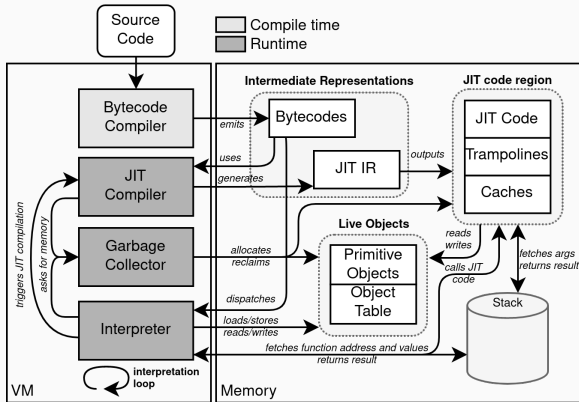


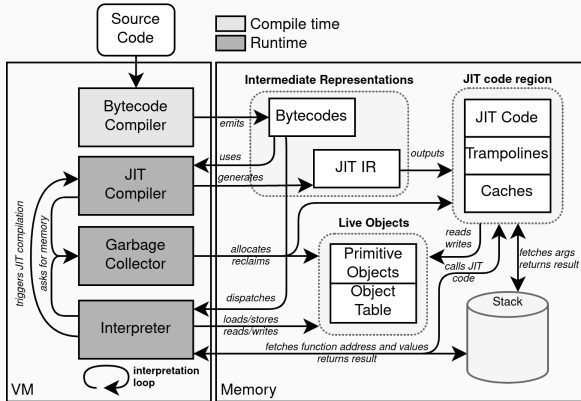
Gigue: A JIT Code Binary Generator for Hardware Testing

Quentin Ducasse, Pascal Cotret, and Loïc Lagadec

November 13, 2023

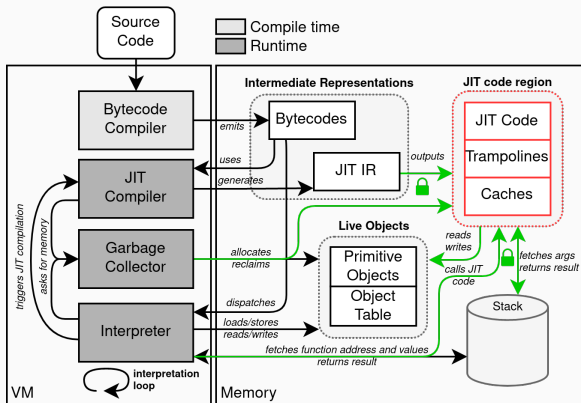
Context & Background





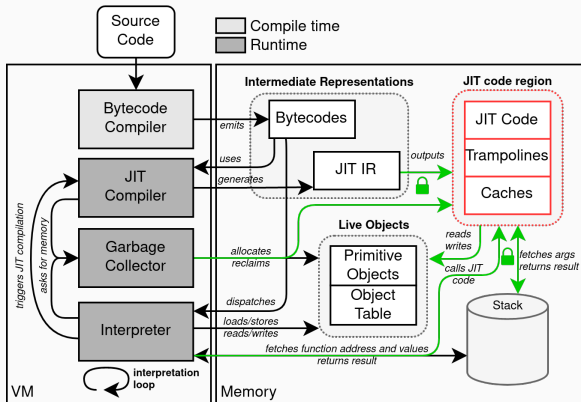
The JIT code region is a key component:

- **Performance-critical** - optimized machine code
- **Security-critical** - writable and executable memory



The JIT code region is a key component:

- **Performance-critical** - optimized machine code
- **Security-critical** - writable and executable memory



The JIT code region is a key component:

- **Performance-critical** - optimized machine code
- **Security-critical** - writable and executable memory

History of attacks [8, 1, 6] and defenses [4, 7] targeting JIT code!

Session K4: Secure Enclaves

CCS'17, October 30-November 3, 2017, Dallas, TX, USA



JITGuard: Hardening Just-in-time Compilers with SGX

Tommaso Frassetto
CYSEC/Technische Universität Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de

Christopher Liebchen
CYSEC/Technische Universität Darmstadt
christopher.liebchen@trust.tu-darmstadt.de

David Gens
CYSEC/Technische Universität Darmstadt
david.gens@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
CYSEC/Technische Universität Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Memory-corruption vulnerabilities pose a serious threat to modern computer security. Attackers exploit these vulnerabilities to manipulate code and data of vulnerable applications to generate malicious behavior by means of code-injection and code-reuse attacks. Researchers already demonstrated the power of data-only attacks by disclosing secret data such as cryptographic keys in the past. A large body of literature has investigated defenses against code-injection, code-reuse, and data-only attacks. Unfortunately, most of these defenses are tailored towards statically generated code and their adaptation to dynamic code comes with the price of security or performance penalties. However, many common applications, like browsers and document viewers, embed just-in-time compilers to generate dynamic code.

