind of dynamically loadable scripted subroutines.

Another aspect of extensibility is the separation of provisioning and execution. This architecture is partly mandated by platform constraints For example, the asymmetric key operation needed to decrypt the provisioning initialization message (ObCP/Init) needs to be separated from the interpreter because the memory restrictions of the limited secure environment do not permit the interpreter such large inputs. However, this separation also provides for interpreter independence from the provisioning system

and vice versa. For the interpretation this makes it possible to support varying business models with regard to provisioning, whereas for our truly open provisioning paradigm the freedom from interpretation implies that a more commonly available and more powerful secure virtual machine (Java, .NET) could be deployed where possible.

5.3 Interpreter code

The interpreter bytecode supports a subset of Lua 2.4, constrained to a degree where the result at least should be considered a variation of the original. The bytecodes could be further simplified if the current reliance on the original LUA compiler is broken, and a compiler for a more space-efficient byte-code is built.

In our custom-made interpreter, stack and variables space borderline conditions as well as bytecode memory constraints are checked as part of every bytecode invocation. All byte-codes related to subroutines, local variables and object orientation are ignored. The only supported data type is a 16-bit unsigned short, and one-dimensional vectors of the same kind. Variables are supported, not just operations on the stack. The bytecodes are:

1. *POP*: Remove topmost stack element.
2. *PUSHUINT*: Push a value onto the stack
3. *PUSHGLOBAL*: Push a named variable onto the stack
4. *PUSHINDEXED*: Push an indexed variable onto the stack
5. *STOREGLOBAL*: Pop from stack and store in a named variable
6. *STOREINDEXED*: Pop from stack and store in an indexed, named variable
7. *CALLFUNC*: Call one of the interal functions
8. *Comparison operations*: Compare on stack: EQ, LT, LE, GT, GE, result left on stack
9. *Arithmetic operations*: Calculate on stack, result on stack: ADD, SUB, MULT, DIV, POW, MOD
10. *Bit operations*: Calculate on stack, result on stack: NOT, OR, AND, XOR
11. *Jumps*: JMP, UPJMP (Jump forward / backward), Conditional jumps on topmost stack element with or without removal, forward or backward.

Additionally, the basic "language library" consists of the following internal functions:

1. *l(x)*: Return the length of an array
2. *d(x)*: Delete an atomic variable or an array
3. *cl()*: Clear invocation buffer for external command.
4. *ap(x)*: Add vector variable x (an atomic value is a vector of size 1) as a data element to the constructed TLV for the external function call.
5. *ex(num)*: Invoke external command identified by num. Success indicated as a return parameter.
6. *rp(x)*: Retrieve the next returned TLV data elements into variable X.
7. *ei(x)*: Read the next input data element into variable x
8. *ei(x)*: Output variable x as the next result data element

A couple of code examples of Lua source code that are using the interpreter extensions are presented in Appendix D.

6 Credentials Manager

Credentials Manager (CM) is a software component (see Figure 1) which provides controlled access to both the provisioning subsystem and the interpreter residing in the secure environment. The main purpose of CM is to provide an easy-to-use API for creating, managing and using credentials for both provisioning clients inserting new credentials into the system and other applications using these credentials, and hide the details of accessing the restricted and device-specific secure environment.[3]

CM also takes care of storing credentials and related metadata in ObC database. CM is aware of certain commonly used credential *attributes* and it can either maintain these attributes or retrieve them when needed and pass these attributes to ObC programs as inputs when they are executed. Additionally, CM supports local PIN-based access control to stored credentials with user customizable *secure UI* which makes PIN stealing by malware more difficult.

This section gives an overview of most important functionality provided by the Credentials Manager component. Full listing of all CM API functions can be found from Appendix B.

6.1 Adding programs

CM supports adding both securely provisioned and plaintext ObC programs into ObC database. The `AddProtectedProgram()` function is used to add a confidential, securely provisioned program. This function would be typically called by a provisioning client after having received a protected program from a provisioning server. The caller of this function must provide ObCP/Init and ObCP/Xfer packages that contain an encrypted root key and encrypted program. CM passes these packages to the provisioning subsystem which decrypts the root key, recovers the program and returns the program in locally sealed format. CM stores the sealed program into its database with associated metadata. `AddProgram()` function adds a plaintext program into database with related metadata without processing it in the provisionign subsystem.

Certain attributes can be assigned to each program when it is added to database. Some of these attributes are static and mainly informational: the provisioner may define a human readable name for the program and attach a logo that identifies the program issuer. Other attributes can indicate that CM should either maintain or dynamically retrieve some information and pass it to the program as an input parameter when the program code is executed in the interpreter. A provisioning client adding a new credential may, for example, define that CM should ask a PIN code from the user and pass that PIN code to the program as an input parameter. Other ObC program attributes that the CM is aware of and can manage are current system time, credential execution sequence number and credential secret service identifier.

The fact that CM can manage certain commonly used programs attributes makes implementation of client applications easier. If CM, for example, automatically inserts a PIN code and a sequence number to the program, the developer of the client application does not have to implement PIN code querying functionality or maintain state between credential executions. Besides easier implementation, letting CM manage these attributes provides more security. In a typical case, CM has higher privileges that the calling client application, and thus it is more likely to have access to reliable system time and CM can also utilize the secure environment when maintaining these attributes.

Besides adding new programs to ObC database, the CM API also provides functions for listing, searching and deleting programs from the database.

6.2 Adding secrets

The API supports adding both securely provisioned and plaintext ObC secrets. `AddProtectedSecret()` function is used to add a securely provisioned secret. This function would be typically called by a provisioning client after having received a protected secret data from a provisioning server. The processing is similar to protected programs: the caller of this function must provide ObCP/Init and ObCP/Xfer

[3] Interfaces to different secure environments may vary greatly and in many cases secure environments are not even targeted for third party application developers and access to the secure environment might be either inhibited by the device manufacturer or poorly documented.

packages. CM passes these packages to the provisioning subsystem which returns the provisioned secret in locally sealed format. Then, CM stores the sealed secret and associated metadata into its database.

`AddSecret()` function is used to add plaintext secrets into ObC system. Typically this function would be used to add secrets that are input by the user. Because secrets are confidential they must be sealed before storing them to the database. To achieve this CM generates the provisioning packages locally. First, CM creates a new random RK, ObCP/Init package containing RK encrypted with PK_D and ObCP/Xfer package containing the secret encrypted with CK derived from the RK. CM passes these packages to the provisioning system which returns the secret in locally selaed format. CM also creates a new random key called *authorization key* (AK). AK is used to encrypt the just created RK and the encrypted RK is stored to the database together with the secret. Finally, AK is returned to the client application. A new credential using this secret can be only created by an application that knows the correct AK (i.e. a new program can only by endorsed to this family defined by the root key RK by an application that knows AK). This mechanism controls that only authorized programs can use the just added secret.

Like with programs, certain attributes can be associated with each secret. Besides name and issuer logo each secret can be assigned an *local access control policy*. This policy defines if a local access control PIN code should be asked when a credential with this secret is used. Each secret may also be assigned a service identifier. CM can automatically pass the service identifier to the program if this option is defined in the program attributes. Finally, each secret may be assigned a lifetime. After the lifetime has expired, CM automatically removes the secret from the database.

Besides adding secrets, functions for listing, searching and deleting secrets from the database are provided.

6.3 Creating and using credentials

In ObC system credentials are created by associating a program with a secret using `CreateCredential()` function. The caller must present correct *authorization data*. If the secret was added in protected format then this authorization data must be an ObCP/Endorse package. CM passes this package to the provisioning subsystem which returns an *endorsement* which provides the program access to the desired secret. The CM saves this endorsement into its database and presents it to the interpreter every time the credential is used. In case of originally plaintext secret (added using `AddSecret()`), this authorization data must be correct authorization key AK. CM uses this key to decrypt the stored root key which it then uses then to generate an endorsement package. CM passes this package to the provisioning system which returns an endorsement for this program.

In some cases a credential program should use more than one secret and due to the size constraints of the secure environment a longer algorithm might have to be split into several short programs. The API support creating credentials that consist of multiple secrets and multiple programs. When a multi-secret credential is executed CM passes all secrets as input parameters to the interpreter. In case of multi-program credential all the programs are executed one after another in the interpreter. In our current system the program developer must take care of sealing and unsealing the intermediary calculation results if needed. Adding automatic support to the interpreter for sealing/unsealing intermediary results without leaking secret information is discussed in more detail in Section 10.

The API supports adding arbitrary metadata to credentials in form of key-value pairs. For example, a provisioning client could add a fixed input parameter to a credential that it has just created. The client application that uses this credential could get this fixed input parameter from the CM database. Using this mechanism the provisioning client and the client application using the credential do not necessarily have to do interprocess communication for sharing.

The API provides two functions for using credentials. Simple programs are used with `UseCredential()` function which has just one application input and one output. When this function is called, CM gets the (possibly sealed) program and sealed secret from its database, creates a TLV block that contains the program code, secret, application provided input, and CM-managed inputs, like system time, if needed. CM also performs local access control check if credential secret requires that. The TLV block is then passed to ObC interpreter which executes the program with both the application provided and CM-managed input parameters and CM passes the resulting output back to the calling application.

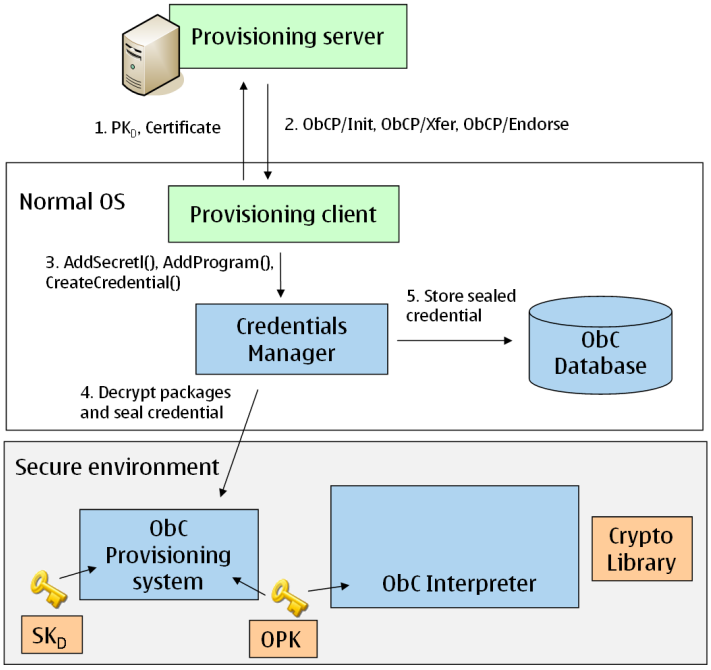

Figure 4: Provisioning example

The ObC system supports also advanced ObC programs with multiple application inputs and outputs. Credentials that use advanced programs should be used with `UseCredentialExtended()` function. Besides handling multiple inputs/outputs this function as an additional input parameter called *manifest*. Basically, manifest in an array of ObC program input parameter identifiers. Using the manifest feature a client application can override previously set program attributes. A client application may, for example, define that CM should query a PIN code from the user and pass that to the program as an input parameter even though the PIN code attribute was not set when the program was added to the ObC system. The manifest feature can also be used to define the order in which the input parameters are passed to the program, if certain ObC programs would like to read the input parameters in some other order than the default order provided by CM. This provides more freedom to ObC programs developers without sacrificing the convenience of having CM to manage certain attributes.

Besides creating and using credentials, the CM API supports listing, searching and deleting credentials.

