Cryptographic Hardware and Embedded Systems (CHES 2004)
(11 – 13 august 2004, Boston, Massachusetts, USA)
M. Joye and J.-J. Quisquater Eds. Springer-Verlag, LNCS 3156, pages 441–454.

How to Disembed a Program?

(Extended Abstract*)

Benoît Chevallier-Mames[1], David Naccache[1], Pascal Paillier[1], and David Pointcheval[2]

[1] Gemplus Card International/Applied Research and Security Center
{benoit.chevallier-mames,david.naccache,pascal.paillier}@gemplus.com
[2] CNRS/ENS – Dépt Informatique – david.pointcheval@ens.fr

Abstract This paper presents the theoretical blueprint of a new secure token called the *Externalized Microprocessor* (XμP). Unlike a smart-card, the XμP contains no ROM at all.
While exporting all the device's executable code to potentially untrustworthy terminals poses formidable security problems, the advantages of ROM-less secure tokens are numerous: chip masking time disappears, bug patching becomes a mere *terminal update* and hence does not imply any roll-out of cards in the field. Most importantly, code size ceases to be a limiting factor. This is particularly significant given the steady increase in on-board software complexity.
After describing the machine's instruction-set we introduce a public-key oriented architecture design which relies on a new RSA screening scheme and features a relatively low communication overhead. We propose two protocols that execute and dynamically authenticate arbitrary programs, provide a strong security model for these protocols and prove their security under appropriate complexity assumptions.

Keywords. Embedded cryptography, RSA screening schemes, ROM-less smart cards, Program authentication, Compilation theory, Provable security, Mobile code.

1 Introduction

The idea of inserting a chip into a plastic card is as old as public-key cryptography. The first patents are now 25 years old but mass applications emerged only a decade ago because of limitations in the storage and processing capacities of circuit technology. More recently new silicon geometries and cryptographic processing refinements led the industry to new generations of cards and more complex applications such as multi-applicative cards [7].

Over the last decade, there has been an increasing demand for more and more complex smart-cards from national administrations, telephone operators and banks. Complexity grew to the point where current cards are nothing but miniature computers embarking a linker, a loader, a Java virtual machine, remote method invocation modules, a bytecode verifier, an applet firewall, a garbage collector, cryptographic libraries, a complex protocol stack plus numerous other clumsy OS components.

This paper ambitions to propose a disruptive secure-token model that tames this complexity explosion in a flexible and secure manner. From a theoretical standpoint, we look back to von Neumann's computing model wherein a processing unit operates on volatile and nonvolatile memories, generates random numbers, exchanges data via a communication tape and receives instructions from a program memory. We revisit this model by alleviating the integrity assumption on the executed program, explicitly allowing malevolent and arbitrary modifications of its contents. Assuming a cryptographic key is stored in nonvolatile memory, the property we achieve is that no *chosen-program* attack can actually infer information on this key or modify its value: only authentic programs, the ones written by the genuine issuer of the architecture, may do so.

Quite customizable and generic in several ways, our execution protocols are directly applicable to the context of a ROM-less smart card (called the Externalized Microprocessor or XμP) interacting with a powerful terminal (Externalized Terminal or XT). The XμP executes and dynamically authenticates external programs of *arbitrary size* without intricate code-caching mechanisms. This approach not only simplifies current smart-card-based applications but also presents immense advantages over state-of-the-art technologies on the security marketplace. Notable features of the XμP are further discussed

* The full version of this work can be found at [6].

in Section 7 and in the full version of this work [6]. We start by introducing the architecture and programming language of the XμP in the next section. After describing our execution protocols in Sections 4 and 5, Section 6 establishes a well-defined adversarial model and assesses their security under the RSA assumption and the collision-intractability of a hash function.

2 The XμP's Architecture and Instruction Set

XJVML. An executable program is modeled as a sequence of instructions $P = (\mathsf{INS}_1, \ldots, \mathsf{INS}_\ell)$ where INS_i is located at address i for $i \in 1, \cdots, \ell$ off-board. These instructions are in essence similar to instruction codes executed by any traditional microprocessor. Although the XμP's instruction set could be similar to that of a 68HC05, MIPS32 or a `MIX` processor [10], we choose to model it as a JVML0-like machine [13], extending this language into XJVML as follows. XJVML is a basic virtual processor operating on a volatile memory RAM, a non-volatile memory NVM, classical I/O ports denoted IO (for data) and XIO (for instructions), an internal random number generator denoted RNG and an operand stack ST, in which we distinguish

