JIT Compiler Security through Low-Cost RISC-V Extension

Quentin Ducasse, Pascal Cotret, Loïc Lagadec

JIT Compiler Security through Low-Cost RISC-V Extension

Quentin Ducasse, Pascal Cotret, Loïc Lagadec

ENSTA Bretagne - Lab-STICC, UMR 6285
firstname.lastname@ensta-bretagne.org

(Preprint version)

Abstract

Language Virtual Machines (VM) need to be extremely efficient and hence use complex engines such as a JIT compiler to speed up the usual bytecode interpretation loop. Their usage of low-level and security-critical tasks make them targets of choice. Enforcing low-cost fine-grained memory isolation has been an important research focus as a countermeasure to the most advanced JIT attacks. Memory isolation splits the components of an application with controlled communication and verified access to other resources. We present how custom instructions linked to hardware-enforced domain-checking could protect JIT code and data. We present incremental solutions and their corresponding custom instructions. The generated machine code and extended RISC-V Rocket come at a low-cost both in performance and intrusiveness.

1 Introduction

Language Virtual Machines (VMs) are runtime engines that hold (at least) an interpreter, a garbage collector and a just-in-time compiler. Those engines leverage the portability of applications and define high-level management tools to handle source languages. A source language is usually compiled in an intermediate representation (namely *bytecodes*) that is then interpreted. Optimizing *bytecode interpretation* is possible through *just-in-time (JIT) compilation*, recompiling frequently used bytecodes blocks as optimized machine code at runtime. Memory management is also handled by VMs as they provide a *garbage collector* to allocate and reclaim memory. Those three components are presented on Figure 1. VMs need to manage memory as well as produce and/or execute machine code and come as targets of choice for attackers.

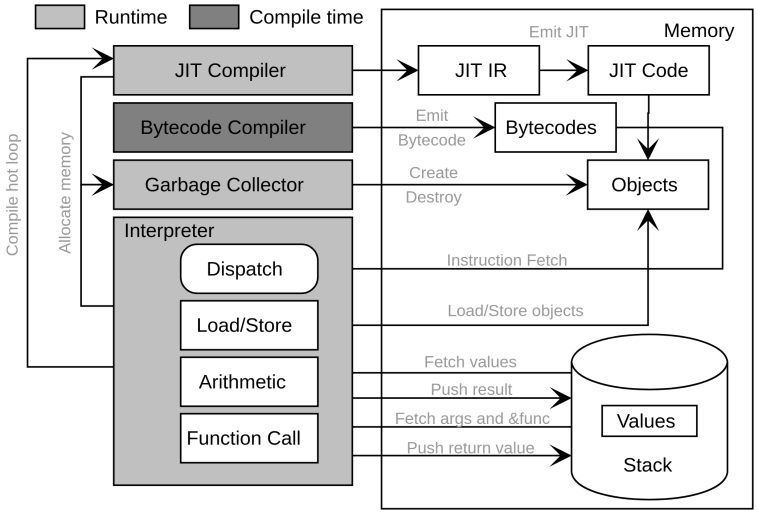

Figure 1: VM components overview.

The scope of attacks against VMs expands above traditional control-flow hijacking attacks as VMs have to handle more untrusted inputs and perform additional security-critical tasks. VM components perform low-level tasks such as attributing and collecting memory or generating optimized machine code and executing it. The JIT compiler in particular has access to writable and/or executable memory and could present a massive security vulnerability [1–3].

Defenses have progressively refined the isolation of VM components. The ones that have been presented as the most resilient with a lesser overhead involve hardware-enforced solutions. Implementation of hardware- and software-based solutions are quantified in terms of performance, area and amount of code needed to instrument the engine.

In the scope of embedded systems, both functionality and security need to be tailored to the needs and usage of the system. The implementation of dedicated solutions comes as a choice of using co-processing units or even extending the *Instruction Set Architecture* (ISA). RISC-V ISA modularity allows development of security measures at a lower cost [4]: it proposes duplicated memory access instructions and an additional component on top of RISC-V Physical Memory Protection (PMP). PMP defines memory regions and their bounded permissions that will later be checked directly by this hardware module when accessed. An additional Domain Memory Protection (DMP) refines PMP with domains and handles duplicated memory access instructions each bound to a given domain.
This paper links security instructions on RISC-V from the JIT compiler generated code to the processor. It presents:

- An application of the security extension RIMI to a RISC-V softcore processor.
- A study of the impact of the added instructions on performance and intrusiveness of the processor.