6.4 Example scenario

Lets consider an example scenario that describes how CM handles provisioning of new credentials and usage of stored credentials. Figure 4 illustrates credential provisioning. A typical provisioning starts when provisioning client software on the target device connects a provisioning server. The client sends device public key PK_D which has been certified e.g. by the device manufacturer (step 1). The provisioning server replies by sending a number of provisioning packages (step 2). In this example the provisioning packages contain program and secret for an authentication protocol, and the needed endorsement to grant the program access to the secret. The provisioning client adds the program and the secret and creates a new credential using the CM API (step 3). CM uses the provisioning subsystem which decrypts the provisioned packages using the private part of the device key SK_D and seals the credential secret (and program if the program is confidential) using the device specific OPK (step 4). Finally, CM stores the sealed secret and program to the ObC database (step 5).

Figure 5 illustrates how the provisioned authentication credential is used. First, an authentication server sends a challenge to an authentication client (step 1). The client finds the correct credential from the ObC database e.g. based on predefined credential name or some information sent by the server. After that, the client uses the credential by giving the received challenge as input to `UseCredential()` function

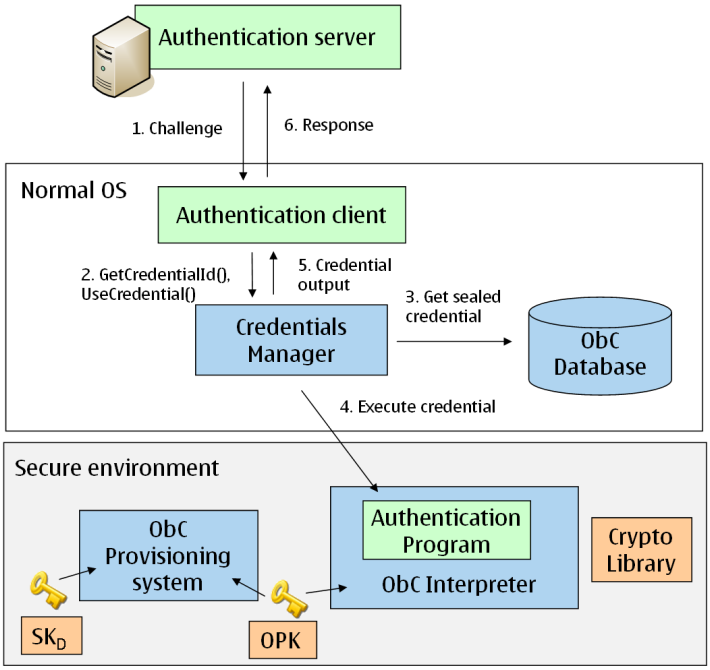

Figure 5: Authentication credential usage example

(step 2). CM obtains the sealed secret (and possibly sealed) program from the database (step 4) and executes the program on the interpreter in the secure environment with both the application provided challenge and the secret as program input parameters. CM can also provide additional input parameters to the program if such program attributes were set at the time the program was added. Once the program execution is completed, CM passes the result to the authentication client (step 5) which uses it to authenticate to the server (step 6). During both provisioning and credential usage the secrets (and confidential programs) are always kept encrypted outside the secure environment.

6.5 API usage example

Now we present a short example that illustrates the basic usage of CM API. In this example a provisioning client first adds securely provisioned HTTP Digest authentication ObC program. The provisioning client provides ObCP/Init and ObCP/Xfer packages, defines the name of the program and indicates that CM should pass a service identifier to the program as input parameter when the program is used (the service identifier contains username, server DNS name and realm and it is used in the HTTP digest computation). After that, the provisioning client adds a securely protected secret. The provisioning client defines a name for the secret and a service identifier. Finally, the provisioning client creates the credential by associating the added program to the secret and providing the ObCP/Endorse package that grants the program access to the secret.

The application that wishes to use the credential first finds the correct credential from the ObC database based on its name. The client calculates HTTP digest response by providing HTTP digest challenge as input parameter. CM passes the service identifier of the secret to the program as input parameter automatically. More examples can be found from Appendix B.6.

```
// Provisioning client:
programId = AddProtectedProgram(protectedProgram, // contains ObCP/Init, ObCP/Xfer
                                "HTTP digest authentication", // program name
                                null, // no logo specified
                                false, // no PIN needed
                                false, // no time needed
```



```
                                false, // no sequence number needed
                                true); // service identifier needed

secretId = AddProtectedSecret(protectedSecret, // contains ObCP/Init, ObCP/Xfer
                              "Example password", // name of the secret
                              false, // no local PIN needed
                              null, // local PIN policy
                              "john@example.com:realm"); // service identifier

credentialId = CreateCredential(programId, // defines the program to use
                                secretId, // defines the secret to use
                                "Example credential", // credential name
                                authData); // contains ObCP/Endorse

// Application using the credential:
credentialId = GetCredentialId("Example credential");

digest_response = UseCredential(credentialId, // identifies credential
                                digest_challenge); // application input
```

6.6 Local access control and secure UI

When a new secret is added to the system, the provisioner may define if this secret requires local access control. If so, whenever a credential with this secret is used, a *local access control* PIN code is asked from the user. The execution of the credential is only permitted if the user provides correct local PIN code. The local PIN code is defined by the user when the system is taken into use for the first time. At the same time the user may also define a *custom picture* and *custom text*. When ever a PIN code dialog is presented to the user, the custom picture and text are shown to the user together with the PIN code dialog. The user is supposed to enter his PIN only if she sees her own personal picture and text.

Besides local access control PIN codes, certain programs may utilize CM-managed PIN codes as described earlier. We call these *server* PIN codes (because they are typically used to authenticate to a server). Altogether there are four possible PIN code alternatives when a credential is used:

1. PIN code is not required by the program or the secret. No PIN code will be asked from the user when a credential is used.
2. Program requires a server PIN code. The CM asks the PIN from the user when a credential is used and passes this PIN to the program.
3. Program does not require PIN code, but local PIN code is needed for local access control related to the secret of the credential. The CM asks the PIN from the user and allows the usage of the credential only if the local PIN code is correct.
4. PIN code is required by both the program and the secret. A case like this should be avoided. Typically the application adding secrets (e.g. provisioning client) is aware of the PIN code requirements of the particular program at hand and should not require local access control PIN code if server PIN code is already needed by the program.

7 Implementations

In this chapter we describe the ObC platform implementations we have built and present some performance measurements. The primary target for our ObC architecture was M-shield based mobile phone, but the design is valid in a larger scope as well, and through a few collaboration projects the design and the core implementations have been validated on two other platforms as well.

7.1 Symbian phone with M-shield secure environment

As our main implementation platform we selected Nokia N95 mobile phone. This device runs Symbian OS v9.2 operating system (with platform security support) on 300 MHz OMAP 2420 platform and it has, like many other Nokia high-end phones, M-shield secure environment built-in.

The interpreter for the M-shield architecture was written in C, and is in its compiled format around 5 kB in size (Thumb 16-bit binary), including necessary initialization, the interpreter core, wrapper code for the cryptographic extension functions, I/O mechanisms and support for sealed input data. On our target device bytecodes of size 1073 bytes can be accommodated, and up to 75 local variables can be used by the script. This is enough to implement e.g. the authentication algorithm for 3G mobile networks. In addition, proprietary credential programs have also been successfully deployed on the platform.

The performance of the interpretation has been measured using the Nokia N95 device. The invocation of the interpreter itself (done once) takes 1.8 ms. The invocation of a minimal script bytecode, i.e. invoking the secure environment takes 1.2 ms. A "typical" script taking a couple of input parameters, one of them sealed, unsealing it, doing an AES operation on the secure side and returning a result takes 8.3 ms to execute. Provisioning an 1 kB secret scripts sets you back 10.3 ms. Thus, although not its primary goal, our architecture in terms of speed compares favourably to smart cards with intepretation where the invocation and operating speeds are one to two orders of magnitude slower [2].

The Credentials Manager component for Symbian OS was implemented in C++ using typical Symbian client-server model: CM is a system server running in its own process and client applications communicate with it by linking a client DLL that provides the API functions and takes care of the inter-process communication between the client application and the server.

The sealing/unsealing mechanism of the interpreter ensures that the stored confidential programs and secrets are never available in plaintext outside the secure environment. However, the interpreter itself does not control who is allowed to create, use and delete credentials. In Symbian platform security each process (UI application or system server) has a set of *capabilities*. CM uses this capability model for providing a coarse-grained access control model to its credentials. A user-grantable low priority capability is required from applications that want to just use credentials. More sensitive operations, like deleting credentials, require a non-user-grantable higher priority capability which means that the application has gone through Symbian Signed process [22]. This coarse-grained access control policy is in line with our openness and security objectives: creating and using credentials is not restricted to few pre-defined applications only and at the same time it provides some level of protection (for example completely untrusted applications cannot just wipe out the whole credentials database). Access to the interpreter and the provisioning subsystem is restricted to the CM. This is enforced using Symbian platform security.

Besides the CM implementation, we have built a few demonstrators on top of this API. To enable secure authentication for web browsing and SIP [18] based VoIP applications we have modified a Symbian OS component called digest authentication filter. This modified filter performs HTTP digest authentication calculation by invoking an HTTP digest authentication credential program through the CM API. The credential program uses sealed passwords that are stored in ObC database.[4] The needed passwords for digest authentication can be either provisioned from OMA DM server (we have modified the existing Symbian OMA DM provisioning client to insert provisioned digest authentication passwords to ObC database instead of storing to Symbian file system) or inserted by the user. Additionally, our research partners have implemented a CT-KIP [17] provisioning client that inserts securely provisioned programs and secrets to ObC database and a S60 client application that uses the CM API for a proprietary token based authentication protocol.

7.2 Linux PC with TPM secure environment

For the PC platform, a prototype was built around TPM and the notion that the linux kernel is trustworthy enough to provide necessary isolation for the credential programs [19]. This approach is straightforward to implement, but has many known security issues. The lack of hardware protection for the secure execution environment is one clear difference to the M-shield approach.

[4] HTTP Digest Authentication credential program does not currently work in some restricted secure environments.

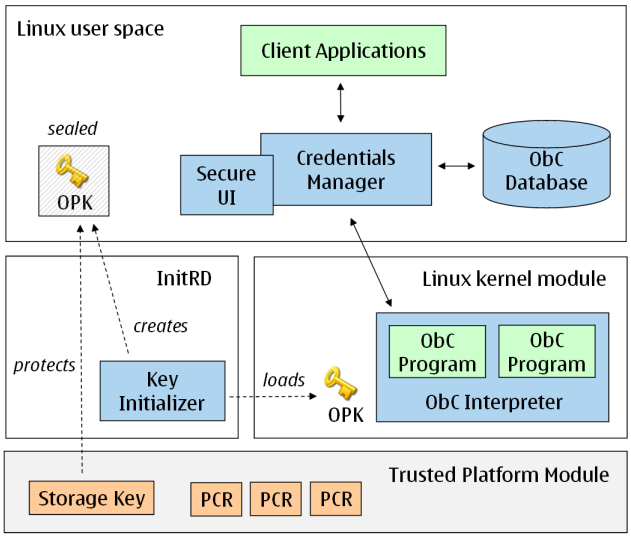

Figure 6: ObC architecture on Linux

In short, we implemented secure booting for the PC platform (TPM enabled IBM T41 laptop), letting the BIOS measure the environment including the Trusted Grub [25] bootloader. The bootloader was augmented to support an encrypted RAMdisk keyed from the TPM - i.e. the sealed RAMdisk key was conditioned to the reference metrics of the bootloader and its configuration, and also to the kernel version. Thus the encrypted InitRD was only decrypted if the bootloader and kernel were unmodified. The RAMdisk further contained the interpreter (in this case encapsulated as a loadable Linux kernel module, but it could as well has been integrated to the kernel). The root key OPK was initialized to a random value during the first boot (by the "Key initializer") the and separately sealed for the platform state before being placed on the hard disk. Future device boot-ups would unseal the OPK during RAMdisk code execution, and initialize the interpreter with it.