- **transfer instructions:** `load` x pushes the current value of RAM$[x]$ (*i.e.* the memory cell at immediate address x in RAM) onto the operand stack. `store` x pops the top value off the operand stack and stores it at address x in RAM. Similarly, `load` IO captures the value presented at the I/O port and pushes it onto the operand stack whereas `store` IO pops the top value off the operand stack and sends it to the external world. `load` RNG generates a random number and pushes it onto the operand stack (the instruction `store` RNG does not exist). `getstatic` pushes NVM$[x]$ onto the operand stack and `putstatic` x pops the top value off the operand stack and stores it into the nonvolatile memory at address x;
- **arithmetic and logical operations:** `inc` increments the value on the top of the operand stack. `pop` pops the top of the operand stack. `push0` pushes the integer zero onto the operand stack. `xor` pops the two topmost values of the operand stack, exclusive-ors them and pushes the result onto the operand stack. `dec`'s effect on the topmost stack element is the exact opposite of `inc`. `mul` pops the two topmost values off the operand stack, multiplies them and pushes the result (two values representing the result's MSB and LSB parts) onto the operand stack;
- **control flow instructions:** letting $1 \leq L \leq \ell$ be an instruction's index, `goto` L is a simple jump to program address L. Instruction `if` L pops the top value off the operand stack and either falls through when that value is the integer zero or jumps to L otherwise. The `halt` instruction halts execution.

Note that no program memory appears in our architecture: instructions are simply sent to the microprocessor which executes them in real time. To this end, a program counter i is maintained by the XμP: i is set to 1 upon reset and is updated by instructions themselves. Most of them simply increment $i \leftarrow i + 1$ but control flow instructions may set i to arbitrary values in the range $[1, \ell]$. To request instruction INS_i, the XμP simply sends i to the XT and receives INS_i via the specifically dedicated communication port XIO.

SECURITY-CRITICAL INSTRUCTIONS. While executing instructions, the device may be fed with misbehaving code crafted so as to read-out secrets from the NVM or even update the NVM at wish (for instance, illegally credit the balance of an e-Purse). It follows that the execution of instructions that have an irreversible effect on the device's NVM or on the external world must be authenticated in some way so as to validate their genuineness. For this reason we single-out the very few machine instructions that send signals out of the XμP[1] and those instructions that modify the state of the XμP's non-volatile memory[2]. These instructions will be called *security-critical* in the following sections and are defined as follows.

[1] Typically the instruction allowing a data I/O port to toggle.
[2] Typically the latching of the control bit that triggers EEPROM/Flash update or erasure.

Definition 1. *A microprocessor instruction is security-critical if it might trigger the emission of an electrical signal to the external world or if it causes a modification of the microprocessor's internal nonvolatile memory. We denote by $\mathcal{S}$ the set of security-critical instructions.*

As we now see, posing $\mathcal{S} = \{\texttt{putstatic}\ x,\ \ \texttt{store}\ \text{IO}\}$ is not enough. Indeed, there exist subtle attacks that exploit i as a side channel. Consider the example below where k denotes the NVM address of a secret key byte $u = \text{NVM}[k]$:

$$P = (\texttt{getstatic}\ k,\ \texttt{if}\ 1000,\ \texttt{dec},\ \texttt{if}\ 1001,\ \texttt{dec},\ \texttt{if}\ 1002,\ \ldots)\ .$$

The XμP will require from the XT a continuous sequence of instructions

$$\mathsf{INS}_1, \mathsf{INS}_2, \ldots, \mathsf{INS}_{u-1}, \mathsf{INS}_u$$

followed by a sudden request of INS_{1000+u} and the value of $u = \text{NVM}[k]$ has hence leaked-out.

Let us precisely formalize the problem: a microprocessor instruction is called *leaky* if it might cause a physically observable variable (*e.g.* the program counter) to take one of several possible values, depending on the data (RAM, NVM or ST element) handled by the instruction. The opposite notion is the one of *data indistinguishability* that characterizes those instructions for which the processed data have no influence whatsoever on environmental variables. Executing a `xor`, typically, does not reveal information (about the two topmost stack elements) which could be monitored from the outside of the XμP. As the execution of leaky instructions may reveal information about internal program variables, they fall under the definition of security-criticality and we therefore include them in $\mathcal{S}$. Following our instruction set, we have $\mathcal{S} = \{\texttt{putstatic}\ x,\ \texttt{store}\ \text{IO},\ \texttt{if}\ L\}$.

3 Ensuring Program Authenticity

Verification per Instruction. To ascertain that the instructions executed by the device are indeed those crafted by the code's author, a naive approach consists in associating a signature to each instruction *e.g.* with RSA[3]. The program's author generates a public and private RSA signature key-pair (N, e, d) and embeds (N, e) into the XμP. The code is enhanced with signatures $P = ((\mathsf{INS}_1, \sigma_1), \ldots, (\mathsf{INS}_\ell, \sigma_\ell))$ where $\sigma_i = \mu(\mathsf{ID}, i, \mathsf{INS}_i)^d \bmod N$, μ denotes a deterministic RSA padding function[4] and ID is a unique program identifier.

