Scheduling Execution of Credentials in Constrained Secure Environments

Jan-Erik Ekberg, N. Asokan, Kari Kostiainen
Nokia Research Center, Helsinki, Finland
{jan-erik.ekberg, n.asokan, kari.ti.kostiainen}@nokia.com

Aarne Rantala
Technical Research Center, Espoo, Finland
aarne.rantala@vtt.fi

ABSTRACT

A new inexpensive approach for using credentials in a secure manner has become available due to the fact that several types of general-purpose secure hardware, like TPMs, M-shield and ARM TrustZone are becoming widely deployed. These technologies still have limitations, one being the limited on-chip secure memory which leads to severe size constraints for credentials that need to execute in secure memories. In this paper, we describe, in the context of a credential provisioning and execution architecture we call On-board Credentials (ObC), a secure scheduling mechanism for overcoming some of the size constraints imposed for the virtual credentials implemented on ObC.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.0 [**Software Engineering**]: General—*Protection Mechanisms*

General Terms

Security

Keywords

On-board Credentials, secure execution environments, trusted hardware, scheduling

1. INTRODUCTION

Cryptographic protocols used in distributed applications and protocols make use of *credentials*, which consist 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. Several types of general-purpose secure hardware have been incorporated into end user devices and are starting to be widely deployed. These include Trusted Platform Modules (TPM) [10] and Mobile Trusted Modules [4] specified by the Trusted Computing Group [9] and other proprietary platforms like M-Shield [8, 7] and ARM TrustZone [1]. All these platforms enable, to different degrees, a strongly isolated secure environment, consisting of secure storage, and in some cases supporting secure execution where processing and memory are isolated from the rest of the system. TPMs are already available on many high-end personal computers. Several mid-range and high-end Nokia phones utilize hardware security features based on M-Shield.

In our earlier work [3, 2], we described how such general-purpose secure hardware can be used to build an inexpensive platform for credentials, which we called On-board Credentials (ObC). The ObC system allows credential logic, in the form of "credential programs" (also known as "ObC programs") to be run inside a secure execution environment. Run-time isolation between credential programs was enforced simply by loading a program in its entirety and running it until completion or until aborted by the ObC system. Since secure execution environments typically have severe limitations on the amount of on-chip memory available, credential program size was restricted to around 1kB.

In this paper we describe a virtual secure execution environment that frees the programmer of the memory constraints described above. True hardware isolation of memory as well as execution are essential for credentials. In this context, the implementation size of secure virtual memory in the traditional sense consumes too much memory to be viable for the problem at hand.

The paper is structured as follows. In section 2, we briefly describe the general ObC architecture and relevant aspects of the current implementation. In section 3, we motivate the need for piece-wise execution and present the design for scheduling credential program execution. Then we describe credential program installation for our system in section 4. We proceed by describing the secured data structures needed for scheduling in section 5. In section 6 we provide an informal security analysis of the system. Sections 7 and 8 provide overviews of program APIs and system management aspects, and in the last sections we wrap up with a some initial performance estimates and conclusions.

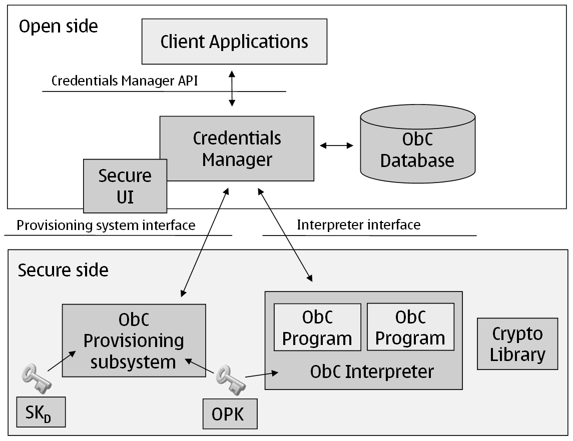

Figure 1: ObC architecture

2. OBC ARCHITECTURE

Figure 1 shows a high-level overview of the initial design of the ObC architecture which is described in more detail in [3, 2]. We describe the components while stating our main architectural design decisions. In the following text we use the term "secure side" to denote the secure, isolated environment where the credentials are run. The rest of the device, including the normal operating system of the device, is denoted "open side" to indicate that its security is not integral to our architecture.

ObC interpreter: A credential consists of *credential secrets* such as keys, and an algorithm that operates on these secrets known as a *credential program*. Isolating credential programs from the secure side resources is one of the prime objectives of the ObC architecture. It is achieved by providing a virtualized environment where the programs can be run. We aim at a very limited RAM usage, and this rules out the use of a general-purpose virtualized execution environment because they cannot be made to fit into the space available. However, a simple bytecode interpreter can be made to run even within these constraints. Thus we have chosen to use a bytecode interpreter as the primary component of the ObC architecture.

Data sealing: In our initial design, only one credential program is allowed to execute on the ObC interpreter at any given time. Therefore, the primary issue in isolating credential programs from one another is with respect to their ability to access persistently stored data. The ObC interpreter has exclusive access to a device specific master key called the *ObC platform key* (OPK). The interpreter provides a sealing/unsealing function for an ObC program using a key derived from OPK and the program text itself. The programs can use sealing to protect secret data to be stored persistently in the open side.

Credentials Manager: Client applications use ObCs via an OS-level component we call the Credentials Manager (CM). The CM has a simple "secure user interface" which the user can recognize by customizing its appearance. It also manages the ObC Database where sealed credential secrets and credential programs can be stored persistently. Only the Credentials Manager is allowed to communicate with the ObC interpreter.