The Credentials Manager on this research platform was not protected by mandatory access control other than a somewhat extended basic Unix security model - we were using LIDS [14] for protecting against unauthorized kernel module loading, and unnecessary root process access to critical devices like */dev/mem* and *dev/kmem*. The driver interface of the interpreter and provisioning subsystem exposed a random key to the first connecting process and all further communication between the Credential Manager and the interpreter was authorized based on the knowledge of this key. CM was started early in the init scripts, and these, as well as the bash shell, could also in principle have been included e.g. in the reference metrics for the encrypted InitRD. In this manner we could protect the credential manager to some degree also on this platform that had no extra security deployed on the operating system side. The Credentials Manager included a user customizable secure UI based on GTK2 and ObC database based on SQLite.

Using this architecture, a demonstration where a Mozilla Firefox 2.0 browser was modified to call the ObC for HTTP-digest authentication was implemented, i.e. the platform served as an extended credential keystore where all critical computations and key management related to the algorithm were done in kernel memory space, on the condition that the kernel and InitRD integrity (as provided by the TPM) were not compromised.

We acknowledge the well known fact that UNIX kernel is too big and has too many interfaces to be considered secure, especially if any processes / daemons are allowed to be executed as root. However, the example shows one deployment option of the ObC architecture that in its way imposes a minimal impact on deployed devices, brings some security (e.g. for user WWW passwords) while, in this case relying only on discretionary access control support by the hardware.

7.3 Linux Tablet with Virtualization

The hardware / software solutions for OnBoard Credentials described above constrain the ObC platform to be usable only when the operating system is integrity checked. For e.g. mobile phones this is acceptable, since the integrity of the device and its operation is regulated by authorities - user safety as well as network integrity must be unconditionally assured. However, there are also many classes of devices that include no critical communication channels, or are constructed in a way that hardware segmentation separates user functionality from e.g. licenced communication. Also, in some user communities, the absense of device "control points" managed by external parties is seen as a benefit, even as a criterion for a purchase decision.

Thus, there is a niche for trying to adapt ObC also for "openness", in the sense that it can be used for such security services where the user is the main stakeholder (like password protection). In this spirit we have also constructed an ObC research prototype built on top of a commercial hypervisor offering (for ARM-11) on a Nokia N800 internet tablet. The Maemo OS 2007 Linux distribution (the default for the N800 Internet Tablet) has been para-virtualized as the main user compartment, whereas the ObC secure enviroment in this prototype constitutes its own compartment, running natively (without an underlying OS) directly on top of the hypervisor.

This specific combination of hardware and software cannot be considered secure. Of the assumptions in section 2.1, the integrity of the environment (e.g. secure boot) cannot be guaranteed by the N800 hardware. As this is a fundamental need for ObC, we "assume" the presence of such a functionality for our prototype (the pictures indicate the presence of secure boot), recognizing that many processor (versions) easily could accommodate it. As in the TPM case we also overlook the fact that the setup runs the ObC byte-code execution in processor-external memory chips, wide open to memory-bus eavesdropping or e.g. on-the-fly modification of already integrity-checked code.

Security in (para)virtualized environments is a research area in itself. Strictly managed access must be in place for all system resources that may cause information leakage between compartments, including virtual memory configuration, interrupts and direct memory access. If all compartments are running on the same processor, timing back-channel attacks may be possible. We leave further discussion on these topics to the references, e.g. [5] or [6], and assume for our purposes that the deployed hypervisor satisfies the isolation property with regard to its compartments.

Next, we briefly describe the prototype in reference to Figure 7. The processor will (as assumed) be equipped with a secure boot functionality by which the bootloader (the first code that the processor runs) is validated. Again, we point out that this is a very typical feature supported by many embedded / DSP processor (cores). If the validation of the bootloader fails, the device does not continue booting.

When the bootloader is active, it in turn validates the tuple (*hypervisor, ObC compartment, OS compartment* - necessary public key information as well as algorithms are part of the bootloader). Based on the measurement result(s) the bootloader will display to the user one of three splash screens outlined in Figure 7. They indicate to the user whether the verification state is that a) all three elements were verified, b) the hypervisor and ObC compartment were successfully verified, or c) some other case. In cases a) and b) the bootloader will convey the OPK as an input from the user (or. e.g. a USB stick) to the ObC compartment at boot-up, in the third case the boot-up simply continues.

When the hypervisor, the ObC compartment and some, potentially verified Linux, have been booted, a character device driver is present on the para-virtualized Linux to communicate with the ObC secure environment. In our demonstrator, the communication was arranged by means of a shared memory page between the two compartments. When an ObC program needs to be activated, the device driver copies any input to the shared memory and activates the ObC compartment by a software (hypervisor-provided) interrupt. In the prototype the hypervisor scheduling is arranged in a manner where the ObC enviroment on activation will enjoy absolute system priority for the duration of the ObC program execution. The return values are finally conveyed back over the shared memory page, and the secure environment remains passive until the next call. The whole ObC environment is running directly on top of the hypervisor, and in total consumes less than 100kB of RAM.

The main difference to previous implementations is the absence of third-party credentials for the ObC platform itself. A device key can (and is) constructed for the environment to support provisioning,

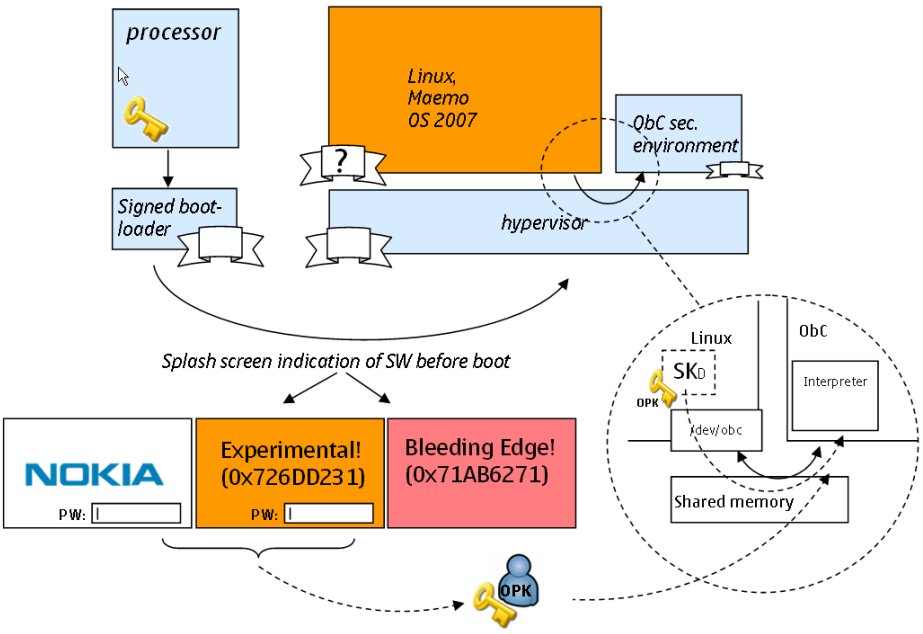

Figure 7: Architecture with virtualization

protected by the OPK when stored in device persistent storage. However, this key cannot reasonably be certified by any third party other than the user himself, since the secrecy of the OPK is ultimately governed by the user. Even with this shortcoming, the setup is perfectly acceptable for a multitude of credentials - all such where the user has an advantage to keep the credentials secret, and where the absence of device binding (making it possible to copy ObC credentials between devices) is not directly harmful. This property is true for e.g. most passwords or for e.g. electronic keys to access the user's private property.

Compared with the TPM-Linux implementation, the system trust now relies mostly on the proper behaviour of the hypervisor only - a more reasonable assumption than to trust the integrity of a complete OS.

Let's also briefly examine what we achieve with this architecture in terms of openness. For the device owner, the only "locked" piece in the system is the bootloader, and even that can be publicly scrutinized to ascertain the absence of any embedded secrets. The bootloader will load any operating system that the user chooses to port to the device. The adventurous user also has the choice to modify the para-virtualized OS in any manner he likes, and still retain the possibility to use the ObC for his own purposes, e.g. for securing passwords. In this scenario he must "remember" the OS integrity check on the "Experimental" splash screen to be certain of the continued integrity of his own OS - otherwise a virus might use the ObC component to e.g. decrypt his address books, love letters or equivalent. At the same time, the default user that only uses manufacturer-signed OS images can to a better degree be assured of device integrity as long as the splash screen remains in its default setting at every boot. In other words, we provide a wide range of flexibility and openness, while still giving users the benefits of the ObC environment, at least for some of its purposes.

8 Development tools

For third party development to be successful as a concept, a good development environment is needed. Even though the development of the credential program itself is most likely only a small subset of a given software project, we still need to make the process of developing and debugging the credential programs

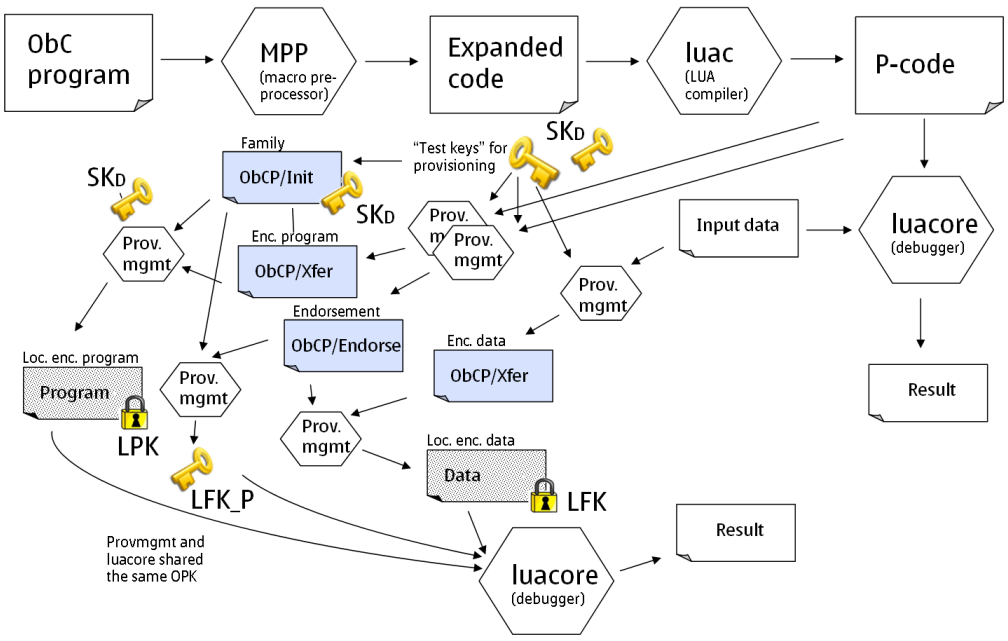

Figure 8: ObC development toolchain

as painless as possible. The developed ObC programs need to be free of errors - a logical error in the bytecode can destroy the security of the entire system solution.

In our reference implementation we use the original Lua language compiler for code generation. As we do not support the entire set of bytecodes, not all language features can be used. The missing byte codes - mapped to language constructs - are separately listed in the development environment documentation - if these are "accidentally" used the interpreter will return a run-time error.

