Hardening a Java Card Virtual Machine Implementation with the MPU

Guillaume Bouffard, Léo Gaspard

Hardening a Java Card Virtual Machine Implementation with the MPU

Guillaume Bouffard and Léo Gaspard
guillaume.bouffard@ssi.gouv.fr
leo@gaspard.io

Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI),
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.

Abstract. In the world of Java Cards, the Firewall guarantees the segregation of applet data and ensures the integrity and confidentiality of each application. In order to be independent from the microcontroller, most Java Card Virtual Machine (JCVM) implementations are not designed to use hardware-based mechanisms.
In this article, we describe how the Memory Protection Unit (MPU) can be used to segregate each Java Card applet from the Operating System (OS) and device drivers. Even if our contribution is designed to fit a specific hardware-based mechanism, our JCVM architecture can be reused for a microcontroller without MPU.

1 Introduction

Developing smart card applications is a long and complex process. Despite several standardization efforts, *e.g.*, concerning power supply, input and output signals, smart card development used to rely on proprietary Application Programming Interfaces (APIs) provided by each manufacturer. The main drawback of this development approach is that the code of the application can then only be executed on a specific platform, thus lowering interoperability.

In order to improve the interoperability and security of smart card software, the Java Card technology has been designed in 1996. It enables Java-based applications to securely run on smart cards and similar low-footprint devices. The trade-offs made on the Java architecture in order to embed the Java Card Virtual Machine (JCVM) on low resource devices concern both functional and security aspects.

1.1 The Java Card Security Model

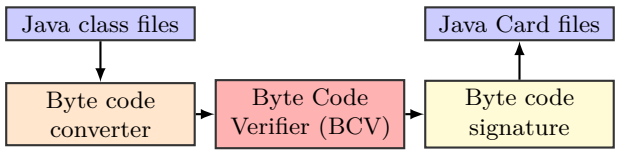
In the Java Card realm, some aspects of software security rely on the Byte Code Verifier (BCV). As Java Card byte code can also be hand-written,

it might violate Java Card safety rules. As a consequence, the Java Card Runtime Environment (JCRE) must not trust the Java Card conversion process. In order to ensure the correctness of the Converted APplet (CAP) file that is to be loaded, the BCV checks:

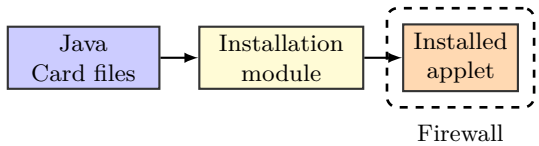
1. that the application structure is valid,
2. that the application byte code neither forges pointers nor violates access restrictions,
3. that the application uses data with the correct type and
4. that the application only accesses objects it owns.

Since the Java Card platform does not support dynamic class loading, byte code verification is performed at loading time, *i.e.* before installing the CAP file onto the card. Moreover, most Java Card platforms do not embed an on-card BCV as it is expensive in terms of memory consumption. As a consequence, byte code verification is performed off-card, either directly by the card issuer if he controls the loading chain, or by a trusted third party that signs the application as a proof of verification.

In addition to static off-card verification enforced by the BCV, the Java Card Firewall performs runtime checks to guarantee applet isolation. It partitions the Java Card platform into separate and protected object spaces called contexts. Each applet is associated with a context, thus preventing instances of an applet from reading or writing another applet's data, unless the accessed applet explicitly exposes functionality through a Shareable Interface Object. The goal is to ensure data confidentiality and integrity. The Java Card security model is summarized in Fig. 1.



(a) Java Card off-card security model



(b) Java Card on-card security model

Fig. 1. Java Card security model

Despite all the security features enforced by the Java Card environment, several attack paths have been found by the Java Card security community [2, 3, 5–7, 11–13, 16, 17, 19].

In this work, we focus on how to harden our JCVM implementation so as to prevent applet isolation violation by a malicious applet that is either an ill-formed applet not properly detected by the BCV — for instance when the Java Card security model is not followed — or a mutant application created by fault injection attacks [24].

