Microarchitecture security, future-proof designs

Habilitation à diriger des recherches
Université de Rennes

Ronan Lashermes

Jury Members:

Reviewers:
- *David Hély*, Grenoble INP Esisar
- *Nele Mentens*, Leiden University & KU Leuven
- *Tamara Rezk*, Inria

Examiners:
- *Régis Leveugle*, Grenoble INP Phelma
- *Isabelle Puaut*, Université de Rennes
- *Arnaud Tisserand*, CNRS

Acknowledgment

I would like to express my deepest gratitude to those who have supported me throughout this journey.

First and foremost, my heartfelt thanks go to Hélène and Gaston, my dearest loves, whose unwavering support and encouragement have been invaluable.

I extend my appreciation to Guillaume Bouffard and Hélène Le Bouder for their meticulous proofreading and insightful feedback, which significantly contributed to improving this manuscript.

I am also grateful to my reviewers, David Hély, Nele Metens, and Tamara Rezk, as well as my examiners, Régis Leveugle, Isabelle Puaut, and Arnaud Tisserand, for their time and invaluable expertise in evaluating my work.

A special thanks to my colleagues at LHS, Michel Hurfin, Ludovic Mé, and Alexandre Sanchez, for their collaboration and support. I also acknowledge Patrick Gros and Éric Poiseau from Inria for enabling the financial support for the defence.

I sincerely thank my collaborators from various projects: Joseph Paturel, Simon Rokicki, and Olivier Sentieys (TARAN); Damien Hardy, Erven Rohou, and Thomas Rubiano (PACAP); and Damien Marion (CAPSULE). Your contributions have been instrumental in shaping this research.

My gratitude extends to my PhD students, Hery Andrianatrehina, Kevin Bukasa, Mathieu Escouteloup, and Amélie Marotta, for the stimulating discussions and shared experiences.

To my friends, Azura, Bichette, Garagnas, Jag, Padmoumou, Rems, Theudeux, Titoon, and Will, thank you for your support, laughter, and companionship.

Lastly, a special mention to my beloved pets, Dune, Mariette, Pixel, and Soupir, who have provided comfort.

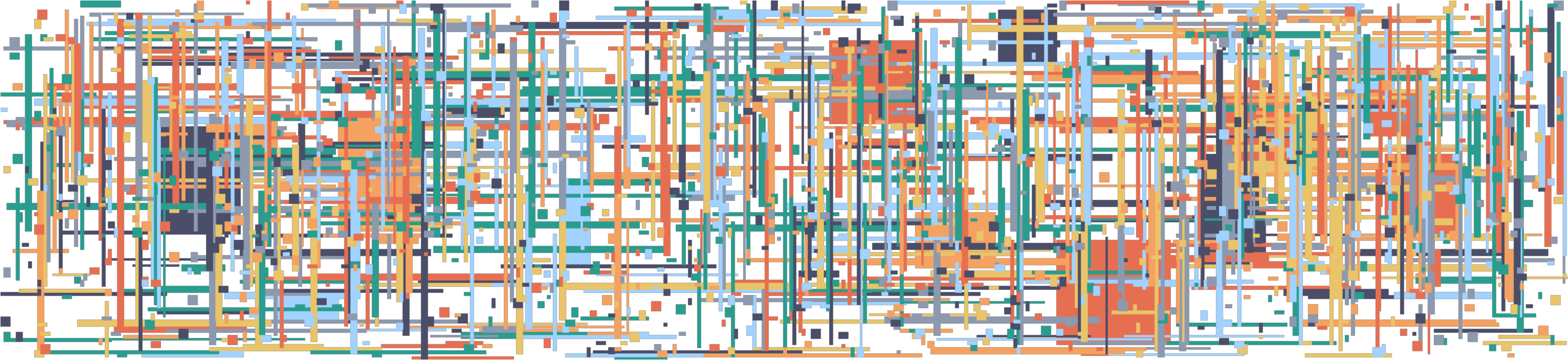
This document uses a latex style derivative of Legrand orange book.
Banner image of an experiment at the LHS taken by C. Morel.
April 11, 2025 edition