8.1 Interpreter emulator for PCs

Additionally we provide an emulation of the secure environment. Essentially the tool is a debugger, where the credential code can be executed in a step-by-step fashion or forwarded with the use of breakpoints. At any time, the entire state of the intepreter and all parameters are visible, as are the current memory consumption values in relation to the values supported in the real environment. We can also provide some relative time estimates of the different bytecodes and external functions, but these will vary between target platforms. All provisioning functions are implemented as command-line tools, and the emulator also can be invoked without any interaction. This makes it possible to construct unit test harnesses (using system scripting facilities) for script developers to validate test cases during development.

The cryptographic API and other auxiliary functionalities are also emulated. The emulator also provides hardcoded keys emulating the device-specific key pair a well as the OPK, i.e. all provisioning functionality can be tested against the emulator as well - everything except for the return and validation of a possible full-fledged platform certificate. To help the single developer, in the absence of a server-end, to construct the provisioning messages and emulate the reception of the same, command-line tools implementing both the construction and local "conversion" of ObCP/Init:s, ObCP/Endorse:s and ObCP/Xfer:s are included.

Currently the development environment (see Figure 8) and emulator/debugger are available for both the Windows platform and Linux operating systems. An example of the debugger interaction is shown in Appendix E.

8.2 Application development for Symbian phones

We have also compiled a version of the ObC platform in which the interpreter runs in Symbian OS, shielded by the platform security architecture. This version includes the full Credentials Manager API towards the client applications and is thus in terms of interfaces completely equivalent to the M-shield based implementation where the interpreter and provisioning components are executed in the secure environment. However, here the stored secrets and confidential programs and the isolated execution of programs are protected by Symbian OS platform security means only [5].

This implementation complements the credentials programs development environment, and can be used for client program development as well as easy prototyping on both the PC based Symbian emulator as well as on any Symbian OS v9.x device independent of the availability of real secure environment.

9 Related Work

From an architectural viewpoint, ObC stands close to the *Small Terminal Interoperable Platform* (STIP) by the GlobalPlatform consortium [8], in that the aspiration of both is to provide an open, well specified platform complete with provisioning support to be used for security services in mobile devices. However, where GPD/STIP relies on a model with rigid security properties proteted by underlying smart-card technology and with a provisioning based relying on the card issuer as a trusted third party. ObC strives to be deployable without additional hardware on existing security architectures on the devices themselves, and proposes a security solution that promotes openness wherever possible, especially in terms of the provisioning solution. ObC also provides provisioning of applications not pre-validated for the environment, be it smart card or mobile phone secure environment. We foresee that the differences will reflect the range of services deployable on the respective platforms - GPD/STIP already has an impressive collection of payment-, DRM and ticketing related trials and deployments all over the world, services where the transaction values motivate the more rigid platform security constraints.

McCune et al [16] describe how the support for dynamic roots of trust in modern processors can be used in conjunction with a TPM to implement a secure exeuction 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 interpreter and the provisioning subsystem.

Gajek et al [6] describe combining a TPM with a virtual machine monitor so that a "wallet" can be implemented as a trusted guest virtual machine. This is similar to the implementation outlined in Section 7.3 combined with a TPM.

Lee et al describe a hardware-assisted architecture for protecting "critical secrets" in microprocessors [13, 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 – the only software allowed to operate on critical secrets are the "trusted software modules" which are authorized by the device owner or issuer.

10 Analysis

10.1 Provisioning

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 secret data. 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 9 shows how ObCP/Init and ObCP/Endorse are modified when digital signatures are used for endorsement.

[5] In Symbian 9.x every process has its private persistent storage access and process execution memory

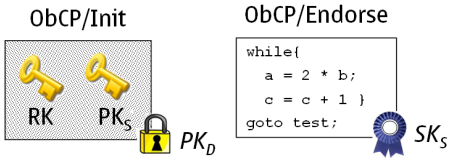

Figure 9: Endorsement using a digital signature

10.2 Interpreter and Security

The logical correctness of the interpreter is of specific importance, as it will be run in environments where e.g. breaking the isolation property with respect to the secure environment could cause significant damage to other services running in that domain. We have written our interpreter with this threat model in mind, and the use of a fairly high-level language (ANSI C) and sparse use of e.g. pointers have made it possible to audit the code for security. Still, we are lacking a (formal) security proof of the isolation property - and this will certainly be worthwhile exercise as near future research.

Regarding footprint minimalism of the virtual machine, there is more to be achieved, although we have fulfilled the goals set for us. Having usability as a starting point and minimizing for deployment reasons has certainly taken us to a different end result than would have been by first designing a minimal virtual machine and then porting a suitable language compiler on top of it. We hope that the architecture put forward in this document stimulates further reserch in this area, in supporting the notion that utility may not always measureable in versatility - less might be more. In addition to isolation, the interpreter controls the resource consumption of all scripts in terms of execution cycles, another of the requirements of our design.

Secure paging

The space reserved for bytecode can be enlarged by securely paging the bytecode – loading only those parts of the bytecode that currently is needed for execution. This form of paging will reveal varying amounts of the application data flow to a cunning eavesdropper. If deployed, the constraints on byte code size can be lifted, at the expense of the required insight by the application developer, who now must take this property into account.

In the same manner the support for extensible macro “libraries” could be implemented. Say, that a specific credential needs a well known cryptographic primitive (algorithm) that is not part of the platform services. The interpretation can be extended with a function to identify the call to such a function, and in case of activation the interpreter must securely store its state as well as the library call parameters, make the swap through the operating system, execute the library function, and return to interpreting byte-code by means of a second “context switch”.

10.3 Credentials Manager API

One of the major architectural design choices that we had to make was whether to design our own custom credentials management API or to adapt the ObC architecture into an existing, possibly standardized, crypto token API. The decision to design our own API was made based on two aspects: First, there is no single crypto token API that is dominant across the wide range of different platforms we are targeting. For example, PKCS #11 [12] and Microsoft CryptoAPI are used by many applications in certain platforms, but neither of them is applicable to the wide range of devices we are targeting: mobile phones, Internet tablets and PCs regardless of the used operating system. Secondly, none of the existing APIs enable the usage of all ObC system features. The existing crypto token APIs typically support only few pre-defined crypto operations, like encryption/decryption and signing/verification, whereas the ObC system enables arbitrary operations to be executed in the secure environment. Additionally, the Credentials Manager API supports secure provisioning, a feature that is missing from the existing APIs.

Naturally, designing a new custom API has its drawbacks. All the applications that want to use the ObC system must be ”‘ObC aware”’, i.e. they must be designed and implemented our API in mind; e.g.

an existing PKCS #11 based application cannot use ObC system without modifications. As future work item, we are considering the implementation of wrappers for existing, well known crypto token APIs. In Symbian platform we could implement a plugin to Symbian operating system Unified Key Store component. This plugin would insert the imported keys into ObC system instead of storing them to Symbian file system or external harware token, and perform the requested cryptographic operations in the secure environment instead normal operating system side or external token. Providing a complete one-to-one match between the Credentials Manager API and Symbian Unified Key Store API is impossible, but the plugin could provide linkage between the commonly supported subset of features. A similar wrapper could be implemented e.g. for PKCS #11 and CM API.

11 Conclusions

In the domain of smart cards there has for several decades been research and development towards the notion of multi-application cards or "white-cards". These concepts have never been widely adopted, possibly because the business models are cumbersome - either the user must pay up front to purchase such a card for applications to reside on, or then one card issuer like a mobile operator, bank or a credit card company must have an incentive (and an architecture in place) to sub-let application space on their respective cards.

We believe that the OnBoard Credentials Platform addresses this issue in a manner that may stimulate larger-scale deployments and a true third-party ecosystem for secure credentials of whatever kind. ObC bootstraps from security components and mechanisms that already are widely deployed for other purposes, and adds the flavour of openness to provisioning - certainly an advantage for small-scale service providers that in the ObC context can go about and build their secure services independently of device manufacturers or other stakeholders. The ObC platform, however, solves only the first pieces of this puzzle. The common execution environment, the credentials manager and provisioning structures must be accompanied by standardized provisioning protocols that take into account not only the credential secrets but also the execution logic. Also needed are common ways of describing and determining the security level of provisioned-to devices and platforms, as well as trust hierarchies to support the provisioning entity in this decision-making.

References

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

[2] Damien Deville, Antoine Galland, Gilles Grimaud, and Sebastien Jean. Smart card operating systems: Past, present and future. In *Proceeding of 5th NORDU/USENIX Conference (NordU2003)*, February 2003.

[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, 2007.

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

[5] Daniel R. Ferstay. Fast secure virtualization for the arm platform. M.Sc. Thesis, University of British Columbia `http://www.cs.ubc.ca/grads/resources/thesis/May06/Ferstay_Daniel.pdf`, 2006.

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

[7] GlobalPlatform. GlobalPlatform's GPD/STIP Solution for Mobile Security. GlobalPlatform White paper `http://www.globalplatform.org/uploads/STIP_WhitePaper.pdf`, August 2007.

[8] GlobalPlatform™. Why the mobile industry is evolving towards security. White paper, http://www.cs.ubc.ca/grads/resources/thesis/May06/Ferstay_Daniel.pdf, 2007.

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

[10] Kaz Kylheku. The macro processor mpp reference manual. http://users.footprints.net/~kaz/mpp-doc.html, 1998.

[11] RSA Laboratories. PKCS 1 v2.1: RSA Cryptography Standard. Available at: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf, June 2002.

[12] RSA Laboratories. PKCS 11 v2.20: Cryptographic Token Interface Standard. Available at: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf, June 2004.

[13] Ruby B. Lee et al. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2005.

[14] LIDS Secure Linux System. http://www.lids.org/.

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

[16] 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.

[17] Magnus Nystrm. Cryptographic Token Key Initialization Protocol (CT-KIP). Technical Report RFC 4758, IETF, November 2006.

[18] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, John Peterson, Robert Sparks, Mark Handley, and Eve Schooler. SIP: Session Initiation Protocol. Technical Report RFC 3261, IETF, June 2002.

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

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

[21] Harini Sundaresan. OMAP™ platform security features. TI White paper http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf, July 2003.

[22] Symbian signed. https://www.symbiansigned.com.

[23] Trusted Computing Group. https://www.trustedcomputinggroup.org/home.

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

[25] Trusted GRUB. http://trousers.sourceforge.net/grub.html.

Appendix

A Glossary and Abbreviations

Authorization Key (AK) Controls access to ObC secrets that are added to CM in plaintext format.

Confidentiality Key (CK) Derived from family root key and used to decrypt confidential data for provisioning.

Credential Manager (CM) Software component that maintains credentials database and provides API for creating and using credentials.

endorsement Granting an ObC program to have access to certain ObC secret.

family The group of ObC programs or ObC secrets that are bound together by one (or several) root keys.

Integrity Key (IK) Derived from RK and used to integrity protected data for provisioning.

OnBoard Platform Key (OPK) A persistent device secret from which local secrets are derived.

ObC OnBoard Credentials, virtual credentials utilizing secure hardware.

ObC interpreter Virtual machine that executes ObC programs and isolates them from rest of the secure environment.

ObC program A bytecode (Lua script) that is executed in ObC interpreter in secure environment.

ObC secret The secret data, like cryptographic key or password, that is used in credential calculation.

ObCP/Init A provisioning package that contains root key RK encrypted with device public key PK_D

ObCP/Xfer A provisioning package that contains either an ObC secret of confidential ObC program. The package is encrypted and integrity protected.

ObCP/Endrose A provisioning package that grants an ObC program access to certain already provisioning ObC secret.

Root Key (RK) 128-bit symmetric AES key that defines a family.

Root Key for Programs (RK_P)

Root Key for Secrets (RK_S)

seal Secure storage facility for a ObC program. An encryption on secret key, diversified on the hash of the program itself.

secure environment A secure execution environment provided by secure hardware. Typically, provides both secure storage and isolated memory for execution.

unseal An operation by which sealed data can be read by the program for which the seal has been made.