Note that the instruction address i appears in the padding function to avoid interchanging instructions in a program. The role of ID is to guard against code mixture attacks in which the i-th instructions of *two* programs are interchanged. The XμP keeps the ID of all authorized programs in nonvolatile memory. We consider the straightforward protocol shown on Figure 1.

```
0.     The X$\mu$P receives and checks ID and initializes $i \leftarrow 1$
1.     The X$\mu$P queries from the XT instruction number $i$
2.     The XT sends $(\mathsf{INS}_i, \sigma_i)$ to the X$\mu$P
3.     The X$\mu$P
   (a)      ascertains that $\sigma_i^e = \mu(\mathsf{ID}, i, \mathsf{INS}_i) \bmod N$
   (b)      executes $\mathsf{INS}_i$
4.     Goto step 1.
```

Fig. 1. The Authenticated XμP **(inefficient)**

This protocol is quite inefficient because, although verifying RSA signatures can be relatively easy with the help of a cryptocoprocessor, verifying one RSA signature per instruction remains resource-consuming.

[3] Any other signature scheme featuring high-speed verification could be used here.
[4] Note that if a message-recovery enabling padding is used, the storage of P can be reduced.

RSA-Based Screening Schemes. We resort to the *screening* technique devised by Bellare, Garay and Rabin in [4]. Unlike verification, screening ascertains that a batch of messages has been signed instead of checking that each and every signature in the batch is individually correct. More technically, the RSA-screening algorithm proposed in [4] works as follows. Given a list of message-signature pairs $\{m_i, \sigma_i = h(m_i)^d \bmod N\}$, one screens this list by simply checking that

$$\left(\prod_{i=1}^{t} \sigma_i\right)^e = \prod_{i=1}^{t} h(m_i) \bmod N \quad \text{and} \quad i \neq j \Leftrightarrow m_i \neq m_j \; .$$

At a first glance, this primitive seems to perfectly suit our code externalization problem where one does not necessarily need to ascertain that all the signatures are individually correct, but rather control that all the code ($\{\mathsf{INS}_i, \sigma_i\}$) seen by the $\mathsf{X}\mu\mathsf{P}$ has indeed been signed by the program's author at some point in time.

Unfortunately the restriction $i \neq j \Leftrightarrow m_i \neq m_j$ has a very important drawback as loops are extremely frequent in executable code (in other words, the $\mathsf{X}\mu\mathsf{P}$ may repeatedly require the same $\{\mathsf{INS}_i, \sigma_i\}$ while executing a given program)[5]. To overcome this limitation, we introduce a new screening variant where, instead of checking that each message appears only once in the list, the screener controls that the number of elements in the list is strictly smaller than e (we assume throughout the paper that e is a prime number) *i.e.* :

$$\left(\prod_{i=1}^{t} \sigma_i\right)^e = \prod_{i=1}^{t} \mu(m_i) \bmod N \quad \text{and} \quad t < e \; .$$

This screening scheme is referred to as μ-RSA. The security of μ-RSA for $\mu = h$ where h is a full domain hash function, is guaranteed in the random oracle model [5] by the following theorem.

Theorem 2. *Let (N, e) be an RSA public key where e is a prime number. If a forger $\mathcal{F}$ can produce a list of $t < e$ messages $(m_1, \ldots, m_t)$ and $0 \leq \sigma < N$ such that $\sigma^e = \prod_{i=1}^{t} h(m_i) \bmod N$ while the signature of at least one of $m_1, \ldots, m_t$ is not given to $\mathcal{F}$, then $\mathcal{F}$ can be used to efficiently extract e-th roots modulo N.*

The theorem applies in both passive and active settings: in the former case, $\mathcal{F}$ is given the list $\{m_1, \ldots, m_t\}$ as well as the signature of some of them. In the latter, $\mathcal{F}$ is allowed to query a signing oracle and may choose the value of the m_is. We refer the reader to [6, Appendix A.1] for a proof of Theorem 2 and detailed security reductions.

Opaque Screening. Signature screening is now used to verify instructions collectively as depicted on Figure 3. At any point in time, ν is an accumulated product of $t < e$ padded instructions $\nu = \prod_i \mu(\mathsf{ID}, i, \mathsf{INS}_i)$. Loosely speaking, both parties $\mathsf{X}\mu\mathsf{P}$ and XT update their own security buffers ν and σ which compatibility (in the sense of $\sigma^e = \nu \bmod N$) is checked before executing any security-critical instruction. Note that a verification is also triggered when exactly $e - 1$ instructions are aggregated in ν.