Contents

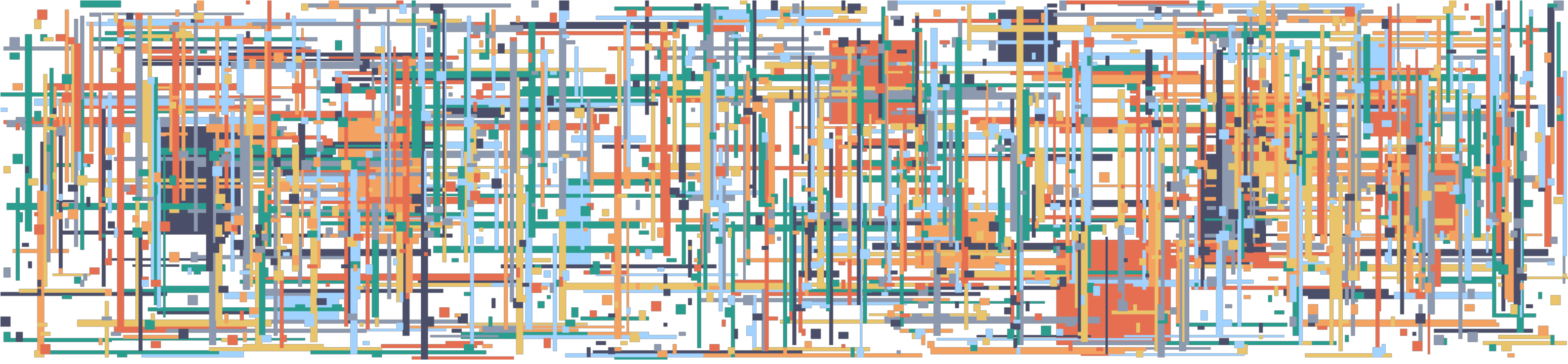
I Introduction and Prerequisites

III Radical New Core Designs for Security

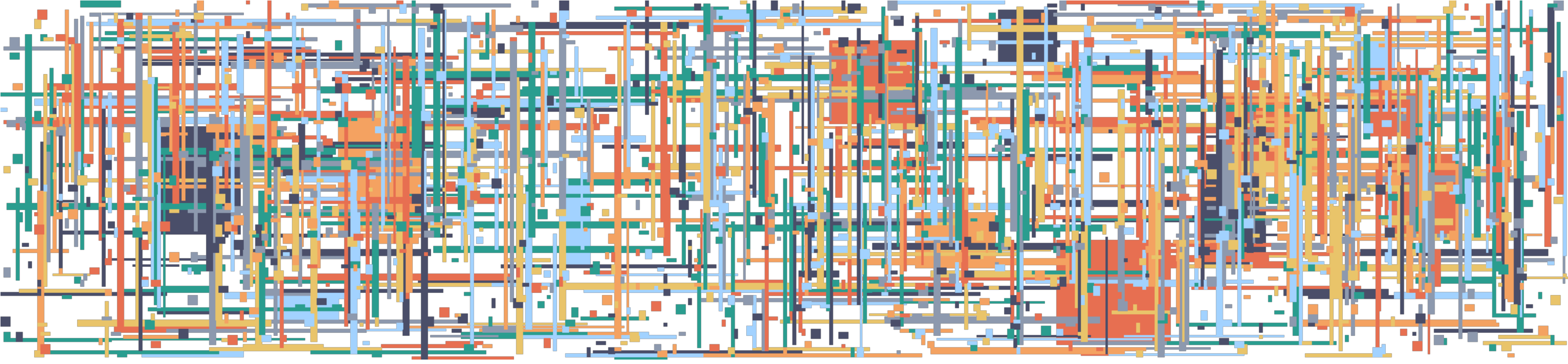
Annexes



List of Figures



List of Tables



List of Listings

Foreword

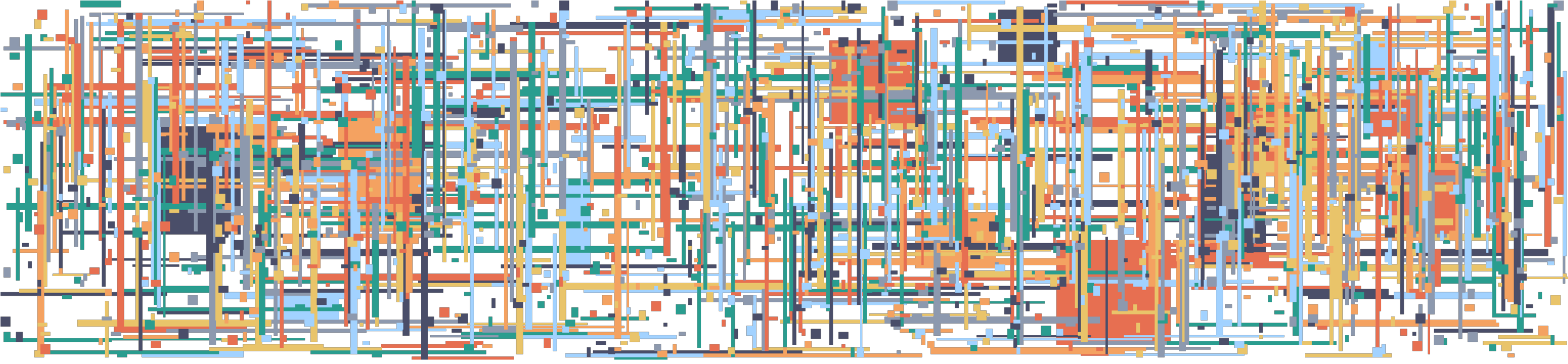
This document is my "Habilitation à Diriger des Recherches (HDR)" thesis. The constraints on the form and substance of an HDR manuscript are much looser than those for a master's or PhD thesis. In this thesis, I aim to describe coherent design schemes for hardened cores suited to various threat models. I take this opportunity to step back from the hyper-focused work typically undertaken for the purpose of publishing scientific papers. I intend to evaluate certain design choices, considering not only their technical merits in terms of performance and security, but also other aspects: does it contribute to the system's complexity? Is it easy for users to understand and reason about? Is there a likelihood that the ecosystem will adopt it?
...
Some of the ideas discussed in this document are not new and are cited accordingly. The works to which I have contributed are explicitly referenced with the corresponding citation enclosed in a red frame, as illustrated in the example below.

> **"Under the Dome: Preventing Hardware Timing Information Leakage"** Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier, and Jean-Louis Lanet. **Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021** [20]

In the digital version, you can click on the icon to download the corresponding paper.

In this document, `instructions` and `registers` are formatted in a specific way to facilitate the identification of the discussed elements. For example, `add rd, rs1, rs2` is an instruction that uses 3 registers.

I Introduction and Prerequisites



1. General Introduction

1.1 What is this document?

This document is my manuscript prepared to obtain my habilitation à diriger des recherches (HDR) diploma, a French qualification that follows a PhD and grants the ability to independently supervise PhD students.

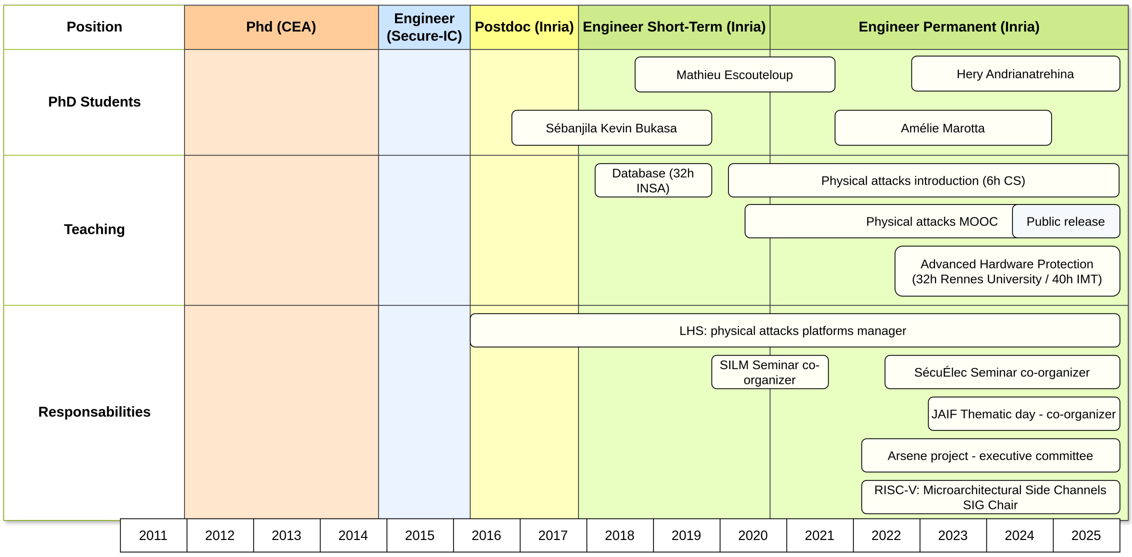
In this manuscript, I focus on the **Design** of secure microarchitectures. Unlike the more traditional technical academic work that I usually do, I aim to take a step back and analyse security solutions, whether developed by myself or others, from a system designer's point of view. Indeed, the design process is influenced by a multitude of factors, not only scientific and technical constraints but also social considerations. Security solutions are not always adopted based solely on their technical superiority but often because they enable social constructs that would otherwise be unfeasible (for instance, the ability to distrust the company that develops an OS, the ability to distrust the user, ...). The most effective design is not necessarily the most technically advanced, especially if its complexity hinders widespread adoption. Conversely, some design decisions are primarily driven by a company's market strategy and do not improve the system's security as advertised.