Open Provisioning: The provisioning of credential programs to the ObC platform in an open manner without the need for any third party approval is described in a related paper [2].

Credential families: In order to keep the ObC system open, we need to allow anyone to be able to write credential programs or provision credential secrets to a device without needing certification or approval from any third party. Instead, we allow an issuer of credentials to create a "family" which consists of a set of credential secrets and a set of credential programs that are allowed access to those secrets. A program is authorized to be a member of a family by having the issuer create an endorsement of the program binding it to a family.

2.1 Initial implementation

The initial design of the ObC interpreter and its bytecode is a compromise between functionality and size while accounting for the needed isolation between credential programs and the underlying platform. The detailed design choices for the interpreter are described elsewhere [3, 2]; this section summarizes its properties.

In terms of program flow the interpreter contains a fairly extensive set of (conditional) jumps that can be used to construct loops and possibly even subroutine constructs. An extensive set of arithmetic and logical operations are available directly as bytecodes, operating on a data stack internal to the interpreter. Credential program data is represented as objects rather than as memory locations, and copying bytecodes are available for moving data between objects and the stack. The unsigned 16-bit short integer is the only supported atomic data type. Vectors of short integers are also supported, and bytecodes for moving vector elements to/from the stack are available. A minimal set of library functions are included as bytecode commands. We have a compilation tool chain for compiling (constrained) Lua-language [6] source code to our bytecode, and our experience indicates that the interpreter appropriately fulfills its primary intent, i.e. to combine cryptographic primitives available on the platform into credential algorithms.

The isolation of credential programs from the secure side resources is implemented in the interpreter by carefully asserting that the program counter stays within the credential program memory space and that the operand stack neither over- nor underflows. No bytecodes mix program flow attributes with data, and data access is only done by reference. The data object space is of fixed sized and internally managed, whereas the object identifiers are local to the credential program.

Currently, the architecture has been implemented on Nokia N95 handsets with the M-shield security architecture. For credential programs, the size constraints imposed on us by M-shield on contemporary processors limits the program size to around 1kB with approximately 200 bytes of data memory to use, especially if we intend to use the architecture on contemporary devices. This is appropriate for many simple credentials, say the HTTP Digest algorithm [5]. Still, if the credential state is complex, or if the credential operation e.g. involves some security-relevant marshaling, the available space quickly becomes too small.

3. DESIGN FOR PIECE-WISE EXECUTION

To enlarge the available code space for the programmer, we need to split the program and execute it in pieces. Given the need to implement support for secure paging, small extensions to that solution will enable us to support other types of piece-wise execution as well. We have identified the following different needs for executing a credential program in pieces, rather than to run it to completion as in our initial design.

1. **On demand bytecode paging**: If the credential program size is larger that the size that fits in secure memory, the program can be partitioned into *slices*, and a return to the open side issued whenever the program counter moves outside of the current slice – i.e. the open side can then schedule the execution at the correct next slice (with the same overall state).

2. **The scheduling of program execution in time**. As we assume complete isolation of the secure side, overall system scheduling (interrupt handling) may be adversely affected by the ObC. Thus, to support real-time requirements there may be a need to return control to the open side after an allotted time, to serve any interrupts or other critical services, and then come back to continue the operation on the secure side where the execution was left off.

3. **The scheduling of input/output**. The amount of I/O volume is a limiting factor in many secure environments; in some it is critical. E.g. for services that operate as filters on large amounts of data, there is a probable need to return to the open side to arrange for I/O buffers to be purged, for more input to be received by the credential program or to flush the output buffers for the production of more output data. This feature also makes it possible from the secure side to select/request specific input data selectively, based on the flow of the credential program.

4. **External native functions implementation**: Commonly useful credential program functionality (like the implementation of common cryptographic algorithms) may be provided as native binary implementations for the secure side. The size limitations of the secure side imply that such external implementations may need to be executed separately from the interpreter itself. Hence, invocation of such external functions also calls for piece-wise execution where the change of control from the interpreter to the external function and back is mediated through the open side.

5. **External bytecode functions support**: Relying on on-demand paging for big programs with subfunction capability has two drawbacks. First, the state, e.g. the reservation of parameter space must be maintained over the whole program. Second, the code slice size is determined by the target device during installation. If the installation system is unaware of the semantics of the credential program being sliced, the border between the slices may occur at a code position that is either inefficient (e.g. in the middle of an innermost loop), and/or at a position where the code flow (which now is partially visible to the open side) reveals security relevant information of the flow or deployed secret parameters. These problems may be alleviated by providing the developer a possibility to define high-level subroutines, written in bytecode, and bound to the same provisioning concepts (endorsements to families) as credential programs at large.

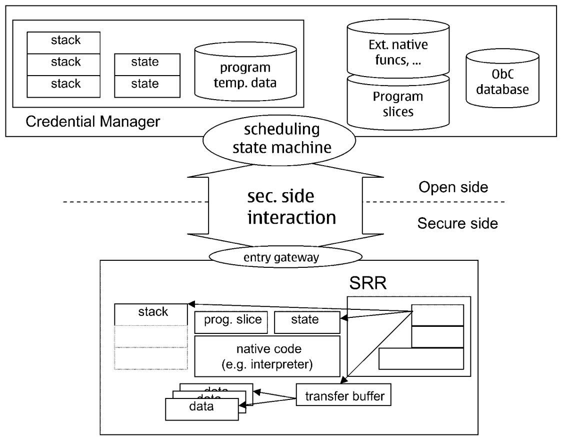

Figure 2: Scheduling of credential executions

All of these different needs call for an architecture to artificially enlarge the available space by executing the program piece-wise in terms of code, data and even time. In this paper, we present such an architecture for securely **scheduling** the execution of credential programs.