To the best of our knowledge, most existing Java Card Firewall implementations do not use any hardware features. The JCVM developers have chosen this architecture in order to be generic over (ie. independent from) the microcontroller where the JCVM will be executed. Thus, without extra effort, smart card developers can move their JCVM implementation to another microcontroller. Our solution aims at proposing an alternative to this approach in order to improve the security of the JCVM implementation with generic hardware-based mechanisms for memory segregation.

Of course, the independence between Java Card applications and hardware-based mechanisms is still ensured by the JCVM. However, our solution does depend on the way to configure hardware-based security mechanisms, which will depend on the microcontroller manufacturer. Changing the target of the JCVM thus requires adapting the software.

We will see that this approach is also compatible with a generic architecture where the Operating System (OS) is able to segregate Java Card security contexts on a microcontroller lacking hardware-based security mechanism and without modifying the JCVM source code.

In the literature, hardware security mechanisms have mainly been studied for the protection they can offer from side-channel and fault injection attacks. The smart card OS is designed to prevent fault injection attacks thanks to defenses like sensors, filters, double executions and so on. Several papers of Lackner et al. focused on hardening the Java Card execution environment with hardware security mechanisms. For instance, they prevent type confusion with a hardware mechanism [14] and introduce hardware checks [15] in the Java Card stack. However, their approach is based on proprietary extensions of a secure component.

In his Master's thesis, Zelle [26] described how to protect the JCRE through hardware mechanisms. His approach is based on a hardware mechanism he implemented on an FPGA. However JCVM are mainly designed in order to not depend on proprietary extensions, for portability

reasons.

This paper is organized as follows. First, Section 2 presents a few hardware security mechanisms embedded in standard components. Next, Section 3 explains how some of those hardware-based mechanisms have been used to improve the security of our JCVM implementation. Section 4 presents our experimental results and then details our secure Java Card OS. Finally, Section 5 concludes and identifies possible future works.

2 Hardware Security Mechanisms to Protect Software Integrity and Data Confidentiality

Central Processing Units (CPUs) include two kinds of hardware security mechanisms. The first one enables the CPU to segregate execution contexts by having code running in different execution contexts (usually kernel vs. user). The other one aims at protecting memory access (reading, writing or executing) from an unauthorized application.

2.1 Context Segregation

Context segregation aims at associating each executed application to a different execution context. The hardware handler checks each access to memory resources to prevent access from a context to another one.

ARM TrustZone is the security extension for ARM-based devices where two virtual processors are separated by a hardware-based access control. The application core can switch between each virtual world. The first state is the secure world where secure and sensitive applications are run in a special OS. For instance, this world aims at starting the boot securely with firmware signature verification. The second state is the normal world where the rich OS (Linux/Android, Windows and so on) is executed. The normal world can communicate with the secure world through a secure buffer. From a security point of view, the normal world may be malicious and may try to leak information from the secure world [4].

The ARM TrustZone technology is currently not embedded in secure components. In addition, it can be corrupted through fault injection attacks as demonstrated in [25].

2.2 Memory Access Protection

When multiple applications are multitasked, checking memory accesses might be required in order to enforce privilege segregation through access rules.

Memory Management Unit Consumer CPUs have an Memory Management Unit (MMU) available for the OS developer to use. The MMU is a hardware component that intercepts all memory accesses performed by the processor and applies translation and authorization.

Translation is the process of taking the requested virtual address, and checking in the page tables to which physical address it maps. The tables involved in the process of address remapping also contain additional data that are used for authorization, by checking that the requesting process (as defined per the processor ring, on Intel x86 processors) is allowed to access in read, write or execute mode the requested address.

It is one of the most important components for providing segregation between the OS and processes, along with the separation between privileged and unprivileged code.