This type of discussion is usually not the place of academic papers on hardware security, and it is a direction I aim to address in this document. I develop and explore mental models for various security features, examining how they interact and influence each other. While the primary focus is on hardware security, effective design cannot occur in isolation. We must also consider instruction set architecture (ISA) design, the roles of compilers, programming languages, and more. I try not to delve into excessive technical details; the referenced papers do so more effectively.

To guide the discussion, I present two design scenarios. In Part II, concepts for enhancing the security of a typical modern application processor are explored. The challenge here is to propose solutions that are compatible with current microarchitectures, requiring minimal modifications. For the sake of realism, I exclude physical attacks from the scope of this section.

In contrast, in Part III, I adopt a more radical approach. How could we design a microarchitecture with enhanced security from scratch? To avoid redundancy, I consider a different threat model, focusing on secure microcontrollers that must withstand physical attacks.

I hope you find this manuscript engaging and thought-provoking. If I succeed in making you pause and reconsider why certain systems are designed the way they are, whether for security or non-technical reasons, I consider my objective accomplished.



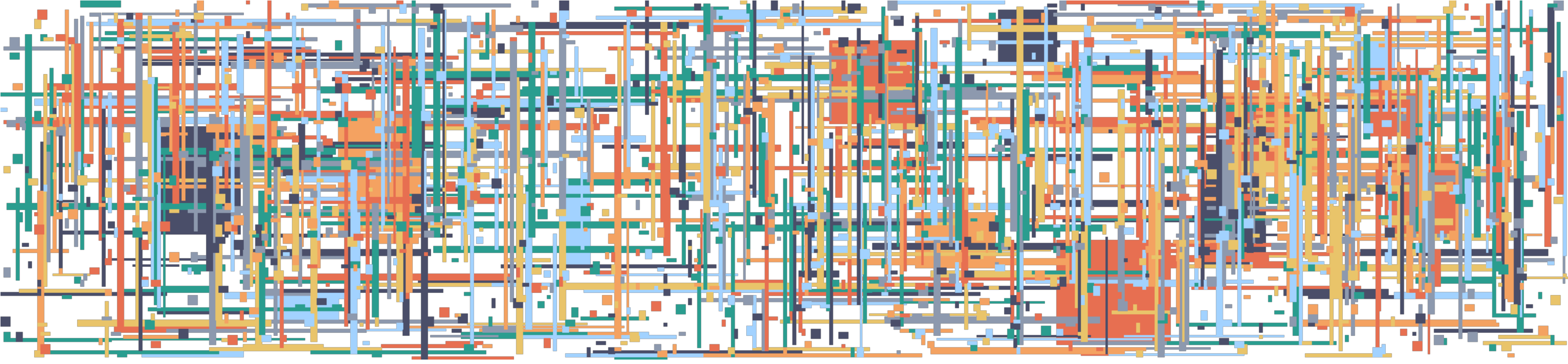
1.2 Who am I?

I am Ronan Lashermes, currently working at Inria as a research engineer. I obtained my PhD from the CEA and UVSQ in 2014, where I focused on the implementation security of pairings, a public cryptographic primitive. During this period, I developed my expertise in physical attacks, particularly fault injection, against cryptographic implementations. I worked as a research engineer (permanent position) at Secure-IC, a hardware security company, before joining Inria in 2016, initially as a postdoc and later as a research engineer.

At Inria, I developed the physical attack benches for the High Security Laboratory (LHS), focusing on electromagnetic (EM) fault injection and EM leakage characterisation. Early on, I was among those who advocated for viewing physical attacks as a generic attack vector against entire systems, not just against cryptographic implementations. When the Spectre attacks (cf section 7.3) were published in 2018, they seemed like a natural and powerful extension of the capabilities of physical attacks.

Today, I specialise in physical attacks and microarchitecture security. Whether from physical attackers or simple logic attackers, there are still numerous ways to compromise the security of modern systems. Enhancing security is not solely a technical challenge, which is why I volunteered as the Microarchitecture Side Channels SIG Chair of the RISC-V Foundation to lead efforts in hardening microarchitecture against these threats.

During my time at Inria, I advised or supervised 4 PhD students, 2 of whom have defended their theses: Kevin Bukasa and Mathieu Escouteloup, while 2 are still ongoing: Amélie Marotta and Hery Andrianatrehina. Kevin and Amélie focused on fault attacks, whereas Mathieu and Hery worked on microarchitecture security. To narrow the scope of this document, I do not discuss my work on physical attacks, except occasionally to justify specific threat models.



2. Modern Cores

In this document, we see how RISC-V cores are designed for security. But before reaching that point, I must introduce how a core capable of executing generic software typically works.

2.1 Instruction Set Architectures

2.1.1 What is an Instruction Set Architecture?

At the core of software abstraction are **instructions**. All software, when compiled, is reduced to a sequence of these elementary operations.

Instructions, and how they should be used, are defined by the instruction set architecture (ISA). The ISA establishes the standard that defines the interface between software (composed of instructions) and hardware (the logic circuits that implement these operations).

