How to Disembed a Program?

Full Version*

Benoît Chevallier-Mames
Gemplus Card International
Applied Research & Security Centre
Avenue des Jujubiers
La Ciotat, F-13705, France

David Naccache, **Pascal Paillier**
Gemplus Card International
Applied Research & Security Centre
34 rue Guynemer
Issy-les-Moulineaux, F-92447, France

{benoit.chevallier-mames,david.naccache,pascal.paillier}@gemplus.com

David Pointcheval
École Normale Supérieure – CNRS
Département d'Informatique
45 rue d'Ulm, F-75230, Paris 05, France
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 will introduce two XμP variants. The first design is a public-key oriented architecture which relies on a new RSA screening scheme and features a relatively low communication overhead at the cost of computational complexity, whereas the second variant is secret-key oriented and relies on simple MACs and hash functions but requires more communication.

For each of these two designs, we propose two protocols that execute and dynamically authenticate arbitrary programs. We also provide a strong security model for these protocols and prove their security under appropriate complexity assumptions.

Keywords. Embedded cryptography, RSA signature screening schemes, ROM-less smart cards, Program authentication, Full-Domain Hash, Secure Tokens, Compilation theory, Provable security, Mobile code, Read-Only Memory.

* An extended abstract of this work can be found under the same title at [8].

Table of Contents

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 [11].

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 $\mathsf{X}\mu\mathsf{P}$) interacting with a powerful terminal (Externalized Terminal or XT). The $\mathsf{X}\mu\mathsf{P}$ 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.

1.1 What Is a Smart-Card

The physical support of a conventional smart-card is a plastic rectangle printed with information concerning the application or the issuer, as well as readable information about the card holder (for instance, a validity date or a photograph). This support can also carry a magnetic stripe or a bar-code.

ISO Standard 7816 specifies that the micromodule must contain an array of eight contacts but only six of these are actually connected to the chip, which is usually not visible. The contacts are assigned to power supplies (V_{cc} and V_{pp}), ground, clock, reset and a serial data communication link commonly called I/O. ISO is currently considering various requests for re-specification of the contacts; notably for dual USB/7816 support.

While for the time being card CPUs are mainly 8 or 16-bit microcontrollers[1] new 32-bit devices have recently become available.

[1] The most common cores are Motorola's 68HC05 and Intel's 80C51.

From a functional standpoint a smart card is a miniature computer. A small on-board RAM serves as a temporary storage of calculation results and the card's microprocessor executes a program etched into the card's ROM at the mask-producing stage. This program cannot be modified or read-back in any way.

For storing user-specific data individual to each card, cards contain EEPROM (Electrically Erasable and Programmable ROM) or flash memory, which can be written and erased hundreds of thousands of times. Java cards even allow the import of executable programs (applets) into their nonvolatile memory according to the card holder's needs.

Finally, the card contains a communication port (serial via an asynchronous link) for exchanging data and control information with the external world. A common bit rate is 9,600 bits per second, but much faster ISO-compliant throughputs are commonly used (from 19,200 up to 115,200 bits per second). The advent of USB cards opens new horizons and allows data throughput to easily reach one megabit per second.

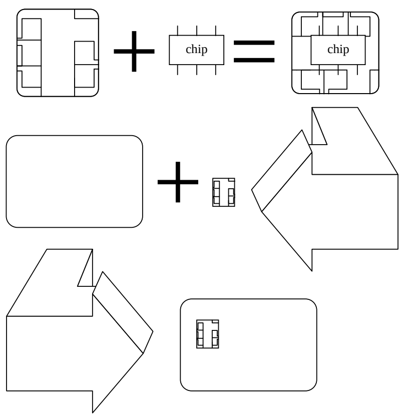

Fig. 1. Smart-Card Manufacturing

To prevent information probing, all these elements are packed into one single chip. If this is not done, the wires linking the system components to each another could become potential passive or active penetration routes [18]. The different steps of smart card manufacturing are shown in Figure 1: wire bonding (chip + micromodule) and potting (chip + micromodule + plastic).

The authors believe that the first smart-card architects did not really brave the wrath of engineering optimal secure portable devices but rather chose the easiest short-term solution: that of *physically hardening* architectures that proved useful in coffee machines or toys.

It seems that both the industry and the research community accepted this endocode[2] (embedded code) paradigm as a truth in itself, which corollary was that subsequent endeavors were mostly devoted to improve the performance of this *existing* architecture[3] instead of looking for *alternative* ways for securely executing embedded code.

[2] $\epsilon\nu\delta o\nu$ = within (*endon*).

[3] Throughout the past decade, the name of the game was larger RAM, ROM and EEPROM capacities, faster coprocessors, lower current consumption, better resistance to side-channel attacks...

1.2 Alternative Designs?

A card never comes alone, it always interacts with an application that implements the 'terminal part' of the protocol. It follows that no matter what application we talk about, terminals *'know'* what functions the cards they must interact with implement. For instance a mobile phone contains the terminal part of the GSM application, ATMs implement the terminal part of the payment application and the same is true for health, gaming, IT security, identity or transport applications.

Some of this century's best discoveries were creative and determined efforts to answer "What if...?" questions. What if people could fly? What if electrical energy could be harnessed to produce light? What if there was an easily accessible, international communication and information network? The answers have resulted in permanent changes: air travel, light bulbs, the Internet. These discoveries have rendered their less effective counterparts to relative extinction from use: gone is the stagecoach, gas lighting, and multi-volume hardbound encyclopedias. These improvements remind us of the research community's option and ability to experiment, re-mold, re-think, and imagine. In that spirit, this article submits a new question:

> *Given that terminals 'know' what functions the cards they must interact with implement, what if terminals could completely contain or help execute a card's code? And if so, could this be done securely and efficiently?*

In this paper we answer the above question by providing the theoretical blueprint of a new secure token called the *Externalized Microprocessor* ($\mathsf{X}\mu\mathsf{P}$) which, unlike a smart-card, 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.

In a nutshell one can compare today's smart cards to Christopher Columbus' caravels that carried all the necessary food, weapons and navigation equipment (ROM) on board whereas the $\mathsf{X}\mu\mathsf{P}$ architecture (ectocode[4]) introduced in this paper is analogous to modern submarines who rely on regular high sea rendezvous and get goods and ammunitions delivered while on assignment.

A basic DSP board $\mathsf{X}\mu\mathsf{P}$ prototype is currently under development.

1.3 Outline of our Work

In Section 2, we progressively refine the machine's architecture. Sections 3 and 5 provide efficient architecture designs for the $\mathsf{X}\mu\mathsf{P}$ relying on RSA. Section 6 introduces a rigorous adversarial model and assesses the security of our execution protocols in it, under adequate complexity assumptions. Section 8 introduces an alternative design based on ephemeral MACs instead of RSA. Further sections extend the instruction set in several directions by introducing powerful instructions while maintaining security. Section 12 provides an implementation of RC4 that illustrates the computational power of our secure platform. Sections 13 and 14 consider various engineering issues related to prototyping the $\mathsf{X}\mu\mathsf{P}$.

[4] $\epsilon\kappa\tau o\varsigma$ = outside (*ectos*).

2 The XμP's Architecture and Instruction Set XJVML

We model the XμP's executable program P as a sequence of instructions:

$$\begin{aligned} 1 &: \ \mathsf{INS}_1 \\ 2 &: \ \mathsf{INS}_2 \\ &\vdots \\ \ell &: \ \mathsf{INS}_\ell \end{aligned}$$

located at addresses $i \in 1, \cdots, \ell$ off-board.

These instructions are in essence similar to instruction codes executed by any traditional microprocessor. Although the XμP's instruction-set can be similar to that of a 68HC05 or a `MIX` processor [15], we have chosen to model it as a JVML0-like machine [22], 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. A toy example of program written in XJVML is given on Figure 2.

Denoting by MEMORY the memory space formed by NVM, RAM and ST altogether[5], the dynamic semantics of our instruction-set are given in Figure 3 (note that there are no rules for `halt` as execution stops when a `halt` is reached).

[5] In other words, MEMORY $= \{\text{RAM}, \text{ST}, \text{NVM}\}$.

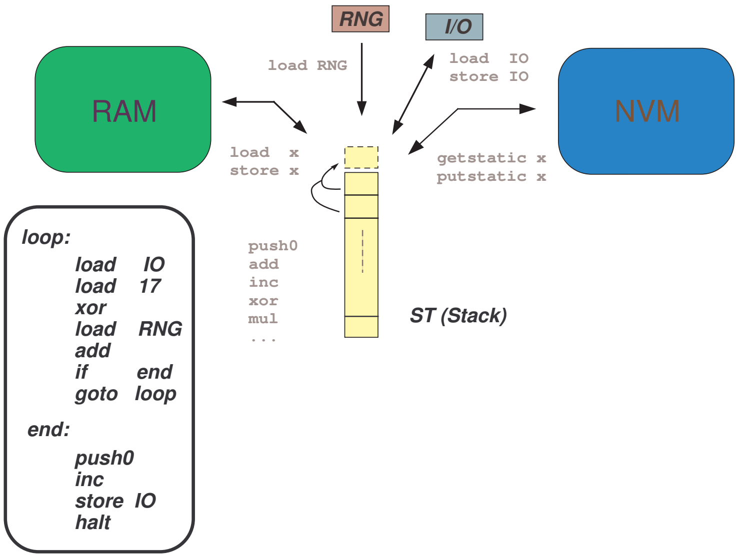

Fig. 2. An Example of Program in XJVML

It is implicitly understood that instructions that read the contents of the stack may throw an interrupt if the stack is empty (*i.e.* $s = 0$) or contains insufficient data (*e.g.* when executing an `xor` while $s = 1$). The following subsections progressively refine the XμP's architecture by presenting successive versions of the machine and explaining the rationale behind each refinement.

2.1 Step 1: The Open XμP

We assume that the program's author deposits in each untrustworthy *Externalized Terminal* (XT) the ectocode:

$$\begin{array}{l} 1: \ \mathsf{INS}_1 \\ 2: \ \mathsf{INS}_2 \\ \vdots \\ \ell: \ \mathsf{INS}_\ell \end{array}$$

The Open XμP is very simple: as execution starts the device resets its program counter ($i \leftarrow 1$) and requires ectoinstruction 1 from the XT. The Open XμP executes INS_1, updates its internal state, determines the next program counter value and repeats this process while $\mathsf{INS}_i \neq$ `halt`. This is nothing but the usual way in which microprocessors execute code stored in external ROMs.

The protocol is formally described on Figure 4 (note that executing INS_i updates i).
As is obvious, the Open XμP lends itself to a variety of attacks. Typically, an opponent could pull-out the contents of the XμP's NVM by sending to the machine the sequence of instructions:

INS_i	**effect on** i	**effect on** RAM	**effect on** ST	**effect on** s
`inc`	$i \leftarrow (i+1)$	none	$\text{ST}[s] \leftarrow (\text{ST}[s]+1)$	none
`dec`	$i \leftarrow (i+1)$	none	$\text{ST}[s] \leftarrow (\text{ST}[s]-1)$	none
`pop`	$i \leftarrow (i+1)$	none	$\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$
`push0`	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow 0$	$s \leftarrow (s+1)$
`load` x	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow \text{RAM}[x]$	$s \leftarrow (s+1)$
`load` IO	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow$ IO	$s \leftarrow (s+1)$
`load` RNG	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow$ RNG	$s \leftarrow (s+1)$
`store` x	$i \leftarrow (i+1)$	$\text{RAM}[x] \leftarrow \text{ST}[s]$	$\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$
`store` IO	$i \leftarrow (i+1)$	$\text{IO} \leftarrow \text{ST}[s]$	$\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$
`if` L	if $\text{ST}[s]=0$ then $i \leftarrow (i+1)$ if $\text{ST}[s] \neq 0$ then $i \leftarrow L$	none	$\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$
`goto` L	$i \leftarrow L$	none	none	none
`xor`	$i \leftarrow (i+1)$	none	$\text{ST}[s-1] \leftarrow \text{ST}[s-1] \oplus \text{ST}[s]$ $\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$
`mul`	$i \leftarrow (i+1)$	none	$\alpha \stackrel{\text{def}}{=} \text{ST}[s-1] \times \text{ST}[s]$ $\text{ST}[s-1] \leftarrow \alpha \bmod 256$ $\text{ST}[s] \leftarrow \alpha \text{ div } 256$	none
		effect on NVM		
`getstatic` x	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow \text{NVM}[x]$	$s \leftarrow (s+1)$
`putstatic` x	$i \leftarrow (i+1)$	$\text{NVM}[x] \leftarrow \text{ST}[s]$	$\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-1)$

Fig. 3. Instruction Set XJVML

0. The XμP initializes $i \leftarrow 1$
1. The XμP queries from the XT ectoinstruction number i
2. The XT sends INS_i to the XμP
3. The XμP executes INS_i
4. goto step 1.

Fig. 4. The Open XμP (Insecure)

$$\begin{array}{l} 1: \texttt{getstatic}\ 1 \\ 2: \texttt{store}\ \mathrm{IO} \\ 3: \texttt{getstatic}\ 2 \\ 4: \texttt{store}\ \mathrm{IO} \\ 5: \texttt{getstatic}\ 3 \\ 6: \texttt{store}\ \mathrm{IO} \\ \vdots \end{array}$$

instead of the legitimate ectoprogram crafted by the XμP's designer (we call such illegitimate sequences of instructions xenoprograms[6]). It follows that the ectocode executed by the machine must be authenticated in some way.

2.2 Step 2: The Authenticated XμP

To ascertain that the ectoinstructions executed by the device are indeed those crafted by the code's author we refine the previous design by associating to each ectoinstruction a digital signature. The program's author generates a public and private RSA signature key-pair $\{N, e, d\}$ and burns $\{N, e\}$ into the Authenticated XμP. The ectocode is enhanced with signatures as follows:

$$\begin{array}{l} 1: \sigma_1: \mathsf{INS}_1 \\ 2: \sigma_2: \mathsf{INS}_2 \\ \vdots \\ \ell: \sigma_\ell: \mathsf{INS}_\ell \end{array}$$

where $\sigma_i = \mu(i, \mathsf{INS}_i)^d \bmod N$ and μ is an RSA padding function[7].

The protocol is enhanced as follows:

0.	The XμP initializes $i \leftarrow 1$
1.	The XμP queries from the XT ectoinstruction number i
2.	The XT sends $\{\mathsf{INS}_i, \sigma_i\}$ to the XμP
3.	The XμP
(a)	ascertains that $\sigma_i^e = \mu(i, \mathsf{INS}_i) \bmod N$
(b)	executes INS_i
4.	goto step 1.

Fig. 5. The Authenticated XμP (Insecure and Inefficient)

While the Authenticated XμP prevents an opponent from feeding the device with xenoinstructions, an attacker could still mix legitimate ectoinstructions belonging to different ectoprograms. In other words, one could successfully replace the 14th opcode of a GSM ectoprogram by the 14th opcode of a Banking ectoprogram.

To avoid this code mixture attack we slightly twitch the design by burning a unique program identifier ID into the device; the existence of ID in the XμP enables the

[6] $\xi\epsilon\nu o\varsigma$ = foreign (*xenos*).

[7] Note that if a message-recovery padding scheme is used, XT storage can be reduced: upon reset the XT can sequentially verify all the σ_i, extract the INS_i and reconstruct the executable part of the ectocode.

execution of the ectoprogram which 'name' is ID. ID is included in each σ_i (*i.e.* $\sigma_i = \mu(\mathsf{ID}, i, \mathsf{INS}_i)^d \bmod N$). IDs are either sequentially generated by the programmer or uniquely produced by hashing the ectoprogram.

2.3 Step 3: The Screening XμP

Although RSA signature verification can be relatively easy, verifying an RSA signature per ectoinstruction is resource-consuming. To overcome this difficulty, 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, assuming that $\mu = h$ is a full domain hash function: 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 the code externalization problem where one does not necessarily need to ascertain that all the signatures are individually correct, but rather control that all the ectocode ($\{\mathsf{INS}_i, \sigma_i\}$) seen by the X$\mu$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 XμP may repeatedly require the same $\{\mathsf{INS}_i, \sigma_i\}$ while executing a given ectoprogram)[8]. To overcome this limitation, we propose 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 smaller than e *i.e.* :

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

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 [6] by the following theorem:

Theorem 1. *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 $\sigma < N$ such that $\sigma^e = \prod_{i=1}^{t} h(m_i) \bmod N$ while the signature of at least one of $m_1, \ldots, m_t$ was not given to $\mathcal{F}$, then $\mathcal{F}$ can be used to efficiently extract e-th roots modulo N.*

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

Noting that $e = 2^{16} + 1$ seems to be a comfortable choice for e here, we devise the protocol shown in Figure 6.

[8] Historically, [4] proposed only the criterion $(\prod \sigma_i)^e = \prod h(m_i) \bmod N$. This version was broken by Coron and Naccache in [14]. Bellare *et alii* subsequently repaired the scheme but the fix introduced the restriction that any message can appear at most once in the list.

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

Fig. 6. The Basic Screening XμP (Insecure)

As one can see, two events can trigger a verification (steps 6a and 6b): the execution of $e - 1$ ectoinstructions (in which case the verification allows to reset the counter t to 1) or the ectoprogram's completion (`halt`). For the sake of conciseness we will denote by CheckOut the test performed in steps 6a and 6b. Namely CheckOut is the XμP-triggered operation consisting in querying from the XT the current value of σ, ascertaining that $\nu = \sigma^e \bmod N$ and halting execution in case of mismatch. We plot this behavior also on Figure 7.

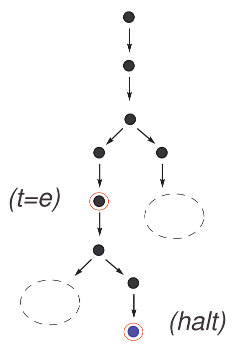

Fig. 7. An Example of Program Execution with the Basic Screening XμP: black dots represent instructions and arrows stand for control flow transitions. Verifications are depicted by small circles around instructions while the event triggering the CheckOut is mentioned within parenthesis.

Unfortunately, this protocol is vulnerable: again, an attacker may feed the device with misbehaving xenocode (*e.g.* the hostile xenocode presented in Section 2.1) crafted so as to never trigger a CheckOut. In other words, as far as the xenocode comprises less than e instructions and never `halts` the attacker can freely 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 ectoinstructions that have an irreversible effect on the device's NVM or on the external world must be preceded by a CheckOut so as to validate the genuineness of the entire list of ectoinstructions *executed so far*.

For this reason we single-out the very few ectoinstructions that send signals out of the XμP (typically the ectoinstruction commanding a data I/O port to toggle) and those ectoinstructions that modify the state of the XμP's non-volatile memory (typically the latching of the control bit that triggers EEPROM update or erasure). These ectoinstructions will be called *security-critical* in the following sections and are defined as follows:

Definition 1. *An ectoinstruction 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 ectoinstructions.*

In our model $\mathcal{S} = \{$`putstatic` x, `store` IO$\}$. We can now twitch the protocol as depicted in Figure 8.

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

Fig. 8. The Screening XμP (Insecure)

We plot an illustration of this protocol on Figure 9.