Input-Output Memory Management Unit The Input-Output Memory Management Unit (IOMMU) is also present on some high-end chipsets. Whereas the MMU is placed so as to intercept memory accesses from the CPU, the IOMMU is placed so as to intercept memory accesses from devices attached to the motherboard. As such, it helps protecting the CPU from malicious devices that may be plugged in, thus mitigating attacks like [10].

Memory Protection Unit On components with limited resources, the main hardware module that improves software security is the Memory Protection Unit (MPU). Designed for the ARM architecture, it is a lightweight version of the MMU that does not perform address remapping but only permission checking. As such, it allows generating an exception (more precisely, a fault) when the access type or address does not match what is allowed by the current ruleset.

Along with the separation between privileged and unprivileged code, this hardware component allows confining the code segment into a sandbox.

When the MPU is enabled, all memory accesses from the microcontroller will be checked according to the following algorithm (simplified from the ARM specification [1]):

Algorithm 1 MPU access checking

Ensure: An operation is allowed to access *addr* with *perm* (read, write or execute)
  Res ← *Reject*
  if DefaultMapEnabled **then**
    Res ← AllowsAccess(*perm*, IsPrivileged, DefaultMap)
  $i \leftarrow 0$
  for $i \rightarrow$ (number of segment − 1) **do**
    if addr in Segment(i) **then**
      Res ← AllowsAccess(*perm*, IsPrivileged, Segment(i))
  return Res

In other words, there is a default memory map (which only allows privileged mode to access all the memory in reading, writing and execution) that can be enabled or not (this being defined at the time of enabling the MPU). Then, all the segments are tested one after the other to check whether they include the accessed address or not. The highest-numbered matching segment then defines the final permissions. As a consequence, the default memory map acts like a "segment −1": it is used only if all other segments do not match the requested address.

There are restrictions on how the segments can be set. Their size must be a power of two at least equal to 32, naturally aligned. Each segment of at least 256 bytes can be split in eight equal-length segments that can be individually enabled or disabled, thanks to the *Sub-Region Disable* bits.

As for the `AllowsAccess` operation, the MPU allows (from a security standpoint) to set for each segment whether it is executable (through the eXecute Never (XN) bit), as well as whether (un)privileged code can access it in read mode, read-write mode, or none (through the Access Permissions (AP) bits).

2.3 JCVM and MPU

To the best of our knowledge, most JCVM implementations do not use hardware-based software security mechanisms even though modern secure components do embed a MPU. Currently, the MPU is used only to prevent the smart card OS, drivers and the Java Card interpreter from corrupting each other.

In order to ensure applet isolation properties, most JCVM implementations use a software mechanism with program counter position verification, bounds checks or application Control Flow Graph (CFG) enforcement [8].

During the development of our JCVM implementation, we focused on how hardware security mechanism can be used to improve security without adding costly additional checks. To be close to a smartcard component, where all JCVMs are executed, the mainstream ARM Cortex-M4 and more specifically the STM32F401RE was chosen as target. This component includes an MPU.

3 When MPU Meets the Java Card Security Model

3.1 Proposed Usage

We propose a way to use the MPU that is closer to what is done by regular computer-based OSs: separate the security context attached to each applet as an actual MPU context. For this purpose, one interpreter is run for each applet. Each is responsible only of correctly interpreting the Java Card byte code for its own applet, so that the OS can segregate between interpreters using the MPU.

Consequently, a malicious (or compromised) applet, even if it manages to compromise the interpreter, will not be able to access or otherwise disrupt the normal behaviour of other applets, except as a Denial of Service (DoS).

3.2 Overall Architecture

We propose the architecture depicted in Fig. 2: the OS handles device drivers as well as hardware support (including the ISO-7816 peripheral, the MPU and the flash device), and exposes their functionality to contexts through system calls (syscalls), while checking authorization. It also provides a way for contexts to interact with each other, again using syscalls.