Today, two major ISAs dominate the landscape: **x86**, developed by Intel and AMD, primarily for laptops, desktops, and servers; and **ARM**, designed by Arm, Apple, Qualcomm, ..., with various implementations for embedded processors, microcontrollers, and increasingly, laptop and server processors. These ISAs are commercially developed and require licensing fees for companies wishing to implement a core based on them.

A new contender ISA, known as **RISC-V**, is gradually gaining traction [104]. RISC-V has a significant advantage in that it is an **open standard**, governed by a dedicated foundation. Anyone can join, like the author of this document, and contribute to its development. While all ISAs have unique features and differ in key areas, in this document, I focus exclusively on the **RISC-V** ISA. It has become the preferred choice for hardware implementations in academic research due to its vast open-source ecosystem, which includes software, hardware, and tooling, allowing for experimentation independent of vendors.

2.1.2 Zooming In on RISC-V

The RISC-V specification [104] defines several "base" integer instruction sets, most notably RV32I and RV64I, which support 32-bit ($\texttt{SXLEN} = 32$) and 64-bit ($\texttt{SXLEN} = 64$) variations of the base instruction set.

2.1.2.1 Registers

The ISA first defines **registers**, which are 32 small memories, each 32-bit wide for RV32I and 64-bit for RV64I, named `x0`, `x1`, `x2`, $\cdots$, `x31`. I use `this formatting` to denote a register. For instance, `x1` holds a 32-bit (or 64-bit for RV64I) value that can be read from or written to by instructions. `x0` is special: it is hardwired to zero and always returns 0 when read.

Certain registers are assigned specific roles for convenience. For example, `x1` is typically used as the *return address*, holding the address where execution should resume after a function call. Although this is merely a convention (`x2` could be used instead), these conventions, collectively referred to as the application binary interface (ABI), allow for performance optimisations. One such optimisation is the return stack buffer (RSB), a hardware feature that speeds up jumps involving the return address stored in `x1`.

2.1.2.2 Instructions

Instructions are represented as 32-bit machine codes that define the operation type, the source and destination registers, and sometimes embed immediate values (literals). Most instructions operate on registers. I use `this formatting` to denote an instruction.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0110011

Figure 2.1: The encoding of the `add` (addition) instruction. `rs1`, `rs2`, `rd` encode the corresponding register numbers on 5 bits (since $2^5 = 32$).

Exemple 2.1 - *Instructions*

```
add t0, sp, t1
load a0, 4(t0)
jal ra, func
```

This small example contains three instructions:

- `add t0, sp, t1` adds the values in registers `sp` and `t1`, and stores the result in register `t0`. The encoding of this instruction is shown in Figure 2.1, with `sp` and `t1` as the source registers `rs1` and `rs2`, and `t0` as the destination register `rd`.
- `load a0, 4(t0)` loads a value from main RAM, using the address obtained by adding 4 to the value in register `t0`, and stores it in register `a0`.
- `jal ra, func` (**j**ump **a**nd **l**ink) jumps to the address `func` and stores the return address (the address of the next instruction) in register `ra`. The `func` address is encoded directly in the instruction as an immediate value.

The base RV32I and RV64I instruction sets contain a limited number of instructions, but there are many official RISC-V extensions that provide additional functionality. For instance, the M extension introduces `mul` and `div` instructions for multiplication and division.

2.1.2.3 Other Features

The ISA also defines additional core behaviours, such as memory models that govern the behaviour of `load` and `store` instructions, control and status registers (CSRs) registers that configure the core, and much more. The unprivileged specification [104] is an extensive document, spanning 670 pages!

2.2 Microarchitecture

The **microarchitecture** is the implementation of the **ISA**, the elements in this latter specification often referred to as the **architecture**.

Definition 2.1 - *Architecture vs microarchitecture*

The registers `x0`, `x1`, ... defined by the ISA are known as *architectural* registers. A common hardware optimisation is to have more *physical* registers than the mandated architectural ones (cf subsection 2.2.2). These physical registers are dynamically assigned to reduce inter-instruction dependencies. These *physical* registers are an implementation detail; they are part of the microarchitecture. *Architectural* registers are mandated by the ISA; they are part of the architecture.

The microarchitecture defines the core's characteristics: different implementations can offer widely varying performance, energy consumption, and security trade-offs. There are many different types of microarchitectures, generally built around pipelines. A pipeline is a series of processing stages where different instructions are handled concurrently, similar to an assembly line in a factory, with each stage completing a part of the instruction. For simplicity, they are usually divided into two categories: in-order microarchitectures and out-of-order microarchitectures.

2.2.1 In-order Microarchitectures

An in-order core executes instructions strictly in the order they appear in the program. This design is typical of microcontrollers and energy-efficient application processors. A classic example is the 5-stage pipeline described in Patterson and Hennessy's book, *Computer Organization and Design* [83], as shown in Figure 2.2.

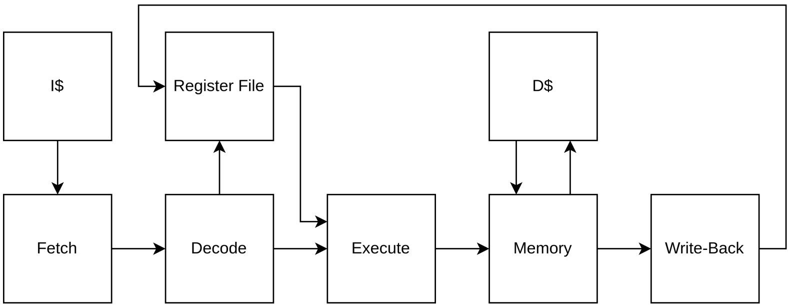


Figure 2.2: Diagram of a typical RISC-V 5-stage in-order pipeline. $ is the short name for a cache memory ($ = cash).

Each of the 5 stages has a distinct function, and at each clock cycle, multiple instructions progress through different stages simultaneously.

1. **Fetch:** Instruction data is fetched from memory according to the current program counter (PC) value.
2. **Decode:** The instruction is decoded, and the relevant information is propagated to other components (e.g. the source registers are identified, and requests are sent to the register file).
3. **Execute:** This is where the actual computation occurs (except for `load` and `store`). For example, if an addition is being performed, the two source values are added. For memory operations, this stage computes the address by adding the base register value to the offset.
4. **Memory:** Memory access is performed for `load` and `store` instructions.
5. **Write Back:** The result is written back to the register file for use by subsequent instructions.