The local impact of scheduling piece-wise execution is significant. When a provisioned program is split into slices, we must guarantee that the execution over the sliced program produces the same result as the original one. The confidentiality, integrity and especially statefulness of any data temporarily stored on the open side as part of the scheduling must also be guaranteed. Because of secure side limitations, the support for these assertions must be implementable in an efficient manner in terms of size while simultaneously being acceptable in terms of performance. We support all of the cases identified above while providing security for the overall credential program execution that is equivalent or close to what is provided by the initial architecture.

In contrast to e.g. Lua virtual machines our interpreter has no built-in support for subroutines, local variables or object orientation. As mentioned, we expose platform services as well as external implementations of commonly used functionality as "external functions". The interpreter itself implements a very limited function interface as part of the instruction set. This interface provides a few support routines (object deletion, resolving array length), ways to perform I/O with the open side and to invoke a given external function. The manner in which this activity is managed and set up is the core contribution of this paper.

The overall execution architecture for our new system is depicted in fig 2. The native code run within the secure side is one of several provisioning functions (see [2]), the interpreter or a native platform service. These are run in an interleaved manner, and the integrity of these components is managed in a platform-specific manner, e.g. by code signing. For the overall execution of a credential program, a sequence of invocations of the interpreter is typically interleaved with the execution of native functions. This overall execution context will be managed from the open side (the operating system) by a *scheduling state machine* described in section 8. The secure side will contain a single persistent record set, the *Session Root Record* (SRR) that asserts the overall program execution state at any given time (further details of the data components and their binding to SSR is given in section 5). All data and program elements are stored in the open side in a sealed form, and provided to the secure side as needed. Prior to execution on the secure side, the credential program and its parameter set are always validated by the entry gateway on the secure side.

3.1 Sealing

A sealing/unsealing facility is the main security service providing integrity and confidentiality protection for the locally stored persistent data and installed credential programs to be executed within the secure side. All sealing is done according to the same template - a 128-bit AES-EAX authenticated encryption with a 16-byte header, a 16-byte nonce and a 16 byte MIC. The header in the seals is used to provide integrity-protected type information for the processing by the open-side scheduling state machine. Fixing the length of header and nonce reduces the secure-side implementation size of the AES-EAX without sacrificing compatibility with full-fledged implementations. This seal is the mechanism by which all parameter seals, program slices and tokens are constructed. Additionally it is also used with stack elements, transfer buffers and state information.

3.2 Keying

An overview of the key hierarchy used for sealing is depicted in figure 3, and in table 1. A key-derivation function KDF is used to construct many of the keys. All keys are 128-bit AES keys. The OPK is a device-specific secret key, and used as a base for all keys that are device-specific, i.e. the *Local Endorsement Key* (LEK) and the *Local Slice Token Key* (LSTK) that is used as an intermediary key to decrypt program slices. The key labeled RK is the root key for a credential family as given in a provisioning message (see [3, 2]), and is used to diversify further keys that are to be family-specific. Examples include the *Local Family key* (LFK) that on the device is common to a specific credential family, and the *Global Communication key* (GCK) that may be used by programs to seal information to external parties in the same program family. A number of randomly generated keys serve special purposes like the *Local Control-stack Key* (LCK) that is bound to the device by being stored inside the secure side for the duration of a single credential program execution or program slicing activity, and the *Capability Sealing Key* (CSK) that provides a similar trust root for the duration of a subroutine call.

4. SLICING THE CREDENTIAL PROGRAMS FOR ON-DEMAND PAGING

The commonality between the different scheduling needs will become clearer when the data elements and state have been considered. However, for the secure on-demand bytecode paging to be successful we need a preprocessing step, where the provisioned credential program is first locally installed and then endorsed by a family. In this section we will describe this activity. In this context the credential program can be either a main program or an external bytecode function.

We intentionally use the terms slicing and program slice for partitioning the credential program into pieces. Currently the slicing is simply based on size and determined by the target device. However, we recognize the benefit of cutting the programs into pieces at the most advantageous places in terms of program flow. This is important for performance but also for security, since minimizing information leakage available to code flow analysis is a clear benefit. Thus we use the term slicing, and intend to improve on this aspect in future work.

4.1 Program installation

We have designed the process of locally installing the provisioned credential program or bytecode function as an iterative one-pass activity over the entirety of a credential program. The overall program digest is calculated during this activity in which on every iteration one program slice - a platform specific slice seal (SS) - will be produced (fig 4). A program slice is cryptographically bound to the program position it represents, i.e. when at the end of the installation activity the credential program hash is determined, the individual slices are mapped to that hash as individually integrity-protected containers with a fixed position in the program context. In conjunction with every slice, a slice-splitting token (SST) containing the hash over the slice seal is produced for the endorsement process. Both seals are done with the local sealing key (LSK) - a randomly generated key kept inside the secure side for the duration of the single local installation activity, and thereafter permanently stored in an endorsement enabling token (EET) for the credential program in question (see figure 4).

Several other optimizations to this basic setup are also done. First, as we support credential programs delivered to the platform in an encrypted manner, all locally installed credential programs will be not only integrity-protected, but also encrypted. Applying this “complete sealing” causes a performance penalty for public credential programs, but simplifies interpreter logic. We argue that this is a sensible trade-off because applications of ObCs are not likely to be bandwidth-intensive or too sensitive to response times; but memory within the secure side is an extremely critical resource. Secondly, the encryption used when provisioning for the platform constitutes an integrity-protected AES CBC-MAC. The piecewise decryption of this encryption will require state-keeping, so rather than maintaining all of the needed state inside the secure side, another token, the slice-splitting-state (SSS), will be used to externalize the state between secure-side invocations needed during the program installation activity. The statefulness of the SSS is guaranteed by the same mechanisms that are used to maintain transfer buffer integrity during credential program execution (section 5.2).