Unfortunately, the Screening XμP lends itself to a subtle attack that exploits i as a side channel. In the example below k denotes the NVM address of a secret key byte $u = \text{NVM}[k]$:

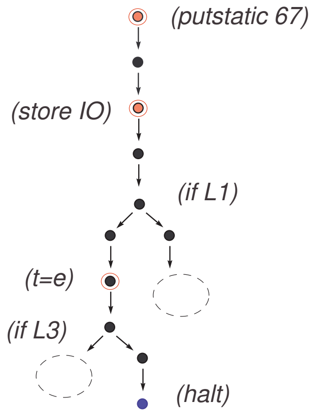

Fig. 9. An Example of Program Execution with the Screening XμP: grey dots now represent security-critical instructions. Verifications are still depicted by small circles around instructions.

$$\begin{array}{rl} 1: & \texttt{getstatic}\ k \\ 2: & \texttt{if}\ 1000 \\ 3: & \texttt{dec} \\ 4: & \texttt{if}\ 1001 \\ 5: & \texttt{dec} \\ 6: & \texttt{if}\ 1002 \\ 7: & \texttt{dec} \\ 8: & \texttt{if}\ 1003 \\ & \vdots \end{array}$$

The Screening XμP will require from the XT a continuous sequence of xenoinstructions followed by a sudden request of xenoinstruction INS_{1000+u} :

$$\mathsf{INS}_1, \mathsf{INS}_2, \ldots, \mathsf{INS}_{u-1}, \mathsf{INS}_u, \mathsf{INS}_{1000+u}, \ldots$$

$u = \text{NVM}[k]$ has hence leaked-out.

Before presenting a solution that eliminates the i side-channel, let us precisely formalize the problem: an ectoinstruction 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 ectoinstruction. The opposite notion is *data indistinguishability* that characterizes those ectoinstructions 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 while, on the contrary, the division ectoopcode `div` is leaky. `div` can be misused to scan secret data as follows: use the unknown variable as a denominator and monitor the occurrence of a 'division by zero' interrupt (when the XμP branches to an interrupt routine it requires from the XT some address different from $i+1$). The attacker can hence decrement the unknown variable until the interrupt is thrown, and count the number of decrements. Note, however, that `div` only leaks information about its denominator and remains data-indistinguishable with respect to the numerator.

INS_i	**effect on** i	**effect on** RAM	**effect on** ST	**effect on** s
`div`	if $\text{ST}[s] \neq 0$ then $i \leftarrow (i+1)$ if $\text{ST}[s] = 0$ then $i \leftarrow$ InterruptAddr	none	$\alpha \leftarrow \text{ST}[s-1]$ div $\text{ST}[s]$ $\beta \leftarrow \text{ST}[s-1]$ mod $\text{ST}[s]$ $\text{ST}[s-1] \leftarrow \alpha, \text{ST}[s] \leftarrow \beta$	none

As the execution of leaky ectoinstructions may reveal information about internal program variables, they fall under the definition of security-criticality and we therefore include them in $\mathcal{S}$. In our ectoinstruction-set (as defined so far), only `if` L and `div` are leaky:

$$\mathcal{S} = \{\texttt{putstatic}\ x,\ \ \texttt{store}\ \text{IO},\ \ \texttt{if}\ \ L,\ \ \texttt{div}\}\ .$$

2.4 Step 4: The Opaque XμP

To deal with information leakage through `if` L, one has several options: the most evident of which consists in simply triggering a CheckOut whenever the XμP encounters any ectoinstruction of $\mathcal{S}$ (Figure 10).

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

Fig. 10. The Opaque XμP (Secure But Suboptimal)

We plot an illustration of this protocol on Figure 11.

As one can easily imagine, `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[9] CheckOut step. This is clearly an overkill: in many cases `if`s can be safely performed on non secret data dependent[10] variables (for instance the variable that counts 16 rounds during a DES computation).

[9] While the execution of a regular ectoinstruction 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.

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

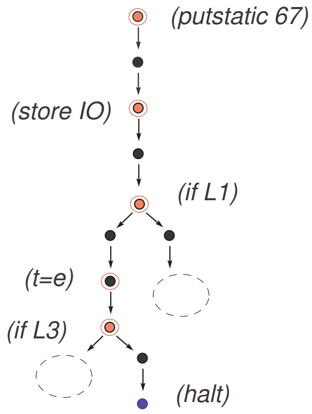

Fig. 11. An Example of Program Execution with the Opaque XμP: `if` instructions are now considered as being security-critical and trigger a CheckOut upon execution.

Efficiency can be improved by adding two popular 80C51 assembly opcodes to the XμP's ectoinstruction-set: `move` $x, \#c$ and `djnz` x, L (the acronym `djnz` stands for *Decrement and Jump if Non Zero*) which dynamic semantics are[11]:

INS_i	**effect on** i	**effect on** RAM	**effect on** ST	**effect on** s
`djnz` x, L	if $\text{RAM}[x] \neq 0$ then $i \leftarrow L$ if $\text{RAM}[x] = 0$ then $i \leftarrow (i+1)$	$\text{RAM}[x] \leftarrow \text{RAM}[x] - 1$	none	none
`move` $x, \#c$	$i \leftarrow (i+1)$	$\text{RAM}[x] \leftarrow \#c$	none	none

Fig. 12. Dynamic Semantics of `djnz` and `move`

It suffices now to create a small array (denoted SRAM, the 'S' standing for 'secure') where the XμP will authorize only the two operations `move` and `djnz` (in other words any ectoinstruction other than `move` and `djnz` attempting to modify the SRAM will cause execution to halt). The SRAM can hence serve to host all the non data-dependent loop counters without triggering a CheckOut:

$$\mathcal{S} = \{\texttt{putstatic}\ x,\ \ \texttt{store}\ \text{IO},\ \ \texttt{if}\ L,\ \ \texttt{djnz}_{x \notin \text{SRAM}} x, L,\ \ \texttt{div}\}\ .$$

This optimization is nothing but an information-flow watchdog that enforces a very primitive *security policy* inside the XμP. Having illustrated our purpose, we now backtrack and *remove* `move` and `djnz` from the ectoinstruction-set and devote the following section to refine and analyse different security policies for reducing the number of CheckOut calls caused by $\mathcal{S}$ as much as possible.

3 Internal Security Policies: Protocol 1

In the next refinement of our architecture a *privacy bit* is associated to each of the XμP's RAM, NVM and stack cells. We denote by $\varphi(\text{RAM}[j])$ the privacy bit associated to $\text{RAM}[j]$, by $\varphi(\text{NVM}[j])$ the privacy bit associated to $\text{NVM}[j]$ and by $\varphi(\text{ST}[j])$ the privacy bit associated to $\text{ST}[j]$. NVM privacy bits are nonvolatile. For the sake of conciseness we denote by Φ the privacy bit space $\Phi = \varphi(\text{MEMORY})$.

[11] $\#c$ represents the constant value c. For instance `move` $x, \#15$ stores the value 15 in $\text{RAM}[x]$.

Informally speaking, the privacy bit's goal 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 stage. Privacy bits of released stack elements are automatically reset to zero and $\varphi(\mathsf{RNG})$ is always stuck to one by definition.

We also introduce simple rules by which the privacy bits of new variables abide (evolve as a function of prior φ values). Transfer ectoinstructions from RAM (`load`) or NVM (`getstatic`) to ST, pushing a memory variable onto the stack, also copy this variable's privacy bit into the privacy bit of the topmost stack element $\varphi(\mathrm{ST}[s])$. Similarly, transfer ectoinstructions from stack to RAM and NVM cells (`store, putstatic`) transfer privacy bits as well. By default, `load` IO sets $\varphi(\mathrm{ST}[s])$ to zero (*i.e.* any external data fed into the XμP is considered as publicly observable by opponents and hence non-private) and `store` IO simply resets $\varphi(\mathrm{ST}[s])$ when returning a data to the external world.

The rule we apply to arithmetical or logical ectoinstructions (and more generally to any ectoinstruction that pops stacked data and/or pushes computation results onto the stack) is *privacy-conservative*; *viz.* 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 computation all output data are tagged as private. As each and every computation is carried out on the stack, it suffices to enforce this rule over the privacy bit subspace $\varphi(\mathrm{ST})$. This rule is easily hardwired as a simple boolean "OR" for binary (two parameter) ectoinstructions; of course, unary ectoinstructions such as `inc` or `dec` leave $\varphi(\mathrm{ST}[s])$ unchanged. For the sake of clarity, we provide in Figure 13 the dynamic semantics of ectoinstructions over Φ.

INS_i	**effect on Φ**
`inc`	none
`dec`	none
`pop`	$\varphi(\mathrm{ST}[s]) \leftarrow 0$
`push0`	$\varphi(\mathrm{ST}[s+1]) \leftarrow 0$
`load` x	$\varphi(\mathrm{ST}[s+1]) \leftarrow \varphi(\mathrm{RAM}[x])$
`load` RNG	$\varphi(\mathrm{ST}[s+1]) \leftarrow 1$
`store` x	$\varphi(\mathrm{RAM}[x]) \leftarrow \varphi(\mathrm{ST}[s])$ $\varphi(\mathrm{ST}[s]) \leftarrow 0$
`load` IO	$\varphi(\mathrm{ST}[s+1]) \leftarrow 0$
`store` IO	$\varphi(\mathrm{ST}[s]) \leftarrow 0$
`if` L	$\varphi(\mathrm{ST}[s]) \leftarrow 0$
`goto` L	none
`xor`	$\varphi(\mathrm{ST}[s-1]) \leftarrow \varphi(\mathrm{ST}[s-1]) \vee \varphi(\mathrm{ST}[s])$ $\varphi(\mathrm{ST}[s]) \leftarrow 0$
`mul`	$\varphi(\mathrm{ST}[s]), \varphi(\mathrm{ST}[s-1]) \leftarrow \varphi(\mathrm{ST}[s-1]) \vee \varphi(\mathrm{ST}[s])$
`div`	$\varphi(\mathrm{ST}[s]), \varphi(\mathrm{ST}[s-1]) \leftarrow \varphi(\mathrm{ST}[s-1]) \vee \varphi(\mathrm{ST}[s])$
`getstatic` x	$\varphi(\mathrm{ST}[s+1]) \leftarrow \varphi(\mathrm{NVM}[x])$
`putstatic` x	$\varphi(\mathrm{NVM}[x]) \leftarrow \varphi(\mathrm{ST}[s])$ $\varphi(\mathrm{ST}[s]) \leftarrow 0$

Fig. 13. Dynamic Semantics Over Φ

Thus, one observes that whatever the ectoprogram actually computes, each and every non-private intermediate value ϑ appearing during its execution must depend only on non-private values stored in NVM and on (observable and hence necessarily non-private) data provided by the XT through the XμP's I/O port. Informally, this means that an external observer could recompute ϑ by herself through a passive observation of ectoinstructions and data emitted by the XT[12], assuming that the XμP's non-volatile non-private information is known[13].

Based on this property which we call *data simulatability*, our security policy allows to process leaky ectoinstructions in different ways depending on whether they are run over private or non-private data. Typically, executing an `if` does not provide critical information if the topmost stack element is non-private because the computation path leading to this value is simulatable anyway. A CheckOut may not be mandatorily invoked in this case. Accordingly, outputting a non-private value via a `store` IO ectoinstruction does not provide any sensitive information, and a CheckOut can be spared in this case as well.

The case of `putstatic` happens to be a bit more involved because skipping CheckOuts in this context gives the ability to freely modify the XμP's NVM, which can be the source of attacks. A typical example is an attack by fault injection, in which a malevolent XT would send to the XμP the xenoinstructions:

```
1 : push0
2 : putstatic 17
3 : halt
```

where data element $\text{NVM}[17]$ is a private DES key byte. Letting the XμP execute this xenocode will partially nullify this key and will thereby allow Differential Fault Analysis [2] to infer the remaining key bits. The fact that data written in NVM is non-private is irrelevant here, because other DFA attacks also apply when storing a private data [3]. This example tells us to require a CheckOut when the NVM cell to be modified is marked as private. But what if the destination is non-private? Well, this depends on the notion captured by what we called privacy in the first place. Assume that we apply the security policy given by Figure 14.

INS_i	**Trigger** CheckOut **if:**
`if` L	$\varphi(\text{ST}[s]) = 1$
`div`	$\varphi(\text{ST}[s]) = 1$
`store` IO	$\varphi(\text{ST}[s]) = 1$
`putstatic` x	$\varphi(\text{NVM}[x]) = 1$

Fig. 14. Read and Write Policy

By virtue of this policy, replacing a non-private NVM data field does not trigger a CheckOut. This means that all non-private NVM objects are left freely accessible to the external world for both reading and writing. This implements a *read and write* policy which might be desirable for objects such as cookies (stored in the device by applications for future use). On the contrary, certain publicly readable objects must not be freely rewritable *e.g.* the balance of an e-purse, an RSA public key[14] and so forth. In which case we enforce the *read only* policy shown on Figure 15.

[12] ϑ's dataflow graph can be easily isolated amongst the stream of ectoinstructions and symbolically executed to retrieve the current value of ϑ.

[13] If this is not the case, a xenoprogram disclosing this NVM information can be easily written by the attacker.

[14] Forcing an RSA e to one would allow to trivially bypass signature verification.

INS_i	**Trigger** $\mathsf{CheckOut}$ **if:**
`if` L	$\varphi(\text{ST}[s]) = 1$
`div`	$\varphi(\text{ST}[s]) = 1$
`store` IO	$\varphi(\text{ST}[s]) = 1$
`putstatic` x	always

Fig. 15. Read-Only Policy

To abstract away the security policy, we introduce the boolean predicate

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

$\mathsf{Alert}(\mathsf{INS}, \Phi)$ evaluates as True when a $\mathsf{CheckOut}$ is to be invoked, *e.g.* following one of the two mechanisms above. We hence twitch our protocol as shown on Figure 16.

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

Fig. 16. Enforcing a Security Policy: Protocol 1

4 The `declassify` and `if_phi` Ectoinstructions

As discussed above, the internal security policy preserves privacy in the sense that any intermediate variable ϑ depending on a private variable ϑ' will be automatically tagged as private. Under many circumstances, final computation results returned by the ectoprogram need to be *declassified* *i.e.* have their privacy bit reset to zero. A typical example is an AES computation: some publicly known plaintext is given to the $\mathsf{X}\mu\mathsf{P}$. The machine encrypts it under a key stored in NVM and marked as private. Because every single variable containing ciphertext bits is a function of all key bits, all the final ciphertext bits will be eventually tagged as private. Outputting or manipulating the ciphertext will hence provoke potentially unnecessary $\mathsf{CheckOut}$s despite the fact that in most protocols and applications ciphertexts are *usually* looked upon as

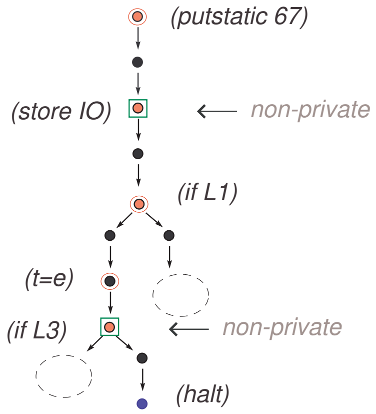

Fig. 17. An Example of Program Execution with Protocol 1: small squares around security-critical instructions denote useless signature verifications saved thanks to the internal security policy.

public data. The same observation applies to public-key signatures, MACs and more generally to any private cryptographic computation which output is public[15].

To allow the programmer easy data declassification, we introduce a specific ectoinstruction that we call `declassify`. This ectoinstruction simply resets the privacy bit $\varphi(\text{ST}[s])$ of the topmost stack element regardless $\text{ST}[s]$'s value. `declassify`'s dynamic semantics are given in Figure 18.

INS_i	effect on i	effect on MEMORY	effect on Φ	effect on s
`declassify`	$i \leftarrow (i+1)$	none	$\varphi(\text{ST}[s]) \leftarrow 0$	none

Fig. 18. Ectoinstruction `declassify`

Of course, `declassify` is security-critical because a malevolent XT could simply send the xenoprogram

$$\begin{array}{l} 1: \texttt{getstatic}\ 17 \\ 2: \texttt{declassify} \\ 3: \texttt{store}\ \text{IO} \end{array}$$

to the $\mathsf{X}\mu\mathsf{P}$ (where 17 is, again, the address of some private NVM data), thereby breaching privacy[16]. We subsequently enrich $\mathcal{S}$ with `declassify` and upgrade the internal security policy to trigger a $\mathsf{CheckOut}$ whenever the $\mathsf{X}\mu\mathsf{P}$ executes this ectoinstruction on a private variable. When the topmost stack element is not tagged as private `declassify` has no effect whatsoever (except incrementing i) and is thus unnecessary to $\mathsf{CheckOut}$. Alert is redefined as on Figure 19.

In the same spirit, programmers may find it handy to dispose of an ectoinstruction that tests privacy bits. The ability to distinguish between private and non-private variables provides a way of treating arbitrary variables in a generic way while relying later on an easy switch between separate ectocode sequences devoted to private or

[15] Note that MACs and ciphertexts are not *necessarily* systematically public, a MAC can for instance serve to generate a secret session key.

[16] Recall that `store` IO does not trigger a $\mathsf{CheckOut}$ when executed on non-private data.

INS_i	**Trigger** CheckOut **if:**
`if` L	$\varphi(\text{ST}[s]) = 1$
`div`	$\varphi(\text{ST}[s]) = 1$
`store` IO	$\varphi(\text{ST}[s]) = 1$
`putstatic` x	always
`declassify`	$\varphi(\text{ST}[s]) = 1$

or

INS_i	**Trigger** CheckOut **if:**
`if` L	$\varphi(\text{ST}[s]) = 1$
`div`	$\varphi(\text{ST}[s]) = 1$
`store` IO	$\varphi(\text{ST}[s]) = 1$
`putstatic` x	$\varphi(\text{NVM}[x]) = 1$
`declassify`	$\varphi(\text{ST}[s]) = 1$

Fig. 19. Security Policies (Including `declassify`)

non-private cases. To this end, we add to the ectoinstruction-set the ectoinstruction `if_phi` L whose dynamic semantics are given in Figure 20.

INS_i	**effect on** i	**effect on** MEMORY	**effect on** ST **and** Φ	**effect on** s
`if_phi` L	if $\varphi(\text{ST}[s]) = 0$ then $i \leftarrow (i+1)$ if $\varphi(\text{ST}[s]) = 1$ then $i \leftarrow L$	none	$\text{ST}[s] \leftarrow$ `undef` $\varphi(\text{ST}[s]) \leftarrow 0$	$s \leftarrow (s-1)$

Fig. 20. Ectoinstruction `if_phi`

In other words, `if_phi` L is similar to `if` L except that the branch is conditioned by the event $\varphi(\text{ST}[s]) = 1$ instead of $\text{ST}[s] = 0$. It is worthwhile to note that `if_phi` L needs not be included into $\mathcal{S}$ because the fact that the program counter jumps to L or $i+1$ does not reveal any information whatsoever about the stacked value other than its privacy status[17].