The register file is typically an SRAM memory with 2 read ports and 1 write port. Its silicon area is often significant.

A performance bottleneck can arise if we have to wait for the write-back stage to complete before reusing a register value. If two consecutive instructions have a dependence on the same register, with one producing the value consumed by the second, several clock cycles may be wasted. A common optimisation, known as *value forwarding*, introduces a data path that feeds the output of the execute stage back into its input to avoid this delay.

2.2.2 Out-of-order Microarchitectures

Out-of-order microarchitectures trade simplicity (and hence energy efficiency) for higher performance. Many instructions within a program are independent and can be executed out of order or in parallel, as long as data dependencies are respected. Additionally, some dependencies are merely *false dependencies*:

```
A: add t0, a0, a1
B: sub t1, t0, t2
C: add t0, a2, a3
```

In this example, it might seem that instruction C must wait for instruction B to complete since they both use `t0`. However, this is merely a naming dependency; instruction C does not depend on the values computed by A or B. They are, in fact, independent.

Out-of-order microarchitectures address this by dynamically renaming registers. These implementations use more physical registers than the architectural ones, and register renaming occurs in a dedicated stage. Instructions are executed by reading from and writing to these physical registers.

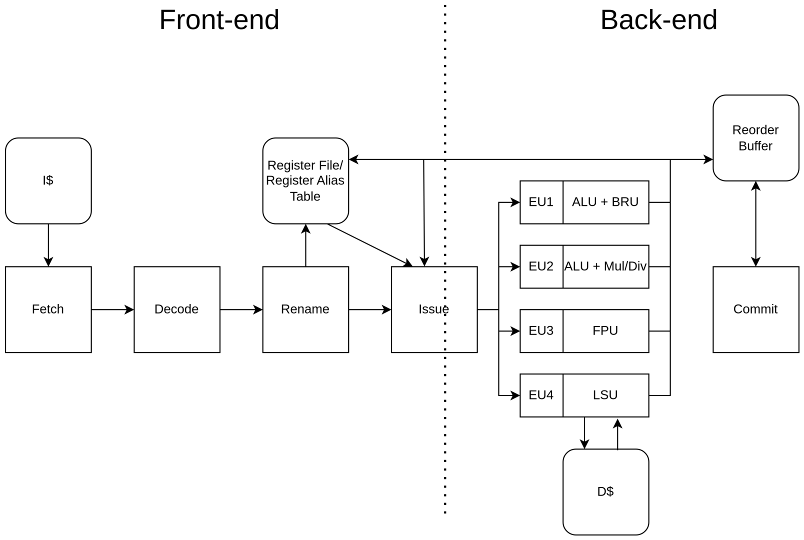


Figure 2.3: Diagram of an out-of-order pipeline.

The out-of-order pipeline, depicted in Figure 2.3, consists of an in-order front end, much like the first stages of an in-order pipeline, followed by an out-of-order back end that executes instructions in parallel and reorders them. **Reordering** is crucial to ensure correctness: if a hardware error (e.g. division by zero) occurs, the system must recover to a valid architectural state. If instructions after the erroneous one have already been executed, they must be cancelled. This is one of the tasks of the reorder buffer. When an instruction is verified (and all preceding ones are also verified), it can be **committed** to the architectural state of the core.

This reordering mechanism also facilitates **speculative execution**. Certain control flow instructions (typically branches and indirect jumps) can take time to resolve. Rather than waiting for these instructions to execute, the microarchitecture makes a prediction and continues executing from that point. If the prediction turns out to be correct, execution proceeds smoothly; if not, the incorrect execution path is discarded.

3. Security Definitions

To discuss the security of microarchitectures, I must define security in our context: what should be protected, and against whom? The main security properties for a system are confidentiality, integrity, and availability (or resilience). These properties only make sense with respect to a specific threat model, which is a conceptualisation of the attacker's capabilities. The considered threat models are presented in section 3.2. The notion of confidentiality is explored in section 3.3, and both integrity and resilience in section 3.4.

In this document, the attackers considered have low social capacities: I consider that the attacker cannot reach physically a server in a data centre. This typically excludes state actors from our threat models.

3.1 Physical Attacks and Microarchitectural Attacks

Two types of attacks are examined in this document.

Physical attacks exploit the fact that the targeted chip has a physical presence that an attacker can interact with (cf. section 5.1). Physical attacks are typically categorised into two subfamilies:

- **Fault injection attacks** involve perturbing the physical environment (e.g. power glitches, clock glitches, EM fault injection, laser fault injection, ...) to induce faults.
- **Observation attacks** (also called side-channel attacks) exploit information leakage through physical quantities such as timing, power consumption, and EM emissions...

Microarchitectural attacks exploit vulnerabilities in the microarchitecture design to extract sensitive information (cf. section 6.1). In particular, **transient attacks** rely on the speculative nature of modern processor cores to leak information through microarchitectural behaviour. Microarchitectural attacks often use timing side channels as a means of extracting data.

3.2 Hardware Threat Models

In this document, I reuse the two threat models typically considered when discussing the hardware security of central processing units (CPUs).

3.2.1 Remote Security on Application Processors

In this scenario, an attacker:

- has **no physical access** to the target chip,

- can usually execute their own program on the target (the privileges of this program may vary depending on the scenario).

In particular, attackers are able to precisely measure the timing of operations in their program. I consider remote physical attacks such as CLKSCREW [64] or Hertzbleed [68] out of scope.

The target chip is a **mobile, desktop or server-class** CPU, meaning that it features out-of-order execution and deep speculative execution, with an **memory management unit (MMU)**.

More specifically, in our case, I target an RVA23X64 RISC-V CPU (RVA23X64 is the RISC-V profile name, with X = U or S).

3.2.2 Physical Security on Microcontrollers

In this second scenario, an attacker:

- has **physical access** to the target chip,
- and can therefore measure physical parameters (power consumption, photoemission, EM emissions, ...) during computation,
- can perform physical fault injection (clock glitches, power glitches, laser and EM fault injection, ...).

The target chip is a **microcontroller**, featuring **in-order** execution, without an MMU. It **may have advanced branch predictors**, but associated speculation is shallow (the speculation window is around the depth of the pipeline). More specifically, in our case, I target an RVM23U32 RISC-V microcontroller.