For instance, interactions with the ISO-7816 peripheral can be provided through an `in_byte` syscall and an `out_byte` syscall, with authorization limited to calls from the APDU parser context. Interactions with the flash can be performed using `read_file` or `write_file` syscalls; they check that the requesting context is indeed allowed to access the requested file.

Given that the Java Card specification [21,22] does not require multiple contexts to be active simultaneously, there is no need for a preemptive scheduler, since only a single context can be active at a single time. Note that this does not prevent DoS attacks; they are inherent to the Java Card specification: an applet can simply start an infinite loop, and the JCRE will not be allowed to kill it.

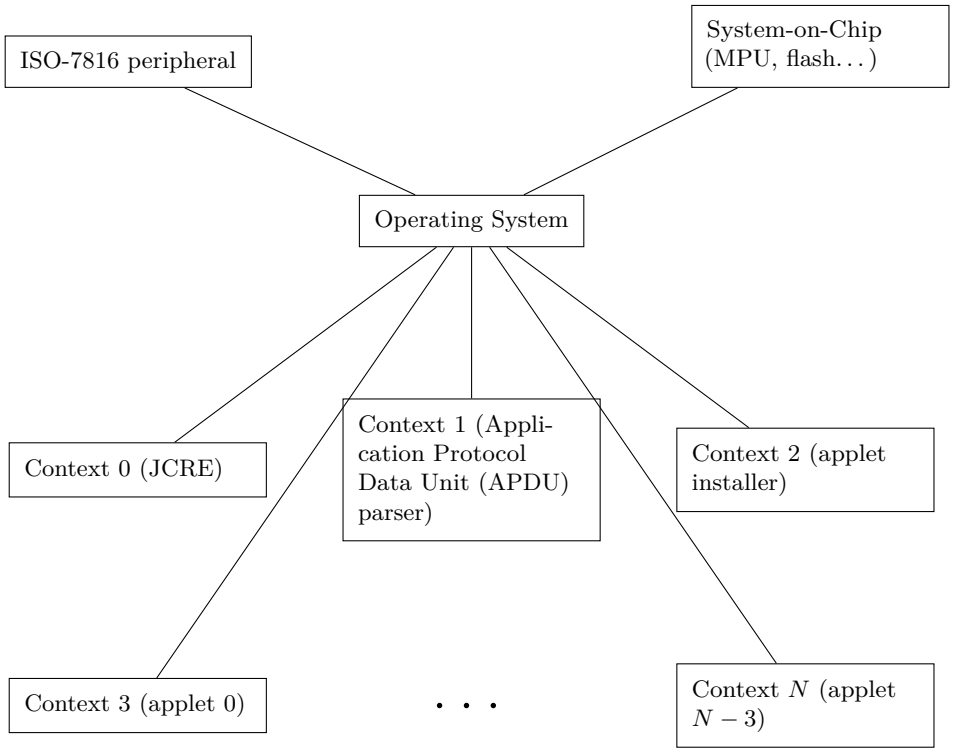


Fig. 2. Overall architecture

Interactions between contexts are performed through the `remote_call` and `remote_result` syscalls. The `remote_call` syscall pushes the current context on the OS stack and calls the remote call handler of another context after context switching. The `remote_result` syscall pops a context from the OS stack to jump back to where execution stopped for the remote call.

3.3 Practical Organization Around the MPU

We chose to have the memory layout split in MPU segments as defined in Fig. 3.

A compiled code segment that always stays readable and executable contains all the compiled code in the interpreter as well as in the OS. At minimum the interpreter has to be readable and executable; the OS has been left readable and executable too in order to simplify the firmware build process and spare some memory.