Interestingly, the ectoinstructions `declassify` and `if_phi` L are powerful enough to allow any computation over privacy bits themselves. For instance, the programmer may need (for some obscure reason) to compute $C \leftarrow A + B$ where $\varphi(C) \leftarrow \varphi(A) \oplus \varphi(B)$. This is not immediate because executing an `add` would compute $A + B$ but the privacy bit of the result would be $\varphi(A) \vee \varphi(B)$. As an illustration, we show how to emulate such an ectoinstruction (`add_mem-xor_phi`) using `declassify` and `if_phi`. Input variables A and B are stored in $\text{RAM}[a]$ and $\text{RAM}[b]$ and remain unmodified throughout the computation, while the output C is stored at address $\text{RAM}[c]$:

[17] This raises an interesting theoretical question: does a multi-level machine where each $\varphi(\text{MEMORY}[i])$ also admits an upper order privacy bit $\varphi(\varphi(\text{MEMORY}[i]))$ makes sense from a security standpoint? Here $\varphi(\varphi(\text{MEMORY}[i])) = 1$ captures a meta-secrecy ('no comment') notion indicating that the machine would even refuse disclosing if the variable $\text{MEMORY}[i]$ is private or not. While the concept can be generalized to higher degrees (*e.g.* $\varphi(\varphi(\ldots\varphi(\text{MEMORY}[i])\ldots)$ its practical significance, applications and semantics seem to deserve clearer definitions and further research.

```
add_mem-xor_phi:
1   : load a
2   : if_phi L2
L1  :
3   : load a
4   : load b
5   : add
6   : goto end
L2  :
7   : load b
8   : if_phi L3
9   : goto L1
L3  :
10  : load a
11  : load b
12  : add
13  : declassify
end :
14  : store c
```

Any other computation over privacy bits is theoretically (and practically) doable (*e.g.* implementing the other boolean operators is left as an exercise to the reader).

5 Authenticating Ectocode Sections: Protocol 2

Following the classical definition of [1,19], 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 techniques [20,19]. 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 ectoinstruction-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 ectoinstruction is an `if` *i.e.* either an `if` L or an `if_phi` L. 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 *ectocode sections*.

Informally, an ectocode section is a maximal collection of basic blocks $\mathsf{B}_1 \Rightarrow \mathsf{B}_2 \cdots \Rightarrow \mathsf{B}_\ell$ such that no ectoinstruction of $\mathcal{S} \cup \{\mathtt{halt}\}$ appears in the blocks except, possibly, as the last ectoinstruction of B_ℓ. The section is then denoted by $\mathsf{S} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_\ell \rangle$. In an ectocode section, very much like in a basic block, the control flow must be deterministic *i.e.* be independent of program variables; thus a section may contain several cascading `goto` ectoinstructions but no data-dependant branches. Ectocode sections, unlike basic blocks, may share ectoinstructions; yet they have a natural graph structure induced by $\mathsf{CFG}(P)$ – which we do not need in the sequel. It is known that computing a program's basic blocks can be done in almost-linear time [20] and it is easily seen that the same holds for ectocode sections. Here is a sketchy way of computing the set $\mathsf{Sec}(P)$ of ectocode sections of an ectoprogram P:

– compute the graph $\mathsf{CFG}(P)$ and associate a section to each and every basic block *i.e.* set

$$\mathsf{Sec}(P) = \text{Vertices}\left(\mathsf{CFG}(P)\right) \ ,$$

– recursively apply the following rules to all elements of $\mathsf{Sec}(P)$:

• if $\mathsf{S} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_\ell \rangle$ and $\mathsf{S}' = \langle \mathsf{B}'_1, \ldots, \mathsf{B}'_{\ell'} \rangle$ are such that $\mathsf{B}_\ell \Rightarrow \mathsf{B}'_1$ in $\mathsf{CFG}(P)$ then unify S and S' into $\mathsf{S} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_\ell, \mathsf{B}'_1, \ldots, \mathsf{B}'_{\ell'} \rangle$,

• if section $\mathsf{S} = \langle \mathsf{B}_1, \ldots, \mathsf{B}_\ell \rangle$ is such that B_i contains $\mathsf{INS} \in \mathcal{S} \cup \{\texttt{halt}\}$, split S into two sections S' and S'' with $\mathsf{S}' = \langle \mathsf{B}_1, \ldots, \mathsf{B}_{i-1}, \mathsf{B}'_i \rangle$ and $\mathsf{S}'' = \langle \mathsf{B}''_i, \mathsf{B}_{i+1}, \ldots, \mathsf{B}_\ell \rangle$ where $(\mathsf{B}'_i, \mathsf{B}''_i)$ is a split of block B_i such that B'_i ends with INS.

For instance, we identify in the toy example given at Figure 21 four sections denoted $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ and S_3. Note that sections S_1 and S_2 have ectoinstructions in common. Ectocode sections are displayed as a graph to depict control flow dependance between them.

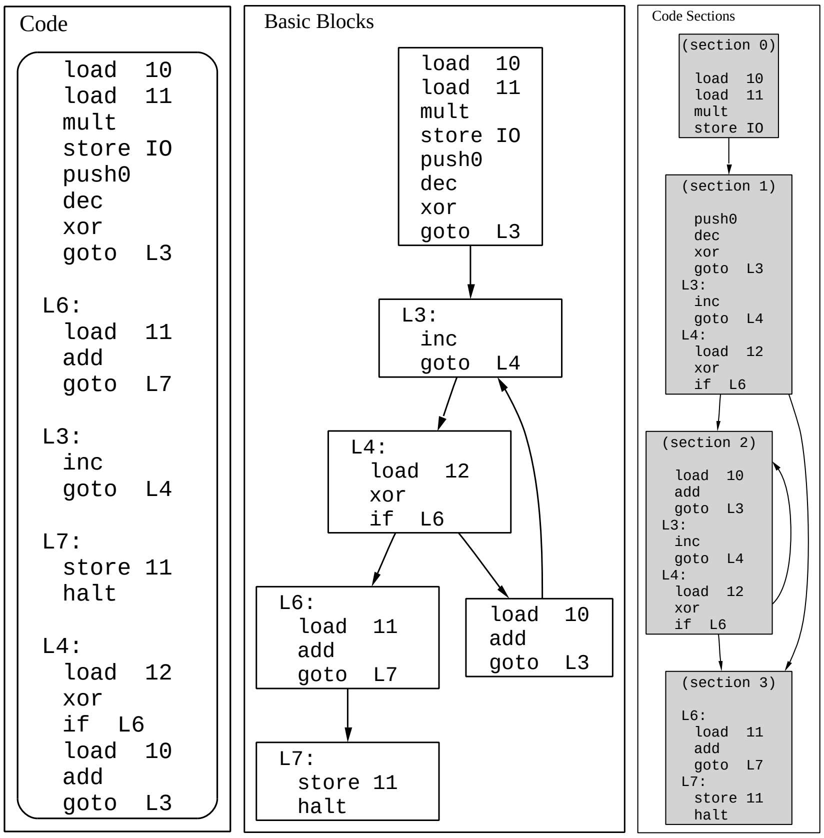

Fig. 21. Example of Determining Code Sections in a Program.

Given that an ectocode section can be regarded as one monolithic composite macro-ectoinstruction, and that they can be computed at compile time, signatures can certify ectocode sections rather than individual ectoinstructions. In other words, a single signature per section suffices (note again that sections that comprise ectoinstructions belonging to $\mathcal{S}$ are chopped at these ectoinstructions).

The signature of an ectocode section S starting at address i is:

$$\sigma_i = \mu(\mathsf{ID}, i, h)^d \mod N \quad \text{with} \quad h = H(\mathsf{INS}_1, \ldots, \mathsf{INS}_k) ,$$

where $\mathsf{INS}_1, \ldots, \mathsf{INS}_k$ are the successive ectoinstructions of S. Here, H is a hash function defined by

$$H(x_1, \ldots, x_j) = F(x_j, F(x_{j-1}, F(\ldots, F(x_2, F(x_1, IV)) \ldots)$$

where $F(x, y)$ is H's compression function and IV an initialization constant[18].

The last ectoinstruction INS_k is:

1. either an element of $\mathcal{S}$ in which case its execution might trigger a $\mathsf{CheckOut}$ or not according to the security policy $\mathsf{Alert}(\mathsf{INS}_k, \Phi)$,
2. or `if_phi`, which does not cause a $\mathsf{CheckOut}$,
3. `halt` which aborts execution.

We summarize the new protocol in Figure 22.

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 an ectocode can be stored in the XT are diverse. The simplest of these consists in representing P as the list of all its signed ectocode 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 ectoprogram* designates an ectoprogram augmented with its signature material $\Sigma(P) = \{\sigma_i\}_i$. Thus, our protocol actually executes authenticated ectoprograms. An ectoprogram is converted into an authenticated executable file via a specific compilation phase involving both code processing and signature generations.

6 Security Analysis

What we are after in this section is a formal proof that our protocols 1 and 2 are secure. The security proof shall have two ingredients: a well-defined security model – describing an adversary's goals and resources – and a reduction to some complexity-theoretic hard problem. As a first investigation, we focus on the protocol of Section 3 in which all ectoinstructions are signed separately. The same results will apply *mutatis mutandis* to the more advanced protocol utilizing signed ectocode sections. We first discuss the security model.

[18] *Iterated* hashing has a *crucial* importance here: iterated hashing allows to pipeline ectoinstructions one by one and thereby allow their on-the-fly hashing and execution. In other words, one does not need to bufferize (cache) an entire section in the $\mathsf{X}\mu\mathsf{P}$ first and execute it next: ectoinstructions arrive one after the other, get hashed and executed.

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

Fig. 22. Ectocode Authentication at Ectocode Section Level: Protocol 2

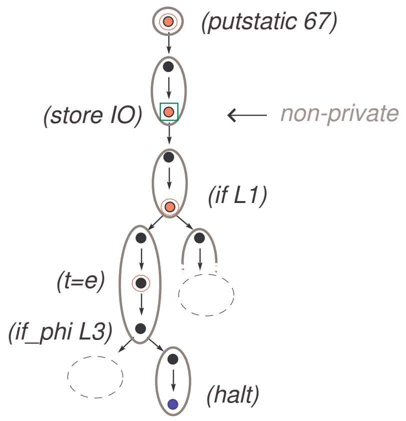

Fig. 23. An Example of Program Execution with Protocol 2: instructions are grouped into code sections at the end of which a CheckOut may or may not be performed.

6.1 The Security Model

We assume the existence of three parties in the game:

- an ectocode issuer CI that compiles XJVML ectoprograms into authenticated executable files with the help of the signing key $\{d, N\}$,
- an $\mathsf{X}\mu\mathsf{P}$ that follows the communication protocol of Section 3 and containing the verification key $\{e, N\}$ matching $\{d, N\}$. 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 will be explained later.

We do not have to include XTs in our model because *in fine* an XT has no particular role in the security model: it contains no cryptographic keys and merely forwards ectoprograms from the CI to the $\mathsf{X}\mu\mathsf{P}$. When the CI sporadically issues compiled ectoprograms, downloading these into XTs can be seen as an *act of publication*. Alternately, we could include XTs in our model and view $\mathcal{A}$ as a malevolent XT.

The Adversary's Resources: Parties behave as follows. The CI crafts polynomially many authenticated ectoprograms of polynomially bounded size and publishes them. We assume no interaction between the 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 xenoinstructions (*i.e.* a sequence not originally issued by the CI).

The Adversary's Goal: The adversary's goal might depend on the role played by the $\mathsf{X}\mu\mathsf{P}$'s cryptographic key k. Of course, inferring information about k – worse, recovering k completely – comes immediately to one's mind, but there could also be weaker (somewhat easier or more subtle) ways of misusing k. For instance if k is a symmetric encryption key, $\mathcal{A}$ might try to decrypt ciphertexts encrypted under k. Similarly, if k is a 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 another. More exotically, the adversary could try to *hijack* the key k *e.g.* use it as an AES key whereas k was intended to be used as an RSA key. $\mathcal{A}$'s goal in this case is a bit more intricate to capture[19]. Hence we do not prohibit that kind of scenario in our security model. Third, the adversary may attempt to modify k, thereby opening the door to fault attacks. To cope with all these subtleties at once and abstract away the cryptographic nature of k, we rely on the notion of *data-flow dependance* [19], starting by introducing the notions our security model will be based upon.

KEY-DEPENDENT EXECUTIONS. Considering an authenticated ectoprogram P, we respectively denote NVM-$\mathsf{In}(P)$ and $\mathsf{In}(P)$ the sets of input state variables and input variables of P; assuming that the stack and the RAM in the $\mathsf{X}\mu\mathsf{P}$ are empty when P starts being executed, NVM-$\mathsf{In}(P)$ only contains non-volatile variables and $\mathsf{In}(P)$ contains variables provided via the external data port (`load` IO). Similarly, NVM-$\mathsf{Out}(P)$ and $\mathsf{Out}(P)$ respectively denote the sets of non-volatile variables written by P and output variables returned through the data port by P. It might be the case that the

[19] To understand the danger here, consider a key used in a 10 round AES. If the device accepts to use the same key for a 12 round AES, an external observer can mount an attack on a $12 - 10 = 2$ round AES.

sets of effectively read or written variables of P vary depending on executions, especially when the control flow graph of P is data-dependent. In this case, we can only apply these definitions to a specific *execution*

$$[P] = P\,(\mathsf{In}, \textsc{nvm}\text{-}\mathsf{In}, \mathsf{RNG})$$

of P, where In is the input tape (IO) of $[P]$, $\textsc{nvm}\text{-}\mathsf{In}$ is the $\mathsf{X\mu P}$'s non-volatile memory and RNG stands for the random tape output by the $\mathsf{X\mu P}$'s Random Number Generator[20]. The sets $\textsc{nvm}\text{-}\mathsf{In}([P])$, $\mathsf{In}([P])$, $\textsc{nvm}\text{-}\mathsf{Out}([P])$, $\mathsf{Out}([P])$ and $\mathsf{RNG}([P])$ are then naturally defined as the subsets of corresponding variable types that are induced by the execution $[P]$.

We identify k as the set of NVM *private* variables containing the secret key bits. Because our XJVML language does not admit branches to addresses obtained from computations[21], there exists a natural bijective mapping between NVM variables and their addresses[22]; we assume that the memory locations of variables in k are publicly known. An execution $[P]$ is said to be *key-independent* if $k \cap \textsc{nvm}\text{-}\mathsf{In}([P]) = \emptyset$ and $k \cap \textsc{nvm}\text{-}\mathsf{Out}([P]) = \emptyset$.

Let ϑ, ϑ' be variables of $[P]$. We denote by $\vartheta \preccurlyeq_\theta \vartheta'$ the data-flow dependence relation [19], meaning that the value written at time $\theta = 1, \ldots, \mathsf{Time}([P])$ in ϑ by $[P]$ is computed as a function $f_{\vartheta,\theta}$ of ϑ', and possibly of other variables. By extension of notations, we denote $\textsc{nvm}\text{-}\mathsf{In}(\vartheta,\theta) \subseteq \textsc{nvm}\text{-}\mathsf{In}([P])$ and $\mathsf{In}(\vartheta,\theta) \subseteq \mathsf{In}([P])$ the sets of state and input variables ϑ' such that $\vartheta \preccurlyeq_\theta \vartheta'$ and $\vartheta = f_{\vartheta,\theta}\,(\mathsf{In}(\vartheta,\theta), \textsc{nvm}\text{-}\mathsf{In}(\vartheta,\theta))$. In a similar way, we denote by $\textsc{nvm}\text{-}\mathsf{Out}(\vartheta,\theta) \subseteq \textsc{nvm}\text{-}\mathsf{Out}([P])$ and $\mathsf{Out}(\vartheta,\theta) \subseteq \mathsf{Out}([P])$ the sets of written state and output variables ϑ' such that $\vartheta' \preccurlyeq_\theta \vartheta$.

EXTENSION TO PROBABILISTIC VARIABLES. A program variable is probabilistic when it is data-flow dependent of a value read on the RNG. It may be the case that the attacker $\mathcal{A}$ attempts to collect information about the random tape of the program through probabilistic variables. A typical case is when the program implements a probabilistic signature scheme *e.g.* DSA. When (with obvious notations) the signatures parts $r = g^h \bmod p \bmod q$ and $s = (x \cdot r + \text{SHA-1}(m))/h \mod q$ have been sent to the external world by a genuine DSA program, an attacker could try to inject extra instructions into the $\mathsf{X\mu P}$ before letting the program reach its final `halt` instruction, so that the random value h (or a piece of it) is returned to $\mathcal{A}$ via the I/O port. The secret key x is then easily extracted from the knowledge of m, q, r, s and h. Of course, the DSA program itself could be written in such a way that the internal value of h is erased before the signature (r, s) is returned: since every part of the signature depends on the private key (x here), the signature is tagged as private and putting in onto the I/O port (`store` IO) will trigger a signature verification, thereby validating the program and also the erasure of h. Nevertheless, it might be the case that cautious erasures be impossible before returning values to the external world, especially when the program implements a 3-pass cryptographic protocol for instance. It appears, as a consequence, that the random values used by the program need to be considered as a part of the private key material k. As said above, this is done by letting $\varphi(\mathsf{RNG})$ take the constant value 1 so that each and every probabilistic variable will be tagged as private due to the privacy-conservative rule applied to computations. Theoretically,

[20] $[P]$ is commonly referred to as the *running code* or *trace* of P.

[21] Only branches and jumps to hard-coded addresses are made possible in our programming language.

[22] This strong property makes pointer analysis vacuous in XJVML and conceptually eases our investigation.

we should then slightly adapt our definition of key-independence above to take the privacy of the random tape into account. For the sake of clarity, though, we simply include all RNG values into the cryptographic material k and maintain our definition as previously discussed for more readability[23]. It is only a matter of form, and this choice does not affect the security proof whatsoever.

Critical ectoinstructions. A variable ϑ of $[P]$ is *externally visible* when it is processed by an ectoinstruction that returns its value to the external world or reveals some information about it: these ectoinstructions are exactly the ones identified as elements of $\mathcal{S}\backslash\{\texttt{putstatic}\}$. Denoting by $\mathsf{Vis}([P])$ the set of externally visible variables of $[P]$, we say that $[P]$ is a *key extractor* when

$$k \cap \bigcup_{\substack{\vartheta\in\mathsf{Vis}([P])\\ 1\leq\theta\leq\mathsf{Time}([P])}} \text{NVM-}\mathsf{In}(\vartheta,\theta) \neq \emptyset\ ,$$

and that P is a *key modifier* if

$$k \cap \text{NVM-}\mathsf{Out}([P]) \neq \emptyset\ .$$

Moreover, during the execution $[P]$, one of the two definitions (or none) is reached first. This happens when some ectoinstruction in $\mathcal{S}\setminus\{\texttt{putstatic}\}$ is executed on a variable depending on k or when executing a `putstatic` on a variable belonging to k. The first ectoinstruction INS_c executed by $[P]$ that classifies $[P]$ into either category is called *critical*. Given that the critical ectoinstruction is unique or does not exist, it follows that key extraction or modification are mutually exclusive properties.