[5] Historically, [4] proposed only the criterion $(\prod \sigma_i)^e = \prod \mu(m_i) \bmod N$. This version was broken by Coron and Naccache in [9]. Bellare *et al.* subsequently repaired the scheme but the fix introduced the restriction that any message can appear at most once in the list.

```
0.    The X$\mu$P receives and checks ID and initializes $i \leftarrow 1$
1.    The X$\mu$P
  (a)     sets $t \leftarrow 1$
  (b)     sets $\nu \leftarrow 1$
2.    The XT sets $\sigma \leftarrow 1$
3.    The X$\mu$P queries from the XT instruction number $i$
4.    The XT
  (a)     updates $\sigma \leftarrow \sigma \times \sigma_i \bmod N$
  (b)     sends $\mathsf{INS}_i$ to the X$\mu$P
5.    The X$\mu$P updates $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \mathsf{INS}_i) \bmod N$
6.    If $t = e$ or $\mathsf{INS}_i \in \mathcal{S}$ the X$\mu$P
  (a)     queries from the XT the current value of $\sigma$
  (b)     halts execution if $\sigma^e \neq \nu \bmod N$ (cheating XT)
  (c)     executes $\mathsf{INS}_i$
  (d)     goto step 1
7.    The X$\mu$P
  (a)     executes $\mathsf{INS}_i$
  (b)     increments $t \leftarrow t + 1$
  (c)     goto step 3.
```

Fig. 3. The Opaque XμP (secure but suboptimal)

As one can easily imagine, this protocol becomes rapidly inefficient when instructions of $\mathcal{S}$ are frequently used. For instance, `if`s constitute the basic ingredient of `while` and `for` assertions which are extremely common in executable code. Moreover, in many cases, `while`s and `for`s are even nested or interwoven. It follows that the Opaque XμP would incessantly trigger the relatively expensive[6] verification stage of steps 6a and 6b (we denote by CheckOut this verification stage throughout the rest of the paper). This is clearly an overkill: in many cases `if`s can be safely performed on non secret data dependent[7] variables (for instance the variable that counts 16 rounds during a DES computation). We show in the next section how to optimize the number of CheckOuts while keeping the protocol secure.

4 Internal Security Policies

We now associate a *privacy bit* to each memory and stack cells, denoting by $\varphi(\textsc{ram}[j])$, $\varphi(\textsc{nvm}[j])$ and $\varphi(\textsc{st}[j])$ the privacy bit associated to $\textsc{ram}[j]$, $\textsc{nvm}[j]$ and $\textsc{st}[j]$. NVM privacy bits are nonvolatile. Informally speaking, the idea behind privacy bit is to prevent the external world from probing secret data handled by the XμP. RAM privacy bits are initialized to zero upon reset, NVM privacy bits are set to zero or one by the XμP's issuer at the production or personalization stage, $\varphi(\text{IO})$ and $\varphi(\text{RNG})$ are always stuck to zero[8] and one by definition and privacy bits of released stack elements are automatically reset to zero.

We also introduce simple rules by which the privacy bits of new variables evolve as a function of prior φ values. Transfer instructions simply transfer the privacy bit of their variable (*e.g.* `getstatic 3` simultaneously sets $\textsc{st}[s] \leftarrow \textsc{nvm}[3]$ and $\varphi(\textsc{st}[s]) \leftarrow \varphi(\textsc{nvm}[3])$ where s denotes the stack pointer and $\textsc{st}[s]$ the topmost stack element). The rule we apply to arithmetical and logical instructions is *privacy-conservative* namely, the output privacy bits are all set to zero if and only if all input privacy bits were zero (otherwise they are all set to one). In other words, as soon as private data enter a

[6] While the execution of a regular instruction demands only one modular multiplication, the execution of an $\mathsf{INS}_i \in \mathcal{S}$ requires the transmission of an RSA signature (*e.g.* 1024 bits) and an exponentiation (*e.g.* to the power $e = 2^{16} + 1$) in the XμP.

[7] Read: non-((secret-data)-dependent).

[8] *i.e.* any external data fed into the XμP is considered as publicly observable by opponents and hence non-private.

computation all output data are tagged as private. This rule is easily hardwired as a simple boolean OR for non-unary operators.

This mechanism allows to process security-critical instructions in different ways depending on whether they run over private or non-private data. Typically, executing an `if` L does not provide critical information if the topmost stack element is non-private. A CheckOut may not be mandatorily invoked in this case. Accordingly, outputting a non-private value via a `store` IO instruction does not provide any sensitive information, and a CheckOut can be spared in this case as well. In fact, one can easily specify a *security policy* that contextually defines the conditions (over privacy bits) under which a security-critical instruction may or may not trigger a collective verification. To abstract away the security policy chosen by the issuer, we introduce the boolean predicate

$$\mathsf{Alert} : \mathcal{S} \times \Phi \mapsto \{\mathsf{True}, \mathsf{False}\}$$