At the end of the credential program installation the program hash has been calculated, and the possible integrity check of a provisioned encrypted program can also be validated at that time. If the MIC matches or if the input program was unprotected, the overall program hash is allowed to determine the formation of a platform-specific, local endorsement key (LEK) by which the key used to seal the SSTs is written to an endorsement enabling token (EET). On failure to install, no EET is produced.

Key	Name	Description of use	Derived from	Kept in
OPK	ObC Platform Key	Device-specific secret	random	Persistent (sec. side)
LEK	Local Endorsement Key	Local proof of endorsement	OPK, program hash	generated on-demand
LSTK	Local Slice Token Key	protection of slice context	OPK, h(slice)	generated on-demand
RK	Root Key	Provisioning	see [3, 2]	used during provisioning
LFK	Local Family Key	Family-specific sealing key	OPK, RK	SET
GCK	Global Communication Key	Secure communication	provisioning secret, code hash	SET
LCK	Local Control-Stack Key	Secure scheduling	random	RAM (sec. side)
LSK	Local Sealing Key	Sealing of slices	random	SET
CSK	Capability Sealing Key	Protection of CapSeals	random	transfer buffer

Table 1: System keys

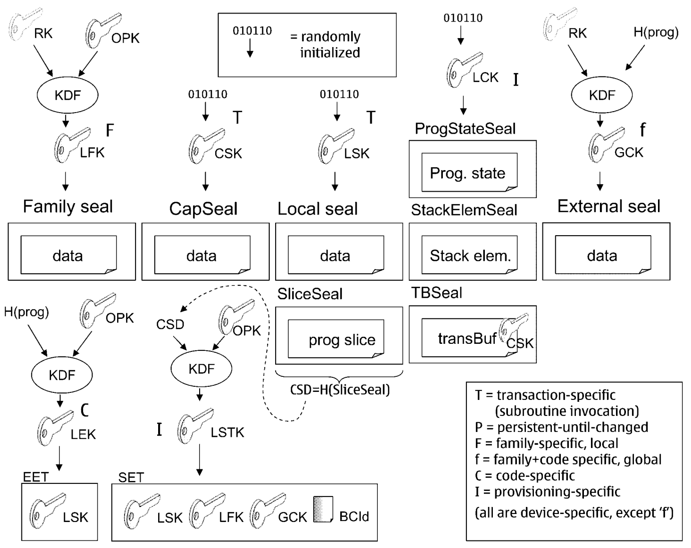

Figure 3: Key hierarchy for sealing

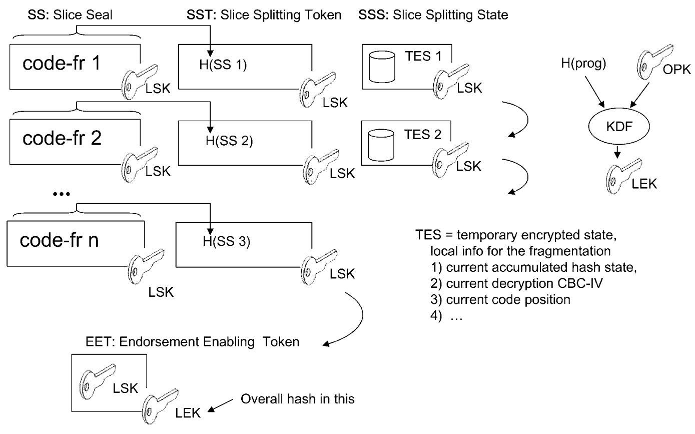
SS: Slice Seal
SST: Slice Splitting Token
SSS: Slice Splitting State
code-fr 1
LSK
H(SS 1)
LSK
TES 1
LSK
code-fr 2
LSK
H(SS 2)
LSK
TES 2
LSK
...
code-fr n
LSK
H(SS 3)
LSK
H(prog)
OPK
KDF
LEK
TES = temporary encrypted state, local info for the fragmentation
1) current accumulated hash state,
2) current decryption CBC-IV
3) current code position
4) ...
EET: Endorsement Enabling Token
LSK
LEK
Overall hash in this

Figure 4: Program slicing during installation

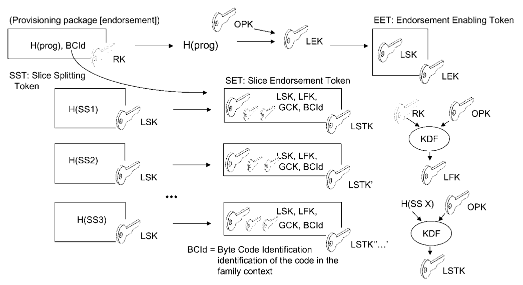
(Provisioning package [endorsement])
H(prog), BCId
RK
H(prog)
OPK
LEK
EET: Endorsement Enabling Token
LSK
LEK
SST: Slice Splitting Token
SET: Slice Endorsement Token
H(SS1)
LSK
LSK, LFK, GCK, BCId
LSTK
RK
OPK
KDF
H(SS2)
LSK
LSK, LFK, GCK, BCId
LSTK'
LFK
H(SS3)
LSK
LSK, LFK, GCK, BCId
LSTK''…'
H(SS X)
OPK
KDF
LSTK
BCId = Byte Code Identification identification of the code in the family context

Figure 5: Program slice endorsement

4.2 Endorsement

Our initial ObC architecture [2] includes the concept of program endorsement, whereby secrets and other context is bound to a credential program by keys determined by the credential issuer. For the scheduling enabled architecture, we upgrade the provisioning data to include a function handle - the ByteCode Identifier (BCId) - to refer to the credential program or bytecode function in the context of the provisioned program family.