Partial executions. While the $\mathsf{X}\mu\mathsf{P}$ is executing some ectoprogram P with identity ID_P, nothing refrains an attacker from suddenly disconnecting the $\mathsf{X}\mu\mathsf{P}$'s power supply, thus causing a disruption in the execution of P. This event is assumed to reset all volatile variables of P, as well as the $\mathsf{X}\mu\mathsf{P}$'s security buffers ν and h. In this case the ectoprogram P' having been executed (the one seen by the $\mathsf{X}\mu\mathsf{P}$) is a partial execution of P and we denote this partial ordering over ectoprogram executions by $[P'] \sqsubseteq [P]$ (note that $\mathsf{ID}_{P'} = \mathsf{ID}_P$). It is understood here that the two executions $[P']$ and $[P]$ are run on the same input data, state variables and random tape. Having $[P'] \sqsubseteq [P]$ means that in two parallel universes where $[P']$ and $[P]$ are running under the same given tapes, everything remains identical until $[P']$ is completed. When $[P'] \not\sqsubseteq [P]$, there must exist at least one ectoinstruction index i such that the i-th ectoinstruction INS_i executed[24] by $[P']$ differs from the i-th ectoinstruction executed by $[P]$. The set of all ectoinstructions of $[P']$ satisfying this property is denoted $\mathsf{Diff}([P'],[P])$. The first ectoinstruction of $\mathsf{Diff}([P'],[P])$ being executed in $[P']$, *i.e.* the one with smallest index, is denoted $\mathsf{INS}_{\neq}$ and is called the *differentiating* ectoinstruction of $[P']$ with respect to $[P]$. In the general case when P and P' are two arbitrary authenticated programs, we define $\mathsf{Diff}([P'],[P])$ as before if $\mathsf{ID}_{P'} = \mathsf{ID}_P$ and $\mathsf{Diff}([P'],[P]) = [P']$ otherwise. This notion is easily extended when $[P] \not\sqsubseteq [P_1],\ldots,[P_\ell]$ by having $\mathsf{INS}_{\neq}$ defined as the first ectoinstruction among $\cap_{1\leq j\leq\ell}\mathsf{Diff}([P],[P_j])$ that gets executed by $[P]$. For $[P] \not\sqsubseteq [P_1],\ldots,[P_\ell]$, we define the *split* of $[P]$ with respect to $[P_1],\ldots,[P_\ell]$ as the unique pair of executions $([P]^-,[P]^+)$ such that $[P]$ is the concatenation of $[P]^-$ and $[P]^+$ and the first ectoinstruction of $[P]^+$ is precisely $\mathsf{INS}_{\neq}$.

[23] *i.e.* we redefine the set of private variables as $k = k \cup \mathsf{RNG}$.

[24] Note that we rely on the index of INS_i, not on its address.

Free Executions. Although we considered only executions of xjvml ectoprograms so far, an adversary communicating with the $\mathsf{X}\mu\mathsf{P}$ is allowed to transmit a sequence of xenoinstructions that cannot be viewed as $[P]$ for any ectoprogram P. Indeed, when a ectoprogram P is adequately executed by the $\mathsf{X}\mu\mathsf{P}$, the same ectoinstruction INS_i (and possibly σ_i) is sent each time the device requests the i-th ectoinstruction of P. This may not be the case for the xenocode sequence transmitted to an $\mathsf{X}\mu\mathsf{P}$ under attack. In fact, the adversary may well benefit from the memoryless behavior of the $\mathsf{X}\mu\mathsf{P}$ and even entirely base her strategy on this constitutional amnesia. We call a sequence ξ of arbitrary xjvml xenoinstructions a *free execution*. It is easily seen that the notions of key extraction, key modification, critical ectoinstruction and differentiating ectoinstruction with respect to a set $[P_1], \ldots, [P_\ell]$ of ectoprogram executions naturally stretch to free executions.

The Attack Scenario The attack is modeled as follows. The CI publishes a collection of valid authenticated ectoprograms $P_1, \ldots, P_\ell$ totalling at most n ectoinstructions. Variables in k are marked as private in NVM and all other non-volatile variables are non-private. The adversary executes Protocol 1 on the $\mathsf{X}\mu\mathsf{P}$ with respect to some free execution ξ and provides input variables $\mathsf{In}(\xi)$ of her choosing. The attack succeeds when

(a) $\xi \not\sqsubseteq [P_1], \ldots, [P_\ell]$,
(b) if (ξ^-, ξ^+) is the split of ξ with respect to $[P_1], \ldots, [P_\ell]$ then ξ^+ is a key extractor or a key modifier,
(c) ξ^+ is not interrupted by the $\mathsf{X}\mu\mathsf{P}$ (cheating terminal) upon the receiving of the critical xenoinstruction INS_c of ξ^+ *i.e.* INS_c passes through the security firewall and gets executed.

We say that $\mathcal{A}$ is an $(\ell, n, \tau, \varepsilon)$-attacker if after seeing at most ℓ authenticated ectoprograms $P_1, \ldots, P_\ell$ totalling at most $n \geq \ell$ ectoinstructions 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 ectoinstruction takes one clock cycle and that all transmissions (ectoinstruction addresses, ectoinstructions, signatures and IO data) are instantaneous.

6.2 Security Proof for Protocol 1

We state:

Theorem 2. *If the screening scheme μ-RSA is (q_k, τ, ε)-secure against existential forgery under a known message attack, then Protocol 1 of Section 3 is $(\ell, n, \tau, \varepsilon)$-secure for $n \leq q_k$.*

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

Proof. Monitoring the communications between the adversary and the $\mathsf{X}\mu\mathsf{P}$, we transform a successful execution ξ into a valid forgery for μ-RSA thereby simulating a forger $\mathcal{F}$. We first notice that the n signed messages

$$\{(\mathsf{ID}_j, i(j), \mathsf{INS}_{i(j)}) \,\big|\, 1 \leq j \leq \ell, 1 \leq i(j) \leq \mathsf{Size}(P_j)\}$$

harvested by $\mathcal{A}$ in $P_1, \ldots, P_\ell$ are all different, thereby complying with the resources of $\mathcal{F}$. We wait until the completion of the attack and observe our transcript, proceeding as follows. When engaging the protocol with the $\mathsf{X}\mu\mathsf{P}$, $\mathcal{A}$ had to send some value for ID, which is recorded. The situation is twofold:

1. either ID corresponds to one of the ectoprograms $P_1, \ldots, P_\ell$, say P_m. According to the conditions for the attack to succeed, we know that there must exist a differentiating ectoinstruction $\mathsf{INS}_{\neq}$ in ξ with respect to $[P_m]$. Let us denote by $i_{\neq}$ the value of i queried by the $\mathsf{X}\mu\mathsf{P}$ right before $\mathsf{INS}_{\neq}$ was sent. $\mathsf{INS}_{\neq}$ splits ξ into (ξ^-, ξ^+) as defined in the previous section;
2. or it corresponds to none of them but ID is nevertheless accepted by the $\mathsf{X}\mu\mathsf{P}$ (we implicitly assume that the set of IDs accepted by a given $\mathsf{X}\mu\mathsf{P}$ is publicly known). In this case, we define $\mathsf{INS}_{\neq}$ as the first ectoinstruction of ξ and set $i_{\neq} = 1$, $\xi^- = \emptyset$ and $\xi^+ = \xi$.

In either case, the following fact holds:

Lemma 1. CI *never signed* $(\mathsf{ID}, i_{\neq}, \mathsf{INS}_{\neq})$.

Besides, following the attack model, ξ^+ must contain a critical xenoinstruction INS_c which categorizes ξ^+ as a key-extractor or a key-modifier. We know that INS_c gets executed by the $\mathsf{X}\mu\mathsf{P}$ with probability at least ε.

Assume that INS_c is executed by the $\mathsf{X}\mu\mathsf{P}$. We rewind time to the latest moment when the $\mathsf{X}\mu\mathsf{P}$ resets $\nu \leftarrow 1$ before $\mathsf{INS}_{\neq}$ is sent by $\mathcal{A}$ and concentrate on the partial execution ξ_0 of ξ starting from reset time until INS_c is executed. Hence

$$\xi_0 = (\mathsf{INS}_1, \ldots, \mathsf{INS}_{p-1}, \mathsf{INS}_p = \mathsf{INS}_{\neq}, \mathsf{INS}_{p+1}, \ldots, \mathsf{INS}_u = \mathsf{INS}_c) \ ,$$

where $p \geq 1$ and $u \geq p$. For $r = 1, \ldots, u$, we denote by add_r the value of i queried by the $\mathsf{X}\mu\mathsf{P}$ before the ectoinstruction INS_r was transmitted. Now the following fact is a direct consequence of the definition of a critical ectoinstruction:

Lemma 2. *For each and every ectoinstruction* INS_r*,* $r = 1, \ldots, u-1$*, either* $\mathsf{INS}_r \notin \mathcal{S}$ *or* $\mathsf{INS}_r \in \mathcal{S}$ *and* $\mathsf{Alert}(\mathsf{INS}_r, \Phi) = \mathsf{False}$.

Thus, to construct ξ_0, the adversary is left free to use arbitrary combinations of data-indistinguishable XJVML ectoinstructions or security-critical instructions which handle non-private variables. Provided that $u < e$, the first $u - 1$ ectoinstructions will be executed without triggering a $\mathsf{CheckOut}$. Indeed, having $\mathsf{INS}_r \in \mathcal{S}$ and simultaneously $\mathsf{Alert}(\mathsf{INS}_r, \Phi) = \mathsf{True}$ would mean that one of the variables of INS_r is private, thereby proving a data-flow dependence between this variable and one of the state variables in k because we assumed that only these are tagged as private in NVM. Then INS_r would be critical by virtue of our definition and we would get $r = u$ by uniqueness of INS_c.

On the other hand, INS_c will trigger a signature verification over all the ectoinstructions $(\mathsf{INS}_1, \ldots, \mathsf{INS}_u)$ of ξ_0. The fact that the $\mathsf{X}\mu\mathsf{P}$ accepts executing INS_c means that $\mathcal{A}$ had to provide a σ satisfying

$$\sigma^e = \prod_{r=1}^{u} \mu(\mathsf{ID}, \mathsf{add}_r, \mathsf{INS}_r) \bmod N \ .$$

Now the right term contains $\mu(\mathsf{ID}, \mathsf{add}_p, \mathsf{INS}_p) = \mu(\mathsf{ID}, i_{\neq}, \mathsf{INS}_{\neq})$ and from Lemma 1, we know that $\mu(\mathsf{ID}, i_{\neq}, \mathsf{INS}_{\neq})^d \bmod N$ is not contained in any of $P_1, \ldots, P_\ell$. Consequently, the set of messages $\{(\mathsf{ID}, \mathsf{add}_r, \mathsf{INS}_r)\}_{1 \leq r \leq u}$ and σ constitutes a valid forgery for μ-RSA. When $u \geq e$, a $\mathsf{CheckOut}$ is performed after INS_{e-1} is received by the $\mathsf{X}\mu\mathsf{P}$. In this case, we must have $p \leq e-1$ for otherwise a $\mathsf{CheckOut}$ would have taken place (thereby resetting $\nu \leftarrow 1$) before $\mathsf{INS}_{\neq}$ is sent, which contradicts the definition of ξ_0. Hence, the $\mathsf{X}\mu\mathsf{P}$ continues execution only if $\mathcal{A}$ provides a σ such that

$$\sigma^e = \prod_{r=1}^{e-1} \mu(\mathsf{ID}, \mathsf{add}_r, \mathsf{INS}_r) \bmod N \ ,$$

and again, $\mu(\mathsf{ID}, i_{\neq}, \mathsf{INS}_{\neq})$ appears in the right term. A valid forgery for μ-RSA is then given by the set of messages $\{(\mathsf{ID}, \mathsf{add}_r, \mathsf{INS}_r)\}_{1 \leq r \leq e-1}$ and σ. Collecting the forgery can actually be done on the fly while the attack is carried out, thus requiring less than τ steps since $\mathsf{Time}(\xi) \leq \tau$.

Finally, when $\mu = \mathrm{FDH}$, outputting a valid forgery is equivalent to extracting e-th roots modulo N as shown in Appendix A. Then Corollary 1 is proved by invoking Theorem 1. $\square$

6.3 Security Proof for Protocol 2

We now move on to the (more efficient) Protocol 2 defined in Section 5. The (μ, H)-RSA screening scheme is defined as in Section 5 with padding function $(x, y, z) \mapsto \mu(x, y, H(z))$. We slightly redefine $(\ell, n, \tau, \varepsilon)$-security as resistance against adversaries that comply with the attack model of Section 6 and have access to at most ℓ authenticated ectoprograms totalling at most n ectocode sections. We state:

Theorem 3. *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 $n \leq q_k$.*

Proof. We adapt the proof of Theorem 2 by considering ectocode sections instead of ectoinstructions. First, we extend the definition of (static) ectocode sections to free executions, as follows: given a sequence of ectoinstructions ξ, we partition ξ into intervals of maximal length ending by an ectoinstruction of $\mathcal{S}$. The ectocode sections of ξ are identified as these intervals. We further define the *differentiating section* $\mathsf{S}_{\neq}$ of ξ with respect to $[P]$ when $\mathsf{ID}_\xi = \mathsf{ID}_P$ and $\xi \not\sqsubseteq [P]$ as the ectocode section of ξ that contains $\mathsf{INS}_{\neq}$. When $\mathsf{ID}_\xi \neq \mathsf{ID}_P$, $\mathsf{S}_{\neq}$ is set to the first ectocode section of ξ. The split (ξ^-, ξ^+) is redefined in a straightforward manner.

Here again, when Protocol 2 starts, the adversary $\mathcal{A}$ has to send some value for ID. If ID corresponds to P_m for $1 \leq m \leq \ell$, the differentiating section $\mathsf{S}_{\neq}$ in ξ splits ξ into (ξ^-, ξ^+). Then $i_{\neq}$ denotes the value of i queried by the $\mathsf{X}\mu\mathsf{P}$ right before the ectocode section $\mathsf{S}_{\neq}$ is transmitted. If ID corresponds to none of $P_1, \ldots, P_\ell$ then $\mathsf{S}_{\neq}$ is the first section of ξ and we set $i_{\neq} = 1$, $\xi^- = \emptyset$ and $\xi^+ = \xi$. Again:

Lemma 3. CI *never signed* $(\mathsf{ID}, i_{\neq}, \mathsf{S}_{\neq})$.

Moreover, ξ^+ must contain a critical xenoinstruction INS_c characterizing ξ^+ as a key-extractor or a key-modifier. We define the critical section S_c of ξ^+ as the section containing INS_c. By the definition of ectocode sections, S_c ends with INS_c since $\mathsf{INS}_c \in$

$\mathcal{S}$. Assuming that INS_c is executed by the $\mathsf{X}\mu\mathsf{P}$, we rewind time to the latest moment where the $\mathsf{X}\mu\mathsf{P}$ resets $\nu \leftarrow 1$ before $\mathsf{S}_{\neq}$ is sent by $\mathcal{A}$ and focus on the partial execution ξ_0 of ξ starting from reset time until INS_c is executed. Hence

$$\xi_0 = (\mathsf{S}_1, \ldots, \mathsf{S}_{p-1}, \mathsf{S}_p = \mathsf{S}_{\neq}, \mathsf{S}_{p+1}, \ldots, \mathsf{S}_u = \mathsf{S}_c) \ ,$$

where again $p \geq 1$ and $u \geq p$. We denote by add_r the value of i sent by the $\mathsf{X}\mu\mathsf{P}$ to $\mathcal{A}$ before the section S_r is transmitted. We state:

Lemma 4. *For each and every ectocode section S_r, $r = 1, \ldots, u-1$, denoting by INS_r the last ectoinstruction of S_r, either $\mathsf{INS}_r \notin \mathcal{S}$ or $\mathsf{INS}_r \in \mathcal{S}$ and $\mathsf{Alert}(\mathsf{INS}_r, \Phi) = \mathsf{False}$.*

The first $\min(e-1, u) - 1$ ectocode sections will be executed without triggering a $\mathsf{CheckOut}$ because having $\mathsf{INS}_r \in \mathcal{S}$ and $\mathsf{Alert}(\mathsf{INS}_r, \Phi) = \mathsf{True}$ for $r \leq \min(e-1, u)$ leads to $\mathsf{INS}_r = \mathsf{INS}_u$, *reductio ad absurdum*. But $\mathsf{INS}_{\min(e-1,u)}$ must trigger a Check-Out of sections $\{\mathsf{S}_1, \ldots, \mathsf{S}_{\min(e-1,u)}\}$ of ξ_0. Having the $\mathsf{X}\mu\mathsf{P}$ executing $\mathsf{INS}_{\min(e-1,u)}$ requires that $\mathcal{A}$ provided σ with

$$\sigma^e = \prod_{r=1}^{\min(e-1,u)} \mu(\mathsf{ID}, \mathsf{add}_r, H(\mathsf{S}_r)) \bmod N \ .$$

The right term therefore contains $\mu(\mathsf{ID}, \mathsf{add}_p, H(\mathsf{S}_p)) = \mu(\mathsf{ID}, i_{\neq}, H(\mathsf{S}_{\neq}))$ and by virtue of Lemma 3, the set of messages $\{(\mathsf{ID}, \mathsf{add}_r, \mathsf{S}_r)\}_{1 \leq r \leq u}$ and σ constitute a valid forgery for (μ, H)-RSA. $\square$

When $\mu(a, b, c) = h(a \| b \| H(c))$ and h is seen as a random oracle, a security result similar to Corollary 1 can be obtained for Protocol 2. However, a bad choice for H could allow $\mathcal{A}$ to easily find collisions in μ via collisions over H. Nevertheless, unforgeability can be formally proved under the assumption that H is collision-intractable. We refer the reader to Theorem 6 given in Appendix B. Associating Theorems 3 and 6, we conclude:

Corollary 2. *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.*

6.4 Further Discussions on the Security Model

Key-dependent variables. The notions of key-extraction or key modification may seem somewhat too strong; according to the $\mathsf{X}\mu\mathsf{P}$'s internal security policy, $a \leftarrow k \oplus k$ is considered as a private variable if k is private. An adversary successful in exporting a from the $\mathsf{X}\mu\mathsf{P}$ without triggering a $\mathsf{CheckOut}$ is then considered as a key extractor even though no real information about the key k has leaked. Similarly, illegally overwriting an NVM private variable with a copy of itself (via `putstatic`) makes the ectoprogram a key modifier although its execution does not really affect the confidentiality of k. We see no simple means by which our security model would treat these specific cases apart, nor why one would need to. As mentioned earlier, the security policy is preservative over the privacy bits of program variables and it is unclear whether weakening this property is feasible, or even desirable.

WHAT ABOUT ACTIVE ATTACKS? Although RSA-based screening schemes may feature strong unforgeability under chosen-message attacks (see 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 issuer CI with messages of her choosing, a trivial attack consists in obtaining the signature:

$$\sigma = \mu(\mathsf{ID}, 1, H(\texttt{getstatic 17}, \texttt{store IO}, \texttt{halt}))^d \mod N$$

where ID is known to be accepted by the XμP and NVM[17] is known to contain a fraction of the cryptographic key k[25]. Similarly, the attacker may query the signature of some trivial key-modifying sequence. Obviously, nothing can be done to resist chosen-message attacks.

PROBABILISTIC PADDINGS. When strong security against active attacks is desired, RSA-based signature and screening schemes rely upon probabilistic padding functions such as PSS or PSS-R [7,5]. These schemes may then feature an optimally tight security reduction in the random oracle model ($\varepsilon' \approx \varepsilon$) and are therefore comparably more secure against active attacks ($\varepsilon' \approx \varepsilon/q_c$ was proven nearly optimal for FDH for instance), while keeping optimal security $\varepsilon' = \varepsilon$ against passive attacks. Given that chosen-message attacks cannot be avoided in our setting, we see no *evident advantage* in using a probabilistic padding for μ.