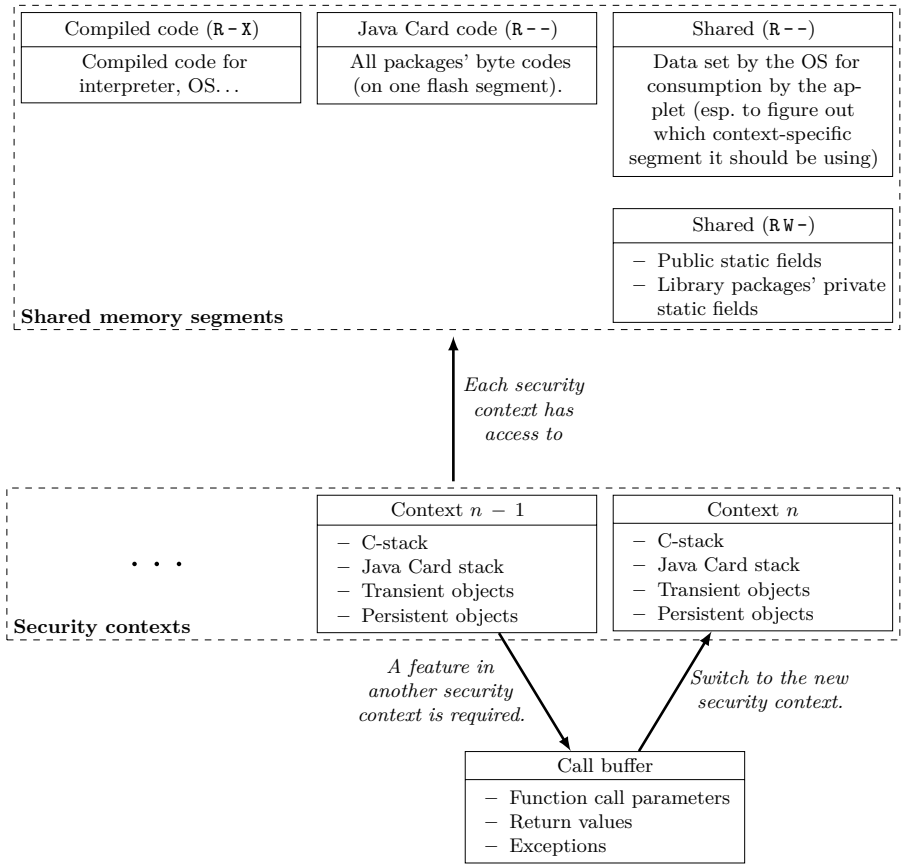


Fig. 3. MPU segments used

An applet code segment that contains all the Java Card bytecode of all installed applets is set read-only. It would be better, from a security point of view, to only allow the applet's reachable code to its security context. However, the applet can call functions in shared packages without changing its security context, which means code reachable from the applet and unreachable from the applet will likely be interleaved.

Having only eight MPU segments available (due to the ARM specification [1], as described in Algorithm 1), especially with the strong constraints on their use (natural alignment and power-of-two size) means that getting a stronger separation of the Java Card byte code is left as future work. The security consequence of this choice is that, in the current implementation, a rogue interpreter can access the Java Card byte code of all other applets, meaning that confidentiality of the code is not ensured.

On the positive side, the MPU still helps preventing modifications to Java Card byte code, and protects the confidentiality as well as the integrity of the data manipulated by other applets.

A shared read-write segment used for the fields that can be modified by all contexts, eg. library packages' static fields.

A shared read-only segment used for all the fields that are set by the OS for use by the applets, yet should not be under their control. For example, this includes the security context identifier, the low and high bounds of the heap to be used by the applet...

A context-specific read-write segment that stores all the data that are actually context-specific: C stack, Java Card stack, `CLEAR_ON_{DESELECT,RESET}` segments, C heap for the interpreter... There is one per security context, so that each security context can only access its own data.

A call buffer is used to pass arguments and return values (or exceptions) across the remote call stack. As one applet can call a function in another applet (defined by a `Shareable` interface by the other applet), the stack may be split across multiple security contexts. In order to achieve this, the OS maintains a stack of which context called which context, and pushes or pops from it as required. However, there is a need to pass arguments and return values to the relevant contexts, and this must be done through some shared space.