2 Attacks and Defenses around VMs

2.1 Attacks

2.1.1 Code injection

The first JIT-specific attack [1] was discovered as a *code-injection* attack. *JIT spraying* attacks include large constants that hide instructions when shifted by one or more bytes. Using a XOR chain with several of these constants would put them all in memory that is executable. A latter disruption of the control flow would trigger the embedded instructions.

2.1.2 Code reuse

JIT code is also vulnerable to *code-reuse* attacks that traverse and dynamically disassemble code pages to generate a *Return Oriented Programming (ROP)* chain at runtime. The attacker identifies gadgets, units of machine code that hold a particular function ending with a `ret` instruction. Linking the gadgets together through a ROP chain allows the attacker to run arbitrary code. This type of attack is particularly suited against JIT compilers as, if the JIT compilation trigger is known, arbitrary gadgets can be put in machine code memory. This process has been automated [2, 3], weaponizing the usage of the JIT compiler.

2.1.3 Data-only

Finally, data-only attacks are forcing the JIT compiler to generate malicious code by corrupting the JIT intermediate representation [5] or bytecodes [6], bypassing protections over JIT code and forcing the JIT compiler in generating the payload.

2.2 Defenses

Defenses in response to code injection (Section 2.1.1) add obfuscation and stricter permissions to the JIT code that were then bypassed by code reuse and data only attacks(Section 2.1.2 and 2.1.3). Solutions to Section 2.1.2 propose control-flow integrity [7] that comes at a considerable overhead cost. Authors of defenses against Section 2.1.3 both suggest a hardware-based isolation strategy that adds reasonable overhead. JITGuard [5] consists of an isolation of the compilation and execution processes of JIT code set up through hardware-based trusted execution environments (namely Intel SGX). NoJITsu [6] locks each critical object in the VM with keys and restricted permissions. Hardware-enforced memory isolation stands out in both cases as the solution that provides the less performance overhead and a way to refine isolation if needed in the future.

2.3 Threat Model

The goal of the adversary is to gain the ability to execute arbitrary code in the VM process. This attack becomes even more concrete as JavaScript or Java VMs are heavily deployed with lots of vulnerabilities [8]. The attacker can use the VM to perform arbitrary (sand-boxed) computations at runtime. The VM enforces a strong $W \oplus X$ policy where memory pages cannot be writable and executable at the same time. This policy disables code-injection attacks. Some part of the VM or the surrounding application contains a memory-corruption bug that enables an adversary to access any part of the program address space. This assumption is considered reliable because most core VM components are written in C or C++ and are not memory-secure.

3 RISC-V Memory Isolation

3.1 ISA Presentation

RISC-V is a *modular*, *extensible* and *open-source* Instruction Set Architecture (ISA) that is gaining increasing attention from both academia and industry. Its modularity comes from the extensions it defines, each over the base RV32I (Integer) instruction set. The most common ones are required to run a fully-featured operating system, they consist of `RV64IMAFDC`. They represent the 64-bits instructions (`RV64I`, a superset of `RV32I`), multiplications and divisions (`M`), atomic instructions (`A`), compressed instructions (`C`) the equivalent of Thumb-2 in ARM, floating-point operations (`F`) and double-precision operations (`D`). Writing an application supported by a specific set of extensions will make it available for broader sets as well. Overall, more than 15 extensions are available, either in a frozen complete state, in the process of being ratified or open to proposals (such as the `J` extension for dynamically translated languages). The standard defines opcode space for custom instructions that extend the base set and reserves hints (base instruction with specific arguments) for custom use. As multiple softcore RISC-V processors are available open-source (Rocket, BOOM, CV32E40P, etc.), they can be extended with custom instructions.

3.2 RISC-V

Its privileged specification [9] presents the PMP mechanism which describes the interface for a standard RISC-V memory protection unit. PMP defines a finite number of memory regions (namely 16), through dedicated control and status registers (CSRs) which can be configured to enforce access permissions. Every region can be set as **R**eadable, **W**ritable and/or e**X**ecutable. If a violation of the permission is raised during decoding, the CPU triggers an exception.