B Credentials Manager API

In this appendix section we describe the CM API in more detail. The function descriptions are presented using generic syntax and one should be able to implement this API using any common programming language.

B.1 Program management

AddProtectedProgram

```
int AddProtectedProgram(string protectedProgram, string name, string logo,
                        bool serverPinNeeded, bool timeNeeded, bool seqNoNeeded,
                        bool serviceIdNeeded)
```


This function adds a securely provisioned program into the Obc database. This function would be typically called by a provisioning client after having received a protected program from a provisioning server. CM passes the `protectedProgram` parameter to the provisioning subsystem which returns the program encrypted using local program key LPK. CM then stores this encrypted program into ObC database with the associated metadata.

Parameters:

`protectedProgram` - Byte array that contains the actual program (bytecode of Lua script) in encrypted and integrity protected format. This byte array is a concatenation of the ObcP/Init package and the ObcP/xfer packages.

`name` - Human redable name of this program. This name is shown to the user when asking PIN code for a credential that uses this program. The name can be also used for finding this program (or credential using this program) from ObC database.

`logo` - A small image that is shown to the user when asking a PIN code for a credential that uses this program. This image could be for example the logo of the company that provisioned this program.

`serverPinNeeded` - Boolean that defines server PIN code attribute for this program. If true the CM will present a PIN code prompt to the user and pass the PIN code to the program as input parameter when a credential that uses this program is executed.

`timeNeeded` - Boolean that defines system time attribute for this program. If true CM will pass the current system time to the program as an input when a credential that uses this program is executed. The time will be provided as an array containing the following elements: origin, year, month, day, hour, minute, second. The origin element defines where CM read the time. Possible values could be e.g. operating system time, secure environment time and network operator time. The program may use this time array in its calculation and possibly adjust its calculation based on the trustworthiness of the time origin.

`seqNoNeeded` - Boolean that defines sequence number attribute for this program. If true, the CM maintains a counter for each credential that uses this program and provides the sequence to the program as an input parameter.

`serviceIdNeeded` - Boolean that defines service identifier attribute for this program. If true, this program can only be used with secrets that have service identifier assigned to them (see `AddProtectedSecret()` and `AddSecret()` function descriptions). When a credential that uses this program is executed, CM passes the service identifier assigned to the secret of the credential to the program as input.

Returns: Unique identifier for this program. In case of error, one of the following error codes: ErrorDecryption / ErrorDatabase / ErrorSecureEnv.

AddProgram

```
int AddProgram(string program, string name, string logo, bool serverPinNeeded,
               bool timeNeeded, bool seNoNeeded, bool serviceIdNeeded)
```

This function adds a non-protected program to the Obc database. The function description is similar to `AddProtectedProgram()` except for the first input parameter. CM stores the program with the associated metadata into ObC database in plaintext format.

Parameters:

`program` - The program (bytecode of Lua script) in plaintext format.

See description of `AddProtectedProgram()` for other parameters.

Returns: Unique identifier for this program. In case of error, one of the following error codes: ErrorDecryption / ErrorDatabase / ErrorSecureEnv.

GetProgramNames

```
int GetProgramNames(string[] programNames)
```

This function returns the names of all programs stored into ObC database.

Parameters:

`programNames` - The names of all programs will be written to this output parameter.

Returns: In case of error, one of the following error codes: ErrorNotFound / ErrorDatabase

GetProgramId

```
int GetProgramId(string programName)
```

This function returns unique identifier for a program with a matching name. If more than one program with the same name is found, identifier of the one added to the database last will be returned.

Parameters:

`programName` - Name of program

Returns: Unique program identifier. In case of error, one of the following error codes: ErrorNotFound / ErrorDatabase

DeleteProgram

```
int DeleteProgram(int programId)
```

This function deletes a program from the OnBoard Credentials database. Deleting a program deletes all the credential using the program automatically.

Parameters:

`programId` - Unique identifier for the program to be deleted.

Returns: Success / ErrorNotFound / ErrorDatabase

B.2 Secret management

AddProtectedSecret

```
int AddProtectedSecret(string protectedSecret, string name, string logo,
                       bool localPinNeeded, string serviceId, int lifetime)
```

This function adds a new securely provisioned secret into Obc database. This function would be typically called by a provisioning client after having received the secret from a provisioning server. CM passes the `protectedSecret` parameter to the provisioning subsystem which returns the secret encrypted using local family key LFK. After that, CM stores the sealed secret and the associated metadata into ObC database.

Parameters:

`protectedSecret` - Byte array that contains the actual secret in encrypted and integrity protected format. This byte array is a concatenation of ObcP/Init and ObcP/xfer packages.

`name` - Human readable name for the secret. The name of the secret is shown to the user when a PIN code is asked for a credential that uses this secret. The name can also be used for finding the right secret (or credential using the secret) from ObC database.

`logo` - Logo provided by the provisioner of this secret. This logo is shown to the user when asking a PIN code for a credential using this secret.

`localPinNeeded` - Boolean that defines whether a local access control PIN code for this secret. If true, CM prompts a PIN code query to the user when a credential with this secret is used and permits the usage of the credential only if the user enters the correct local access control PIN. One should note that this local access control check is independent of the possible program related server PIN requirement.

`serviceId` - Identifies the service that this secret is used for. Typically, this service identifier would consist of username, remote server DNS name and realm formatted in the following fashion (but in principle the provisioning client is free to choose any kind of service identifier format): username@server-name:realm. A client application can use the service identifier to select the correct secret if it does not know the secretId.

`lifetime` - Defines the lifetime of this secret. Once the lifetime has expired, CM automatically deletes the secret and all credentials using this secret from the ObC database.

Returns: Unique identifier for this secret. In case of error: ErrorDecryption / ErrorDatabase / ErrorSecureEnv.

AddSecret

```
int AddSecret(string secret, string name, string logo,
              bool localPinNeeded, string serviceId, int lifetime, string authKey)
```

This function adds a plaintext secret to the Obc system. This function would be typically called by a an application after having asked the user to input a secret (like a password). The secret must be sealed before storing it. First, CM generates two new random keys: authorization key AK for this secret and a new family root key RK. Then, it produces ObcP/Init and ObcP/xfer packages using RK. CM passes these two packages to the provisining subsystem which returns the secret encrypted using local family key LFK. CM encrypts RK using AK, creates a MAC for integrity protection and stores these. Authorization key AK is returned to the calling application. Later, a program can only be assigned to this secret if the calling application presents correct AK.

Parameters:

`secret` - The actual secret in plaintext format.

`authKey` - The authorizationKey AK for this secret is written into this output parameter.

For other parameters see `AddProtectedSecret()` description.

Returns: Unique identifier for this secret. In case of error: ErrorDecryption / ErrorDatabase / ErrorSecureEnv.

GetSecretNames

```
int GetSecretNames(string[] secretNames)
```

This function returns names of all secrets stored into ObC database.

Parameters:

`secretNames` - A list of all secret names is written into this output parameter.

Returns: Success / ErrorNotFound / ErrorDatabase

GetSecretId

```
int GetSecretId(string secretName)
```

This function returns unique identifier for a secret with matching name. If more than one secret is found with the same name, the identifier for one inserted last to the database is returned.

Parameters:

`secretName` - The human readable name of the secret.

Returns: Unique identifier for this secret. In case of error: ErrorNotFound / ErrorDatabase.

DeleteSecret

```
int DeleteSecret(int secretId)
```

This function deletes a secret from the ObC database. When a secret is delered all the credentials using this secret are deleted automatically too.

Parameters:

`secretId` - The secret identifier

Returns: Success / ErrorNotFound / ErrorDatabase / ErrorInUse

B.3 Credential management

CreateCredential

```
int CreateCredential(int programId, int secretId, int authorization, string name)
```

This function creates a new credential instance by associating a program to a secret. Typically this program would be called by the same provisioning client that added the secret since the caller has to provide correct authorization for the secret.

First, the CM checks from database how the secret was added to the system. If the secret was provisioned in protected format the authorization parameter must contain ObcP/endorse package. CM passes this package to the provisioning subsystem which returns local family key encrypted using local endorsement key: $Enc_{LEK}(LFK)$. This *endorsement* is stored into the database together with other credential metadata.

If the secret of this credential was added in plaintext format, CM gets the stored encrypted root key for secrets $Enc_{AK}(RK)$ and the stored MAC and decrypts RK if the given AK is correct. Then, CM generates the ObcP/Init and ObcP/endorse packages using the RK. These two packages are passed to the provisioning subsystem which returns endorsement ($Enc_{LEK}(LFK)$).

Parameters:

`programId` - The program that should operate on the secret

`secretId` - The secret that is used

`authorization` - Authorization data, i.e. the ObCP/Endorse package if the secret was provisioned in protected format or authorization key (AK) if the secret was added in plaintext format.

`name` - Human readable name for the created credential. The name can be used for finding the correct credential from the ObC database.

Returns: Unique identifier for this credential. In case of error: ErrorSecureEnv / ErrorDatabase / ErrorAuthorization

AddSecretToCredential

```
int AddSecretToCredential(int credentialId, int secretId)
```

This function adds an additional secret to an existing credential. The additional secret must belong to the same family as the original secret.

Parameters:

`credentialId` - The credential that the additional secret should be added to

`secretId` - The secret that is added

Returns: Success / ErrorSecureEnv / ErrorDatabase

AddProgramToCredential

```
int AddProgramToCredential(int credentialId, int programId, int authorization)
```

This function adds an additional program to an existing credential. Also the new program must be endorsed to use the secret(s) of the credential. If the secret(s) of the credential were securely provisioned, the authorization data must be correct ObCP/Endorse package. If the secret(s) of the credential were added in plaintext format, the authorization data must be correct authorization key AK. The provisioning subsystem is used like in `CreateCredential()` function.

Parameters:

`credentialId` - The credential that the additional program should be added to programId The program that is added (this program should belong to the same family as the original program of this credential)

`programId` - The program that should be added to an existing credential.

`authorization` - Authorization data. Contains ObCP/Endorse package if the secret of this credential was provisioned in protected format or authorization key (AK) if the secret of this credential was added in plaintext format.

Returns: Success / ErrorSecureEnv / ErrorDatabase / ErrorAuthorization

AddCredentialMetadata

```
int AddCredentialMetadata(int credentialId, string key, string value)
```

This function adds an arbitrary metadata in form of key-value pair to an existing credential. For example, a provisioning client could use this function to add a fixed input parameter or some other credential usage hitn to a credential that it has just created. The client application that uses this credential could query this credential metadata from the ObC database using `GetCredentialMetadata()` function.

Parameters:

`credentialId` - The credential that the additional metadata should be added to.

`key` - Identifier for the metadata.

`value` - The actual metadata value.

Returns: Success / ErrorDatabase

GetCredentialMetadata

```
int GetCredentialMetadata(int credentialId, string key, string value)
```

This function reads metadata assigned to a credential.

Parameters:

`credentialId` - Identifier of the credential.

`key` - Identifier of the metadata.

`value` - The actual metadata will be written to this output parameter.

Returns: Success / ErrorNotFound / ErrorDatabase

UseCredential

```
int UseCredential(int credentialId, string input, string output)
```

This function executes credential in the secure environment. The calling application may provide one input parameter and the function returns one output. Most credentials should be used with this function (credentials with advanced programs should be used with function `UseCredentialExtended()`).

First, CM performs a local access control check if one is needed by credential secret. Then CM executes the program with the credential secret, application provided input and possible CM-managed inputs as parameters. CM will return the program output to the calling application.