Earlier, we recognized the use case for unendorsed credential programs. With the design enhancement described in this paper, we will require the endorsement of all programs to be executed on the interpreter. The reason for this is found in the realization that any program invoking an external bytecode function needs the endorsement to map the identity of the called function. Thus, only credential programs invoking nothing but external native functions could be exempt of endorsement, but we see that as a small enough program subclass that we rather emphasize interpretation simplicity in the matter. Note that an endorsement for a public credential program can be completely local to the device, and thus any added complexity can and is hidden behind CM interfaces.

In principle the endorsement activity could be done by locally sealing any family specific secrets (and bytecode identifiers) by any imaginary local seal that includes the credential program digest e.g. as seal contents. In such a case the interpretation (of a program slice) would consist of opening such a "family secrets" seal, and based on the program hash, further opening and validating the EET, SST and finally the SS. A natural optimization is to combine the family secrets with the contents of the EET and the SST into a slice- and family-specific token containing all necessary information needed by the program slice. The slice endorsement token (SET) is exactly that, and the endorsement process is structured as another one-pass provisioning activity performing the abovementioned transformation. The key protecting the SET is simply a derivation of platform-specific key with the encrypted slice digest as the diversifier. In SET we also include the GCK for credential program communication back to the issuer.

Also note that the local installation is separate from the endorsement activity. The set of SS's, slice-specific SST's and an EET can be independently endorsed by anybody with knowledge of the credential program hash. The biggest data elements - the SSs - need to be locally stored as a single representation only, independently of the number of endorsements done for the credential program.

5. DATA STRUCTURES FOR SCHEDULING

The scheduling activity fundamentally impacts credential program data and its protection. Where earlier all data was maintained inside the secure side until the execution completed and the results were exported, now the state must be intermittently stored in insecure memory. The inclusion of external bytecode routines requires special handling for intermediary I/O parameters. The protection against slice re-ordering attacks demands tight state-control. The end result seems complex, but we believe that the result is a balance between uniformity between scheduling activities (one solution fits all cases, keeping implementations small) and usability, i.e. the programmers perspective.

Let us further elaborate on the data elements in figure 2. As was earlier mentioned, the 28-byte session root record (SRR) containing a counter, a reference metric and a key is the only volatile memory construct that is maintained inside the secure side between scheduling invocations.

The interpreter **state** contains a snapshot of the interpreter dynamic data including all local variables, the operand stack, the program counter etc. Note that the interpreter proper does not include a call stack. We introduce this concept as an add-on "on top" of the interpreter. Every unique credential program will have its own state, i.e. external bytecode functions will have a state that is different from that of their calling code.

An abstract memory construct - the **call stack** - will also be provided by the scheduling architecture. Although administered by the open-side scheduling state machine, a reference to the topmost (i.e. current) stack element is always kept in the SRR, making the combination the primary guarantee for the statefulness of the execution. A scheduled secure side invocation may require the inclusion of up to two (topmost) stack elements, and may also produce up to two new ones.

Trust for subroutine parameters and I/O in general is handled by an entity denoted the **transfer buffer**. The transfer buffer is always directly related to the topmost stack element, and as such need to be remembered in the open side only until the next secure side invocation. The integrity-protected, plaintext header of the transfer buffer seal is also the main signaling channel between the secure side and the open side regarding what parameters are needed by the secure side at a given invocation.

The secure side code comes on two abstraction layers. The whole provisioning system, the interpreter code itself as well as external native functions are written directly for the underlying platform, and their integrity is guaranteed e.g. by code signing. For all interpreted code, i.e. slices of credential programs, irrespectively of whether they are parts of a main program or an external bytecode function, the integrity is provided through the mechanism of local installation and endorsement, as described in section 4. The integrity-check is performed at the entry gateway.

5.1 The call stack

5.1.1 *Stack internals*

The intention of including the call stack as a specific architectural security component is to provide statefulness between interpreter and external function invocations, but also with other scheduling events. We will first examine the stack construct itself, and then proceed to describe its use in ObC.

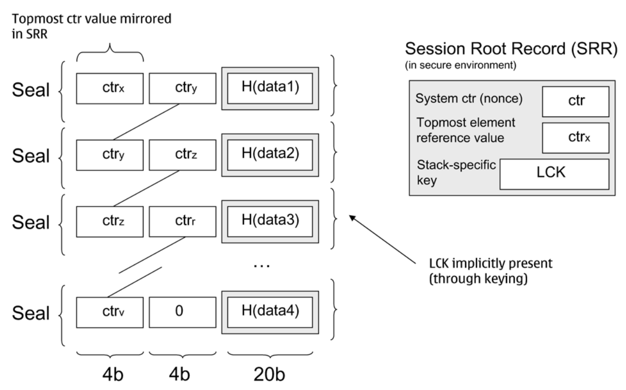

Figure 6: The call stack

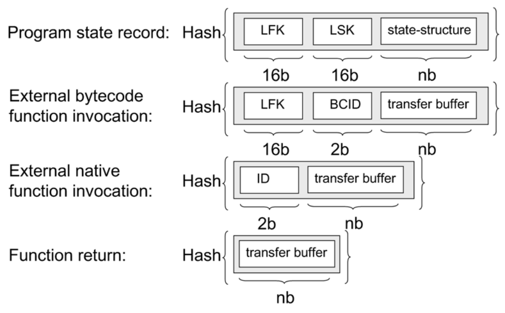

Figure 7: Data structures used in the stack

The stack is based on SRR which is a persistent, volatile record in the secure side (see fig 6). The record is initialized on every new main program invocation. The initialization consists of a new Local Call Stack Key (LCK) being constructed, the counter and reference values being cleared. This operation invalidates all old data records bound to the stack, i.e. everything except the family-, platform specific and global seals.