3.2.3 Threat Model Justification

One may wonder why I consider only these two models. In particular, why am I not considering physical attacks on application processors or transient execution attacks (which require deep speculative execution) on microcontrollers?

Today, we are already having difficulties proposing efficient solutions for the two simpler threat models. These models allow us to isolate the two main classes of issues we face today:

1. physical attacks,
2. microarchitectural attacks.

Fortunately, these two threat models make sense: our attacker has no access to servers to perform physical attacks (low social capacity is assumed), and the microarchitecture of microcontrollers is currently too simple to allow transient execution attacks.

However, some important use cases do not fit into these models:

- Mobile phones and laptops have application processors that may be accessed by an attacker.
- Microcontrollers are evolving, featuring increasingly complex microarchitectures, and deep speculative execution is not unimaginable in the coming years.

> **!** Beware: achieving good security for these use cases is currently out of reach, and the possibility is not even being considered! The best that can be done is to build a System-on-Chip (SoC) with both security-aware application processors and hardened microcontrollers and execute applications on the correct computation substrate (cf. section 4.1).

3.3 Confidentiality: What Should Be Secret?

One of the core functions of secure applications is to ensure confidentiality: some data must remain secret from unauthorised entities. In order to build a threat model, we should ask ourselves what data must stay secret, and with respect to whom.

Here are examples of confidential data of various types:

- **Monetary advantages**: This can include information that may bias a competition (e.g., player positions in a video game competition) or where any bias may impact the monetary gains of participants.
- **Privacy-related data**: This includes private information that may harm someone if made public (e.g., medical information, personal address, bank account numbers, ...).
- **Industrial/State secrets**: These are assets that provide a competitive advantage over competitors: the secret recipe of a nearby restaurant, or the position of a state's military assets.
- **Authentication tokens**: Passwords and personal identification numbers (PINs) are required to authenticate oneself to a system. They are usually gates to unlocking some functionality. In most cases, they should not be shared.
- **Cryptographic keys**: These are required to communicate confidentially with others. They may be shared (in symmetric cryptography) with selected entities but must be kept secret from others to prevent them from accessing the communication content.

The nature of a confidential asset is tied to different properties:

1. The severity of the asset (from secret defence to grandma's cookie recipe).
2. The confidentiality **lifetime**, which is the duration for which confidentiality must be maintained. This lifetime cannot realistically be infinite.
3. The **data volume** of the secret. This information is important since cryptographic primitives are rated for a maximum data volume per key.

Thanks to encryption, the confidentiality of data can be reduced to the confidentiality of cryptographic keys. To protect one company's secrets, encrypt them. They then only need to protect the cryptographic key, which is much smaller, and ensure the confidentiality of the process when dealing with this data (e.g. decrypting and editing). This reduction property is used to define a Root-of-Trust in section 4.1.

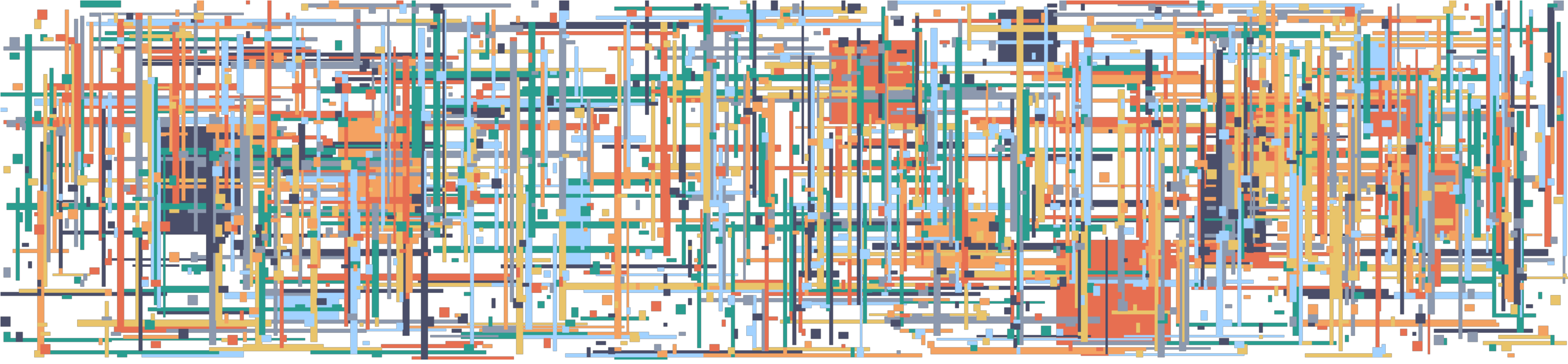
3.4 Integrity and Resilience in the Microarchitecture

Similarly, we want to prevent attackers from altering our data. Several kinds of integrity are important:

- Data integrity: The attacker cannot alter data.
- Instruction integrity: The attacker must not modify the instructions that are executed.
- Control-flow integrity: The attacker must not be able to modify how instructions are chained together.
- Architectural state integrity: The architectural configuration (privilege level, MMU configuration, ...) must not be altered.
- Microarchitectural state integrity: The microarchitectural state (branch prediction, cache policy manager, ...) must not be altered.

Execution integrity is the combination of Instruction, Control-Flow, Architectural, and Microarchitectural integrities.

In addition to these integrity issues, we would like our chip to continue working normally in the presence of naturally occurring faults, for example from cosmic radiation, while we would like to erase secret data upon an active fault (e.g. from EM fault injection). Since it is generally not possible to distinguish between the two cases, we want a resilient core that is not only capable of detecting an integrity violation but also capable of remediating it. Resilient microarchitectures are discussed in chapter 12.



4. Security Beyond the Microarchitecture

The security of the microarchitecture must align with its intended use. In this chapter, I present how secure microarchitectures are used.

4.1 System-on-Chips

The processing circuit that implements the ISA is called a core, excluding peripherals, L2+ caches, etc. A core, whether in a microcontroller or an application processor, is typically embedded in a broader system called a SoC. The SoC may include multiple cores, peripherals, memory, and other components.

A large and complex system like a modern SoC presents a significant attack surface, often coupled with non-resilient dynamic random-access memory (DRAM) memory, as discussed in section 3.4.