where Φ denotes the set of all privacy bits $\Phi = \varphi(\text{RAM}) \cup \varphi(\text{NVM}) \cup \varphi(\text{ST})$. $\mathsf{Alert}(\mathsf{INS}, \Phi)$ evaluates as True when a CheckOut is to be invoked. We hence twitch our protocol as now shown on Figure 4.

```
0.    The X$\mu$P receives and checks ID and initializes $i \leftarrow 1$
1.    The X$\mu$P
  (a)        sets $t \leftarrow 1$
  (b)        sets $\nu \leftarrow 1$
2.    The XT sets $\sigma \leftarrow 1$
3.    The X$\mu$P queries from the XT instruction number $i$
4.    The XT
  (a)        updates $\sigma \leftarrow \sigma \times \sigma_i \bmod N$
  (b)        sends $\mathsf{INS}_i$ to the X$\mu$P
5.    The X$\mu$P updates $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \mathsf{INS}_i) \bmod N$
6.    If $t = e$ or $(\mathsf{INS}_i \in \mathcal{S}$ and $\mathsf{Alert}(\mathsf{INS}_i, \Phi))$ the X$\mu$P
  (a)        CheckOut
  (b)        executes $\mathsf{INS}_i$
  (c)        goto step 1
7.    The X$\mu$P
  (a)        executes $\mathsf{INS}_i$
  (b)        increments $t \leftarrow t + 1$
  (c)        goto step 3.
```

Fig. 4. Enforcing a Security Policy: Protocol 1

5 Authenticating Code Sections Instead of Instructions

Following the classical definition of [1,11], we call a *basic block* a straight-line sequence of instructions that can be entered only at its beginning and exited only at its end. The set of basic blocks of a program P is usually given under the form of a graph $\mathsf{CFG}(P)$ and computed by the means of control flow analysis [12,11]. In such a graph, vertices are basic blocks and edges symbolize control flow dependencies: $\mathsf{B}_0 \rightarrow \mathsf{B}_1$ means that the last instruction of B_0 may handover control to the first instruction of B_1. In our instruction set, basic blocks admit at most two sons with respect to control flow dependance; a block has two sons if and only if its last instruction is an `if`. When $\mathsf{B}_0 \rightarrow \mathsf{B}_1$, $\mathsf{B}_0 \Rightarrow \mathsf{B}_1$ means that B_0 has no son but B_1 (but B_1 may have other fathers than B_0). In this section we define a slightly different notion that we call *code sections*.

Informally, a code section is a maximal collection of basic blocks $\mathsf{B}_1 \Rightarrow \mathsf{B}_2 \cdots \Rightarrow \mathsf{B}_\ell$ such that no instruction of $\mathcal{S} \cup \{\texttt{halt}\}$ appears in the blocks except, possibly, as the last instruction of B_ℓ. The section is then denoted by $\mathsf{S} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_\ell \rangle$. In a code section, the control flow is deterministic *i.e.* independent from program variables; thus a section may contain several cascading `goto` instructions. Code sections, unlike basic blocks, may share instructions; yet they have a natural graph structure induced by $\mathsf{CFG}(P)$ which we do not use in the sequel. It is known that computing a program's basic blocks can be done in almost-linear time [12] and it is easily seen that the same holds for code sections. We refer to the full version of this work for an algorithm computing the set $\mathsf{Sec}(P)$ of code sections of a program P.

Given that instructions in a code section are executed sequentially, and that sections can be computed at compile time, signatures can certify sections rather than individual instructions. In other words, a single signature per code section suffices. The signature of a code section S starting at address i is:

$$\sigma_i = \mu(\mathsf{ID}, i, h)^d \bmod N \ ,$$

with $h = H(\mathsf{INS}_1, \ldots, \mathsf{INS}_k)$ where $\mathsf{INS}_1, \ldots, \mathsf{INS}_k$ are the successive instructions in S. Here, H is an iterative hash function recursively defined by $H(x_1, \ldots, x_j) = F(x_j, H(x_1, \ldots, x_{j-1}))$ and $H(x_1) = F(x_1, IV)$ where $F(x, y)$ is H's compression function and IV an initialization constant. We summarize the new protocol on Figure 5.

```
0.    The $\mathsf{X\mu P}$ receives and checks $\mathsf{ID}$ and initializes $i \leftarrow 1$
1.    The $\mathsf{X\mu P}$
  (a)     sets $t \leftarrow 1$   (t now counts code sections)
  (b)     sets $\nu \leftarrow 1$
2.    The $\mathsf{XT}$ sets $\sigma \leftarrow 1$
3.    The $\mathsf{X\mu P}$
  (a)     sets $h \leftarrow IV$
  (b)     queries the code section starting at address $i$
4.    The $\mathsf{XT}$
  (a)     updates $\sigma \leftarrow \sigma \times \sigma_i \bmod N$
  (b)     sets $j = 1$
5.    The $\mathsf{XT}$
  (a)     sends $\mathsf{INS}^i_j$ to the $\mathsf{X\mu P}$
  (b)     increments $j \leftarrow j + 1$
6.    The $\mathsf{X\mu P}$
  (a)     receives $\mathsf{INS}^i_j$,
  (b)     updates $h \leftarrow F(\mathsf{INS}^i_j, h)$
7.    If $\mathsf{INS}^i_j \in \mathcal{S}$ and $(\mathsf{Alert}(\mathsf{INS}^i_j, \Phi)$ or $t = e)$ the $\mathsf{X\mu P}$
  (a)     sets $\nu = \nu \times \mu(\mathsf{ID}, i, h) \bmod N$
  (b)     CheckOut
  (c)     executes $\mathsf{INS}^i_j$
  (d)     goto step 1
8.    Else if $\mathsf{INS}^i_j \in \mathcal{S}$ then the $\mathsf{X\mu P}$
  (a)     sets $\nu = \nu \times \mu(\mathsf{ID}, i, h) \bmod N$
  (b)     increments $t \leftarrow t + 1$
  (c)     executes $\mathsf{INS}^i_j$
  (d)     goto step 3
9.    Else the $\mathsf{X\mu P}$
  (a)     executes $\mathsf{INS}^i_j$
  (b)     increments $j \leftarrow j + 1$
  (c)     goto step 5.
```