The stack elements (see figure 6), contain a fixed 20-byte data part (used as storage for a hash value), and two counter values. The counters in each stack element contain the unique counter value for the element itself and a "link", i.e. the unique counter value of the element one level below in the stack, or 0, if this element is the bottom-most element. Elements are sealed for the platform using the LCK.

When a stack element is given to the secure side, the match between the element's unique counter and the reference value inside the SRR is validated. If the stack is popped, the reference value is replaced by the removed element's "link" value. If the stack is pushed, a new element is generated with the old secure side reference value constituting the "link" value, and the new element's unique counter value, taken from the updated SRR counter, will be put as the SRR's reference value.

The reliance on the seal for the stack element protection was chosen for implementation simplicity, but it also protects the confidentiality of the stack data. The linkage between elements could equally well have been arranged e.g. by chains of cryptographic hashes, but the current option has an element state-keeping overhead of only 8 bytes - smaller than an acceptable size of a digest as statistical uniqueness is needed.

The ObC system will use four different types of stack records, shown in figure 7, to handle scheduling. The record contents are hashed into the 20-byte space allocated for data in the stack element. The stack as a construct will guarantee protection against re-ordering or removal of elements, and the records themselves concatenate whatever information is needed to define the current scheduled context. The stack records are:

1. **Program state record:** The hash will contain LFK and LSK, both retrievable from the SET, to bind the invocation to a family and through the endorsement to a specific code identity. The stack record will lock the state and thus also the program counter. The program state record is used when an external bytecode function or native function returns control to the caller and for all types of credential program self-scheduling (on-demand paging, I/O and execution partitioning).

2. **External function invocation:** As we want logical separation between the caller entity and the function, only the function identifier, as indicated by the caller, is identified along with the family. Integrity-protecting the transfer buffer makes it possible to transfer parameters to the subroutine in a stateful fashion. Bytecode invocations include the LFK for the family tie-in. The BCId identifier is used for bytecode functions, whereas for native functions the identifier is globally known rather than provisioned.

3. **Function return:** Whenever a function returns data, or a credential program schedules itself, a return record will contain a record integrity-protecting the returned transfer buffer. There is no binding to a specific family or routine id - these parameters are only validated on invocation, and indirectly conveyed in the return through the existence of the stack, i.e. the fact that the second topmost element is a program state record.

The operation of the call stack is best explained in the view of the three different scheduling needs: 1) When a self-scheduling return is triggered from a credential program, the interpreter will produce two records on the stack - a program state record, and on top a function return record. 2) When an external function (bytecode or native) is called from a credential program, a function invocation record is put on top of the program state record. 3) The external function will end by cleaning its invocation record and replace it with a function return return record. It is trivial to see that this invocation mechanism is logically recursive, i.e. an external function may further call other external functions without endangering the statefulness of the stack.

On any invocation, the stack is the part that glues together the overall secure side entry validation in the entry gateway (see fig. 2). For a bytecote invocation, the entry gateway begins by unsealing SET and SS and depending on the invocation type also a possible transfer buffer and program state record. At that stage the interpreter should have access to all data needed for the stack record, the data of which is hashed to the 20-byte data part of the stack element. The invocation type exclusively determines the

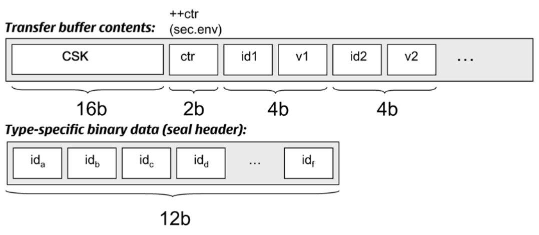

Figure 8: Transfer buffer

amount and order of stack elements, which one at a time are unsealed, structurally checked against the SRR, and their data validated against the unsealed data reference. On success, the stack element is popped, on error the execution terminates. For credential programs, the execution is only allowed to continue if the stack checks match AND the code position given in the state matches the position information in the SS header. For function returns and self-scheduling, we rely on the uniqueness of LFK to identify the program to be run.

5.2 The transfer buffer

A system for state-enforcing parameter transfer to and from external functions has been included in the architecture rather than to put the implementation burden on the credential programs themselves. The setup includes a specific seal type for the purpose, as well as the transfer buffer construct. The parameter setup is designed to handle deep invocations, i.e. that external functions call other such functions.

The transfer buffer is a variable-length buffer that is initialized at main program invocation (at the same time the program state is initialized), and which contains a randomly generated *capability sealing key* (CSK). The intention is that the life-time of the transfer buffer spans one or a few function invocations from the main credential program. The transfer buffer will gradually increase in size, and therefore resetting it (from the main credential program) on occasion between function invocations makes good sense.

Shown in figure 8, the transfer buffer internals consists of the above-mentioned key, and a list of parameter identifiers and version ordinals of all seals done in the transfer buffer context. The same information is part of the integrity-protected seal headers - thus the fact that a seal unseals with the CSK and that the parameter id and version number in the seal matches with the respective entry in the transfer buffer provides the state guarantee for the parameter in question. As the transfer buffer itself is sealed and bound to the call stack, the freshness guarantee is bound all the way to SRR.