We have seen in section 3.3 how ensuring the confidentiality of a large dataset is challenging, leading to a strategy of encrypting the data and protecting the much smaller cryptographic key.

Similarly, a potential approach to secure a SoC is to concentrate security features, such as confidentiality and integrity, on a smaller part of the SoC. The hardware security module (HSM) is the SoC component responsible for critical security services: it can store cryptographic keys and perform cryptographic operations, like encryption, autonomously. This component is typically responsible, during boot time, for verifying the boot image via a signature check before execution. The boot image can then verify the next image, such as the operating system (OS) image, and execute it, establishing a chain-of-trust (CoT). This process, called *secure boot*, places the entire system's security responsibility on the HSM; one must trust that the HSM works correctly, but they *only* need to trust it. In this context, the HSM is known as the root-of-trust (RoT).

One possible implementation of the HSM is a secure element (SE), a security-oriented microcontroller similar to those used in secure smartcards. A SE should feature a dual-lockstep microcontroller with hardware security hardening (e.g., attack sensors, component duplication, memory encryption).

Sometimes, the trusted execution environment (TEE) handles the security role of the HSM. TEEs, such as Arm's TrustZone or Intel SGX, often execute secure applications on the same core as unsecure ones but in a different execution context.

4.2 The Responsibility of Security

Security is a multifaceted concept, and system designers must make compromises with opposing constraints. First and foremost, the device should ensure the security of the **users**: the confidentiality of their data, the integrity of their applications, ... However, we must recognise other antagonistic goals. The **device owner** may want the possibility to recover data; for example, when an employee leaves the company. **Service providers** (e.g. video and music streaming services) may want to store assets on the device that must not be accessible to both the owner and the user.

Ethics suggest that the loyalty of a device should go to the user [17, 77], while commercial success often favours the device owner (assuming they are the buyer), and users want to use the device to access data from service providers. Meanwhile, law enforcement would like to have access to everything, when a device is recovered from a crime scene for example. These opposing interests, which are ultimately a political struggle, have influenced the technical architecture of modern SoCs. Not everyone was a fan. In his famous speech to the CCC, Cory Doctorow [89] explicitly described the risks resulting from these conflicting interests. From these arguments, the Linux community initially rejected the Secure Boot model relying on a HSM that would allow the HSM provider to choose what can boot on the system.

In the rest of this section, the security model underlying the modern SoC organisation is examined from this new perspective.

4.2.1 Limitations to the Secure Boot Threat Model

When discussing techniques to ensure data confidentiality, I emphasise that instead of trying to guarantee that the entire system memory is tamper-resistant, one should encrypt the data instead. Then, one only needs to ensure that the memory storing the cryptographic keys is immune to attacks, resulting in a much smaller attack surface. This process is relevant only for data at rest. If one needs to edit the data, it must be decrypted and modified in the clear. Thus, a secure method, resilient to attacks, is needed to process confidential data.

Drawing a parallel with the CoT involved in secure boot, something remains missing in this approach. The RoT may be immune and can guarantee that the boot image has not been tampered with, but it is not a sufficient countermeasure against an attacker targeting the hardware. While verifying and measuring the boot process and then comparing it to a golden reference (a technique called *measured boot*) may increase confidence that no attack has occurred, it is not foolproof. Even if measured boot is perfectly secure, an attacker can still target any application or OS functionality to gain system control, for example through a fault attack during execution [91].

Even if it complicates matters for the attacker, if secure boot can be bypassed through fault injection [22], it remains an insufficient solution given the complexity of both hardware and software in the *secure boot* mechanism. What is the cost of securing the Nintendo Switch boot process compared to the cost of an attack demonstrated in [91]?

4.2.2 The Threat Model Might Not Be The One Marketed

A possible explanation is that modern SoC security architecture is not designed to protect the user running the SoC against an attacker as the first objective; it is designed to allow service providers to store confidential data within the SoC while distrusting most of the system, including the users themselves. Service providers are not directly impacted if a user loses their personal data to attackers, but they are much more concerned if they lose their own assets to attackers (see for example the AACS encryption key controversy[105]). Consider the movie or music industries as examples, where DRM keys are stored in the HSM.

Secure boot should be understood as a mechanism that primarily enables third parties to distrust the user. We should recognise that the design process is driven by conflicting interests, often resolved through commercial success as a form of evolutionary pressure. If one needs a HSM to watch their favourite movies, they will buy the SoCs specified by the service provider. An example is Windows 11's requirement for TPM 2.0, a specific HSM improving security with respect to previous generation TPM, rendering obsolete many otherwise functional laptops and desktops.

A counter-argument is the attacks [22, 91] discussed earlier, which pose a real threat to service provider assets. Indeed, I believe hardware attacks are often downplayed in threat evaluations because these attacks have significant limitations.

1. They do not scale. A glitch attack against a Nintendo Switch provides access only to that specific device[91].
2. Assets within the HSM remain secure. The DRM keys are usually safe; it is the media on the compromised device that can be decrypted. This represents a loss for the service provider but is preferable to exposing the keys publicly[105].

Despite these limitations, the poor track record of hardware security [91] underscores that it should be taken more seriously.

4.2.3 The Sociology of Security in the Design Process

The technical and sociological aspects of design are always intertwined and must be recognised as such. The merits of a particular solution should therefore be evaluated based on how well it fits the intended use cases, in addition to its technical efficacy.

Exemple 4.1 - *Good technical solutions may not be enough*

The data integrity issue, caused by cosmic and radioactive radiation, highlights the importance of social constructs in security design. It is not enough to devise a working technical solution; it is also necessary to create the conditions for its widest possible diffusion.

In [58], the authors analyse the fault rates of DRAM memory observed in data centre servers over a period of 2.5 years. They observe the rate of correctable faults to be $2000 - 6000$ per GB per year, a significant value. These faults are correctable since server-class DRAM memories typically feature error correcting code (ECC), a mechanism that trades some memory space for data redundancy, allowing for recovery from isolated bit flips due to radiation.

Interestingly, ECC is not featured in most desktop-class DRAM memories. Indeed, fault resilience is used as a market discriminatory feature: only knowledgeable buyers are supposed to care about this, and those buyers can pay more. ECC should be more expensive for the same memory capacity because of the redundancy, but not at the level seen commercially. This trend is slowly changing for the better, and now high-end desktop-class CPUs from AMD and Intel can support ECC DRAM.