The contribution of this paper is twofold: first, we propose a generic data-only attack against JIT compilers, dubbed DOJITA. In contrast to previous data-only attacks that aimed at disclosing secret data, DOJITA enables arbitrary code-execution. Second, we propose JITGuard, a novel defense to mitigate code-injection, code-reuse, and data-only attacks against just-in-time compilers (including DOJITA). JITGuard utilizes Intel's Software Guard Extensions (SGX) to provide a secure environment for emitting the dynamic code to a secret region, which is only known to the JIT compiler, and hence, inaccessible to the attacker. Our proposal is the first solution leveraging SGX to protect the security critical JIT compiler operations, and tackles a number of difficult challenges. As proof of concept we implemented JITGuard for Firefox's JIT compiler SpiderMonkey. Our evaluation shows reasonable overhead of 9.8% for common benchmarks.

website creators to dynamically change the content of the current web page without requesting a new website from the web server. For efficient execution modern run-time environments include just-in-time (JIT) compilers to compile JavaScript programs into native code.

Code-injection/reuse. Unfortunately, the run-time environment and the application that embeds dynamic languages often suffer from memory-corruption vulnerabilities due to massive usage of unsafe languages such as C and C++ that are still popular for compatibility and performance reasons. Attackers exploit memory-corruption vulnerabilities to access memory (unintended by the programmer), corrupt code and data structures, and take control over the targeted software to perform arbitrary malicious actions. Typically, attackers corrupt code pointers to hijack the control flow of the code, and to conduct code-injection [2] or code-reuse [45] attacks.

While code injection attacks have become less appealing, mainly due to the introduction of Data Execution Prevention (DEP) or writable xor executable memory (W^X), state-of-the-art attacks deploy increasingly sophisticated code-reuse exploitation techniques to inject malicious code-pointers (instead of malicious code), and chain together existing instruction sequences (gadgets) to build the attack payload [51].

Code-reuse attacks are challenging to mitigate in general because it is hard to distinguish whether the execution of existing code is benign or controlled by the attacker. Consequently, there exists a large body of literature proposing various defenses against code-reuse attacks. Prominent approaches in this context are code randomization and control-flow integrity (CFI). The goal of code randomization [54] schemes is to prevent the attacker from learning addresses of any gadgets. However, randomization techniques re-

Intel SGX: Secure Enclave with encrypted content and controlled IOs

NoJITSU: Locking Down JavaScript Engines

Taemin Park*, Karel Dhondt[†], David Gens*, Yeoul Na*, Stijn Volckaert[†], Michael Franz*

*Department of Computer Science, University of California, Irvine

[†]Department of Computer Science, imec-DistriNet, KU Leuven

Abstract—Data-only attacks against dynamic scripting environments have become common. Web browsers and other modern applications embed scripting engines to support interactive content. The scripting engines optimize performance via just-in-time compilation. Since applications are increasingly hardened against code-reuse attacks, adversaries are looking to achieve code execution or elevate privileges by corrupting sensitive data like the intermediate representation of optimizing JIT compilers. This has inspired numerous defenses for just-in-time compilers.

Our paper demonstrates that securing JIT compilation is not sufficient. First, we present a proof-of-concept data-only attack against a recent version of Mozilla's SpiderMonkey JIT in which the attacker only corrupts heap objects to successfully issue a system call from within bytecode execution at run time. Previous work assumed that bytecode execution is safe by construction since interpreters only allow a narrow set of benign instructions and bytecode is always checked for validity before execution. We show that this does not prevent malicious code execution in practice. Second, we design a novel defense, dubbed NoJITSU to protect complex, real-world scripting engines from data-only attacks against interpreted code. The key idea behind our defense is to enable fine-grained memory access control for individual memory regions based on their roles throughout the JavaScript lifecycle. For this we combine automated analysis, instrumentation, compartmentalization, and Intel's Memory-Protection Keys to secure SpiderMonkey against existing and newly synthesized attacks. We implement and thoroughly test our implementation using a number of real-world scenarios as well as standard benchmarks. We show that NoJITSU successfully thwarts code-reuse as well as data-only attacks against any part of the scripting engine while offering a modest run-time overhead of only 5%.