7 Variants of our Protocols

Our protocols 1 and 2 are general enough to allow many variations in different directions. What we describe in this section is a couple of variants. The first one is a variation of Protocol 1 which lowers the verification cost by relying on Rabin's signature scheme instead of RSA. The other is a variant of Protocol 2 and reduces the number of verifications by memorizing correct sections in cache memory.

7.1 A Variant with Fast Signature Verification

THE PRINCIPLE. As seen in the previous sections, the XμP's CheckOut procedure is basically a modular exponentiation to the power e, and a comparison. Because RSA-screening imposes that at most $e-1$ instructions (resp. sections) be repeated, our protocols count the number of instructions (resp. sections). However, to be applicable to real life programs that intensively use loops, the public exponent value e must be set to a large enough prime number. A typical value for e is $2^{16}+1$, meaning that a signature verification is roughly equivalent to 17 modular multiplications.

Alleviating the restriction on signature screening, we show that e can be set to 2. Not only is the variant faster (signature verification reduces to a single modular squaring), but also the security level is improved: RSA-screening is based on the RSA (or root extraction) problem, while Rabin-screening relies on integer factoring.

The basic principle of the variant consists in keeping a counter u that counts the number of backward jumps executed since the last CheckOut occurred. Updating u

[25] The `halt` is even superfluous as the attacker can power off the device right after the `store` gets executed.

can be easily hardwired as it amounts to a simple address comparison between the input and output values of i. Wlog, we may assume that the value of i and u are automatically updated during the execution of instructions.

CONCRETE PROTOCOL WITH $e = 2$. The reason for defining u is explained in the following protocol, which follows from Protocol 1. Here, the program is not stored in the XT as a collection $(\{\mathsf{INS}_i, \sigma_i\})$, but rather as $(\{\mathsf{INS}_i, \{\sigma_{i,u}\}_{0 \le u \le U}\})$ for some parameter U. We recall that the execution of INS_i also updates u.

0.	The $\mathsf{X\mu P}$ receives and checks ID and initializes $i \leftarrow 1$	
1.	The $\mathsf{X\mu P}$	
(a)		sets $u \leftarrow 0$
(b)		sets $\nu \leftarrow 1$
2.	The XT sets $\sigma \leftarrow 1$ and $u' \leftarrow 0$	
3.	The $\mathsf{X\mu P}$ queries from the XT ectoinstruction number i	
4.	The XT	
(a)		updates $\sigma \leftarrow \sigma \times \sigma_{i,u'} \bmod N$
(b)		sends INS_i to the $\mathsf{X\mu P}$
5.	The $\mathsf{X\mu P}$ updates $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \mathsf{INS}_i, u) \bmod N$	
6.	The XT updates u' with the knowledge of INS_i	
7.	if $u = U$ or $(\mathsf{INS}_i \in \mathcal{S}$ and $\mathsf{Alert}(\mathsf{INS}_i, \Phi))$ then the $\mathsf{X\mu P}$	
(a)		$(\mathsf{CheckOut})$
		queries from the XT the current value of σ
		halts execution if $\nu \neq \sigma^2 \bmod N$ (cheating XT)
(b)		executes INS_i
(c)		goto step 1
8.	The $\mathsf{X\mu P}$	
(a)		executes INS_i
(c)		goto step 3.

Fig. 24. Rabin-based Variant: Protocol 1.1

The advantages and drawbacks of this protocol are quite clear: on one hand, the program material $\Sigma(P)$ is multiplied in size by a factor nearly U while on the other hand, the $\mathsf{CheckOut}$ stage only requires a single modular multiplication, thereby leading to a 95% speed-up when compared to Protocol 1 with $e = 2^{16} + 1$. As usual, the XT is supposed to have virtually unlimited storage resources.

The security of this variant follows from combining the security proof of Protocol 1 with the following theorem:

Theorem 4. *Let N be an RSA modulus. If a forger $\mathcal{F}$ can produce a list of t messages $\{m_1, \ldots, m_t\}$ and $\sigma < N$ such that $\sigma^2 = \prod_{i=1}^{t} h(m_i) \bmod N$ while the Rabin signature of at least one of $m_1, \ldots, m_t$ was not given to $\mathcal{F}$, then $\mathcal{F}$ can be used to efficiently factor N.*

The proof of Theorem 4 is detailed in Appendix C.

7.2 A Variant with a Caching Mechanism

Independently of minimizing the cost of a signature verification, one could also want to reduce the number of signature verifications. Authenticating code sections in Protocol

2 allowed to reduce the number of modifications applied to the verification accumulator (*i.e.* the register ν). Here, we come up with a new improvement consisting in remembering the signature of correct sections.

Informally, we use a cache of sections that were already recognized as valid by the XμP in the past, and consequently for which future verifications are useless. Better than storing the whole contents of code sections, we cache hash values of these sections under a collision-resistant hash function of small output size $\mathcal{H}_{160}$. The protocol also uses a cache memory CACHEthat should be of type LIFO (Last In - First Out). We make use of a function AddInCache allowing to append a data in cache memory[26]. The size $\Upsilon$of the cache memory has a direct impact on the efficiency of this variant.

The protocol is then as depicted on Figure 25.

0.	The XμP receives and checks ID and initializes $i \leftarrow 1$	
1.	The XμP	
	(a)	sets $t \leftarrow 1$
	(b)	sets $\nu \leftarrow 1$
2.	The XT sets $\sigma \leftarrow 1$	
3.	The XμP	
	(a)	sets $h \leftarrow IV$
	(b)	queries the ectocode section starting at address i
4.	The XT	
	(a)	updates $\sigma \leftarrow \sigma \times \sigma_i \bmod N$
	(b)	sets $j \leftarrow 1$
5.	The XT	
	(a)	sends INS^i_j to the XμP
	(b)	increments $j \leftarrow j+1$
6.	The XμP	
	(a)	receives INS^i_j,
	(b)	updates $h \leftarrow F(\mathsf{INS}^i_j, h)$
7.	if $\mathsf{INS}^i_j \notin \mathcal{S}$, then the X$\mu$P	
	(a)	executes INS^i_j
	(b)	increments $j \leftarrow j+1$
	(c)	goto step 5.
8.	The XμP sets $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, h) \bmod N$	
9.	if $\neg\mathsf{Alert}(\mathsf{INS}^i_j, \Phi))$ then the XμP increments $t \leftarrow t+1$	
10.	if $t = e$ or $(\mathsf{Alert}(\mathsf{INS}^i_j, \Phi))$ then the XμP	
	(a)	computes $\kappa \leftarrow \mathcal{H}_{160}(\nu)$
	(b)	if $\kappa \notin$ CACHE, CheckOut
	(c)	executes INS^i_j
	(d)	AddInCache(κ)
	(e)	goto step 1
11.	The XμP	
	(a)	executes INS^i_j
	(b)	goto step 3

Fig. 25. Variant with Cache Mechanism: Protocol 2.2

[26] AddInCache can be implemented in several ways (*e.g.* with a cycling buffer).

The main advantage of this protocol is that if the cache table is large enough, most of sections are verified only once, thereby speeding up the execution of very repetitive programs. Finally, we mention that the table CACHE could also be stored in NVM in order to memorize the hash values of already verified sections. In this respect, this table could also be initialized during the personalization step. This in turn results in that critical (*i.e.* overused) functions will not trigger a signature verification when executed.

8 MAC-Based Ectoprogram Authentication

Interestingly, public-key cryptography is not mandatory for implementing the concept described in this paper. This section describes a simpler variant based on symmetric cryptography. In this section $\mu_K(x)$ denotes a MAC function where K is the key and x the MAC-ed data. $\mathcal{H}_1$ and $\mathcal{H}_2$ denote hash functions (*e.g.* SHA-1) with respective compression functions H_1 and H_2 and initialization vectors IV_1 and IV_2. Finally, ℓ the number of ectoinstructions in the ectoprogram P. We assume that $\mathsf{ID} = \mathcal{H}_1(P)$. The protocol is shown at Figure 26.

-2. The XμP generates a random session key K and initializes $h \leftarrow \mathrm{IV}_1$
-1. for $i \leftarrow 1$ to ℓ
  (a) The XμP queries from the XT ectoinstruction number i
  (b) The XT sends INS_i to the XμP
  (c) The XμP computes $\sigma_i \leftarrow \mu_K(i, \mathsf{INS}_i)$ and updates $h \leftarrow H_1(h, \mathsf{INS}_i)$
  (d) The XμP sends σ_i to the XT (no copies of σ_i or INS_i are kept in the XμP)
  (e) The XT stores σ_i

0. The XμP ascertains that $h = \mathsf{ID}$ (abort if mismatch) and initializes $i \leftarrow 1$
1. The XμP sets $\nu \leftarrow \mathrm{IV}_2$
2. The XT sets $\sigma \leftarrow \mathrm{IV}_2$
3. The XμP queries from the XT ectoinstruction number i
4. The XT
  (a) updates $\sigma \leftarrow H_2(\sigma, \sigma_i)$
  (b) sends INS_i to the XμP
5. The XμP updates $\nu \leftarrow H_2(\nu, \mu_K(i, \mathsf{INS}_i))$
6. if $\mathsf{INS}_i \in \mathcal{S}$ and $\mathsf{Alert}(\mathsf{INS}_i, \Phi)$ then the XμP
  (a) CheckOut: query σ from the XT and ascertain that $\sigma = \nu$
  (b) executes INS_i
  (c) goto step 1
7. The XμP
  (a) executes INS_i
  (b) goto step 3.

Fig. 26. MAC-Based (Ectoinstruction Level) Protocol: Protocol 3

In steps -2 and -1 the XμP does two operations:

1. Hash the entire program presented by the XT to ascertain that this program indeed hashes into the reference digest ID, burned into the device at production time.

2. MAC each and every instruction under an ephemeral key K and send the resulting MACs to the XT for storage.

8.1 Security Analysis

Following the security model defined in Section 6, the security of Protocol 3 can be formally assessed. Before assessing the security of our protocol, we define a weak form of forgery for MAC functions.

Weak Forgeries for Symmetric Signatures. Classical notions of security for symmetric signatures are given in Appendix D. Informally, a *weak forgery* for a given MAC function μ_K with respect to a given hash function $\mathcal{H}$ is a list $M = (m_1, \ldots, m_t)$ of messages and a value h such that

$$\mathcal{H}\left(\mu_K(m_1), \ldots, \mu_K(m_t)\right) = h$$

whereas the signature $\mu_K(m_i)$ of m_i was never given to the forger for at least one value of $i \in [1, t]$. This security notion comes with different flavors, depending on the attack model, *i.e.* whether the forger is allowed to make adaptive signature queries or not. In the sequel, we only consider the case of passive attacks: the forger $\mathcal{F}$ is given a list of message-signature pairs M_0 and attempts to produce (M, h) as above such that $M \subsetneq M_0$.

It is quite easy to show that a weak forgery is equivalent to a forgery in the random oracle model that is, when $\mathcal{H}$ is seen as a random oracle. The proof of equivalence is omitted here and left as an exercise for the reader.

More formally, we define a (q_k, τ, ε)-weak forger for μ_K as a probabilistic polynomial-time Turing machine $\mathcal{F}$ such that $\mathcal{F}$ returns a weak forgery (M, h) as above with some probability ε after at most τ elementary steps, given as an input a list M_0 of q_k message-signature pairs. The MAC function μ_K is said to be (q_k, τ, ε)-secure against weak forgeries with respect to $\mathcal{H}$ when there is no (q_k, τ, ε)-weak forger for μ_K.

Security Proof for Protocol 3. We recall the attack model of Section 6.3, saying that Protocol 3 is $(\ell, n, \tau, \varepsilon)$-secure if any adversary $\mathcal{A}$ having access to at most ℓ authentic programs totalling at most n ectoinstructions and running in at most τ steps succeeds with probability at most ε. Here yet again, $\mathcal{A}$ succeeds when the first ectoinstruction belonging to $\mathcal{S}$ of the given code sequence ξ is accepted and executed by the $\mathsf{X}\mu\mathsf{P}$. We claim:

Theorem 5. *If μ_K is (q_k, τ, ε)-secure against weak forgeries with respect to $\mathcal{H}_2$, then Protocol 3 is $(\ell, n, \tau, \varepsilon)$-secure for $n \le q_k$ under the collision-freeness of $\mathcal{H}_1$.*

Proof. We transform a successful free execution ξ created by an $(\ell, n, \tau, \varepsilon)$-attacker $\mathcal{A}$ into a weak forgery for μ_K with respect to $\mathcal{H}_2$ or a collision for $\mathcal{H}_1$. Before starting, we note that $\mathcal{A}$ is given (after the protocol has executed the preliminary steps) no more than n different signed messages $\{(i(j), \mathsf{INS}_{i(j)}) \,|\, 1 \le j \le \ell, 1 \le i(j) \le \mathsf{Size}(P_j)\}$, thereby complying with the resources of a known-message weak forger for μ_K.

As in the proofs of previous sections, we launch $\mathcal{A}$ and monitor all communications between $\mathcal{A}$ and the $\mathsf{X}\mu\mathsf{P}$. Now, when Protocol 3 starts, the adversary $\mathcal{A}$ sends some program P that hashes into some value $\mathsf{ID} = \mathcal{H}_1(P)$. ID necessarily corresponds to $\mathcal{H}_1(P_m)$ for some $1 \le m \le \ell$ otherwise the attack cannot be successful. The attacker

then sends a free execution ξ to the $\mathsf{X}\mu\mathsf{P}$. Again, there must be a differentiating ectoinstruction $\mathsf{INS}_{\neq}$ attesting that $\xi \not\sqsubseteq [P_m]$. Then $i_{\neq}$ denotes the value of i queried by the $\mathsf{X}\mu\mathsf{P}$ right before the ectoinstruction $\mathsf{INS}_{\neq}$ is sent by $\mathcal{A}$.
We define by E the event according to which the $\mathsf{X}\mu\mathsf{P}$ did not send $\mu_K(i_{\neq}, \mathsf{INS}_{\neq})$ at step -1, *i.e.* the instruction $\mathsf{INS}_{\neq}$ was not given to the $\mathsf{X}\mu\mathsf{P}$ during the preliminary stage. If E is false, then the program P contains the instruction $\mathsf{INS}_{\neq}$ at address $i_{\neq}$ leading to

$$\mathcal{H}_1(P) = \mathcal{H}_1(\dots, \mathsf{INS}_{\neq}, \dots) = \mathsf{ID} = \mathcal{H}_1(P_m) = \mathcal{H}_1(\dots, \mathsf{INS}, \dots)\ ,$$

for some instruction $\mathsf{INS} \neq \mathsf{INS}_{\neq}$ of P_m. We then stop and output (P, P_m) as a collision for $\mathcal{H}_1$. If E is true, we call INS' the $i_{\neq}$-th instruction MAC-ed by the $\mathsf{X}\mu\mathsf{P}$ at step -1 and we proceed as follows. We know by definition of the security model that ξ must contain a critical ectoinstruction $\mathsf{INS}_c \in \mathcal{S}$ sent to the $\mathsf{X}\mu\mathsf{P}$ after $\mathsf{INS}_{\neq}$. When the attack succeeds, INS_c is executed by the $\mathsf{X}\mu\mathsf{P}$ after a $\mathsf{CheckOut}$ verification. At the moment of this verification, the transcript contains the partial execution $\xi' \sqsubseteq \xi$ (all instructions executed until that point in time). Now when the verification occurs, the $\mathsf{X}\mu\mathsf{P}$ compares its digest

$$\nu = \mathcal{H}_2\left(\sigma_1, \dots, \sigma_{i_{\neq}-1}, \mu_K(i_{\neq}, \mathsf{INS}_{\neq}), \dots, \mu_K(i_c, \mathsf{INS}_c)\right)$$

with the value σ sent by $\mathcal{A}$. Here, the first $i_{\neq -1}$ instructions are the common instructions of P_m and ξ. Since the event E is true, the MAC of $(i_{\neq}, \mathsf{INS}_{\neq})$ was not given to $\mathcal{A}$, so that (M, σ) with

$$M = \left\{\sigma_1, \dots, \sigma_{i_{\neq}-1}, \mu_K(i_{\neq}, \mathsf{INS}_{\neq}), \dots, \mu_K(i_c, \mathsf{INS}_c)\right\}$$

constitutes a valid weak forgery for μ_K with respect to $\mathcal{H}_2$. □

The very same technique is applicable to the authentication of ectocode sections. In this case, sections are MAC-ed as

$$\sigma = \mu_K(i, \mathcal{H}_3(\mathsf{S}_i))\ ,$$

where, as before, $\mathcal{H}_3$ is a hash function that processes one by one the ectoinstructions of S_i. The extension of the security proof to this variant is straightforward, and we get the same security level under the additional assumption that $\mathcal{H}_3$ is collision-free.

8.2 Hashing Tree Variant

Even if the exchange of digests can be limited to one digest per ectocode section, before execution starts, the entire programme must be pipelined into the $\mathsf{X}\mu\mathsf{P}$ before execution starts. This is clumsy and time consuming. Steps -2 and -1 can be eliminated by resorting to tree hashing. Tree hashing is a well known cryptographic technique allowing to ascertain that a word belongs to a message which digest value is ID without re-hashing the entire message.
The technique is illustrated in Appendix C where one can see that:

$$\mathsf{ID} = H(P) = h_{1,2,3,4,5,6,7,8} = H(H(h_{1,2}, (H(\mathsf{INS}_3), h_4)), h_{5,6,7,8})$$

For the sake of clarity we illustrate the idea with individual instructions rather than with ectocode sections and denote by Δ_i the partial hash values required to reconstruct ID given INS_i (in our example $\Delta_3 = \{h_4, h_{1,2}, h_{5,6,7,8}\}$).

0.	The $\mathsf{X}\mu\mathsf{P}$ initializes $i \leftarrow 1$
1.	The $\mathsf{X}\mu\mathsf{P}$ queries from the XT ectoinstruction number i
2.	The XT sends the data Δ_i and INS_i to the $\mathsf{X}\mu\mathsf{P}$
3.	The $\mathsf{X}\mu\mathsf{P}$
(a)	checks that $\mathsf{HashTree}(\mathsf{INS}_i, \Delta_i) = \mathsf{ID}$
(b)	executes INS_i
(c)	goto step 1

Fig. 27. Hash-Tree Protocol: Protocol 4

9 The `if_skip` and `restart` Ectoinstructions

Many cryptographic operations require secret-data-dependent[27] `if`s. RSA square-and-multiply is one such typical example where different secret bits trigger different INS_i requests.

While several side-channel protection techniques [12] allow an easy $\mathsf{X}\mu\mathsf{P}$ implementation of such routines, it may appear handy to have a specific instruction that allows the programmer to disable the execution of a sequence of ectoinstructions but still accumulate them in $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \mathsf{INS}_i) \bmod N$.

We introduce two ectoinstructions called `if_skip` and `restart` which work as follows. On executing `if_skip`, the $\mathsf{X}\mu\mathsf{P}$ checks the topmost stack element $\textsc{st}[s]$. If $\textsc{st}[s] \neq 0$ the ectoinstruction has no particular effect[28]. If $\textsc{st}[s] = 0$ however, the device suspends the execution of all ectoinstructions following the `if_skip` while maintaining their modular accumulation in ν until the ectoinstruction `restart` is encountered. Regular execution mode is then recovered.