Parameters:

`credentialId` - The identifier of the credential

`input` - The input for program calculation, e.g. challenge in HTTP digest authentication protocol

`output` - The result of the program calculation, e.g. response in HTTP digest calculation, will be written into this output parameter

Returns: Success / ErrorNotFound / ErrorDatabase / ErrorPinQuery / ErrorSecureEnv

Capabilities: NetworkServices (user-grantable)

UseCredentialExtended

```
int UseCredentialExtended(int credentialId, string manifest,
                          string[] inputs, string[] outputs)
```

This function is an extended version of the basic `UseCredential()` function. This function should be used to execute credentials with advanced programs. This version of the function has an additional input parameter called `manifest`. The manifest can be used to override the default order in which the CM passes inputs to the program and it can also be used to override program attributes (which defines the input parateres that CM should pass to the program automatically). Additionally, this extended function accepts an arbitrary number of inputs and outputs.

Parameters:

`credentialId` - The identifier of the credential.

`manifest` - Defines which input parameters the CM should pass to the program and in which order.

`inputs` - An array containing inputs from the application to the program.

`outputs` - An array containing outputs from the program to the application.

Returns: Success / ErrorNotFound / ErrorDatabase / ErrorPinQuery / ErrorSecureEnv

GetCredentialNames

```
int GetCredentialNames(int programId, string[] credentialList)
```

This function returns names for all credentials that use the program given as input parameter. If the `programId` parameter is left out, the names of all credentials in the system are returned.

Parameters:

`programId` - The program that the credential uses.

`credentialList` - A list of credential identifiers. Contains credential identifiers for all credentials that uses the program given as input (or all credentials in the system).

Returns: Success / ErrorNotFound / ErrorDatabase

GetCredentialId

```
int GetCredentialId(string name)
```

This function returns identifier of the credential that has matching name.

Parameters:

`name` - The name of the credential.

Returns: Unique identifier for matching credential or in case of error: ErrorNotFound / ErrorDatabase

GetCredentialId

```
int GetCredentialId(int programId, string serviceId)
```

This function returns identifier of the credential that uses the specified program and a secret with the specified service identifier.

Parameters:

`programId` - The program that the credential uses.

`serviceId` - Service identifier assigned to the secret that the credential uses.

Returns: Unique identifier for matching credential or in case of error: ErrorNotFound / ErrorDatabase

DeleteCredential

```
int DeleteCredential(int credentialId)
```

This function deletes a credential from the Obc database.

Parameters:

`credentialId` - The identifier of the credential that should be deleted.

Returns: Success / ErrorNotFound / ErrorDatabase

B.4 Device public key and certificate

GetDevicePublicKey

```
int GetDevicePublicKey(string publicKey)
```

Returns device public key PK_D. In the current implementation we use RSA keys. The RSA key pair is generated when the CM is used for the first time, but this key pair could be also created at device manufacturing time. The private part (SK_D) is always kept secret.

Parameters:

`publicKey` - DER encoded PKCS format RSA key will be written into this output parameter.

Returns: Success / ErrorSecureEnv

Capabilities: ReadDeviceData (not user-grantable)

GetDeviceCertificates

```
int GetDeviceCertificates(string[] deviceCertificateList)
```

This function returns a list of certificates for device public key PK_D. A typical device has either one device certificates at all (the public key has been certified at device manufacturing time by the device manufacturer). Of course, it is possible that the device public key has been certified by more than one party.

`Parameters:`

`deviceCertificateList` - A list of device certificates (in X.509 format)

Returns: Success / ErrorSecureEnv

B.5 Local access control settings

SetAccessControlSettings

```
int SetAccessControl(string localPin, string personalText, string personalPicture)
```

This function should be called before any credential is used. This function defines the settings that are used for local access control checks. This function would be typically called by some kind of startup wizard or control panel application.

Parameters:

`localPIN` - The PIN code that is asked from the user when a local access control is required by a certain secret.

`personalText` - A personal text string. This string is shown as a part of the UI prompt when asking for a PIN code from the user. The user should enter the PIN to the prompt only when she sees her personal string as a part of the UI prompt.

`personalPicture` - A personal picture. Used the same way as `personalText`.

Returns: Success / ErrorSecureEnv / ErrorDatabase

B.6 API usage examples

In Section 6.5 we presented an example of creating and using a simple credential. Now, we show few more complicated examples.

Multi-program credential with metadata: Due to the limited resources in the secure execution environment, the programs have to be quite small in size (the maximum size is around one kilobyte in our primary target platform). Thus, it is likely that some more complicated algorithms will have to be split into several programs that will be executed in sequential order. This example shows how to create such a credential and how to use credential metadata.

First, a normal credential is created by adding a program and a secret and assigning them together. [6] Because the secret is added in plaintext format, authorization key used as authorization data when creating the credential. After that, another program is added and this program is added to the already created credential using the same authorization data. A serial number is attached to the credential using `AddCredentialMetadata()` function.

The application using the credential first finds it based on credential name. Then, the application reads the serial number metadata and executes the credential with the serial as input parameter.

```
// creating the credential
program1 = AddProgram(programData1, "first half of program logic");

secret = AddSecret(secret, "example secret name", authKey);

credential = CreateCredential(program1, secret, authKey);

program2 = AddProgram(programData2, "second half of program logic");

AddProgramToCredential(credential, program2, authKey, "example credential");

AddCredentialMetadata(credential, "serial", "12345678");

// finding and using the credential
credential = GetCredentialId("example credential");

GetCredentialMetadata(credential, "serial", serial);

output = UseCredential(credential, serial);
```

When `UseCredential()` is called, CM starts interpreter which executes the first program in the secure environment. The first program should pass out intermediary calculation result to CM in sealed format. CM then starts interpreter with the second program and passes the intermediary results to the second program as input. The second program finalizes the calculation and passes all the needed results to the application.

C Provisioning and local data sealing

C.1 Provisioning keys

As described Section 4, the family concept is based on family root key (RK) which can internally be diversified into a RK_S for secrets and RK_P for confidential programs. The symmetric 128-bit AES root key defines key hierarchy for provisioning packages, but more important is that the provisioning subsystem and interpreter interpret all provisioning information ultimately secured under either of these keys to belong to a family. Within a family version updates of programs can be implemented, and all data belonging to a family can be read by all subjects of that family.

[6] Notice that in both `AddProgram()` and `AddSecret()` function calls the optional parameters are left out and thus default values for those parameters are used.

In provisioning the root key RK is not used for anything as such, but it is used to derive confidentiality key CK and integrity key IK using a SHA-1 based HMAC function. A *provisioning identifier* (PID) is concatenated to RK in all key derivations. The provisioning identifier is reserved for possible future development and shall be currently set to zero. In key derivation HMAC function produces 20 bytes of which first 16 bytes are used. CK and IK are derived as follows:

```
CK = HMAC('Confident', RK|PID)
IK = HMAC('Integrity', RK|PID)
```

In the beginning of the provisioning process, the provisioning entity (typically a provisioning server) acquires the device public key PK_D of the target device. We assume that the provisioning entity can authenticate PK_D e.g. by receiving the public key from the target device itself accompanied with a device certificate issued by the device manufacturer or by retrieving the public key from a database maintained by the device manufacturer.

In our main implementation platform, Nokia mobile phones with M-shield secure environment, each device has a device-specific ElGamal key pair stored in the secure environment. The ElGamal public key consists of three values: a large prime number p (1024 bits), a small integer *alpha* (32 bits) and a large integer x. The first two values are system wide constants. In our environmetn $alpha = 2$ and p has the following randomly generated value:

```
0xa1 0xac 0xb6 0x8f 0xe6 0x4c 0xf0 0x2f 0x73 0xd5 0x4f 0xdb 0x1e 0xe8 0x01 0xc2
0x4e 0x2e 0xb4 0x85 0xb5 0x43 0x1e 0x98 0x69 0x4a 0x60 0x78 0x5e 0x87 0x35 0x43
0x98 0x09 0x79 0x73 0x8f 0x85 0xde 0x2e 0xaa 0x7f 0xe2 0xeb 0x0c 0xdd 0x72 0xe1
0x52 0x6d 0x06 0x5c 0x55 0x9c 0x08 0xe0 0x63 0xd7 0x28 0x19 0xea 0x66 0x86 0x85
0xc4 0x2c 0x06 0xad 0x0f 0x67 0x11 0x47 0x9c 0x35 0x46 0x98 0x50 0xac 0x67 0x75
0x97 0x4e 0x15 0x19 0x13 0xe0 0x66 0xf6 0xc3 0x3c 0xbd 0x6f 0x2e 0x75 0x07 0xad
0x60 0x09 0xd4 0x78 0x3c 0x73 0x41 0x44 0x75 0x63 0x39 0x71 0x5f 0xf2 0xd5 0x55
0xc0 0x93 0xbe 0x77 0x0b 0x71 0xcf 0x34 0x9e 0xd9 0xe1 0x8f 0xec 0xf0 0xd0 0xb7
```

Because p and *alpha* are system wide constants only x is needed to define an ElGamal public key. We use the following encoding for ElGamal public keys in our system: x is preceeded by a two-byte length value[7]. The length is encoded most significant byte first and the same byte order is used in all other encodings as well.

```
PK_ElGamal = len/2B | x/nB
```

Besides ElGamal device keys, which are natively supported by the secure environment of our primary target platform, we support RSA device keys. An RSA public key consists of modulus n and public exponent e. In our system $e = 2^{16} + 1$. The RSA public key is encoded as the 128 byte modulus n only.

```
PK_RSA = n/128B
```

C.2 Provisioning packages

Figure 10 shows the different provisioning messages in ObC architecture. ObCP/Init contains a family root key RK from the provisioning entity to the secure environment, ObCP/Xfer transfers encrypted data (either confidential ObC program or ObC secret) to a device under a given RK, and Obc/Endorse grants the right to access secret data under a specific family to a given Obc program (based on the hash of the bytecode of the program).

ObCP/Init package consists of the root key RK, a four-byte provisioning identifier and their SHA-1 (for integrity protection). The package is padded according to PKCS#1 v1.5 [11] for encryption: a block of 88 bytes starting with a value 0x02 and ending with 0x00, and containing 86 bytes of non-zero

[7] In our current implementations the maximum length is 134 bytes, although calculations are performed using 144 bytes big-ints

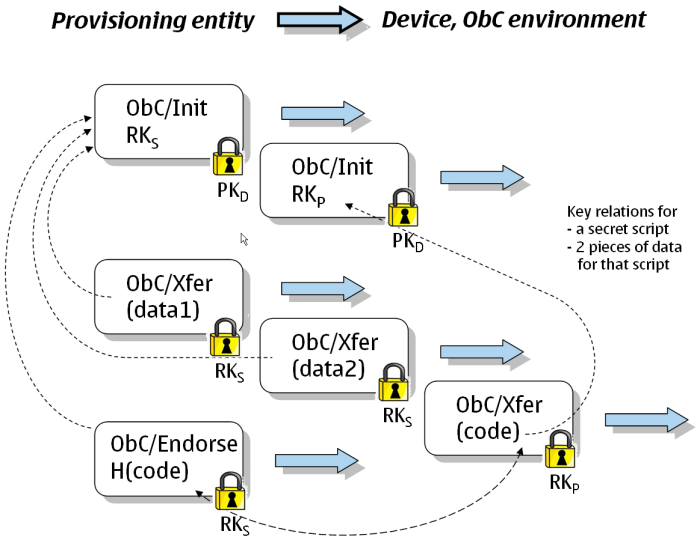

Figure 10: Provisioning packages

random data is prepended to the payload so that the total length of the package is 128 bytes. This data is protected by encrypting if for the target device using ElGamal or RSA encryption. The ElGamal-encrypted ObCP/init package consists of two big-ints gamma g and delta d. Both are prepended with two-byte length values. In RSA encryption the resulting ciphertext c is 128 bytes long byte string.

```
Init_data = 0x02 | rand/86B | 0x00 | RK/16B | PID/4B | SHA(RK|PID)/20B
ObCP/Init_ElGamal = g_len/2B | g/nB | d_len/2B | d/nB
ObCP/Init_RSA = c/128B
```

ObCP/Xfer package consists of a *tag*, the length of the payload, the payload itself (either confidential ObC program or ObC secret), a version number and a filler to make the data length multiple of 16 bytes. This data is AES-CBC-encrypted using confidentiality key CK. The AES initialization vector (IV) prepends the encrypted data and integrity of the secret is protected by calculating a SHA1-based HMAC using integrity protection key IK of the encrypted data and IV.

```
Xfer_data = tag/1B | payload_len/2B | payload/nB | version/2B | filler/xB
ObCP/Xfer = IV/16B | AES_CK(Xfer_data)/N*16B | MAC_IK(IV | AES_CK(Xfer_data))/20B
```

In the general case ObC programs are not deemed to be confidential, and their integrity is indirectly assured, since all their persistent critical data should be sealed and a modified ObC program is not able to handle data sealed to the original one without a new ObCP/Endorse. However, if the access control system provider wants to protect the confidentiality of the ObC program, it can be provisioned in an encrypted form, like any other secret. The one-byte tag defines the type of the payload. For ObC secrets we use 0x30 and for confidential ObC programs we use 0x21.