Initially, these exploits focused on the JIT compiler itself. This compiler transforms interpreted bytecode into natively executed machine code. When JavaScript JIT compilers first became popular, they wrote all run-time generated code onto memory pages that were simultaneously writable and executable throughout the execution of the script. This trivially enabled code-injection attacks [18], [55]. Later JIT engines added support for W \oplus X policies by doubly-mapping JIT pages instead. This meant that JIT code could no longer be found on memory pages that were simultaneously writable and executable. While this undeniably improved security, attackers repeatedly demonstrated that JIT engines could still be attacked. *JIT spraying*, for example, lets an attacker inject small arbitrary instruction sequences into JIT pages without writing directly to the pages [7], [13], [37]. Defenders quickly thwarted these attacks through the use of constant blinding [13], constant elimination and code obfuscation [19], code randomization [32], or control-flow integrity [46].

Successfully defending JIT engines against code-reuse attacks proved more challenging, however, since an adversary can leverage memory disclosure vulnerabilities to iteratively traverse and disassemble code pages to dynamically generate a ROP chain at run time (an attack known as JIT-ROP [56]). A number of schemes protect against such attacks by leveraging randomization and execute-only memory [8], [9], [23], [29].

More recently, several efforts independently demonstrated that an adversary may still be able to inject code despite all of the above defenses being in place by resorting to data-only attacks. Both Theori et al. [62] and Frassetto et al. [27]

Intel MPK: Intra-process Isolation with Memory Protection Keys



CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation

Mengyao Xie
SKLP, Institute of Computing
Technology, CAS
University of Chinese Academy of
Sciences
Beijing, China
xiemengyao@ict.ac.cn

Chenggang Wu
SKLP, Institute of Computing
Technology, CAS & University of
Chinese Academy of Sciences
Zhongguancun Laboratory
Beijing, China
wucg@ict.ac.cn

Yinqian Zhang
Department of Computer Science and
Engineering, SUSTech
Research Institute of Trustworthy
Autonomous Systems, SUSTech
Shenzhen, China
yinqianz@acm.org

Jiali Xu
SKLP, Institute of Computing
Technology, CAS
University of Chinese Academy of
Sciences
Beijing, China
xujiali@ict.ac.cn

Yuanming Lai
SKLP, Institute of Computing
Technology, CAS
University of Chinese Academy of
Sciences
Beijing, China
laiyuanming@ict.ac.cn

Yan Kang
SKLP, Institute of Computing
Technology, CAS
University of Chinese Academy of
Sciences
Beijing, China
kangyan@ict.ac.cn

Wei Wang
SKLP, Institute of Computing
Technology, CAS
Beijing, China
wangwei2021@ict.ac.cn

Zhe Wang*
SKLP, Institute of Computing
Technology, CAS
Zhongguancun Laboratory
Beijing, China
wangzhe12@ict.ac.cn

ABSTRACT

Intel control-flow enforcement technology (CET) is a new hardware feature available in recent Intel processors. It supports the coarse-grained control-flow integrity for software to defeat memory corruption attacks. In this paper, we retrofit CET, particularly the write-protected shadow pages of CET used for implementing shadow stacks, to develop a generic and efficient intra-process memory isolation mechanism, dubbed CETIS.

To provide user-friendly interfaces, a CETIS framework was developed, which provides memory file abstraction for the isolated memory regions and a set of APIs to access said regions. CETIS also

KEYWORDS

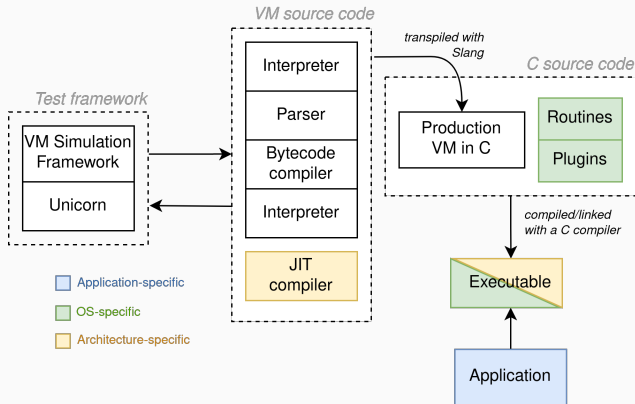
Intra-process Memory Isolation, Intel CET, Memory File Abstraction

ACM Reference Format:

Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, November 7–11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3548606.3539344>

Intel CET: Control-Flow Enforcement using a Shadow Stack and Indirect Branch Tracking