In order to minimize the risk of leaks if context A calls context B with 128 bytes of arguments, and context B calls context C with 32 bytes of arguments, wrapper functions are provided to retrieve or put data in this call buffer by resetting it all to zeros. This call buffer is technically only a part of the shared read-write segment, but its implementation is specific enough that it is handled as though it was in another segment.

Switching process. When switching from a context to another (eg. during a `remote_call` syscall), all these MPU segments are reset by the OS to the relevant addresses of the new applet, thus allowing a number of applets only limited by the memory of the microcontroller and the fact all segments must be of power-of-two size.

4 Experimental Results

4.1 Implementing our embedded OS

In order to implement our OS, the use of a security-oriented programming language that works on limited-resources components is required. Our target, the STM32F401RE, embeds a 32-bit CPU running at 84 MHz, with 512 kB of Flash memory and 96 kB of RAM.

In the embedded world, programs are widely developed in C, C++ and, more rarely, in Ada. Due to being low-level, programs written in the C or C++ language often contain bugs undetectable by the toolchain. Recent C compilers add sanitizing options (for instance memory leak or overflow detector) that can be enabled in order to prevent some bugs at runtime.

In order to prevent bugs during execution, the Ada toolchain also adds runtime checks. SPARK is a formal programming language based on the Ada language. It aims at ensuring properties statically during the build process. Thus, since each property is statically proved, the SPARK program can then be built without any runtime check in order to improve its footprint.

Recently, Mozilla Research sponsored the Rust programming language as a safe, concurrent and practical language. The Rust toolchain aims at checking several security properties at build time. It is based on the LLVM compiler, which turns out to have ARM assembly available as a binary target.

As described by Couprie and Chifflier [9], the Rust language has:

- managed memory without any garbage collector,
- type safety,
- thread safety, even though this feature is not required for embedded development,
- native code with zero-copy feature,
- easy integration with C and C++ code,
- isolation of unsafe fragments of code (unsafe code can be used to write low-level code), and
- a minimal runtime.

Due to the limited resources of our target, we decided to develop our OS with the Rust programming language with the possibility to link with C/C++ code, especially for the board API. Moreover, the Rust runtime being really lightweight is an interesting feature available for the embedded world.

4.2 Tests

In order to have the JCVM being as secure as possible, there is a need to have as complete as possible unit tests. As the interpreter also aims at running on x86 machines for development purposes, it makes sense to execute the unit tests on x86 too.

From this point on, two ways forward can be chosen. The first one is to port the OS to x86 even though x86 had no MPU: let all the functions that change the MPU be no-ops, and do not test the MPU part.

The second one, which we picked, is to use the Linux `mprotect` function to emulate an MPU, as this allows simulating more precisely how things work in the Cortex-M4 CPU. However, `mprotect` only protects memory per block of 4 KiB, whereas the MPU allows protecting blocks as small as 32 bytes. This means that in order to simulate the MPU using `mprotect`, a trick has to be used.

The trick was to `mprotect` the whole 4 KiB, and catch the segmentation faults that will arise even in case of valid access. Then, analyzing the fault, we check whether it is on an allowed or disallowed address, and if it is on a disallowed address, we report the violation. If it is on an allowed address, we can then `mprotect` the segment back to allow its use, single-step, then `mprotect` the segment to lock it again.

So as to do this, the Linux `ptrace` system call gives all the required primitives with the cost of not being able to run the tests under another debugger. As debugging tests that fail is important too, the `SIGILL` signal is used to trigger a core dump of the child, and all analysis is performed post-mortem.

Implementing system calls is the last thing that had to be done in order to test the OS on an x86 desktop computer. These were implemented using the `int3` x86 instruction, and thus perform a kind of "debugger call" that is then used in order to implement all the other primitives that normally require some help from the hardware (or, in this case, the debugger).

Thanks to the tests this framework allowed us to write, we discovered a few bugs in our implementation, including some that would not have been detected if the MPU was not emulated. We also hope it will help prevent future regressions.