ObCP/endorse package is used to grant a program access to a secret. The package contains program hash encrypted and integrity protected using keys derived from the family root key used in the provisioning of the secret. Version defines the generation of family secrets that this program is authorized to access.

```
Endorse_data = SHA(ObC program)/20B | version/2B | filler/10B
ObCP/Endorse = IV/16B | AES_CK(Endorse_data)/32B | MAC_IK(IV|AES_CK(Endorse_data))/20B
```


C.3 Provisioning version control and naming

The version parameters in ObCP/Xfer and ObCP/Endorse are interpreted in a straigh-forward manner. ObC secrets are assigned a *minimal* version identifier that they can be applied to by the provisioning system. For ObCP/endorse messages, the version number indicates the *maximum* version that a given program can be given. In practice this means that a provisioning operation will be successful if version(ObCP/Xfer) $\leq$ version(ObCP/Endorse). Versioning can be used for credential lifecycle management, but it does not provide a complete solution for it. For example, the locally stored secrets are devoid of any provisioned version information.

As can be seen from the provisioning packages, there is no inclusion of naming or even referencing between packages that are to be used together. This is intentional - naming and management metadata is assumed to either be a part of the provisioning protocol, or be known to participating parties in some other way.

C.4 Provisioning example

A simple example may help to clarify the provisioning concepts. Let's assume that a provisioner P has a functionality that consists of three ObC programs that share data (e.g, the programs might be pipelined to achieve a wanted end result). Let us also assume that P wishes to transmit his programs in encrypted format, i.e., the algorithm in the program is assumed to be confidential.

The provisioner gets hold of a certificate containing the device public key PK_D.[8] 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.

For each of the three programs, an ObCP/Xfer by the root key RK_P is needed. These will encapsulate the encrypted bytecode for the respective programs. Likewise, for each of the three programs, an ObCP/Endorse by the root key RK_S is needed. 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.

There are three noteworthy issues: (1) In this example there are two families - one for confidential programs and another for secrets. At least the credential family endorsement should be common for any set of programs that need to share information without program-specific transfer code. (2) It is a managerial issue for the client executable to provision data correctly, i.e., with programs that share a family, any tuple (ObCP/init, ObCP/xfer, ObCP/endorse) where the parameters match would result in the secret data being sealed so that it is accessible to the endorsed program. If this is a security issue, the data itself (i.e., the last value of the array) can indicate which script is the right target (and the script in question can duly check this fact). (3) 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.

C.5 Local data sealing formats

The sensitive data (both secrets and confidential programs), is stored outside the secure environment in an encrypted and integrity protected format. 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 uses 128-bit AES encryption, and therefore the basic unit of data is a block of 16 bytes. The sealed data consists of a header of one block (16 bytes) and one or more data blocks. The (unencrypted) header consists of a length field (2 bytes), an integrity protection field (first 10 bytes of SHA-1 of the cleartext data) and a random field (4 bytes). If the length of the actual data is not a

[8] This is simply an x.509 certificate, and of no specific relevance to the protocol at hand, except that the device key can be confirmed to belong to a given device (and person).

multiple of 16 bytes, the end of the last data block is padded with unspecified values. The same format is used when sealing program specific secrets, family specific secrets and confidential programs - the difference is in the used encryption keys and their derivation.

```
SealedData = len/2B | SHA(data)/10B | rand/4B | AES_key(data | filler)/N*16B
```

Program and device specifig *local endorsement key* (LEK) is derived from OPK and program hash using SHA-1 based key derivation.[9] LEK is used as an AES (CBC mode) key with initilization vector (IV) set to all zeroes.

Secrets belonging to a family are provisioned in an encrypted form and on request sealed to all programs endorsed to handle secrets belonging to that family. The family specific sealing key, *local family key* (LFK), is derived from the root key for secrets (RK_S), the provisioning identifier (PID), the family version number and the OPK. [10] LFK is then encrypted for each program endorsed to handle secrets belonging to this family using LEK. No header is used (length is always 16 bytes) and IV is set to all zeroes. Only these encrypted family keys (LFK_P) will be stored outside the secure environment.

```
LEK = SHA1(OPK | SHA1(Program) | OPK) (first 16 bytes)
LFK = HMAC_SHA(RK_S | PID | version, OPK)  (first 16 bytes)
LFK_P = AES_LEK(LFK)
```

When a program handles family specific secrets, it needs as input the family key encrypted for it (LFK_P). The execution environment first derives LEK, uses it to decrypt LFK_P, and then stores that inside the secure environment for subsequent sealing and unsealing family secrets (IV is again zeroed). If a program does not get LFK_P as input, then it uses LEK for sealing and unsealing.

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. Again, the one block header contains the actual length of the program and its truncated SHA-1. 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  = SHA1(OPK|‘‘SecretCod’’|OPK)
```

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, the whole process when a confidential program handles sealed data belonging to a family the course of events is as follows:

1. A sealed program is detected - the device specific LPK is derived, and used to decrypt the program.
2. SHA-1 of the decrypted program is calculated and the program specific LEK is derived from it.
3. The family specific LFK is acquired by decrypting LFK_P using LEK. LEK (or LPK) are no more needed, but LFK is stored for further use, when unsealing and sealing family secrets during program execution.

We also have some, non-complete support for the following alternate configurations: In a resource starved situation it is possible to omit the header of the sealed data. On the other hand, if a very fine-grained compartmentalization is required (and secure environment resources are plentiful enough to tolerate the incurred code and data footprint), it is possible to implement an optional program specific sealing in parallel with family specific sealing. In this case the family specific LFK is used as previously, but the first 16 bytes of the SHA-1 of the program are used as IV, instead of the zeroed block used in every other case.

[9] This somewhat non-standard key derivation is used, bacause LEK derivation is needed in ObC interpreter, and using e.g. HMAC-SHA1 based key derivation would increase the interpreter size significantly since the HMAC implementation is not available in our primary target secure environment i.e. M-shield.

[10] This key derivation needs to be done during provisioning only, and in the provisioning system we assume to have proper HMAC implementation available, and thus we can used normal HMAC based key derivation.

C.6 Provisioning subsystem interface

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

Confidential program: Confidential ObC programs need to be provisioned in encrypted and integrity protected format. The provisioning subsystem converts the provisioned program into locally sealed data structure. The needed input data 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).[11].

Secret data: 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 data are: ObCP/Init (containing RK_S) and ObCP/endorse (containing the encrypted hash of the program). The provisioning subsystem produces the LFK_P (LFK encrypted using LEK). Each ObC program needs its own LFK_P in order to be able to access encrypted family data, 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 secret data belonging to the previous version (encrypted using the previous version LFK). The provisioning subsystem produces sealed secret data belonging to the new version (encrypted using the new LFK - new, 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 LFK_P.

D Program examples

D.1 Input and output handling

In ObC architecture Credentials Manager starts the interpreter which executes ObC programs in the secure environment. Programs may use different kind of input parameters and they produce one or more outputs. The program developers can use `#env_in()` macro for reading the inputs one after another. Program results are communicated out using `#env_out()` macro. The number of inputs (and sometimes even the order in which the inputs are provided) depends on the scenario.

Simple programs are used with `UseCredential()` function. In this case, the first input parameter that CM provides to the program is the credential secret. If the credential is expected to have more than one secrets, then the program developer should invoke `#env_in()` macro several times to read in all the secrets. The secrets are in sealed format and the application developer is expected to unseal the secret(s).[12] The input from the application is provided next. For simple programs there is only onen application input. CM-managed inputs are provided last. The default order in which these inputs are provided is serverPin, systemTime, sequenceNumber, serviceId.

[11] 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

[12] One of our future work intems is to add automatic unsealing to the interpreter so that program developers would not have to unseal secrets manually.

Below is an example of a simple program with that is always used with UseCredential function. Simple programs always use a single application input and simple programs are expected to know the number of secrets (in this example one). This particular example program also expects serverPin and systemTime inputs from the ObcManager.The program provides single output back to the client application.

```
-- Simple example program

-- read the sealed secret ss
#env_in(ss)

-- unseal the sealed secret ss, the result is secret s
#unseal(i, ss, s)

-- delete sealed secret ss to free memory
#delete(ss)

-- read the application input i
#env_in(i)

-- read the server PIN p
#env_in(p)

-- read the system time t
#env_in(t)

-- do the actual calculation using provided inputs (not shown here)

-- write the calculation output o back to the application
#env_out(o)
```

Advanced programs are used with the UseCredentialExtended function. Advanced program may handle multiple application inputs and provide multiple outputs back to the application. Additionally, for advanced program CM provides an extra input parameter called manifest which is always the first input. Manifest contains information about rest of the inputs. Basically, manifest is an array of input parameter indicators. The first element defines how many secrets the program should read (default value is one), the second element defines the number of application inputs (default is one), and the rest of the manifest contains identifiers that defines the number and order of inputs provided by CM.

If a program developer developer does not override program...

```
-- read manifest m and ignore it
#env_in(m)
#delete(m)

-- read in secret and unseal it
#env_in(ss)
#unseal(i, ss, s)

-- this program expects three application inputs
#env_in(i1)
#env_in(i2)
#env_in(i3)

-- actual calculation

-- this program produces two outputs
#env_out(o1)
#env_out(o2)
```

If the program does not know the number and type of all inputs before hand, it has to parse the manifest and adjust its input reading accordingly. Below is an example program that is sometimes used with

ServerPin and sometimes without serverPin. The example program first reads manifest. The first short of the manifest defines the number of secrets and the second defines the number of application inputs. If the manifest has a third short, then serverPin is provided by the ObcManager.

```
-- read manifest m
#env_in(m)

-- read secret sealed s
#env_in(s)

-- read application input i
#env_in(i)

-- get the length l of manifest m
l = #length(m)

-- if manifest length l is three the serverPin p is provided by ObcManager
if (l == 3)
    #env_in(p)

-- do the actual calculation

-- write back the calculation o
#env_out(o)
```

Below is another example of an advanced program. This program is capable of handling one to three secrets and application inputs. This example program also provides three outputs. It should be noted that this kind of more flexible input handling consumes a considerable portion of the available code space and thus using several application inputs and secrets should be used only when absolutely necessary.