Name	Defense mechanism	VM	Code Injection [34], [35], [39]	Code Reuse [38], [42], [43]	Data Injection [10], [53]
INSeRT [56]	Diversification	V8	✓	✗	✗
RIM [57]	Diversification	Tamarin	✓	✗	✗
librando [61]	Diversification	Hotspot, V8	✓	✗	✗
- [62]	(Re)-Diversification	JikesRVM	✓	~	✗
JITDefender [77]	Transient Protection	Tamarin, V8, JSCore	✓	✗	✗
JITSafe [58]	Transient Protection, Diversification	Tamarin, V8, JSCore	✓	✗	✗
XnR [81]	XnR	Application-Agnostic	✗	✓	✗
Readactor [60]	XnR, Diversification	V8	✓	✓	✗
Lobotomy [80]	Process Isolation	SpiderMonkey	✓	✗	✗
ACG [89]	Process Isolation	Microsoft Edge	✓	✗	✗
NaCl [86]	Sandboxing	V8, Mono Runtime Engine	✓	~	✗
SDCG [78]	Sandboxing	V8	✓	~	✗
JITSec [91]	syscall Filtering	Application-Agnostic	~	~	✗
JITScope [22]	CFI, Transient Protection	SpiderMonkey	✓	✓	✗
RockJIT [94]	CFI	V8	✓	✓	✗
JITGuard [53]	Intel SGX Enclave	SpiderMonkey	✓	✓	~
Libmpk [104]	Intel MPK for Transient Protection	SpiderMonkey, ChakraCore, V8	✓	✗	✗
NoJITsu [10]	Intel MPK for Fine-Grain Isolation	SpiderMonkey	✓	✓	✓



The Pharo VM uses an **indirect threaded interpreter** and a **linear non-optimising method-based JIT compiler**, recently ported to RISC-V [3].

The **RISC-V ISA** [9, 10] defends three main objectives:

- **Open-source:** open standards SoC and core implementations
- **Modular:** instruction groups to support a wide range of applications
- **Extensible:** several standard-allocated spaces for extensions

The **RISC-V ISA** [9, 10] defends three main objectives:

- **Open-source:** open standards SoC and core implementations
- **Modular:** instruction groups to support a wide range of applications
- **Extensible:** several standard-allocated spaces for extensions

Groups of instructions

- I-nteger
- M-ultiplication
- A-tomics
- ...
- RV64IMAFDC supports an OS!

The **RISC-V ISA** [9, 10] defends three main objectives:

- **Open-source:** open standards SoC and core implementations
- **Modular:** instruction groups to support a wide range of applications
- **Extensible:** several standard-allocated spaces for extensions

Groups of instructions

- I-nteger
- M-ultiplication
- A-tomics
- ...
- RV64IMAFDC supports an OS!

Extension vectors

- Opcodes `custom0-3`
- Hints e.g. `lui x0, val`

⇒ Guarantee of compatibility with future evolutions of the standard

Which custom instructions?

Three examples that will be added at different levels in Gigue:

- **E1:** `ror[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)

```
1  
2 rori t1, t1, 2  
3 ror t1, t1, t2  
4 roli t1, t1, 2  
5 rol t1, t1, t2
```

Which custom instructions?

Three examples that will be added at different levels in Gigue:

- **E1:** `ror[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** `cficall, cfiret`
shadow-stack instructions [2]
(+ 2 instructions)

```
1 method1:
2 # Store the return
   ↪ address
3 cficall
4 # Call method2
5 call method2
6 # Load the return
   ↪ address
7 cfiret
```

```
1 method2:
2 ...
3 ret
```

Which custom instructions?

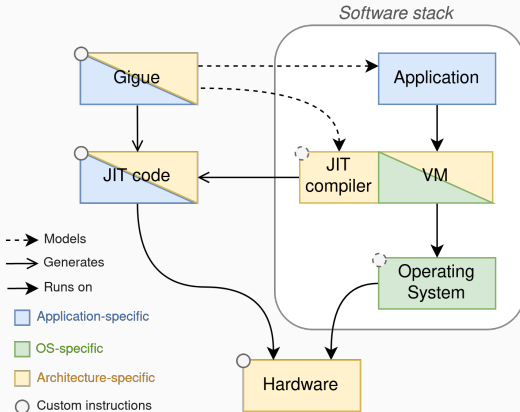
Three examples that will be added at different levels in Gigue:

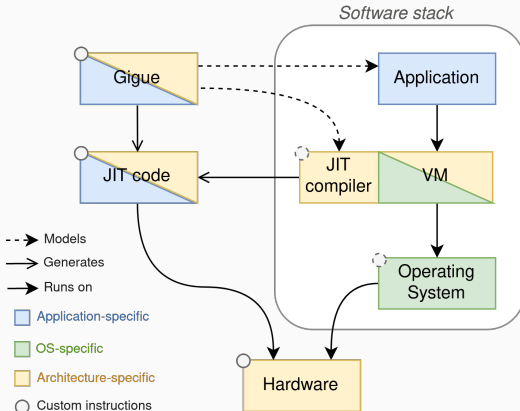
- **E1:** `ror[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** `cficall, cfiret`
shadow-stack instructions [2]
(+ 2 instructions)
- **E3:** `chdom, retdom, l*1, s*1`
dedicated domains and associated
memory accesses [5]
(+ 15 instructions)

```
1 interpretation_loop:
2 ...
3 # Calling a JIT method
4 la    t1, jit_method
5 chdom x0, 0(t1)
6 ...
```

```
1 jit_method:
2 # Loading JIT data
3 lw1   t0, 24(s0)
4 # Storing JIT data
5 sw1   t0, 24(s0)
6 ...
7 retdom ra, 0(ra)
```

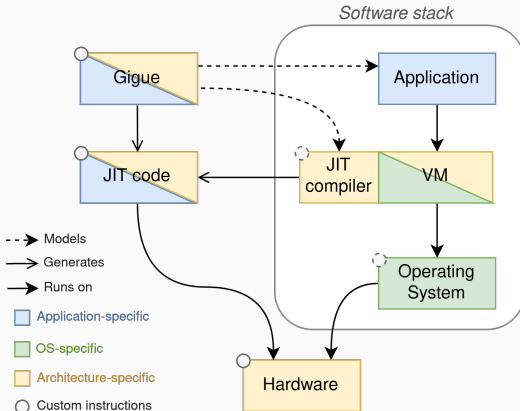
Motivation & Design





Assumptions:

- (1) JIT and AOT compiler(s) are the only components modifying machine code.
- (2) A snapshot of the JIT code region is representative of changes made by those components.



Assumptions:

- (1) JIT and AOT compiler(s) are the only components modifying machine code.
- (2) A snapshot of the JIT code region is representative of changes made by those components.

Motivation

Flattening the software stack significantly speeds up **hardware development** to support VM-specific **custom instructions**.

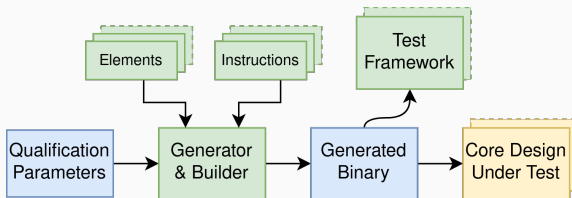
Workload Generator with Custom Instructions

Gigue is a *random workload generator* that produces an executable file modeled after the *JIT code region*, with *custom instructions*, ready to execute on top of extended cores.

Workload Generator with Custom Instructions

Gigue is a *random workload generator* that produces an executable file modeled after the *JIT code region*, with *custom instructions*, ready to execute on top of extended cores.

- **Parametrization:** Diverse application and VM qualification
- **Modularity:** Instructions and JIT elements extensions
- **Testing:** Sanity checks and custom execution model



Three main components integrated in Gigue JIT code region are:

- **Methods:** Filled with random instructions and calls
- **PICs:** Type-guards, switch to the corresponding methods
- **Trampolines:** Routine machine code stubs

Method	
address:	int
body_size:	int
call_nb:	int
call_depth:	int
used_s_regs:	List[int]
local_vars_nb:	int

PIC	
address:	int
case_nb:	int
hit_case_reg:	int
cmp_reg:	int

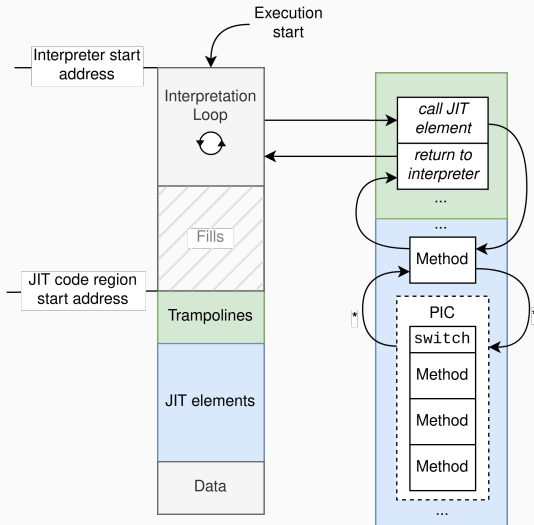
Trampoline	
address:	int
name:	str

(1) **Interpretation Loop** that calls all JIT elements in a random order.