It is easy to see that `if_skip` and `restart` allow to program data-dependant routines without explicit branches: instead of executing separate functions and relying on control switches, the programmer can ordain the ectoprogram to inhibit a fraction of itself depending on input values (without altering the authentication process though).

From a computational standpoint, control switches and ectocode inhibition have comparable effects and are equivalently powerful. For the programmer, changing from using one to the other is a mere question of programming habits.

What we want to ascertain, however, is the fact that data-dependent ectocode inhibition is really data-indistinguishable; in other words, we require that no information about the (private) topmost stack element should leak out of the device. To illustrate different leakage hazards, we consider the following ectoprogram where $\textsc{ram}[a]$ and $\textsc{ram}[b]$ are respectively private and non-private variables:

```
1 : load a
2 : if_skip
3 : load b
4 : inc
5 : store b
6 : push0
7 : restart
  ⋮
```

[27] (secret-data)-dependent.

[28] Other than $i \leftarrow (i+1)$, $\textsc{st}[s] \leftarrow \texttt{undef}$ and $s \leftarrow (s-1)$.

As is obvious, RAM$[b]$ is incremented when RAM$[a] = 0$ and is left unchanged otherwise. Since RAM$[b]$ is non-private, its value before the `if_skip` execution can be retrieved (see Section 3). Therefore, it suffices to consult the value of RAM$[b]$ just after the `restart` is executed[29] to probe whether RAM$[a] = 0$ or not.

WRITTEN VARIABLES. This observation tells us to force to one the privacy bit of each and every variable *written* by the `if_skip` sequence *i.e.* by ectoinstructions located after `if_skip` and before `restart`. Indeed, one observable effect of executing a sequence of ectoinstructions is the modifications induced by these in memory variables. We therefore twitch our `if_skip` mechanism so that ectoinstructions that write variables (`store` x and `putstatic` x) appearing in the `if_skip` sequence (be it executed or not) set $\varphi(\text{RAM}[x])$ or $\varphi(\text{NVM}[x])$ to one.

SECURITY-CRITICALITY. Another danger stems from security-criticality: an attacker may send to the XμP, in the middle of an `if_skip` sequence, a xenocode such as

$$\begin{array}{ll} i & : \texttt{push0} \\ i+1 & : \texttt{store IO} \end{array}$$

and inspect what comes-out at the XμP's IO port. The value zero will appear on the data port if and only if the sequence is executed. Similarly, and for the same confidentiality reasons, ectoinstructions that might trigger a CheckOut must be forbidden in an `if_skip` sequence. We must therefore force the XμP to abort the protocol (returning a "cheating terminal" error) if a security-critical ectoinstruction is encountered in an `if_skip` sequence.

JUMPS AND BRANCHES. In the same spirit, an attacker may insert a jump into an `if_skip` sequence, for instance with the xenoinstructions

$$\begin{array}{ll} i & : \texttt{push0} \\ i+1 & : \texttt{goto 1} \end{array}$$

Executing the jump would make the XμP query the contents of address 1, thus revealing execution. The same holds for `if_phi`. Therefore, branches and jumps must be excluded from the set of ectoinstructions that the XμP is authorized to legitimately encounter while treating an `if_skip` sequence.

STACK-BASED ATTACKS. Another observable witness of executions is their effect on the stack level. In our toy example, the `if_skip` sequence ends with a `push0` ectoinstruction. As a result, the value zero is pushed onto the stack (and s incremented by one) when the sequence gets executed, which is not the case when the sequence is not executed. An attacker willing to probe if RAM$[a] = 0$ can simply send to the XμP, right after `restart`, a xenocode that pops off all the stack elements until the stack is emptied, in which case an interrupt is invoked. A simple count will reveal whether the sequence was executed or not.

At a first glance, the impact of this observation would be twofold. First, the `if_skip` sequence designed by the programmer seems to require that the stack level be left unchanged; we call a sequence of ectoinstructions featuring this property *stack-level*

[29] By sending {`load` b, `store` IO} to the XμP.

invariant. Second, no stack-level variant sequence should be created in an on-the-fly manner by an attacker while an `if_skip` sequence is being treated. Indeed, adding the xenoinstruction `push0` right before sending `restart` would render the sequence stack-level variant, thereby leading to a security breach.

Instead of guarantying that `if_skip` sequences are stack-level invariant, we choose, in order to thwart stack-related attacks, to introduce a conceptually simpler mechanism that we describe later.

Interrupt-based attacks. Forcing a CheckOut or writing on the IO port are not the only ways in which one can breach the confidentiality of an `if_skip`'s input. One may also provoke dummy interrupts by injecting into the `if_skip` sequence interrupt-generating xenoinstructions. For instance, the xenocode

$$\begin{aligned} i &: \texttt{push0} \\ i+1 &: \texttt{push0} \\ i+2 &: \texttt{div} \end{aligned}$$

throws in a *division-by-zero* interrupt when executed. From the above, we know that we already excluded `div` given its security-criticality; nevertheless, interrupts can also be generated by non-security-critical operators such as `xor` or `add`. These instructions, indeed, are fed with the stack's contents and may well throw an interrupt when the stack is empty or contains a single element. The attacker may then modify the `if_skip` sequence and send a series of `xor`s:

$$\begin{aligned} i &: \texttt{xor} \\ i+1 &: \texttt{xor} \\ &\vdots \end{aligned}$$

It is easy to see that, whatever the ectocode executed by the XμP is, an attacker can retrieve the stack level s at any point in time throughout the protocol. In the present attack, the attacker recovers the value of s before the `if_skip` is executed, rewinds (reruns) the device, sends a sequence of $s+1$ `xor`s, and waits for the interrupt to occur. The interrupt shows up when the XμP requests the interrupt address (instead of $s+i+1$) from the XT. In our xjvml language, as defined so far, the 'empty stack' interrupt is the only one that can be generated by non-critical ectoinstructions.

Stack-indistinguishability. What we actually require from the `if_skip` and `restart` ectoinstructions is the fact that the ectocode sequence that they define, when inhibited, effectively handles the operand stack the same way they do when executed. In other words, when the XμP enters an `if_skip` sequence, ectoinstructions will manipulate the stack regardless their being executed or not. Thus, provoking an 'empty stack' interrupt is watertight, because it would occur whatever the mode (skip or execution) the XμP is actually works in. Additionally, such a mechanism completely alleviates the constraint of having stack-level invariant `if_skip` sequences as discussed above.

Putting it all together. Taking all the above into account, the simplest way of implementing the skip mode consists in

– Aborting the protocol (cheating terminal) when a security-critical, branch or jump ectoinstruction is encountered after an `if_skip` and before a `restart`.

– Inhibiting memory-writing: a `store` x ectoinstruction behaves the same way as in execution mode except that $\text{RAM}[x]$ is left unchanged and its privacy bit $\varphi(\text{RAM}[x])$ reset to one,
– Letting arithmetical, logical and transfer operators (other than `store` x) act on the operand stack exactly the same way they do in execution mode, except that the privacy bit of all variables pushed onto the stack is automatically set to one.

Finally, note that we do not catalog the ectoinstructions `if_skip` and `restart` as security-critical.

10 Indirect Addressing: `loadi` and `storei` Ectoinstructions

The XJVML language we have been investigating so far does not allow indirect addressing. Namely, one cannot transfer from memory to the operand stack (or the other way around) the contents of a variable whose address is itself a variable. The purpose of this section is to show how the ectoinstruction set and security policy of the XμP can be extended to allow indirect addressing.

DESCRIPTION. We denote by `loadi` x and `stori` x the indirect versions of ectoinstructions `load` x and `store` x, whose dynamic semantics are defined in Figure 28.

INS_i	**effect on** i	**effect on** RAM	**effect on** ST	**effect on** s
`loadi` x	$i \leftarrow (i+1)$	none	$\text{ST}[s+1] \leftarrow \text{RAM}[\text{RAM}[x]]$	$s \leftarrow (s+1)$
`stori` x	$i \leftarrow (i+1)$	$\text{RAM}[\text{RAM}[x]] \leftarrow \text{ST}[s]$	$\text{ST}[s] \leftarrow$ **undef**	$s \leftarrow (s-1)$

Fig. 28. Dynamic Semantics of Indirect Addressing Transfers `loadi` and `storei`

These ectoinstructions are properly executed only when $\text{RAM}[x]$ contains data compatible with the format of a memory address ("falling off" RAM is not allowed). Therefore, the value of $\text{RAM}[x]$ is transparently converted into a valid RAM address right before the transfer becomes effective: for instance, if $|\text{RAM}|$ denotes the size of the XμP's volatile memory space, the value of $\text{RAM}[x]$ could be reduced modulo $|\text{RAM}|$ before transferring data to or from this address.

RELATED SECURITY POLICY. Obviously, care must be taken when the contents handled by a `loadi` or `storei` is private. Similarly, privacy must be conserved also when the address variable $\text{RAM}[x]$ itself is private. We devise the XμP-internal security policy given in Figure 29.

INS_i	**effect on** Φ
`loadi` x	$\varphi(\text{ST}[s+1]) \leftarrow \varphi(\text{RAM}[x]) \vee \varphi(\text{RAM}[\text{RAM}[x]])$
`stori` x	$\varphi(\text{RAM}[\text{RAM}[x]]) \leftarrow \varphi(\text{ST}[s]) \vee \varphi(\text{RAM}[x])$

Fig. 29. Dynamic Semantics of `loadi` and `storei` Over Φ

What the security policy we have chosen means is that, for both ectoinstructions, the privacy bit updated during execution (in RAM for `stori` x, on the stack for `loadi` x) depends not only on the privacy of the transferred data but also on the privacy of the address hosting the data. An illustrative example of this paradigm is the following:

Assume the ectocode works with a non-private S-box S and that at some point the value $S[k]$ is required, where k is (directly related to) a private key. As S is publicly known, $S[k]$ provides information about k meaning that $S[k]$ itself has to be treated as a secret data precisely because of the secrecy of the indirection k. For the same reason, pushing onto the stack an element of a private table T located at a non-private index j in T, the stacked value $T[j]$ must be considered private as it obviously reveals information about T.

SECURITY CRITICALITY. As `loadi` and `storei` operate only on the device's volatile memory, they are not considered security-critical. This consideration holds under the hypothesis that memory locations dedicated to specific processing operations (RNG, IO, stack, ...) cannot be accessed via these instructions, which are consequently limited to general-purpose memory cells.

11 Reading ROM Tables

Reading constant data tables from ROM is a very frequent operation. While the treatment of this operation is in principle similar to the execution of any other ectoinstruction, here particular care must be taken to allow the XμP to authenticate the contents of ROM tables as a proper part of the ectoprogram. We propose two mechanisms for doing so, depending on the way a given ROM table is accessed.

11.1 Accessing Privately Located Entries

We assume that the XμP's ectocode works with an array T of absolute constants so that during computation, T is accessed at a variable location j. In this respect, we rely on the indirect addressing mode provided by `loadi` x as follows. The ectocode writes successively at consecutive RAM addresses the constants $T[0], T[1], \ldots, T[n]$ using regular XJVML ectoinstructions. Keeping the address add_T of $T[0]$ in memory, $T[j]$ is accessed given any j using `loadi` x where RAM$[x]$ is previously initialized to $\mathsf{add}_T + j$. This simple mechanism is effective whatever the privacy status of j is; its only limitation resides in the size of the XμP's volatile memory, *i.e.* one cannot have $n > |\text{RAM}|$.

11.2 Accessing Non-Private Locations

We now turn to the description of a second mechanism by which the ectoprogram can access the table T without having to store its entire contents in RAM. Access to T will only be possible at non-private, immediate locations.

We extend our ectoinstruction-set to include a specific table-reading operator: the ectoinstruction `push` add_T`,` j, where add_T is the address of T located in ROM (*i.e.* in the XT) and j a constant. The ectoinstruction is implemented as follows (assuming authentication at the ectoinstruction level):

- The XμP sends i to the XT and gets $\mathsf{INS}_i = $ `push` add_T`,`j in response;
- The XμP requests the ROM contents corresponding to (add_T, j);
- The XT replies with $T[j]$ and updates

$$\sigma \leftarrow \sigma \times \sigma_{\langle \texttt{push } \mathsf{add}_T, j\rangle} \mod N \quad \text{or} \quad \sigma \leftarrow H(\sigma, \sigma_{\langle \texttt{push } \mathsf{add}_T, j\rangle}) ,$$

while the XμP updates

$$\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \langle \texttt{push add}_T, j\rangle, T[j]) \mod N$$

or

$$\nu \leftarrow H(\nu, \mu_K(\mathsf{ID}, i, \langle \texttt{push add}_T, j\rangle, T[j]) ,$$

depending on the chosen execution protocol.

Thus, the contents of T are authenticated by the same technique as for ectoinstructions. The push add_T, j operation never requires to trigger a CheckOut on execution since j is inherently non-private. Hence we do not add this instruction to $\mathcal{S}$. Note that when Protocol 3 is implemented, the pre-execution phase has to access $T[j]$ while MAC-ing the ectoinstruction push add_T, j.

12 A Software Example: Ectoprogramming RC4

12.1 Extra Ectoinstructions

Before giving the ectocode of a very-basic implementation of the RC4 for the XμP, we introduce a handful of new ectoinstructions that are equivalent in term of security to other ectoinstructions in our XJVML language. For each of these new ectoinstructions, we give an ectoinstruction which effect on Φ is equivalent.

INS_i	**effect on** i	**effect on** RAM	**effect on** ST	**effect on** s	φ **equivalence**
push v	$i \leftarrow (i+1)$	none	$\mathrm{ST}[s+1] \leftarrow v$	$s \leftarrow (s+1)$	push0
add	$i \leftarrow (i+1)$	none	$\mathrm{ST}[s-1] \leftarrow \mathrm{ST}[s-1] + \mathrm{ST}[s]$	$s \leftarrow (s-1)$	xor
add256	$i \leftarrow (i+1)$	none	$\mathrm{ST}[s-1] \leftarrow \mathrm{ST}[s-1] + \mathrm{ST}[s] \bmod 256$	$s \leftarrow (s-1)$	xor
mod	$i \leftarrow (i+1)$	none	$\mathrm{ST}[s-1] \leftarrow \mathrm{ST}[s] \bmod \mathrm{ST}[s-1]$	$s \leftarrow (s-1)$	mul

Fig. 30. Some More Ectoinstructions.

We remind that RC4 is a stream cipher devised by RSA Data Security: its specifications can be found in [25].

12.2 Ectoprogram and Brief Analysis

The ectoprogram works as follows: parts 1, 2 and 3 implement the key schedule whilst the fourth and last part is dedicated to the encryption function itself. First, LoopA is very simple: it uses two counters, stored in RAM[259] and RAM[260]: the first is a value running down from 256 to 0, while the second runs up from 0 to 256. RAM[259] is in fact the loop index. RAM[260] is the value stored in a buffer called RC4-STATE, used for key schedule. The value is stored from RAM[0] to RAM[255].

Once the initialization step is done, the ectoprogram uses RC4-KEY (which is supposed to be stored from NVM[0] to NVM[8], with NVM[0] = 8 corresponding to the key length). It copies this key RC4-KEY into RAM, from RAM[300] to RAM[307]. Finally, it initializes a certain number of counters in RAM: $x =$ RAM[257], $y =$ RAM[258], $i_1 =$ RAM[260], $i_2 =$ RAM[261], $i =$ RAM[263] are all reset to zero; RAM[259] is initialized to 256.

LoopB is the key schedule's second step: RC4-KEY[i_1] is stored in RAM[262]. Then, RC4-STATE[i] is loaded, and RC4-STATE[i]+RC4-KEY [i_1]$+i_2$ mod 256 is computed

Code for a basic RC4: Key Schedule and Cipher

(part 1)

```
Key Schedule:
push0
store       260
push        256
store       259

LoopA:
load        260
stori       260
load        260
inc
store       260

load        259
dec
store       259
load        259
if          LoopA
goto        Part2
```

(part 2)

```
Part2:
getstaticx    0
store       264
getstaticx    1
store       300
getstaticx    2
store       301
getstaticx    3
store       302
getstaticx    4
store       303
getstaticx    5
store       304
getstaticx    6
store       305
getstaticx    7
store       306
getstaticx    8
store       307

push0
store       257
push0
store       258
push0
store       260
push0
store       261

push0
store       263

push        256
store       259

goto        LoopB
```

(part 3)

```
LoopB:
push        300
load        260
add
store       262
loadi       262

loadi       263

add
load        261
add256
store       261

loadi       263
store       262
loadi       261
stori       263
load        262
stori       261

load        264
load        260
inc
mod
store       260

load        263
inc
store       263

load        259
dec
store       259
load        259
if          LoopB
goto        Cipher
```

(cipher)

```
Cipher:
load         IO
store       259

LoopC:
load        257
push          1
add256
store       257

load        258
loadi       257
add256
store       258

loadi       257
store       260
loadi       258
stori       257
load        260
stori       258

loadi       258
loadi       257
add256
store       260

load         IO
loadi       260
xor
store        IO

load        259
dec
store       259
load        259
if          LoopC
halt
```

Fig. 31. Software Example: Ectoprogramming an RC4

and stored in RAM[261]. Follows the exchange of RC4-State[i] and RC4-State[i_2], through a temporary memory variable RAM[262]. i_1 is incremented and taken modulo the key length stored in RAM[264]. Finally, i is incremented and the loop counter (in RAM[259]) is decremented. The loop is re-done if this counter is nonzero.

The last part is the stream cipher itself: the ectoprogram loads (from the IO) the length of the plaintext to encrypt and stores it in RAM[259] then it begins a loop as long as the plaintext to encrypt: It loads x (in RAM[257]) and increments it modulo 256. It updates y (in RAM[258]) by adding to it RC4-State[x] modulo 256, exchanges RC4-State[x] and RC4-State[y], using a temporary variable RAM[260], computes

$$\text{RC4-State}[x] + \text{RC4-State}[y] \mod 256$$

and stores this quantity in $t = $ RAM[260]. Finally the ectoprogram gets from the IO the value to encrypt: it just xors this value with RC4-State[t], and sends the encrypted byte to the IO. Finally, it decrements the loop counter, and resumes the loop if needed.

12.3 How many CheckOuts are needed?

It appears that the enforcement of the security policy in the above example slows-down execution only negligibly: indeed, during execution the authors noticed that the XT was only asked for signatures during the `store IO` phase.

This drove us to introduce yet another improvement in the device, namely an ectoinstruction allowing to send not only one value to the IO, but an array. In our case, this would reduce the number of signature queries from one per byte to just one for the *entire* message. This ectoinstruction is described in Figure 32.

INS_i	effect on i	effect on RAM	effect on ST	effect on s
`export`	$i \leftarrow (i+1)$	$\text{IO} \leftarrow \text{ST}[s-1]$ $\text{IO} \leftarrow \text{ST}[s-2]$ $\ldots$ $\text{IO} \leftarrow \text{ST}[s-\text{ST}[s]]$	$\text{ST}[s-1] \leftarrow$ `undef` $\text{ST}[s-2] \leftarrow$ `undef` $\ldots$ $\text{ST}[s-\text{ST}[s]] \leftarrow$ `undef` $\text{ST}[s] \leftarrow$ `undef`	$s \leftarrow (s-\text{ST}[s]-1)$

Fig. 32. Store Large Results on IO.