Fig. 5. Authentication of Code Sections: Protocol 2

This protocol presents the advantage of being far less time consuming, because the number of $\mathsf{CheckOuts}$ (and updates of ν) is considerably reduced. The formats under which the code can be stored in the XT are diverse. The simplest of these consists in representing P as the list of all its signed code sections $P = (\mathsf{ID}, (1, \sigma_1, \mathsf{S}_1), \ldots, (k, \sigma_k, \mathsf{S}_k))$. Whatever the file format used in conjunction with our protocol is, the term *authenticated program* designates a program augmented with its signature material $\Sigma(P) = \{\sigma_i\}_i$. Thus, our protocols actually execute authenticated programs. A program is converted into an authenticated executable file via a specific compilation phase involving both code processing and signature generations.

6 Security Analysis

What we provide in this section is a formal proof that the protocols described above are secure. The security proof shall have two ingredients: a well-defined security model describing an adversary's goal and resources, and a reduction from some complexity-theoretic hard problem. Rather than rigourously introducing the numerous notions our security model is based upon (which the reader may find in [6], as well as the fully detailed reductions), we give here a high-level description of our security analysis.

The Security Model. We assume the existence of three parties in the game:

– a code issuer $\mathcal{CI}$ that compiles xjvml programs into authenticated executable files with the help of the signing key (N, d),
– an $\mathsf{X}\mu\mathsf{P}$ that follows the communication protocol given in Section 4 and contains the verification key (N, e) matching (N, d). The $\mathsf{X}\mu\mathsf{P}$ also possesses some cryptographic private key material k stored in its NVM,
– an attacker $\mathcal{A}$ willing to access k using means that are discussed below.

Adversarial Goals. Depending on the role played by the $\mathsf{X}\mu\mathsf{P}$'s cryptographic key k, the adversary's goals might be of different nature. Of course, inferring information about k (worse, recovering k completely) comes immediately to one's mind, but there could also be weaker (somewhat easier) ways of having access to k. For instance if k is a symmetric encryption key, $\mathcal{A}$ might try to decrypt ciphertexts encrypted under k. Similarly, if it is a public-key signature key, $\mathcal{A}$ could attempt to rely on the protocol engaged with the $\mathsf{X}\mu\mathsf{P}$ to help forging signatures in a way or an other. More exotically, the adversary could try to *hijack* the key k *e.g.* to use it (or a part of it thereof) as an AES key whereas k was intended to be employed some other way. $\mathcal{A}$'s goal in this case is a bit more intricate to capture, but we see no reason why we should prohibit that kind of scenario in our security model. Third, the adversary may attempt to modify k, thereby opening the door to fault attacks [2,3].

The Attack Scenario. Parties behave as follows. The $\mathcal{CI}$ crafts polynomially many authenticated programs of polynomially bounded size and publishes them. We assume no interaction between the $\mathcal{CI}$ and $\mathcal{A}$. Then $\mathcal{A}$ and the $\mathsf{X}\mu\mathsf{P}$ engage in the protocol and $\mathcal{A}$ attempts to make the $\mathsf{X}\mu\mathsf{P}$ execute a sequence of instructions ξ that was not originally issued by the $\mathcal{CI}$. The attack succeeds when ξ contains a security-critical instruction that handles some part of k which the $\mathsf{X}\mu\mathsf{P}$ nevertheless executes.

We say that $\mathcal{A}$ is an $(\ell, n, \tau, \varepsilon)$-attacker if after seeing at most ℓ authenticated programs $P_1, \ldots, P_\ell$ totalling at most $n \geq \ell$ instructions and processing at most τ steps, $\Pr[\mathcal{A} \text{ succeeds}] \geq \varepsilon$. In this definition, we include in τ the execution time $\mathsf{Time}(\xi)$ of ξ, stipulating by convention that executing each instruction takes one step and that all transmissions (instruction addresses, instructions, signatures and IO data) are instantaneous.