Having functionally tested our OS and JCVM implementation, we evaluated our solution's footprint on the targeted architecture, so as to gather performance results that could help in balancing security gains against performance loss.

4.3 Timing

With all this functionality implemented and without any interaction with the interpreter (that is, measuring only the time of calling a no-op C function), calling a function takes 90 ns on the STM32F401RE board.

This measurement giving the baseline of the processor speed, we then measured the performance of system calls (in order to measure the approximate speed of the core to push and pop the required registers), which took 2.6 µs each.

Finally, we measured the time taken to perform a full remote call, that is a context switch to another security context which then returned a zero-valued result to the original security context through another context switch. Given the duration of a system call, this time had to be at least of 5 µs, as each remote call implies two system calls (one to enter the other security context, and one to come back to the first one). The final measure was 20 µs. In order to understand it, we looked at how long a write in RAM took on our test platform, and it came up at 71 ns. This means the implementation performs about 200 RAM writes for the context switch in addition to the two system calls that already take 5 µs, which, given the used data structures, could probably be improved but looks like an acceptable loss.

Even though this performance difference (90 ns vs. 20 µs) could look like a huge performance loss (approximately two hundred times slower), this is to be put in perspective by the time that will be spent in the interpreter (cost of going through a `Shareable` interface, etc.) as well as the tiny number of times such remote calls will occur in practice.

4.4 Analyzing our Contribution

More often than not, memory protection mechanisms are used in computer security to prevent corruption of legitimate applications from a malicious one. This mechanism is mainly implemented to segregate each process from the OS point of view.

In our contribution, we reorganized the JCVM architecture, from one based on an interpreter which executes a set of applets, to one interpreter per applet. With this approach, each applet is managed as an application from the OS point of view.

Segregating Java Card applets as applications improves memory protection granularity without costly software checks. Our solution prevents malicious Java Card applets (as well as rogue interpreters) from accessing other applets' context data. This approach also blocks native memory

overflows that would be hard to stop (without specifically built toolchains or low-level pieces of code), like stack or heap overflows across security contexts.

Finally, our JCVM architecture can be used even if the microcontroller has no MPU mechanism. Indeed, each access to the resources is handled by OS syscalls. For such an architecture, segregation would be ensured by the OS and Java Card interpreter using more costly software checks.

5 Conclusion and Future Works

In this article, we introduced an approach to improve the confidentiality and integrity of Java Card applets' data along with reducing the amount of checks. We based our contribution on the MPU, a hardware security mechanism provided by ARM chips.

Merging the MPU with our JCVM requires reorganizing the architecture to one where each Java Card applet is segregated in its own context. The advantage of this new architecture is that it reduces the risk of memory corruption from both the native and the Java Card world without adding software checks.

If our JCVM implementation must run on a microcontroller without MPU, the designed architecture allows adding software verification in the OS at the syscalls level, although it wouldn't protect as effectively against an interpreter gone completely rogue. This approach offers a way to ensure security properties across several microcontrollers effortlessly.

Now that we merged the MPU implementation into our JCVM, we are focusing on how to guarantee Control Flow Integrity (CFI) for each Java Card applet installed on the card. Several studies were completed in this direction [8], where the CFG is checked through verifications implemented at software level. We want to improve on this approach using hardware mechanisms. Nyman et al. proposed in [20] a CFI implementation based on the ARM-TrustZone mechanism. We are currently studying on how to adapt their approach for our microcontroller that does not have the ARM-TrustZone mechanism.

References

1. *ARMv7-M Architecture Reference Manual*. ARM Limited, 2014.
2. Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In Prouff [23], pages 297–313.

3. Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, pages 148–163, 2010.
4. Bits, Please! Extracting Qualcomm's KeyMaster Keys – Breaking Android Full Disk Encryption. `http://bits-please.blogspot.fr/2016/06/extracting-qualcomms-keymaster-keys.html`, 2016. [Online; accessed 17-July-2017].
5. Guillaume Bouffard. *A Generic Approach for Protecting Java Card Smart Card Against Software Attacks*. PhD thesis, University of Limoges, Limoges, France, October 2014.
6. Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In Prouff [23], pages 283–296.
7. Guillaume Bouffard and Jean-Louis Lanet. The ultimate control flow transfer in a Java based smart card. *Computers & Security*, 50:33–46, 2015.
8. Guillaume Bouffard, Bhagyalekshmy N. Thampi, and Jean-Louis Lanet. Security automaton to mitigate laser-based fault attacks on smart cards. *IJTMCC*, 2(2):185–205, 2014.
9. Geoffroy Couprie and Pierre Chifflier. Writing parsers like it is 2017. *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2017.
10. Maximillian Dornseif. 0wn3d by an iPod: Firewire/1394 Issues. PacSec 2004.
11. Emilie Faugeron. Manipulating the Frame Information with an Underflow Attack. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications - 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*, volume 8419 of *Lecture Notes in Computer Science*, pages 140–151. Springer, 2013.
12. Samiya Hamadouche, Guillaume Bouffard, Jean-Louis Lanet, Bruno Dorsemaine, Bastien Nouhant, Alexandre Magloire, and Arnaud Reygnaud. Subverting Byte Code Linker service to characterize Java Card API. In *Seventh Conference on Network and Information Systems Security (SAR-SSI)*, pages 75–81, May 22rd to 25th 2012.
13. Samiya Hamadouche and Jean-Louis Lanet. Virus in a smart card: Myth or reality? *Journal of Information Security and Applications*, 18(2-3):130–137, 2013.
14. Michael Lackner, Reinhard Berlach, Johannes Loinig, Reinhold Weiss, and Christian Steger. Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection. In Mangard [18], pages 1–15.
15. Michael Lackner, Reinhard Berlach, Reinhold Weiss, and Christian Steger. Countering type confusion and buffer overflow attacks on Java smart cards by data type sensitive obfuscation. In Jens Knoop, Valentina Salapura, Israel Koren, and Gerardo Pelosi, editors, *Proceedings of the First Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2014, Vienna, Austria, January 20, 2014*, pages 19–24. ACM, 2014.
16. Julien Lancia. Java Card Combined Attacks with Localization-Agnostic Fault Injection. In Mangard [18], pages 31–45.
17. Julien Lancia and Guillaume Bouffard. Java Card Virtual Machine Compromising from a Bytecode Verified Applet. In Naofumi Homma and Marcel Medwed, editors, *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, volume 9514 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2015.

18. Stefan Mangard, editor. *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*. Springer, 2013.
19. Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.
20. Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-supported Call and Return Enforcement for Commercial Microcontrollers. *CoRR*, abs/1706.05715, 2017.
21. Oracle. *Java Card 3 Platform, Runtime Environment Specification, Classic Edition*. Number Version 3.0.5. Oracle, September 2011.
22. Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition*. Number Version 3.0.5. Oracle, 2015.
23. Emmanuel Prouff, editor. *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*, volume 7079 of *Lecture Notes in Computer Science*. Springer, 2011.
24. Tiana Razafindralambo, Guillaume Bouffard, and Jean-Louis Lanet. A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard. In Nora Cuppens-Boulahia, Frédéric Cuppens, and Joaquín García-Alfaro, editors, *DBSec 2012, Paris, France, July 11-13,2012. Proceedings*, volume 7371 of *Lecture Notes in Computer Science*, pages 122–128. Springer, 2012.
25. Aurélien Vasselle, Hugues Thiebeauld, Adèle Morisset, Quentin Maouhoub, and Sebastien Ermeneux. Laser Induced Fault Injection on Smartphone Bypassing the Secure Boot. *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.
26. Michael Zelle. Design and implementation of a hardware supported memory protection for the java card firewall. Master's thesis, Graz University of Technology, April 2015.