This instruction, security-critical, would then be treated like a simple `store` IO, except that the CheckOut is triggered when one (or more) of the privacy bits

$$\varphi(\text{ST}[s]), \varphi(\text{ST}[s-1]), \ldots, \varphi(\text{ST}[s-\text{ST}[s]])$$

is equal to one *i.e.*:

$$\bigvee_{i=0}^{\text{ST}[s]} \varphi(\text{ST}[s-i]) = 1$$

13 Deployment Considerations

From a practical engineering perspective, the new architecture is likely to deeply impact the card industry. Today, this industry's interests (the endocode *i.e.* the mask's contents) are inherently protected against alien scrutiny by the card's tamper-resistant features initially meant to protect the *client's* NVM secrets. By deploying ectocode in terminals, the card manufacturers' role is likely to evolve and focus on personalization. The card's intelligence being entirely in the terminal, terminal manufacturers will gain independence and face the usual challenges of the software industry (separation between code and hardware, ectocode must be protected by obfuscation against reverse-engineering *etc*).

This section attempts to foresee a few expectable consequences of the concept introduced in this paper.

13.1 Speed Versus Code Size

A dilemma frequently faced by smart card programmers is that of striking an effective balance between endocode size and speed. The main cost-factor in on-board ROM is not storage itself but the physical hardening of this ROM against external attacks. Given that in the new architecture external (*distrusted* and hence cheaper) virtually unlimited ROM can be used to securely store ectocode, ectocode can be optimized for speed. 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.*

13.2 Code Patching

One of the major advantages of the XμP is the fact that a bug in an ectoprogram does not imply the roll-out of devices in the field but a simple terminal update.

The bug patching mechanism that we propose consists in encoding in ID a backward compatibility policy signed by the CI that either instructs the XμP to replace its old

ID by a new one and stop accepting older version ectoprograms or allow the execution of new or the old ectocode (each at a time, *i.e.* no blending possible). The description of this mechanism is straightforward and omitted here.

In any case, the race against hackers becomes much easier. In a matter of hours the old ectocode can be rolled out whereas today, card roll-out can take months or even years. Patching a future smart card can hence become as easy as patching a PC.

13.3 Code Secrecy

It is a common practice in the telecom Industry to use proprietary A3A8 algorithms [24]. Given that the XT contains the application's code, our architecture assumes that the algorithm's specifications are public.

It should be pointed out that while the practice of keeping algorithms secret *does not* fall under the standard setting within which system security is traditionally assessed[30], it is still possible to reach *some* level of ectocode secrecy by encrypting the XT's ectocode under a key (common to all XμPs). 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 ectocode could remain confidential if programmed appropriately. Note that compromising one XμP will reveal the ectocode's encryption key, but this is no different from the traditional smart-card setting where a successful attack on one card suffices to reveal the endocode common to all cards. Also, it should be stressed that compromising the ectocode encryption key does not allow to feed the XμP with aggressive xenocode (ectocode integrity and ectocode confidentiality being two different functions). Finally, from a practical standpoint it is expected that current SIM cards will be progressively replaced by 3G ones on the long run. 3G uses a public AES-based authentication algorithm (Milenage) which specifications are public [23].

However, from the user's perspective, the authors consider that the XμP architecture offers much better (yet not perfect) privacy guarantees against back-doors by exposing the executable ectocode to public scrutiny[31]. We assume that the mere possibility for a user to inspect the exchanges between his XμP and the XT offer privacy guarantees that stretch far beyond those offered by traditional smart-cards.

13.4 Limited Series

Consider a Swede traveling to China and using his card in an ATM there. Using current technology, a mask deployed in Sweden can contain user instructions[32].

If the user card's were an XμP, Chinese terminals would have to *also* contain user instructions in Swedish (in fact, in any possible language) or, alternatively, user instructions should have been personalized in the device's NVM.

13.5 Simplified Stock Management

Given that a GSM XμP and an electronic-purse XμP differ only by a few NVM bytes (essentially ID), by opposition to smart-cards, XμPs are real commodity products (such

[30] Security must stem from the key's secrecy and not from the algorithm's confidentiality.

[31] We do not get here into the philosophical debate of wether or not the ectocode input into the device is indeed the one executed by the device.

[32] ASCII strings such as "`Insert card`" or "`Enter PIN code`" in Swedish ($\equiv$ "`Stoppa in kortet`", "`Mata in din personliga kod`").

as capacitors, resistors or Pentium processors) which stock management is greatly simplified and straightforward.

In essence, when a card manufacturer[33] finishes the coding of a traditional off-the-shelf mask (*e.g.* a SIM card), the card manufacturer buys a few millions of masked chips from the chip manufacturer[34] and constitutes a *stock*. This stock is an important risk factor as the card manufacturer must forecast sales with accuracy: a market downturn, a standard change or unrealistic marketing plans can cause very significant financial losses.

When constituting an XμP stock the risk is greatly reduced. The only important factor is the card manufacturer's *global* sales volume (per NVM size), which is *much easier* to forecast than the sales volume per product.

For MAC-based XμPs, the manufacturer can even migrate into the device's ROM the list $\{H(P_i)\}$ corresponding to the entire company history: *i.e.* the hash values of all applications coded by the manufacturer so far - 160 bits per application. At personalization time, the manufacturer can simply burn into the device the index i that enables the execution of a given P_i.

Alternatively, the XT can contain a digital signature on $\{i, H(P_i)\}$ and the XμP can dispense with the storage of $H(P_i)$.

13.6 Reducing The Number of Cards

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.

13.7 Faster Prototyping

Note that a PC, a reader and an off–the-shelf application-independent XμP are sufficient for prototyping applications.

14 Engineering and Implementation Options

A large gamut of trade-offs and variants is possible when implementing the architecture described in this paper. This section describes a few such options.

14.1 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.

[33] *e.g.* Gemplus, Oberthur, G&D or Axalto.
[34] *e.g.* Philips, Infineon, ST Microelectronics, Atmel or Samsung.

14.2 Speeding The Accumulation With Fixed Padding

Speeding-up the operation $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, h) \bmod N$ is crucial for the efficiency of the protocols proposed in this paper. This paragraph suggests a candidate μ for which the accumulation operation is particularly fast:

$$\mu(x) = 2^\kappa + h(x) \quad \text{for} \quad 2^\kappa < N < 2^{\kappa+1} \quad \text{and} \quad 2^{\frac{\kappa}{4}} < h(x) < 2^{\frac{\kappa}{4}+1}$$

Indeed, any attack against this padding function will improve the $\frac{\kappa}{3}$ fixed padding bound described in [10].

The advantage of this padding function is that while the multiplication of two κ-bit integers requires κ^2 operations, multiplying a random ν by $\mu(x)$ requires only $\kappa^2/4$ operations.

14.3 Using a Smaller e

Implementers wishing to use a smaller e can use several t counters and a hash function h. Here the idea is that instead of incrementing t, the $\mathsf{X}\mu\mathsf{P}$ increments $t_{h(i,\mathsf{INS}_i)}$. Whenever any of the t_j-counters reaches $e-1$ the $\mathsf{X}\mu\mathsf{P}$ triggers a $\mathsf{CheckOut}$. Denoting by λ the size of h's digests (in bits), one $\mathsf{CheckOut}$ per $e \times 2^{\lambda-1}$ ectoinstruction queries will be expectedly triggered, on the average.

14.4 Smart Usage of Security Hardware Features

Most of secure tokens in use today contain hardware-level countermeasures thwarting physical attacks relying on power analysis or related techniques. As detailed in the past sections, the $\mathsf{X}\mu\mathsf{P}$ essentially runs in two modes, depending on the privacy bit of the current variable being processed (unless the $\mathsf{X}\mu\mathsf{P}$ is parallelized, it is guarantied that only one variable is processed at a given point in time). When the current variable is non-private, an attacker is theoretically capable of recovering its value by symbolically executing the transmitted piece of ectocode related to this variable. Being vacuous, hardware protections shall not necessarily be operating at that moment. On the contrary, the $\mathsf{X}\mu\mathsf{P}$ could (selectively?) activate these protections whenever a private variable is handled or forecasted to be used a few cycles later.

14.5 High Speed XIO

A high-speed communication interface is paramount for servicing the extensive information exchange between the $\mathsf{X}\mu\mathsf{P}$ and the XT .

Let $|\mathsf{INS}|$ and $|i|$ respectively denote the bitsizes required to encode the ectoinstructions and their addresses in the XT. A typical example being $|\mathsf{INS}| = |i| = 32$. We denote by $\mathsf{TrT}(n)$ the time required to exchange n bits between the $\mathsf{X}\mu\mathsf{P}$ and the XT and by ExT the average time it takes to execute an instruction[35] (*latency*).

Then, the $\mathsf{X}\mu\mathsf{P}$'s external operating frequency f_{ext} is:

$$f_{\text{ext}} = \frac{1}{\mathsf{TrT}(|\mathsf{INS}| + |i|) + \mathsf{ExT}} \quad \text{Hz}$$

[35] including the computation of $\nu \leftarrow \nu \times \mu(\mathsf{ID}, i, \mathsf{INS}_i) \bmod N$.

While the machine is actually run internally at:

$$f_{\text{int}} = \frac{1}{\mathsf{ExT}} \quad \text{Hz}$$

One can remark that whenever the ectoinstruction is not a test (`if` L or `if_phi` L) or a `goto`, addresses are just incremented by one. It follows that transmission can be significantly slashed in most cases as the sending of i becomes superfluous. This observation also allows to parallelize the execution of INS_i and the reception of INS_{i+1} for most ectoinstructions.

More specifically, even when INS_i is a test the XT can still send to the XμP the ectoinstruction that would be queried next if the test were negative. Should the test be positive (miscache) the XμP can simply send a control bit to the XT who will reply with the correct successor of INS_i[36].

Neglecting the miscache bit's cost, the frequency formula becomes:

$$f_{\text{ext}} = \frac{1}{(1-p) \times \max\{\mathsf{TrT}(|\mathsf{INS}|), \mathsf{ExT}\} + p \times \max\{\mathsf{TrT}(|\mathsf{INS}| + |i|), \mathsf{ExT}\}} \quad \text{Hz}$$

where p is the average proportion of `goto`s in the code.

For the sake of illustration, we evaluated the above formula for a popular standard, the Universal Serial Bus (USB). Note that USB is unadapted to our application as this standard was designed for good bandwidth rather than for good latency.

In USB High Speed mode transfers of 32 bits can be done at 25 Mb/s which corresponds to 780K 32-bit words per second. When servicing our basic protocol, this corresponds approximately to a 32-bit XμP working at 390 KHz; when parallel execution and transmission take place, one gets a 32-bit machine running at 780 KHz.

An 8-bit USB XμP (where transfers of 8 bits can be done at 6.7 Mb/s), would correspond to 830K 8-bit words per second. This yields a parallel execution and transmission 8-bit machine running at 830 KHz.

15 Further Research

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 ectocode by competing parties (*i.e.* mechanisms for the secure handover of execution from ectoprogram ID_1 to ectoprogram ID_2), the devising of faster ectoexecution protocols or the improvement of those described earlier in this paper.

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.

16 Acknowledgements

The authors are indebted to Julien Brouchier, Éric Deschamps, Markus Jakobsson, Anne-Marie Praden, Ludovic Rousseau, Jacques Stern as well as anonymous reviewers of CHES'04 for their useful remarks and feedback on this work.

[36] Note that we have chosen the negative test to be the fast one as tests are mostly used in loops. Hence the protocol is optimal for all loop iterations except the last; the reverse choice would have slowed-down all the loop tests except the last one.

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, *PSS: Provably Secure Encoding Method for Digital Signatures*, Submission to IEEE P1363a, August 1998, `http://grouper.ieee.org/groups/1363/`.
6. 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.
7. M. Bellare and P. Rogaway, *The Exact Security of Digital Signatures – How to Sign with RSA and Rabin*, Eurocrypt'96, LNCS 1070, pages 399–416. Springer-Verlag, Berlin, 1996.
8. B. Chevallier-Mames, D. Naccache, P. Paillier and D. Pointcheval, *How to Disembed a Program?*, CHES 2004, Springer-Verlag, 2004.
9. G. Bilardi and K. Pingali, *The Static Single Assignment Form and its Computation*, Cornell Univ. Technical Report, 1999, `www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps`.
10. É. Brier, C. Clavier, J.-S. Coron and D. Naccache, *Cryptanalysis of RSA signatures with fixed-pattern padding* par In Advances in Cryptology – Crypto 2001, LNCS 2139, pages 433–439, Springer-Verlag, 2001.
11. Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, The Java Series, Addison-Wesley, 2000.
12. B. Chevallier-Mames, M. Ciet and M. Joye, *Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity*, Cryptology ePrint Archive, Report 2003/237, `http://eprint.iacr.org/`.
13. J.-S. Coron, *On the Exact Security of Full-Domain-Hash*, Crypto'2000, LNCS 1880, Springer-Verlag, Berlin, 2000.
14. 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.
15. D.E. Knuth, *The Art of Computer Programming, vol. 1, Seminumerical Algorithms*, Addison-Wesley, Third edition, pages 124–185, 1997.
16. S. Goldwasser, S. Micali, and R. Rivest. A "Paradoxical" Solution to the Signature Problem. In *Proc. of the 25th FOCS*, pages 441–448. IEEE, New York, 1984.
17. S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptative Chosen-Message Attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.
18. O. Kommerling and M. Kuhn, *Design principles for tamper-resistant smartcard processors*, Proceedings of USENIX Workshop on Smartcard Technology, 1999, pp. 9–20.
19. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
20. G. Ramalingam, *Identifying Loops in Almost Linear Time*, ACM Transactions on Programming Languages and Systems, 21(2):175-188, March 1999.
21. R. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, 21(2):120–126, February 1978.
22. 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/`.
23. TS 35.206, *"3G Security; Specification of the MILENAGE algorithm set: An example algorithm Set for the 3GPP Authentication and Key Generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Algorithm specification"*, `http://www.3gpp.org/ftp/Specs/html-info/35206.htm`.
24. K. Vedder, *GSM: Security, Services, and the SIM*, State of the Art in Applied Cryptography, LNCS 1528, pages 224–240, 1997.
25. RC4 Stream Cipher, `http://burtleburtle.net/bob/rand/isaac.html#RC4`, `http://www.wisdom.weizmann.ac.il/~itsik/RC4/rc4.html`.

A Unforgeability of FDH-RSA Screening (Proof of Theorem 1)

We treat separately the case of passive and active attacks.

A.1 Known Message Attacks

In the passive attack model, we consider a forger $\mathcal{F}$ allowed to make q_h queries to the hash oracle h and q_k known-message queries and outputting a list of t messages $(m_1, \dots, m_t)$ as well as $\sigma < N$. We assume that with probability at least ε,

$$\sigma^e = \prod_{i=1}^{t} h(m_i) \bmod N ,$$

whereas the signature of at least one of the messages $m_1, \dots, m_t$ has never been provided to $\mathcal{F}$, meaning that the e-th root $h(m_j)^d$ of $h(m_j)$ for some $1 \le j \le t$ is unknown to $\mathcal{F}$. We show how to use this adversary to break RSA. More precisely, we build a reduction $\mathcal{R}$ that uses $\mathcal{F}$ to compute the e-th root of an arbitrary $y \in \mathbb{Z}_N^\star$ with probability $\varepsilon' = \varepsilon$.

The reduction $\mathcal{R}$ works as follows. On input (y, e, N) where e is a prime number, $\mathcal{R}$ invokes $\mathcal{F}$ and transmits (e, N) to $\mathcal{F}$. Then $\mathcal{R}$ simulates the random oracle h as well as the signing oracle S_k which returns upon request up to q_k message-signature pairs (m_i, σ_i) with $\sigma_i^e = h(m_i) \mod N$. These simulations are performed as follows.

Simulation of S_k. Each time $\mathcal{F}$ requests a message-signature pair, $\mathcal{R}$ chooses (according to any arbitrary distribution) some message $m \in \{0,1\}^\star$ such that m does not appear in $\mathcal{R}$'s transcript, picks a random $r \in \mathbb{Z}_N^\star$, defines $h(m) \leftarrow r^e \bmod N$, updates its transcript accordingly, and outputs the pair (m, r).

Simulation of h. Whenever $\mathcal{F}$ requests $h(m)$ for some $m \in \{0,1\}^\star$, $\mathcal{R}$ checks in its transcript if $h(m)$ is already defined, in which case $h(m)$ is returned. If $h(m)$ is undefined, $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$, defines $h(m) \leftarrow r^e y \mod N$, updates the transcript and returns this value to $\mathcal{F}$.

These simulations never fail to respond to $\mathcal{F}$'s queries and the distributions of answers are statistically indistinguishable from the ones $\mathcal{F}$ expects. After at most q_h hash queries and q_k message-signature queries, $\mathcal{F}$ outputs $(m_1, \dots, m_t)$ and σ within some time bound τ. Then $\mathcal{R}$ queries $h(m_i)$ for $i = 1, \dots, t$ to its own simulation of h and checks whether $\sigma^e = \prod_{i=1}^t h(m_i) \bmod N$.

Extraction of y^d. Since each and every message $m_i \in \{m_1, \dots, m_t\}$ has been queried to the hash oracle (either by $\mathcal{F}$ or $\mathcal{R}$), $\mathcal{R}$ knows an r_i such that $h(m_i) = r_i^e y \bmod N$ or $h(m_i) = r_i^e \mod N$. Letting A (respectively B) denote the set of indices i such that $h(m_i) = r_i^e y \mod N$ (resp. $h(m_i) = r_i^e \mod N$), we know that the messages m_i for $i \in B$ are among the ones given by $\mathcal{R}$'s simulation of S_k to $\mathcal{F}$ throughout the experiment. By definition of $\mathcal{F}$, $B \subsetneq \{1, \dots, t\}$ and hence $|A| \neq 0$. Consequently, if the verification succeeds then

$$\sigma^e = \prod_{i \in A} r_i^e y \prod_{i \in B} r_i^e = \left(\prod_{1 \le i \le t} r_i \right)^e y^{|A|} \mod N ,$$

meaning that

$$\left(\sigma/\prod r_i\right)^e = y^{|A|} \mod N \ .$$

Since $0 < |A| \le t < e$ and e is prime, there exist integers (α, β) such that $\alpha|A|+\beta e = 1$. Then

$$y = y^{\alpha|A|+\beta e} = \left(\left(\sigma/\prod r_i\right)^\alpha y^\beta\right)^e \mod N \ ,$$

and $\mathcal{R}$ returns $y^d = (\sigma/\prod r_i)^\alpha y^\beta \bmod N$ with probability one. Summarizing, since $\mathcal{F}$ outputs a valid forgery with probability at least ε within τ steps, our reduction $\mathcal{R}$ returns y^d with probability $\varepsilon' = \varepsilon$ after at most $\tau' = \tau + (q_h + q_k)\mathcal{O}(\log^3 N)$ steps.

A.2 Chosen-Message Attacks

In an active adversarial model, the forger $\mathcal{F}$ is allowed, in addition to h and S_k, to query at most q_c times a signing oracle S_c for messages of her choosing. Relying on a technique introduced by Coron [13], we modify the reduction $\mathcal{R}$ as follows.