Security Proof for Protocol 1. We state:

Theorem 3. *If the screening scheme μ-RSA is (q_k, τ, ε)-secure against existential forgery under a known message attack, then Protocol 1 is $(\ell, n, \tau, \varepsilon)$-secure for $n \leq q_k$.*

Moreover, when $\mu = \text{FDH}$, outputting a valid forgery is equivalent to extracting e-th roots modulo N as shown in [6, Appendix A.1]. The following corollary is proved by invoking Theorem 2.

Corollary 4. *If μ is a full domain hash function, then Protocol 1 is secure under the RSA assumption in the random oracle model.*

Security Proof for Protocol 2. We now move on to the (more efficient) Protocol 2 defined in Section 5. (μ, H)-RSA is defined as being the RSA screening scheme with padding function $(x, y, z) \mapsto \mu(x, y, H(z))$. We slightly redefine $(\ell, n, \tau, \varepsilon)$-security as the resistance against adversaries that have access to at most ℓ authenticated programs totalling at most n code sections. We state:

Theorem 5. *If the screening scheme (μ, H)-RSA is (q_k, τ, ε)-secure against existential forgery under a known message attack, then Protocol 2 is $(\ell, n, \tau, \varepsilon)$-secure for $n \leq q_k$.*

When $\mu(a, b, c) = h(a \| b \| H(c))$ and h is seen as a random oracle, a security result similar to Corollary 4 can be obtained for Protocol 2. However, a bad choice for H could allow the adversary $\mathcal{A}$ to easily find collisions over μ via collisions over H. Nevertheless, unforgeability can be formally proved under the assumption that H is collision-intractable. We refer the reader to the corresponding theorem given in [6, Appendix B]. Associating this result with Theorem 5, we conclude:

Corollary 6. *Assume $\mu(a, b, c) = h(a \| b \| H(c))$ where h is a full-domain hash function seen as a random oracle. Then Protocol 2 is secure under the RSA assumption and the collision-intractability of H.*

What about active attacks? Although RSA-based screening schemes may feature strong unforgeability under chosen-message attacks (see [6, Appendix A.2] for such a proof for FDH-RSA), it is easy to see that our protocols cannot resist chosen-message attackers whatever the security level of the underlying screening scheme happens to be. Indeed, assuming that the adversary is allowed to query the code issuer $\mathcal{CI}$ with messages of her choosing, a trivial attack consists in obtaining the signature

$$\sigma = \mu(\mathsf{ID}, 1, H(\mathsf{INS}_1, \mathsf{INS}_2, \mathsf{INS}_3))^d \bmod N$$

of a program P where ID is known to be accepted by the $\mathsf{X\mu P}$ and the single-section program P is

$$P = (\texttt{getstatic 17}, \texttt{store IO}, \texttt{halt})$$

wherein NVM[17] is known to contain a fraction of the cryptographic key k, the value 17 being purely illustrative here[9]. Similarly, the attacker may query the signature of some trivial key-modifying code sequence. Obviously, nothing can be done to resist chosen-message attacks.

7 Deployment Considerations and Engineering Options

From a practical engineering perspective, our new architecture is likely to deeply impact the smart card industry. We briefly discuss some advantages of our technology.

Code Patching. A bug in a program does not imply the roll-out of devices in the field but a simple terminal update. Patching a future smart card can hence become as easy as patching a PC. A possible bug patching mechanism consists in encoding in ID a backward compatibility policy signed by the $\mathcal{CI}$ that either instructs the $\mathsf{X\mu P}$ to replace its old ID by a new one and stop accepting older version programs or allow the execution of new or old code (each at a time, *i.e.* no blending possible). The description of this mechanism is straightforward and omitted here.

Code Secrecy. Given that the XT contains the application's code, our architecture assumes that the algorithm's specifications are public. It is possible to reach *some* level of secrecy by encrypting the XT's program under a key (common to all $\mathsf{X\mu P}$s). Obviously, morphologic information about the algorithm will leak out to some extent (loop structure *etc.*) but important elements such as S-box contents or the actual type of boolean operators used by the code could remain confidential if programmed appropriately.

Simplified Product Management. Given that a GSM $\mathsf{X\mu P}$ and an electronic-purse $\mathsf{X\mu P}$ differ only by a few NVM bytes (essentially ID), by opposition to smart-cards, $\mathsf{X\mu P}$s are real commodity products (such as capacitors, resistors or Pentium processors) which stock management is greatly simplified and

[9] The `halt` instruction is even superfluous as the attacker can power off the device right after the second instruction is executed.

straightforward. Given the very small NVM room needed to store an ID and a public-key, a single XμP can very easily support several applications provided that the sum of the NVM spaces used by these applications does not exceed the XμP's total NVM capacity and that these NVM spaces are properly firewalled. From the user's perspective the XμP is tantamount to a key ring carrying all the secrets (credentials) used by the applications that the user interacts with but *not* these applications themselves.