3.3 Memory Isolation

Several memory protection and isolation solutions on RISC-V are direct alternatives of principles from other architectures. For example, Keystone [10] is a Trusted Execution Environment (TEE) for RISC-V, Donky [11] and SealPK [12] are implementations of Memory Protection Keys (MPK). Co-processing units add other guarantees such as filtering with FlexFilt [13], monitoring with PHMon [14] or built-in control-flow integrity modules like FIXER [15]. Other methods modify the processor directly by adding instructions in the decoder or using hints. RIMI [4] duplicates memory access instructions and defines domains from which these instructions can be used specifically. Bratter [16] defines a control-flow integrity mechanism purely based on the hints available and a CSR. Stolz et al. [17] add instructions to hash and verify the integrity of basic blocks as well as encode and decode code and data pointers.

3.4 Discussion and Position

The mentioned solutions all use custom instructions. Usually, the intrusiveness of the solution is measured along with the overhead in performance, assessing the impact on usual applications. However, comparisons between solutions are rare as the domain and application are crucial to the requirements and acceptability of a solution. The main differences rely on four criteria: the *type* of the solution, whether it is a co-processor or an extension of the main core; the *backward-compatibility* of the solution; the *intrusiveness* of the solution and its *scalability*. Since the JIT compiler regenerates machine code at runtime, we can take advantage of this to generate custom instructions with the higher-level knowledge it has at compilation time. We can take advantage of RIMI instruction isolation by adding few modifications to the JIT compiler and protect its data accesses.

4 Design of the solution

4.1 RISC-V Modifications

Taking advantage of RISC-V ISA extensibility and PMP feature, Kim et al. present a mechanism, called *RIMI*, to enforce memory isolation at instruction level [4]. It adds a *Domain Memory Protection* (DMP) mechanism that comes on top of the existing PMP. Similarly to PMP, control and status registers are used to define memory regions called domains that consist of a code and a data region. Memory access and control transfer instructions are duplicated for each domain. For example, the `lw` instruction (load word) is duplicated as the original `lw` and another `lw1`. At decode stage, and without overriding the inner PMP, the processor checks if the domain is allowed to execute this instruction. Instructions are then processed identically to their original counterpart after the decode phase and do not interfere with internal states of the RISC-V core. The switch between domains is operated through two new instructions: `jalx` to jump to a new domain and `jalrx` to return from a domain. This mechanism allows a low-area implementation of in-process isolation.

4.2 Processor Modifications

An RV64GC ISA is needed to run a VM: this work focuses on the 64-bit Rocket [18] and uses a memory protection scheme similar to [4] where an RV32 ISA has been used, in simulation. There are three main modifications to perform:

4.2.1 Decoder logic

10 instructions are needed: load/stores (for various data widths) and inter-domain jumps (branches are not taken into account). Furthermore, the decoded instruction is exported to the domain checking logic in order to verify the domain property of the code currently executed. As a consequence, only the CPU instruction decoder needs to be modified: 2 jumps and 8 load/stores instructions are added with fixed opcodes.

4.2.2 Register file

4 CSRs must be added in order to manage memory isolation thanks to a domain separation. Each register, known as `domaincfg`, allows a developer to configure code sections in one of the two domains (the lowest two bits known as `dmpcfg` field). As explained earlier, the domain setting is a complementary security mechanism to the existing PMP proposed in RISC-V specifications. A bit-masking operation helps to have both protections working together (see Figure 2).

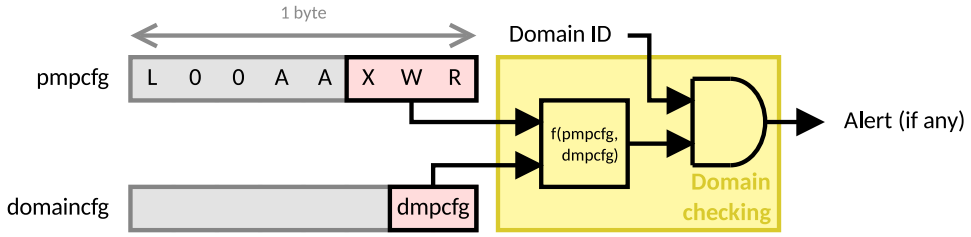

Figure 2: Domain checking verification.

3 inputs are needed:

- Domain ID given by the decoder. It is obtained through instruction bit masking.
- `pmpcfg` is a byte of a `pmpcfgN` CSR (see Section 3.7 of [9], volume 2). It contains read/write/execute rights for a given address space.
- `dmpcfg` represents the lowest two bits of a byte in the `domaincfgN` CSR, 1 bit per domain.