(2) Each **JIT element** calls a number of other elements.

(3) Both have access to **trampolines** for routines.

⇒ The resulting binary is compiled using the binary framework provided by RISC-V assembly tests, `riscv-tests`.



☰ README.md

Gigue: Benchmark Setup and Code Generator for JIT code on RISC-V [↗](#)

 build **passing**

Gigue (*french for jitter*) consists of a machine code generator for RISC-V that mimics the execution of JIT code in concordance with an interpretation loop. The objective is to compare memory isolation memory on a simple model and easily (re-)generate the corresponding machine code parts for both the interpretation loop and JITed code. The base model generates an interpretation loop, a succession of calls to the JIT code. It generates a static binary with both binaries (interpretation loop and JIT elements) along with data the JIT elements use and basic OS facilities to run on top of Verilator emulators from open-source cores (the [Rocket CPU](#) and [CVA6](#) were tested!).

Installation [↗](#)

The project was developed using [pipenv](#) and Python 3.9, whose installation is presented below as well:

- *pyenv* installation:

```
# Install required library headers for pyenv
sudo apt-get install build-essential zlib1g-dev libffi1-dev libssl-dev libbz2-dev libreadline-dev lib

# Install pyenv to manage Python versions
curl https://pyenv.run | bash

# Update PATH (append these to ~/.bashrc)
export PYENV_ROOT="$HOME/.pyenv"
command -v pyenv >/dev/null || export PATH="$PYENV_ROOT/bin:$PATH"
eval "$(pyenv init -)"
```

Figure 1: <https://github.com/QDucasse/gigue>

Two main elements are used, (1) a Generator responsible for **higher level structure handling** and (2) a Builder for **instruction emission**.

1. Instantiate trampolines
2. Determine method base size - $size_{JIT}$, $nb_{methods}$
3. Instantiate elements - $weights_{elts}$
 - **Method** - $base_{size}$, μ_{size} , σ_{size}
 - **PIC** - nb_{cases}
4. Fill elements with random instructions - $weights_{instrs}$, $regs$
5. Patch calls - μ_{calls} , σ_{calls} , λ_{depth}
6. Generate data - $size_{data}$, $generator$
7. Link in a static self-contained binary

The random nature of the generated binaries requires **sanity checks**:

- **Controlled Registers:** fixed at generation.
- **Sanitized Jumps/Branches:** to prevent call graph changes.
- **Data Accesses:** indirect access through a dedicated base register.
- **Call Patching:** call numbers and depth are attributed at JIT element instantiation to fix the call graph
⇒ patched once all elements are filled.

+ Testing!

Modularity & Test Framework

Using the three previous examples:

- **E1:** `rot[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** `cficall, cfiret`
shadow-stack instructions [2]
(+ 2 instructions)
- **E3:** `chdom, retdom, l*1, s*1`
dedicated domains and associated
memory accesses [5]
(+ 15 instructions)

Using the three previous examples:

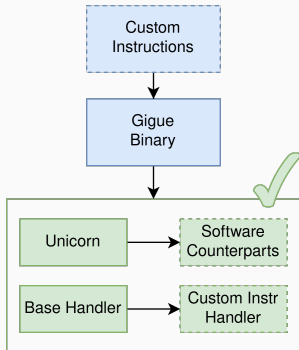
- **E1:** `rot[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** `cficall, cfiret`
shadow-stack instructions [2]
(+ 2 instructions)
- **E3:** `chdom, retdom, l*1, s*1`
dedicated domains and associated
memory accesses [5]
(+ 15 instructions)
- **Impact:** Added to the random
generation of I and R instructions.
(+161/12 loc)

Using the three previous examples:

- **E1:** rot[i], rol[i]
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** cficall, cfiret
shadow-stack instructions [2]
(+ 2 instructions)
- **E3:** chdom, retdom, l*1, s*1
dedicated domains and associated
memory accesses [5]
(+ 15 instructions)
- **Impact:** Added to the random
generation of I and R instructions.
(+161/12 loc)
- **Impact:** Added to method
epilogues and call generation.
(+154/12 loc)