A wide range of trade-offs and variants is possible when implementing the architecture described in this paper. Referring to the extended version of this work [6] for more, a few engineering options are considered here.

Speeding up modular operations. While the multiplication of two κ-bit integers theoretically requires κ^2 operations, multiplying a random ν by $\mu(x)$ may require only $\kappa^2/4$ operations when μ is adequately chosen. Independently, an adequate usage of RAM counters allows to decrease the value of e without sensibly increasing the expected number of CheckOut on the average.

Replacing RSA. Clearly, any signature scheme that admits a screening variant (*i.e.* a homomorphic property) can be used in our protocols. RSA features a low (and customizable) verification time, but replacing it by EC-based schemes for instance, could present some advantages.

Code Size versus Execution Speed. The access to a virtually unlimited ROM renders vacuous the classical dilemma between optimizing code size or speed. Here, for instance, one can cheaply unwind (inline) loops or implement algorithms using pre-computed space-consuming look-up tables instead of performing on-line calculations *etc.*

Smart Usage of Security Hardware Features. Using the Alert predicate, the XμP could selectively activate hardware-level protections against physical attacks whenever a private variable is handled or forecasted to be used a few cycles later.

High Speed XIO. A high-speed communication interface is paramount for servicing the extensive information exchange between the XμP and the XT. Evaluating transmission performances for a popular standard, the Universal Serial Bus (USB)[10], we found that transfers of 32 bits can be done at 25 Mb/s in USB High Speed mode which corresponds to 780K 32-bit words per second. When servicing Protocol 1, this corresponds approximately to a 32-bit XμP working at 390 KHz; when parallel execution and look-ahead transmission take place, one gets a 32-bit machine running at 780 KHz. An 8-bit USB interface leads to 830 KHz. There is no doubt that these figures can be greatly improved.

8 Further Work

The authors believe that the concept introduced in this paper raises a number of practical and theoretical questions. Amongst these is the safe externalization of Java's *entire* bytecode set, the safe co-operative development of code by competing parties (*i.e.* mechanisms for the secure handover of execution from program ID_1 to program ID_2), or the devising of faster execution protocols.

Interestingly, the paradigm of signature screening on which Protocols 1 and 2 are based also exists in the symmetric setting, where RSA signatures are replaced by MACs and a few hash functions. Security can also be assessed formally in this case under adequate assumptions. We refer the reader to [6] for details.

This paper showed how to provably securely externalize programs from the processor that runs them. Apart from answering a theoretical question, we believe that our technique provides the framework of novel practical solutions for real-life applications in the world of mobile code and cryptography-enabled embedded software.

[10] Note that USB is unadapted to our application as this standard was designed for good bandwidth rather than for good latency.

References

1. A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
2. E. Biham and A. Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, In Advances in Cryptography, Crypto'97, LNCS 1294, pages 513–525, 1997.
3. I. Biehl, B. Meyer and V. Müller, *Differential Fault Attacks on Elliptic Curve Cryptosystems*, In M. Bellare (Ed.), Proceedings of Advances in Cryptology, Crypto 2000, LNCS 1880, pages 131–146, Springer Verlag, 2000.
4. M. Bellare, J. Garay and T. Rabin, *Fast Batch Verification for Modular Exponentiation and Digital Signatures*, Eurocrypt'98, LNCS 1403, pages 236–250. Springer-Verlag, Berlin, 1998.
5. M. Bellare and P. Rogaway, *Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols*, Proceedings of the first CCS, pages 62–73. ACM Press, New York, 1993.
6. B. Chevallier-Mames, D. Naccache, P. Paillier and D. Pointcheval, *How to Disembed a Program?*, IACR ePrint Archive, `http://eprint.iacr.org/2004/138`, 2004.
7. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.
8. J.-S. Coron, *On the Exact Security of Full-Domain-Hash*, Crypto'2000, LNCS 1880, Springer-Verlag, Berlin, 2000.
9. J.-S. Coron and D. Naccache, *On the Security of RSA Screening*, Proceedings of the Fifth CCS, pages 197–203, ACM Press, New York, 1998.
10. D.E. Knuth, *The Art of Computer Programming, vol. 1, Seminumerical Algorithms*, Addison-Wesley, Third edition, pages 124–185, 1997.
11. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
12. G. Ramalingam, *Identifying Loops in Almost Linear Time*, ACM Transactions on Programming Languages and Systems, 21(2):175-188, March 1999.
13. R. Stata and M. Abadi, *A Type System for Java Bytecode Subroutines*, SRC Research Report 158, June 11, 1998, `http://www.research.digital.com/SRC/`.