4.2.3 Other logic

Some logic is mandatory for bit-masking both PMP and DMP features needed for memory isolation. Furthermore, when an instruction is decoded, it is also exported to the domain checking logic. In case a domain verification fails, an interrupt is raised. Regarding the threat model, it is assumed that CSRs required for memory isolation are accessible only in machine mode.

5 Use case study

The main goal of an isolation defense around a VM is to protect JIT code and data memory regions. The usual runtime behavior of a VM consists of calls between the interpretation loop and JIT code. The main use cases of RIMI on this type of execution scenario are the following:

1. A shadow stack only used by the JIT code that is hidden in a domain only accessible through duplicated instructions.

2. Isolating the JIT region in a domain, making the JIT data only accessible through JIT code using the dedicated memory access instructions.

3. Both solutions combined to hold both a shadow stack in its own domain and a memory region only accessible by JIT code.

5.1 Scenario 1

An addition of two instructions is required, namely `sws` and `lws` that respectively store and load the return address in the shadow stack and increment or decrement the value in the register holding the shadow stack pointer. This register is either an existing one in the RISC-V architecture that has to be cleared out from its other usages (*i.e.* compiled with the `-ffixed-<register>` from GCC) or a custom register created for this usage. In the example, we use a *shadow stack pointer register* called `str`. The shadow stack is stored in a dedicated RIMI domain only accessible through the `sws` and `lws` instructions.

Listing 1: Scenario 1 code

```
jit_method:
# Prologue                  # Shadow stack load
addi sp, sp, -16            lws  ra, 0 (str)
sw   s0, 0 (sp)             addi str, str, 4
# Shadow stack save         # Epilogue
addi str, str, -4           lw   s0, 0 (sp)
sws  ra, 0 (str)            addi sp, sp, 16
...                         ret
```

5.2 Scenario 2

Making the JIT data only accessible through the JIT code implies that all memory accesses are performed from the JIT code using the duplicated memory accesses, namely `sw1` and `lw1` and their varying widths alternatives (*i.e.* `-b`, `-h`, `-w` and `-d`). The execution or not of those instructions depends on the current domain, either the base domain or the JIT memory region domain. The change is explicit in the VM code when the interpreter triggers a call to an already JITted method. The instruction `jalx` is used to change domain and `jalrx` to return from JIT code to the interpreter. This solution adds ten and thirteen (adding `lwu`, `ld` and `sd`) instructions for 32 and 64-bits architectures respectively.

Listing 2: Scenario 2 code

```
                        jit_method:
interpreter_loop:       # Loading JIT data
...                     lw1   t0, 24(s0)
sw   t0, 0(sp)          # Storing JIT data
jalx jit_method         sw1   t0, 24(s0)
lw   t0, 0(sp)          ...
...                     jalrx ra, 0(ra)
```

6 Evaluation of the proposed solution

6.1 Hardware implementation

In order to evaluate the hardware impact of this work, synthesis were done on Xilinx Vivado 2018.2 targeting an Artix-7 xc7a100t device. Area results are presented in Table 1. For each metric, left value is a bare Rocket CPU core: no modification, synthesized from the v1.5 release with small core settings while right value is a Rocket core modified to be compatible with the approach presented in Section 4. `BPU` is a *Breakpoint Unit*, `div` is the multiplication/division unit, `ibuf` is a set of several buffers and `CSRs` is a submodule describing all the control/status registers as well as implementing the decoding logic. For `div` and `ibuf`, it is assumed that optimizations were made by both Scala-to-Verilog transformation and synthesis tool.

Table 1: Implementation results on a xc7a100t FPGA device. For each column, left are results for a bare Rocket core and right for a modified Rocket core.

		LUTs	**Registers**	**DSPs**
CPU	**ALU**	627 | 627	0 | 0	0 | 0
	BPU	67 | 67	0 | 0	0 | 0
	CSRs	1066 | 1045	741 | 741	0 | 0
	div	761 | 765	214 | 214	4 | 4
	ibuf	187 | 187	53 | 53	0 | 0
	Other	1318 | 1358	609 | 609	0 | 0
Total		4,026 | 4,049 (+0.57%)	1,617 | 1,617 (+0.00%)	4 | 4 (+0.00%)

Modifications for the modified core produces a small area overhead (around 1% in terms of LUTs). As this work is based on custom instructions and additional CSRs, the main overhead can be seen in the `CSRs` submodule. Then, a power analysis is presented in Table 2.