5.2.1 Function-specific capability handles

In some scenarios, there is also a need to provide a more persistent capability for a sealed data element. The main example is external native functions that produce (sealed) output parameters that cannot be imported into the interpreter due to size constraints. An RSA key generation, signing and decrypting external native function is a good example of such a service - the keys them self are too long to comfortably fit into interpreter local variables.

In these cases, we expect the native function to return a specific secret handle, sealed using CSK and version control, for the generated data to the caller. The referenced data is in turn sealed together with the secret handle, e.g. for the native function and the device. During later invocations the secret handle is used as the access control mechanism for further use of the referenced data. We will see in section 8 that the parameter naming combined with the I/O flow control scheme nicely supports this notion of function-specific capabilities.

6. SECURITY ANALYSIS

From a security perspective, the analysis of the interleaved ObC system can be examined from the following angles:

1. **Isolation**: The isolation of persistent data between credential families is maintained also in the augmented solution - the family endorsement is extended to include also bytecode functions. The isolation between the secure side and the credential programs remains unchanged from the initial architecture [2].

2. **Confidentiality, integrity**: The whole security solution is founded on sealing using diversifications of the OPK, the RK and a few randomly generated keys. The binding to the call stack and the 28 bytes of secure-side memory that constitutes the SRR provides rollback protection. The AES-EAX seal is a well-known construct for authenticated encryption. The rollback and statefulness of the solution is built around easy-to-verify primitives such as counters for the stack construct and parameter versions, the LSK and code position parameters for binding program slices to each other and the use of the program hash as the ultimate identifier for the credential programs. The LFK used for defining the endorsement to a family is no different from the one in the initial design.

3. **Replay**: The call stack and the transfer buffer together guarantee the replay protection for the code flow. Since each new main program invocation starts with a fresh LCK and CSK, there is no relation between main program invocations. The parameters protected by local, family and global seals have no built-in replay / version protection and if needed, mechanisms like the one presented in section 5.2.1 could be deployed e.g. with version control primitives mirrored against a trusted server.

4. **Information leakage**: The initial ObC design isolates the complete execution and the related data as a matter of definition. The scheduling solution will leak more information about the execution, e.g. because of the on-demand paging - analysis of the trace of slices being scheduled may in the worst case reveal secrets, especially if the attacker can select the input. As a counterbalance, including support for external bytecode functions provides one tool for the programmer to structure his program to force the scheduling decision to happen in places with the least possible risk of algorithm state. We also claim that the programmer is not significantly worse off than before - small and simple “old-style” credentials will still be scheduled as a single invocation - the extension mainly gives

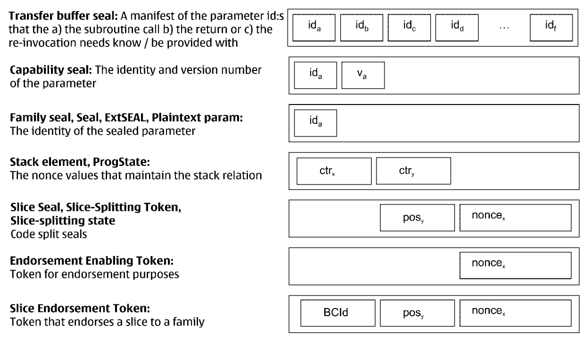
Transfer buffer seal: A manifest of the parameter id:s that the a) the subroutine call b) the return or c) the re-invocation needs know / be provided with
id_a id_b id_c id_d … id_f
Capability seal: The identity and version number of the parameter
id_a v_a
Family seal, Seal, ExtSEAL, Plaintext param: The identity of the sealed parameter
id_a
Stack element, ProgState: The nonce values that maintain the stack relation
ctr_x ctr_y
Slice Seal, Slice-Splitting Token, Slice-splitting state
Code split seals
pos_y $nonce_x$
Endorsement Enabling Token: Token for endorsement purposes
$nonce_x$
Slice Endorsement Token: Token that endorses a slice to a family
BCId pos_y $nonce_x$

Figure 9: Data in seal header

the programmer more power but unfortunately comes with additional responsibility and a requirement for a deeper insight.

Some keys, like the OPK, will in this architecture be used quite extensively (e.g. as the key seed for each and every LSTK). An additional level of key indirection could be used to address this issue.

7. CREDENTIAL PROGRAM INTERFACE

The programmer of the credential program cannot give hints for the on-demand code paging, i.e. he can only affect the splitting of the credential program by dividing his code into external byte-code functions. For the isolation of data, the selection of seals (or no seal at all) for any imported or exported parameter is determined by the credential program. A seal may 1) be local, i.e. specific to the program or subroutine and device, 2) family-specific, i.e. tied to the family and device or 3) global, i.e. tied to the family and program. Additionally, subroutine parameters and their return values preferably should be "capability-sealed", i.e. bound to the transfer-buffer secrets. I/O functions are used to export or import local variables / arrays of a given seal type with a specific parameter identity. Error handling of I/O is not visible to credential programs - an error immediately leads to the interpreter returning an error. Additionally the credential program has the interface to execute an external function (by its id), and to re-initialize the transfer buffer state.

The external function input parameters are defined by a parameter manifest, i.e. an array of parameter identities. From the credential program perspective, the parameter identifiers are collected in an array and written to the "environment" by exporting the array to a fixed, well-known parameter id prior to executing the external function. On return, the same well-known id can be imported into an array, from which the identifiers of the return parameters can be determined. We assume that most external functions actually get an array of their return parameter identities as a specific input parameter, in which case the calling side typically knows the return parameter identities up front.