```
-- Advanced example program

-- read the manifest m first
#env_in(m)

-- at least one sealed secret ss is always provide so that can be
-- read without any checks and unsealed
#env_in(ss)
#unseal(i, ss, s1)

-- for possible second and third secret we must check first short of manifest
if (m[0] > 1)
   #env_in(ss)
   #unseal(i, ss, s2)

if (m[0] > 2)
   #env_in(ss)
   #unseal(i, ss, s3)

-- at least one application input i1 is always provided so that can
-- be read without any checks
#env_in(i1)

-- for possible second and third application input we must
-- check second byte of manifest
if (m[1] > 1)
   #env_in(i2)

if (m[1] > 2)
   #env_in(i3)
```



```
-- do the actual calculation (not shown here)

-- write back the outputs o1, o2 and o3
#env_out(o1)
#env_out(o2)
#env_out(o3)
```

As a summary, here is the order in which the ObC programs can read input parameters:

- **manifest** - provided when credential used with `UseCredentialExtended()` function
- **secret(s)** - all the secrets assigned to the credential
- **application inputs** - the input parameters from the calling client application
- **CM-manages attributes** - the input parameters that are automatically inserted by the Credentials Manager

D.2 Password checking

In devices with integrated platform security (like Symbian 9.1 and forward) in the operating system, there is fairly good protection in place for process isolation and processes may run on different security levels depending on their task and interfacing needs. However, when the device is powered down, if no encryption is deployed on the file system, any process specific secrets can be extracted by simply reading and analyzing the filesystem in another device.

The following very simple credential program (abstractly) serves as an "/etc/passwd" file for some local service, where we now assume that when the device is active there is enough integrity in place for the caller to determine that the credential program response is authentic, and to address the confidentiality of the entered password all the way to the script. The heavily commented script takes a mode parameter (enter password / check password), whereby new passwords can be accepted (the encrypted version is given back to the caller) and a later entered password can be matched agaist the original. Note that the encrypted result can well be augmented to include some state information, additional meta-data or other related information.

```
-- ----------------------------------
-- Password storage and check. The password
-- should be less than 15 bytes long
--
-- (c) Nokia Research Center
--     orig. J-EE 2006
--     Mods by ARa 2007
--
-- Parameters:
-- ===========
-- i1: mode 0=set passwd / 1=normal op.
--
-- For mode 0:
-- i2: plaintext password
--     (one byte per word)
-- o1: a sealed response (encr. passwd)
--
-- For mode 1:
-- i2: sealed password
-- i3: plaintext password
--
-- o1: Result (1=Ok, 0=Nok)
```



```
--
-- ----------------------------------


-- ----------------------------------
-- Input mode (0=seal PW, 1=operation)
-- ----------------------------------
#env_in(m)

-- --------------------------------
-- Sealing operation
-- --------------------------------
if m == 0 then

  -- ------------------------------------
  -- Input data for sealing. Input
  -- is in byte format - one byte in
  -- one word
  -- ------------------------------------
  #env_in(ii)


  -- --------------------------------
  -- Seal and return
  -- --------------------------------
  #seal(n,ii,ww)

  #env_out(ww)
end

-- --------------------------------
-- Check password
-- --------------------------------
if m == 1 then
  -- ------------------------------------
  -- Common processing, now input the
  -- sealed password
  -- ------------------------------------
  #env_in(ii)
  #unseal(n,ii,kk)
  c = #length(kk)


  -- ------------------------------------
  -- Input data for checking. Input
  -- is in byte format - one byte in
  -- one word
  -- ------------------------------------
  #env_in(jj)
  a = #length(jj)

  -- Check password
  b = 0
  dd[0] = 1
  if a == c then
    while b < c do
      if kk[b] ~= jj[b] then
        dd[0] = 0
        b = c
      end
      b = b + 1
```

```
    end
  else
    dd[0] = 0
  end

  dd[1] = a
  dd[2] = c


  -- ------------------------------
  -- Return check value
  -- ------------------------------
  #env_out(dd)

end
```

D.3 Milenage 3G

As an example of a real-world algorithm that has been implemented for the ObC architecture, here is presented the core part of the Milenage/3G algorithm - the UMTS/3G authetication function. The code is Lua, and the native compiler is used after a pre-processing step with the MPP pre-processor. The code is commented, but note the *env_in(x)* functions for inputting data, *env_out(x)* for presenting the result, and the macro functions (resolving into external function calls) identifiable by the '#'-prefix. The 'unseal' and 'seal' operations unwrap (and re-wrap) encrypted data for this script, and e.g. *#aes_enc* is a typical example of the invocation of an external cryptographic function.

On a practical level, this function constitutes the security core of a UICC - the 3G SIM card. Thus, in principle, this code, suitably provisioned with keys and operator constants, could be used to authenticate to a mobile phone operator.

```
-- ------------------------------------
-- Milenage 3G security  kernel
-- (c) Nokia Research Center 2007
-- ------------------------------------

-- ------------------------------------
-- Input data (key) and unseal
-- Input is expected to be 8 shorts (16 bytes).
-- ------------------------------------
#env_in(ii)
#unseal(n,ii,kk)

-- ------------------------------------
-- Challenge input for Milenage kernel,
-- another 8 shorts.
-- ------------------------------------
#env_in(rn)

-- ------------------------------------
-- Operator Variant Algorithm Configuration
-- Field, another 8 shorts.
-- ------------------------------------
#env_in(ii)

-- ------------------------------------
-- Function number, scalar.
-- ------------------------------------
```



```
#env_in(fn)

-- ------------------------------
-- Run the Kernel itself
-- ------------------------------

i = 0

while i < 8 do
  rn[i] = rn[i] ^ ii[i]
  i = i + 1
end

#aes_enc(n, kk, rn, ww)

while 0 < i do
  i = i - 1
  ww[i] = ww[i] ^ ii[i]
end

if fn == 2 then
   rf = 0
   cf = 1
elseif fn == 3 then
   rf = 2
   cf = 2
elseif fn == 4 then
   rf = 4
   cf = 4
else
   rf = 6
   cf = 8
end

while i < 8 do
  wz[i] = ww[(i+rf)%8]
  i = i + 1
end

wz[7] = wz[7] ^ cf

#aes_enc(n, kk, wz, ww)

while 0 < i do
  i = i - 1
  wz[i] = ww[i] ^ ii[i]
end

-- ------------------------------
-- Return f(n)
-- ------------------------------
#env_out(wz)

end
```

E Debugger Example

In this section we show two example fragments of an interactive session with the emulator applied on the following piece of code:

```
-- ------------------------------------------------------------
-- Simple, debugged program, testing while loop and arithmetic
-- ------------------------------------------------------------

#env_in(b)
a = #length(b)

jj = 0
while jj < a do
  c[jj] = b[jj] + 121  -- just a magic value added
  jj = jj + 1
end

#env_out(c)
```

We see that for each invocation of a byte code, the consumed stack and tuple space are displayed in terms of percentages, as are the program-counter and consumed code-steps. The current byte-code, along with any resolved parameter names are displayed.

```
%luacore.exe fe10.lc arrayinput.tlv arrayoutput.tlv s
+------------------------------------------------------------+
| OnBoard Program emulator  v. 0.93, 24.10.07                +
| (c) Nokia Research Center 2007.         Jan-Erik Ekberg    +
+------------------------------------------------------------+
-----------------------------------------------
PC:   0 C:     0 S:  0% T:  1%(14 01 00 ) PUSHGLOBAL env_in
((PUSH a GLOBAL variable ( or function ptr) on the stack))
Before:>h
b [value]: Continue, and break at specific PC value
c [value]: Continue until steps=[value]
c:         Continue until the end
t:         Show all global tuples (arrays)
t [value]: Show specific global tuple array
s:         Display stack
y:         Display symbols (symbol mappings)
i:         H(i)de line-by-line output
Any other key + return single steps
Before:>y
SYMBOLS:
001: env_in
002: b
003: a
004: length
005: jj
006: c
007: env_out
Before:>t
Used 3/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
length          =           7      7
env_out         =           6      6
Before:>s
```



```
Used 0/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
Before:>
After :>
-----------------------------------------------
PC:   3 C:     2 S: 16% T:  1%(14 02 00 ) PUSHGLOBAL b
((PUSH a GLOBAL variable ( or function ptr) on the stack))
Before:>
After :>s
Used 4/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
    0      1    1          5    5
    1      2    2          0    0
After :>
-----------------------------------------------
PC:   6 C:     4 S: 33% T:  1%(40 01 00 ) CALLFUNC (1:0)
((CALLFUNC: call a function))
Before:>t
Used 3/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
length          =           7      7
env_out         =           6      6
Before:>s
Used 4/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
    0      1    1          5    5
    1      2    2          0    0
Before:>
After :>t
Used 6/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
length          =           7      7
env_out         =           6      6
b           ( 0)=           1      1
b           ( 1)=           2      2
b           ( 2)=           3      3
After :>s
Used 0/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
After :>
-----------------------------------------------
PC:   9 C:     6 S:  0% T:  1%(14 04 00 ) PUSHGLOBAL length
((PUSH a GLOBAL variable ( or function ptr) on the stack))
Before:>

... move to the middle of the program, where b[1] gets
    an increment of 121

-----------------------------------------------
PC:  43 C:    64 S: 33% T:  1%(14 02 00 ) PUSHGLOBAL b
((PUSH a GLOBAL variable ( or function ptr) on the stack))
Before:>
After :>
-----------------------------------------------
PC:  46 C:    66 S: 50% T:  1%(14 05 00 ) PUSHGLOBAL jj
((PUSH a GLOBAL variable ( or function ptr) on the stack))
Before:>
After :>
-----------------------------------------------
```



```
PC:  49 C:    68 S: 66% T:  1%(15 ) PUSHINDEXED
((PUSHINDEXED: Index and variable on stack. Replace by value.))
Before:>
After :>
-----------------------------------------------
PC:  50 C:    70 S: 50% T:  1%(06 79 00 ) PUSHUINT 121
((PUSH a 16-bit word on the stack))
Before:>
After :>
-----------------------------------------------
PC:  53 C:    72 S: 66% T:  1%(30 ) ADDOP
((Add two topmost elements of stack, answer on stack))
Before:>t
Used 9/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
a               =           3      3
length          =           7      7
jj              =           1      1
env_out         =           6      6
b           ( 0)=           1      1
b           ( 1)=           2      2
b           ( 2)=           3      3
c           ( 0)=         122     7A
Before:>s
Used 8/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
    0     70   46        122   7A
    1      5    5          1    1
    2      0    0          2    2
    3      0    0        121   79
Before:>
After :>t
Used 9/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
a               =           3      3
length          =           7      7
jj              =           1      1
env_out         =           6      6
b           ( 0)=           1      1
b           ( 1)=           2      2
b           ( 2)=           3      3
c           ( 0)=         122     7A
After :>s
Used 6/12 stack elements
STACK: (ID dec/hex)   (VAL dec/hex)
    0     70   46        122   7A
    1      5    5          1    1
    2      0    0        123   7B
After :>
-----------------------------------------------
PC:  54 C:    74 S: 50% T:  1%(23 ) STOREINDEXED0
((STOREINDEXED0: Value, index and variable on stack. Store val in var))
Before:>
After :>t
Used 10/75 locations
TUPLE:    INDEX:        (DEC)  (HEX)
env_in          =           5      5
a               =           3      3
length          =           7      7
```



```
jj              =           1      1
env_out         =           6      6
b           ( 0)=           1      1
b           ( 1)=           2      2
b           ( 2)=           3      3
c           ( 0)=         122     7A
c           ( 1)=         123     7B
After :>

...
```