Table 2: Global power of Rocket sub-modules (in W).

	Bare Rocket	**Modified Rocket**	
ALU	3.584	3.543	(- 1.14%)
BPU	0.177	0.176	(- 0.56%)
CSRs	2.087	2.315	(+ 10.92%)
div	10.235	10.507	(+ 2.66%)
ibuf	4.327	4.674	(+ 8.02%)
Overall power	30.919	32.285	(+ 4.42%)

Regarding the overall power consumption of the modified core, similar assessments can be made: when the modified Rocket power consumption is lower, it is assumed to be due to prior optimizations by the synthesis. However, by looking at the overall power of the core, its power consumption is less than 3% higher than a bare softcore which is an acceptable overhead.

6.2 Discussion

On the hardware side, modifications needed in the processor both come out as cheap (less than 10%, even in the worst case). Furthermore, the memory-protected CPU is still able to decode existing instructions: standard applications are not affected by domain security settings. However, the generated secure code using duplicated instructions is not backward-compatible.

On the software side, the instrumentation cost is low as it only requires to generate duplicated memory access instructions in the machine code that may differ for 1 bit only. The domain change instructions should be used through trampolines that connect the interpreter with the jitted machine code. The shadow stack used by the JIT code comes as additional instructions instrumenting calls and returns.
Only a quarter of the `domaincfg` register is used and as an extension, it is assumed that it can be used to extend the memory isolation feature with up to 8 domains as long as dedicated instructions are implemented in the decode stage of the CPU. In our case, 15 instructions were added, 11 for memory access, 2 for domain changes and an additional 2 to handle the shadow stack. They should be generated by the JIT compiler and decoded then executed on the processor. Simpler domains (needing less specific instructions) could be added to the existing pipeline at a lower cost.

7 Conclusion and perspectives

Language virtual machines compile and manipulate native machine code through their JIT compiler. As attackers have presented various JIT attacks, the most advanced defenses comes as strict memory isolation. While this solution protects the system and runtime application efficiently, they come as a cost, whether in performance overhead, intrusiveness of the instrumentation or scalability. We looked at the impact of an instruction-level memory isolation (RIMI) on the Rocket RISC-V softcore. We presented the impact of the presence of two domains on the underlying hardware architecture. Simple additions to the main decode logic and PMP implementation allow the implementation of two scenarios of isolation. The goal is to clearly separate JIT code and other critical components in separate domains with their own dedicated instructions. We plan on conducting further investigation on the impact of such measures in context and compare different RISC-V instruction-based isolation solutions in the context of JIT compilation.

References

[1] Dionysus Blazakis. Interpreter exploitation. In *WOOT*, 2010.

[2] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[3] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *NDSS*, 2015.

[4] Haeyoung Kim, Jinjae Lee, Derry Pratama, Asep Muhamad Awaludin, Howon Kim, and Donghyun Kwon. Rimi: instruction-level memory isolation for embedded systems on risc-v. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[5] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2405–2419, 2017.

[6] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Nojitsu: Locking down javascript engines. In *NDSS*, 2020.

[7] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328, 2014.

[8] Choongwoo Han. A collection of JavaScript engine CVEs with PoCs, 2020.

[9] The RISC-V instruction set manual - volume I and volume II. `https://riscv.org/technical/specifications/`.

[10] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[11] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient in-process isolation for risc-v and x86. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1677–1694, 2020.

[12] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. Sealpk: Sealable protection keys for risc-v. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1278–1281. IEEE, 2021.

[13] Leila Delshadtehrani, Sadullah Canakci, William Blair, Manuel Egele, and Ajay Joshi. Flexfilt: towards flexible instruction filtering for security. In *Annual Computer Security Applications Conference*, pages 646–659, 2021.

[14] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. Phmon: A programmable hardware monitor and its security use cases. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 807–824, 2020.

[15] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. Fixer: Flow integrity extensions for embedded risc-v. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 348–353. IEEE, 2019.

[16] Seonghwan Park, Dongwook Kang, Jeonghwan Kang, and Donghyun Kwon. Bratter: An instruction set extension for forward control-flow integrity in risc-v. *Sensors*, 22(4):1392, 2022.

[17] Florian Stolz, Marc Fyrbiak, Pascal Sasdrich, and Tim Güneysu. Recommendation for a holistic secure embedded isa extension. *Cryptology ePrint Archive*, 2023.

[18] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.