Simulation of h. Whenever $\mathcal{F}$ requests $h(m)$ for some $m \in \{0,1\}^\star$, $\mathcal{R}$ checks if $h(m)$ is already defined, in which case $h(m)$ is returned. If $h(m)$ is undefined, $\mathcal{R}$ selects a random bit $b \in \{0,1\}$ with a certain bias δ, *i.e.* b is set to zero with probability δ. Then $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$, memorizes (m, b, r), defines $h(m) \leftarrow r^e y^b \mod N$ and returns this value to $\mathcal{F}$.

Simulation of S_k. The simulation of S_k is unchanged: each time $\mathcal{F}$ requests a message-signature pair, $\mathcal{R}$ chooses some arbitrary, fresh message $m \in \{0,1\}^\star$ and a random $r \in \mathbb{Z}_N^\star$, defines $h(m) \leftarrow r^e \bmod N$, memorizes $(m, 0, r)$ and outputs the pair (m, r).

Simulation of S_c. When $\mathcal{F}$ requests the signature of $m \in \{0,1\}^\star$, $\mathcal{R}$ checks in its own transcript if some value is defined for $h(m)$. If $h(m)$ is undefined, $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$, defines $h(m) \leftarrow r^e \mod N$, memorizes $(m, 0, r)$ and returns r. Otherwise the transcript contains a record $(m, b \in \{0,1\}, r)$. If $b = 0$, $\mathcal{R}$ returns r. If $b = 1$, $\mathcal{R}$ aborts.

Again, the simulations of h and S_k are perfect. However the simulation of S_c may provoke an abortion before the game comes to an end. Let us assume that no abortion occurs. After at most q_h hash queries, q_k known-message queries and q_c chosen-message queries, $\mathcal{F}$ outputs $(m_1, \ldots, m_t)$ and σ within some time bound τ. Then, again, $\mathcal{R}$ queries $h(m_1), \ldots, h(m_t)$ to its own simulation of h and checks if $\sigma^e = \prod_{i=1}^t h(m_i) \bmod N$.

Extraction of y^d. Every message $m_i \in \{m_1, \ldots, m_t\}$ corresponds in the transcript to a pair (b_i, r_i) such that $h(m_i) = r_i^e y^{b_i} \mod N$. By definition of $\mathcal{F}$, there is at least one message m_j that was neither output by S_k nor queried to S_c. Suppose that $b_j = 1$ and that the verification is successful. Then

$$\sigma^e = \prod r_i^e y^{b_i} = \left(\prod r_i\right)^e y^{\sum b_i} \mod N \ ,$$

with $\sum b_i \ge b_j = 1$. Since $0 < \sum b_i \le t < e$ and e is prime, there exist integers (α, β) with $\alpha \sum b_i + \beta e = 1$. Then $\mathcal{R}$ returns $y^d = (\sigma/\prod r_i)^\alpha y^\beta \bmod N$ with probability one.

Reduction Cost Analysis. Our reduction $\mathcal{R}$ succeeds with probability (taken over the probability spaces of $\mathcal{F}$ and $\mathcal{R}$):

$$\begin{aligned}\varepsilon' &= \Pr[\mathcal{F} \text{ forges} \wedge \neg\text{abortion} \wedge b_j = 1] \\ &= \Pr[\mathcal{F} \text{ forges} \wedge \neg\text{abortion}] \Pr[b_j = 1] \\ &= \Pr[\mathcal{F} \text{ forges} \mid \neg\text{abortion}] \Pr[\neg\text{abortion}] \Pr[b_j = 1] \\ &= \varepsilon\delta^{q_c}(1-\delta) \ ,\end{aligned}$$

where the equalities stem from the pairwise independence of the random coins b and their independence from the forger's view. The optimal value for $\delta^{q_c}(1-\delta)$ is reached for $\delta = 1 - 1/(q_c + 1)$. Then,

$$\varepsilon' = \frac{\varepsilon}{q_c}\left(1 - \frac{1}{q_c+1}\right)^{q_c+1} \geq \frac{\varepsilon}{4q_c} \quad \text{for } q_c \geq 1 \ .$$

The reduction $\mathcal{R}$ returns y^d or aborts within time bound $\tau' = \tau + (q_h + q_k + q_c)\mathcal{O}(\log^3 N)$ steps.

B Unforgeability of (FDH, H)-RSA Screening (Proof of Theorem 6)

Theorem 6. *We set $\mu(a,b,c) = h(a\|b\|H(c))$ where h is a full-domain hash function seen as a random oracle. Then μ-RSA is existentially unforgeable under a known-message attack assuming that RSA is hard and H is collision-intractable.*

Proof. We build a reduction algorithm $\mathcal{R}$ that uses an $(q_h, q_k, t, \tau, \varepsilon)$-forger $\mathcal{F}$ to compute the e-th root of $y \in \mathbb{Z}_N^\star$ with probability ε'_1 and simultaneously a collision of H with probability ε'_2, where $\varepsilon'_1 + \varepsilon'_2 \geq \varepsilon$. The reduction $\mathcal{R}$ works as follows. Given (y, e, N), e prime, $\mathcal{R}$ transmits (e, N) to $\mathcal{F}$ and simulates the random oracle h and the signing oracle S_k as follows.

Simulation of S_k. Upon request, $\mathcal{R}$ chooses some arbitrary fresh message $m = a\|b\|c \in \{0,1\}^\star$, computes $\gamma = H(c)$, and makes sure that $a\|b\|\gamma$ does not appear in the transcript. If it does, the simulation of S_k is restarted. Otherwise, $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$, defines $h(a\|b\|\gamma) \leftarrow r^e \bmod N$, memorizes $\langle m, a\|b\|\gamma, r\rangle$ and outputs the pair (m, r).

Simulation of h. Whenever $\mathcal{F}$ requests $h(a\|b\|\gamma)$ for some triple $(a, b, \gamma) \in \{0,1\}^{|a|} \times \{0,1\}^{|b|} \times \mathsf{Im}(H)$, $\mathcal{R}$ checks if $h(a\|b\|\gamma)$ is already defined, in which case the value defined is returned to $\mathcal{F}$. Otherwise, $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$, memorizes $\langle \perp, a\|b\|\gamma, r\rangle$, defines $h(a\|b\|\gamma) \leftarrow r^e y \bmod N$ and returns $h(a\|b\|\gamma)$ to $\mathcal{F}$.

These simulations are perfect. After some time τ, $\mathcal{F}$ outputs σ and a t-uple $(m_1, \ldots, m_t)$ with $m_i = a_i\|b_i\|c_i$. Then $\mathcal{R}$ queries $h(a_i\|b_i\|H(c_i))$ for $i = 1, \ldots, t$ to the simulation of h and tests whether $\sigma^e = \prod_{i=1}^t h(a_i\|b_i\|H(c_i)) \bmod N$.

Extraction of y^d or extraction of a collision in H. Assume that $\mathcal{F}$ outputs a correct forgery. Then there is for each message $m_i \in \{m_1, \ldots, m_t\}$ at least one record $\langle x_i, a_i\|b_i\|H(c_i), r_i\rangle$ that appears in the transcript where $x_i \in \{m_i, \perp\}$. The messages m_i for which $x_i = m_i$ were given by the simulation of S_k to $\mathcal{F}$ during the experiment. Noting $A = \{i \mid x_i = \perp\}$, two cases appear.

$|A| \neq 0$: then $\mathcal{R}$ returns $y^d = (\sigma / \prod r_i)^\alpha y^\beta \bmod N$ where $\alpha|A| + \beta e = 1$;

$|A| = 0$: because at least one message $m_j = a_j \| b_j \| c_j$ for $j = 1, \ldots, t$ appearing in the forgery was not output by S_k, we get that the record $\langle x_j, a_j \| b_j \| H(c_j), r_j \rangle$ contains a message $x_j = a_j \| b_j \| c'_j$ featuring $H(c'_j) = H(c_j)$. $\mathcal{R}$ then outputs $\mathsf{coll} = (c_j, c'_j)$.

Reduction Cost Analysis. Letting $\varepsilon'_1 = \Pr[\mathcal{R} \text{ outputs } y^d]$ and $\varepsilon'_2 = \Pr[\mathcal{R} \text{ outputs } \mathsf{coll}]$, we get

$$\begin{aligned}
\varepsilon'_1 &= \Pr[\mathcal{R} \text{ outputs } y^d] = \Pr[\mathcal{R} \text{ outputs } y^d \,|\, \mathcal{F} \text{ forges} \wedge |A| \neq 0] \Pr[\mathcal{F} \text{ forges} \wedge |A| \neq 0] \\
&\quad + \Pr[\mathcal{R} \text{ outputs } y^d \,|\, \neg(\mathcal{F} \text{ forges} \wedge |A| \neq 0)] \Pr[\neg(\mathcal{F} \text{ forges} \wedge |A| \neq 0)] \\
&\geq \Pr[\mathcal{R} \text{ outputs } y^d \,|\, \mathcal{F} \text{ forges} \wedge |A| \neq 0] \Pr[\mathcal{F} \text{ forges} \wedge |A| \neq 0] \\
&= 1 \cdot \Pr[\mathcal{F} \text{ forges} \wedge |A| \neq 0] \ ,
\end{aligned}$$

and

$$\begin{aligned}
\varepsilon'_2 &= \Pr[\mathcal{R} \text{ outputs } \mathsf{coll}] = \Pr[\mathcal{R} \text{ outputs } \mathsf{coll} \,|\, \mathcal{F} \text{ forges} \wedge |A| = 0] \Pr[\mathcal{F} \text{ forges} \wedge |A| = 0] \\
&\quad + \Pr[\mathcal{R} \text{ outputs } \mathsf{coll} \,|\, \neg(\mathcal{F} \text{ forges} \wedge |A| = 0)] \Pr[\neg(\mathcal{F} \text{ forges} \wedge |A| = 0)] \\
&\geq \Pr[\mathcal{R} \text{ outputs } \mathsf{coll} \,|\, \mathcal{F} \text{ forges} \wedge |A| = 0] \Pr[\mathcal{F} \text{ forges} \wedge |A| = 0] \\
&= 1 \cdot \Pr[\mathcal{F} \text{ forges} \wedge |A| = 0] \ ,
\end{aligned}$$

whereby:

$$\varepsilon'_1 + \varepsilon'_2 \geq \Pr[\mathcal{F} \text{ forges} \wedge |A| \neq 0] + \Pr[\mathcal{F} \text{ forges} \wedge |A| = 0] = \Pr[\mathcal{F} \text{ forges}] = \varepsilon \ .$$

The reduction $\mathcal{R}$ returns y^d or coll in at most $\tau' = \tau + (q_h + q_k)\mathcal{O}(\log^3 N)$ steps. □

C Unforgeability of FDH-Rabin-Screening (Proof of Theorem 4)

We only consider the case of passive attacks. In the passive attack model, we consider a forger $\mathcal{F}$ allowed to make q_h queries to the hash oracle h and q_k known-message queries and outputting a list of t messages $(m_1, \ldots, m_t)$ as well as $\sigma < N$. We assume that with probability at least ε,

$$\sigma^2 = \prod_{i=1}^{t} h(m_i) \bmod N \ ,$$

whereas the signature of at least one of the messages $m_1, \ldots, m_t$ has never been provided to $\mathcal{F}$, meaning that no square root of $h(m_j)$ for some $1 \leq j \leq t$ is known to $\mathcal{F}$. We show how to use this adversary to extract square roots. More precisely, we build a reduction $\mathcal{R}$ that uses $\mathcal{F}$ to extract a square root of an arbitrary $y \in \mathbb{Z}_N^\star$ with probability $\varepsilon' = \varepsilon/2$.

The reduction $\mathcal{R}$ works as follows. On input (y, N), $\mathcal{R}$ invokes $\mathcal{F}$ and transmits N to $\mathcal{F}$. Then $\mathcal{R}$ simulates the random oracle h as well as the signing oracle S_k which returns upon request up to q_k message-signature pairs (m_i, σ_i) with $\sigma_i^2 = h(m_i) \bmod N$. These simulations are performed as follows.

Simulation of S_k. Each time $\mathcal{F}$ requests a message-signature pair, $\mathcal{R}$ chooses (according to any arbitrary distribution) some message $m \in \{0,1\}^\star$ such that m does not appear in $\mathcal{R}$'s transcript, picks a random $r \in \mathbb{Z}_N^\star$, defines $h(m) \leftarrow r^2 \bmod N$, updates its transcript accordingly, and outputs the pair (m, r).

Simulation of h. Whenever $\mathcal{F}$ requests $h(m)$ for some $m \in \{0,1\}^\star$, $\mathcal{R}$ checks in its transcript if $h(m)$ is already defined, in which case $h(m)$ is returned. If $h(m)$ is undefined, $\mathcal{R}$ picks a random $r \in \mathbb{Z}_N^\star$ and a random bit $b \in \{0,1\}$, defines $h(m) \leftarrow r^2 y^b \bmod N$, updates the transcript and returns this value to $\mathcal{F}$.

These simulations never fail to respond to $\mathcal{F}$'s queries and the distributions of answers are statistically indistinguishable from the ones $\mathcal{F}$ expects. After at most q_h hash queries and q_k message-signature queries, $\mathcal{F}$ outputs $(m_1, \dots, m_t)$ and σ within some time bound τ. Then $\mathcal{R}$ queries $h(m_i)$ for $i = 1, \dots, t$ to its own simulation of h and checks whether $\sigma^2 = \prod_{i=1}^t h(m_i) \bmod N$.

Extraction of a square root of y. Since each and every message $m_i \in \{m_1, \dots, m_t\}$ has been queried to the hash oracle (either by $\mathcal{F}$ or $\mathcal{R}$), $\mathcal{R}$ knows an r_i such that $h(m_i) = r_i^2 y^{b_i} \mod N$ (if m_i was requested to h) or $h(m_i) = r_i^2 \mod N$ (if m_i was requested to S_k). Letting A (respectively B) denote the set of indices i such that m_i was requested to h (resp. to S_k), we know that the messages m_i for $i \in B$ are among the ones given by $\mathcal{R}$'s simulation of S_k to $\mathcal{F}$ throughout the experiment. By definition of $\mathcal{F}$, $B \subsetneq \{1, \dots, t\}$ and hence $|A| \neq 0$. Consequently, if the verification succeeds then

$$\sigma^2 = \prod_{i \in A} r_i^2 y^{b_i} \prod_{i \in B} r_i^2 = \left(\prod_{1 \le i \le t} r_i \right)^2 y^B \mod N ,$$

with $B = \sum_{i \in A} b_i$ meaning that

$$\left(\sigma / \prod r_i\right)^2 = y^B \mod N .$$

Since all bits b_i are mutually independent and uniformly distributed over $\{0,1\}$, we have that B is odd with probability $1/2$. When B is odd, there exist integers (α, β) such that $\alpha B + 2\beta = 1$. Then

$$y = y^{\alpha B + 2\beta} = \left(\left(\sigma / \prod r_i\right)^\alpha y^\beta \right)^2 \mod N ,$$

and $\mathcal{R}$ returns the root $x = (\sigma / \prod r_i)^\alpha y^\beta \bmod N$ with probability one. Summarizing, since $\mathcal{F}$ outputs a valid forgery with probability at least ε within τ steps, our reduction $\mathcal{R}$ returns x such that $x^2 = y \bmod N$ with probability $\varepsilon' = \varepsilon/2$ after at most $\tau' = \tau + (q_h + q_k)\mathcal{O}(\log^3 N)$ steps.

D Security Model for Signatures and Macs

D.1 Signature Schemes

A signature scheme $\mathsf{SIG} = (\mathsf{SIG.Key}, \mathsf{SIG.Sign}, \mathsf{SIG.Verify})$ is defined by the three following algorithms:

- The *key generation algorithm* $\mathsf{SIG.Key}$. On input 1^k, the algorithm $\mathsf{SIG.Key}$ produces a pair $(\mathsf{pk}, \mathsf{sk})$ of matching public (verification) and private (signing) keys.

- The *signing algorithm* $\mathsf{SIG.Sign}$. Given a message m and a pair of matching public and private keys $(\mathsf{pk}, \mathsf{sk})$, $\mathsf{SIG.Sign}$ produces a signature σ. The signing algorithm might be probabilistic.
- The *verification algorithm* $\mathsf{SIG.Verify}$. Given a signature σ, a message m and a public key pk, $\mathsf{SIG.Verify}$ tests whether σ is a valid signature of m with respect to pk.

Several security notions have been defined about signature schemes, mainly based on the seminal work of Goldwasser *et al* [16,17]. It is now classical to ask for the impossibility of existential forgeries, even for adaptive chosen-message adversaries:

- An *existential forgery* is a new message-signature pair, valid and generated by the adversary. The corresponding security level is called *existential unforgeability* (EUF).
- The verification key is public, including to the adversary. But more information may also be available. The strongest kind of information is definitely formalized by the *adaptive chosen-message attacks* (CMA), where the attacker can ask the signer to sign any message of its choice, in an adaptive way.

As a consequence, we say that a signature scheme is secure if it prevents existential forgeries, even under under adaptive chosen-message attacks. This is measured by the following success probability, which should be small, for any adversary $\mathcal{A}$ which outputs a new pair (m, σ), within a reasonable running time and at most q_s signature queries to the signature oracle:

$$\mathsf{Succ}^{\mathsf{euf-cma}}_{\mathsf{SIG}}(\mathcal{A}, q_s) = \Pr\left[\begin{array}{c}(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{SIG.Key}(1^k), (m, \sigma) \leftarrow \mathcal{A}^{\mathsf{SIG.Sign}(\mathsf{sk};\cdot)}(\mathsf{pk}) : \\ \mathsf{SIG.Verify}(\mathsf{pk}; m, \sigma) = 1\end{array}\right].$$

D.2 Message Authentication Codes

A Message Authentication Code $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Verify})$ is defined by the two following algorithms, with a secret key sk uniformly distributed in $\{0,1\}^\ell$:

- The *MAC generation algorithm* $\mathsf{MAC.Sign}$. Given a message m and secret key $\mathsf{sk} \in \{0,1\}^\ell$, $\mathsf{MAC.Sign}$ produces an authenticator μ. This algorithm might be probabilistic.
- The *MAC verification algorithm* $\mathsf{MAC.Verify}$. Given an authenticator μ, a message m and a secret key sk, $\mathsf{MAC.Verify}$ tests whether μ has been produced using $\mathsf{MAC.Sign}$ on inputs m and sk.

As for signature schemes, the classical security level for MAC is to prevent existential forgeries, even for an adversary which has access to the generation and the verification oracles. This is measured by

$$\mathsf{Succ}^{\mathsf{euf-cma}}_{\mathsf{MAC}}(\mathcal{A}, q_s, q_v) = \Pr\left[\begin{array}{c}\mathsf{sk} \stackrel{R}{\leftarrow} \{0,1\}^\ell, (m, \mu) \leftarrow \mathcal{A}^{\mathsf{MAC.Sign}(\mathsf{sk};\cdot),\mathsf{MAC.Verify}(\mathsf{sk};\cdot,\cdot)} : \\ \mathsf{MAC.Verify}(\mathsf{sk}; m, \mu) = 1\end{array}\right],$$

where the adversary can ask up to q_s and q_v queries to the generation and verification oracles $\mathsf{MAC.Sign}$ and $\mathsf{MAC.Verify}$ respectively. It wins the game if it outputs a *new* valid authenticator.

E Code Certification With a Hash-Tree