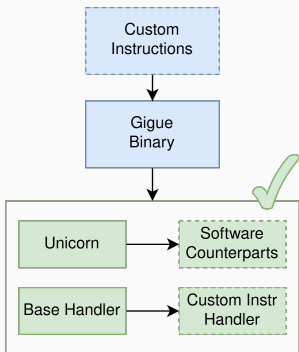
Using the three previous examples:

- **E1:** `rot[i], rol[i]`
rotation instructions (still in draft)
(+ 4 instructions)
- **E2:** `cficall, cfiret`
shadow-stack instructions [2]
(+ 2 instructions)
- **E3:** `chdom, retdom, l*1, s*1`
dedicated domains and associated
memory accesses [5]
(+ 15 instructions)
- **Impact:** Added to the random
generation of I and R instructions.
(+161/12 loc)
- **Impact:** Added to method
epilogues and call generation.
(+154/12 loc)
- **Impact:** Replaced random
generation of S and L, added to
method epilogues and call
generation.
(+502/54 loc)



Unicorn defines flexible wrappers triggered on events such as **exceptions**, **memory accesses**, or **register values**.

⇒ We extend them to catch *custom instructions*!



Unicorn defines flexible wrappers triggered on events such as **exceptions**, **memory accesses**, or **register values**.

⇒ We extend them to catch *custom instructions*!

Using the three previous examples:

- **E1:** Software rotation implementation.
- **E2:** List containing the shadow stack.
- **E3:** Domain checking and duplicated instructions.

Setup & Use Case

Implementation of domains over the physical memory protection settings for JIT code regions.

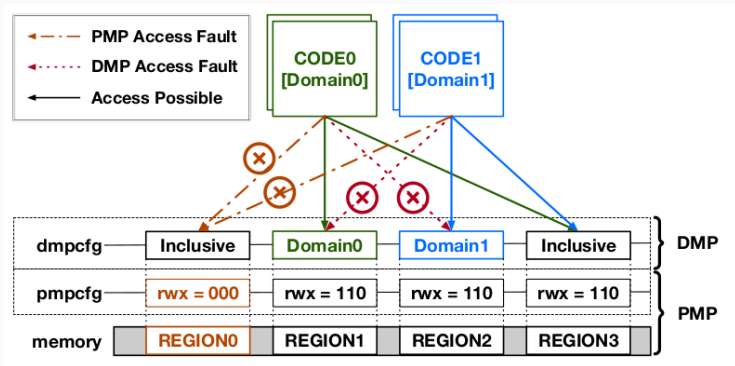
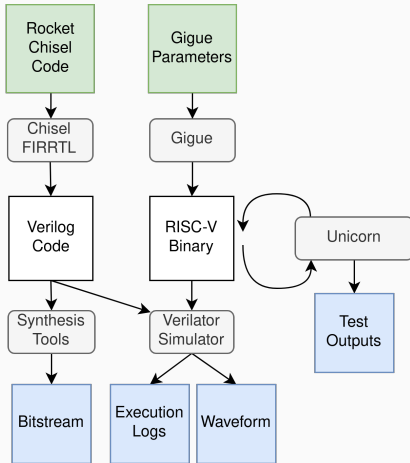


Figure 2: Kim, H., Lee, J., Pratama, D., Awaludin, A. M., Kim, H., & Kwon, D. (2020). RIMI. Proceedings of the 39th International Conference on Computer-Aided Design <https://doi.org/10.1145/3400302.3415727>



The hardware development stack is already complex and involves several steps:

1. Core definition in a high-level HDL (*Chisel*)
2. Compilation to a lower-level HDL (*SystemVerilog/VHDL*)
3. Cycle-accurate Verilator simulator compilation (*C++*)

⇒ Execute the Gigue'd binary

CVA6 implementation (in progress!)