8. MANAGING THE SCHEDULING

The scheduling of a program is driven by the return values from the secure side and by information present in the plaintext token headers (fig 9). The return value consists of a type, and optionally one or two 16-bit parameters. The parameter values contain an error code or the BCId and program position of the next program invocation. The types include the error return code and the ready return value, which tells the Credentials Manager (CM) either that the execution of a main program successfully terminated or that a subroutine is done and we should continue execution where the calling entity left off - in accordance with the contents of the stack. Additional return types indicate external function invocation and the re-loading of the same program slice and state. The latter one is encountered when the interpreter is scheduling in time or, if a parameter request is present in the transfer buffer seal header, that a parameter was missing and that the CM should provide that parameter at the next invocation. The final return type indicates with parameters that the on-demand code paging needs another program slice to continue.

Slice seal headers are labeled with a program-specific nonce in addition to the position parameter of the slice. The same nonce is present in the SET header, which also contains the BCId. Thus, the CM can always select the right SET and SS for execution based on the return parameter.

Data parameters are identified by 16-bit parameter IDs visible both in the credential programs when sealing or unsealing, as well as in the seal headers, and in the transfer buffer header. The parameter names are not segmented based on seal type, i.e. it is up to the credential program and external functions to produce and read seals correctly based on parameter name - the CM will not consider seal types as part of parameter management.

Inside the interpreter, the parameter Id 0x0000 is reserved for credential program R/W access to the transfer buffer manifest. Orthogonal to this, external functions typically (but not necessarily) determine their input parameters based on the order of parameters in the transfer buffer manifest, rather than based on parameter naming.

The operation of the CM provides intermediary storage of the call stack and related state records. Those can be indexed and "stacked" based on the counter values in the header of the stack element and state record seals. Additionally, for the duration the execution of a main credential program, the CM maintains a big storage set, indexed by parameter id, containing the set of the latest versions (last seen) of all parameters. The CM handles the parameter exchange based on the header part of the transfer buffer seal. The header in question contains the parameter manifest for the next invocation irrespectively of invocation type.

9. CONCEPTUAL EXAMPLE

This section illustrates one case of split execution in terms of processing seals, tokens and decisions made by the interpreter or related software from the secure-side perspective:
Calling an external native function: The credential program prepares its output by the initializing the transfer buffer, and capability-sealing parameters to be sent to the function. The parameter names can quite freely selected by the program, but eventually their respective IDs are set into the transfer buffer manifest in the order the function expects them. Some sealed output parameter may also as its value indicate the parameter id(s) the local function should use for its return value(s). Finally the script calls the native function with a specific byte code command (indicating the

BCId of the function). The interpreter terminates with a request to load a subfunction, specifying the required function and producing two stack records, the sealed state and the transfer buffer.

The BCId identifies the called (native) function, but it does not define how it is implemented. Therefore, CM must know how to map native function code to their BCIds. The CM also reads the id:s of required parameters from the manifest - i.e. the transfer buffer header - and tries to find those that are not already supplied by the calling script in its return parameters. The input buffer contents is constructed according to the manifest, and finally the called function is invoked with the topmost stack element and transfer buffer returned by the previous secure side invocation. The native function must on entry check the stack element and transfer buffer, read and unseal its input.

If input parameters could be validated, the native function does its thing and produces its outputs, capability-sealing them with id:s plausibly given to it by the caller as input parameters. The global function returns with a "function ended" return code, and on top of the output seals also produces the function return stack element and a new transfer buffer, e.g. listing the capability-sealed return parameters. Upon this, the CM invokes interpreter again, giving it SS, SET, the two topmost stack elements, the state and the transfer buffer. The input is constructed according to the manifest provided by the subroutine at return.

10. PERFORMANCE

As the solution heavily uses sealing as part of the security for scheduling, a relevant concern is the performance impact of the system, as the use of the sealing primitive quickly adds up. E.g, starting a smallish program slice of 128 bytes that takes one sealed input consumes 36 AES block operations. As a rough estimate, one tested library runs an AES block operation on a 300MHz ARM core in 0.1ms with some additional overhead for key scheduling. Thus, the initialization overhead induced by the solution is in the range of milliseconds on a Nokia N95 handset. More complex scheduling, e.g. calling an external function, will bring a much higher penalty.

A back of the envelope calculation of invoking a small external bytecode function that takes two input parameters and returns one sets us back 258 AES block calculations for the complete call sequence - a time overhead of tens of milliseconds. Accounting for the fact that the main argument for the ObC platform is to be able to execute credentials securely, we argue that the overhead is acceptable, but see a potential danger with increasingly complex credentials actually causing user-noticeable delay. The delay issue is directly addressable by a applying a computationally lighter seal or hardware acceleration, thus it does not impact the architecture as a whole.

11. CONCLUSIONS AND FUTURE WORK

This paper describes how a minimal scheduling solution for a size-constrained secure environment can be implemented based a sealing primitive by using a few carefully chosen simple algorithmic structures - the call stack and the transfer buffer.

We intend to implement and test the complete scheduling architecture on a legacy mobile device with a secure element to derive more accurate performance results. Additionally we have identified the issue of slicing the credential programs in the most advantageous places as the primary topic for further consideration. This is a topic with clear security relevance, as it is instrumental for the programmer and credential issuer to be able to have some security guarantees regarding information leakage from the credential algorithms run in the secure environment.

12. REFERENCES

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

[2] Jan-Erik Ekberg et al. On-board Credentials with Open Provisioning, draft paper 2008. Technical Report NRC-TR-2008-007, Nokia Research Center, August 2008. `http://research.nokia.com/files/NRCTR2008007.pdf`

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

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

[5] John Franks et al. HTTP Authentication: Basic and Digest Access Authentication. Technical Report RFC 2617, IETF, June 1999.

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

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

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

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

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