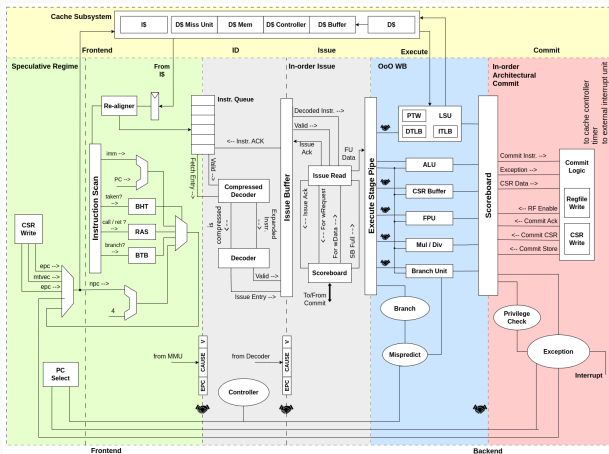


Figure 3: CVA6 default schematic. Current modifications:
<https://github.com/QDucasse/cva6/tree/jitdomain>

```
1: rocket@rocket-VirtualBox:~/Documents/jitdom
[R] - Writing test suite result..
csr/: 4/4
  1 expected successes
  3 expected failures
  0 failures
  0 errors

domain-change/: 5/6
  1 expected successes
  4 expected failures
  1 failures
  0 errors

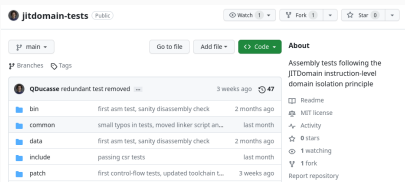
flush/: 2/2
  1 expected successes
  1 expected failures
  0 failures
  0 errors

mem-access/base/: 48/48
  4 expected successes
  44 expected failures
  0 failures
  0 errors

mem-access/duplicated/: 47/58
  3 expected successes
  44 expected failures
  11 failures
  0 errors

mem-access/shadow-stack/: 10/12
  7 expected successes
  0 expected failures
  2 failures
  0 errors

syscall/: 2/4
  0 expected successes
  2 expected failures
  2 failures
  0 errors
```



The screenshot shows the GitHub repository page for 'jitdomain-tests'. The repository is public and has 47 stars. The description states: 'Assembly tests following the JITDomain instruction-level domain isolation principle'. The repository contains several folders: 'bin', 'common', 'data', 'include', and 'patch'. The 'bin' folder has a commit from 2 months ago with the message 'first aom test, sanity disassembly check'. The 'common' folder has a commit from last month with the message 'small typos in tests, moved linker script an...'. The 'data' folder has a commit from 2 months ago with the message 'first aom test, sanity disassembly check'. The 'include' folder has a commit from last month with the message 'passing csr tests'. The 'patch' folder has a commit from 3 weeks ago with the message 'first control-flow tests, updated toolchain L...'. The repository also has a README, MIT license, and 1 fork.

<https://github.com/QDucasse/jitdomain-tests>

We presented *Gigue*, a workload generator for hardware testing:

- **Parametrizable** - Qualify VMs and applications
- **Modular** - Simplified addition of elements and instructions
- **Testing** - Software guarantees and ease of custom handling

We are implementing different security solutions in the CVA6 code: from open-source code, solutions released in open-source repositories.

Tools

<https://github.com/QDucasse/gigue>

<https://github.com/QDucasse/jitdomain-tests>

Notes about the software and the hardware

CVA6: <http://pcotret.gitlab.io/blog/tags/cva6/>

Rocket: <https://qducasse.github.io/tags/rocket/>

Gigue: A JIT Code Binary Generator for Hardware Testing

Quentin Ducasse, Pascal Cotret, and Loïc Lagadec

November 13, 2023



M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis.

The devil is in the constants: bypassing defenses in browser JIT engines.

In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*. NDSS, 2015.



A. De, A. Basu, S. Ghosh, and T. Jaeger.

FIXER: Flow integrity extensions for embedded RISC-V.

In *Proceedings of the 26th Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*, pages 348–353. IEEE, 2019.



Q. Ducasse, G. Polito, P. Tesone, P. Cotret, and L. Lagadec.

Porting a JIT compiler to RISC-V: Challenges and opportunities.




In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR'22)*, pages 112–118. ACM, 2022.






T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi.

JITguard: hardening just-in-time compilers with SGX.

In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, pages 2405–2419. ACM, 2017.

-  H. Kim, J. Lee, D. Pratama, A. M. Awaludin, H. Kim, and D. Kwon.
RIMI: instruction-level memory isolation for embedded systems on RISC-V.
In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*, pages 1–9. ACM, 2020.
-  W. Lian, H. Shacham, and S. Savage.
A call to ARMs: understanding the costs and benefits of JIT spraying mitigations.
In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*. NDSS, 2017.
-  T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz.
NoJITsu: locking down JavaScript engines.
In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*. NDSS, 2020.

-  A. Sintsov.
Writing JIT-spray shellcode for fun and profit.
Technical report, DSecRG: Digital Security Research Group, 2010.
-  A. Waterman, K. Asanovic, and S. Inc.
The RISC-V instruction set manual, Volume I: unprivileged ISA, 2019.
-  A. Waterman, K. Asanovic, and S. Inc.
The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, 2021.