Remember, if naturally occurring radiation can corrupt data, an attacker can do so with even greater precision.

The design features of a technical solution can influence its sociological impact and must be considered accordingly. Following Donald Norman's principles [82], good design has the following properties:

1. Design your solution in a way that allows the user to create a mental model of it.
2. In the best case, this mental model already exists and is shared among your target audience.
3. Make it hard to use it wrong, aka *friction*.

4. Make it easy to use it right.

Exemple 4.2 - *`unsafe` Rust*

A great example is the explicit goal of adding *friction* to undesirable constructs in the Rust programming language. Typically, the `unsafe` keyword is used to bypass the usual memory safety rules enforced by the language, sometimes necessary for interfacing with libraries written in C. This is a dangerous feature that must be used sparingly. As a common practice, crates (Rust's term for libraries/packages) are often divided into unsafe ones that interface with unsafe C libraries, typically named with a `-sys` suffix. These unsafe crates are then wrapped within a safe crate, with an application programming interface (API) leveraging all the safety features provided by Rust.

Based on these principles, the technical content of this document is organised into two main parts, aimed at improving system security for users.

In Part II, I consider technical solutions that may fit today's sociological security models. With these techniques, there is no need to alter the organisational structure of entities producing the devices. These are incremental changes, following a "business as usual" approach.

In contrast, Part III explores more radical approaches that could shift the responsibilities of stakeholders within the security ecosystem. For example, developers may now need to annotate confidential variables (cf. chapter 10), or a new late phase of compilation, called *installation*, must be performed in-situ on the device (cf. section 12.3). Such radical changes should be recognised as such, immediately rendering these solutions unrealistic in the short term, even if they are scientific achievements.

Many elements must align to achieve a secure system. In this document, I focus on the cores themselves, leaving aside other critical components (RNG, secure memories, ...).

Thesis In my view, the typical SoC design, focused around an HSM and application cores with minimal security features, is not sustainable for the future as it fails to provide sufficient protection against attackers, particularly those targeting hardware vulnerabilities. The underlying reasons for these technical shortcomings often lie in non-technical influences on the design process, where the ability to distrust external actors, while enhancing supply chain and provisioning security, does not necessarily improve the security for the SoC user. Recognising this "Chesterton's fence", we can embark on a blank slate redesign of the cores to obtain future-proof designs for microarchitecture security.

II

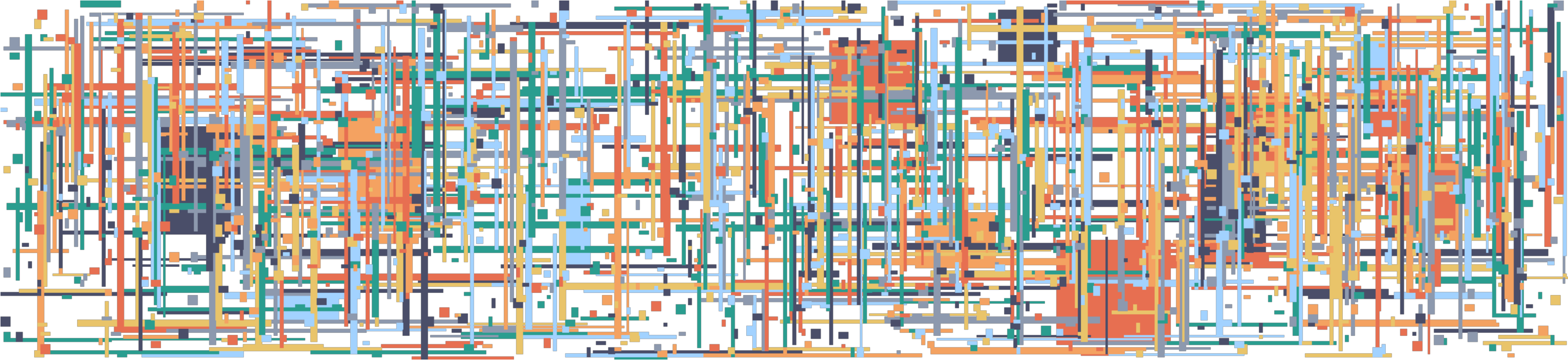
Improving Security in Today's Cores

As outlined in Part I, the security of modern microarchitectures requires significant improvement. Numerous attacks continue to be discovered, and mitigation measures are often applied reactively. These mitigations frequently lack a principled approach. They aim to prevent specific exploits while leaving underlying vulnerabilities exposed, which may still be exploited through other means. This exemplifies suboptimal management of system security.

Several factors contribute to this challenge, one of the primary being that security is just one among many competing design metrics. Furthermore, other security weaknesses are often prioritised, with good reasons. For instance, memory safety was identified as the root cause of 70% of vulnerabilities in 2019 [45]. However, gradual improvements are being made toward memory safety with the adoption of better programming languages such as Rust, Ada, Frama-C, ...more advanced vulnerability detection tools (e.g., fuzz testing becoming standard practice), and the inclusion of static analysis passes in compilers (e.g. introduced in GCC in 2020).

With these advancements, enhancing microarchitecture security remains imperative, as attackers continue to seek new ways to exploit weaknesses when memory safety is properly implemented.

This part examines strategies to improve the microarchitectural security of modern cores without necessitating a complete redesign, a process that would be prohibitively costly given the complexity of current architectures. To limit the scope, this part focuses on techniques for mitigating **covert channels** (chapter 5 and 6) and **transient execution attacks** (chapter 7 and 8) using solutions that are realistically applicable to existing cores.



5. Using Timing Measurements to Exfiltrate Information

The first family of vulnerabilities explored in this document are observation attacks on applicative processors leveraging timing measurements as the leaking physical quantities.

They are called **timing channels** and refer to the possibility for an attacker to exfiltrate data through variations in the execution time of certain operations.

5.1 Covert and Side Channels

The ability of an attacker to exfiltrate and retrieve a secret can be viewed as the establishment of a communication channel. The attacker's goal is to establish this communication channel across an architectural security boundary. For example, this could allow a user-mode process to read a secret manipulated during a system call executed in supervisor mode.

The sender in this communication is generally referred to as a **Trojan**, while the receiver is called a **Spy**, as illustrated in Figure 5.1.

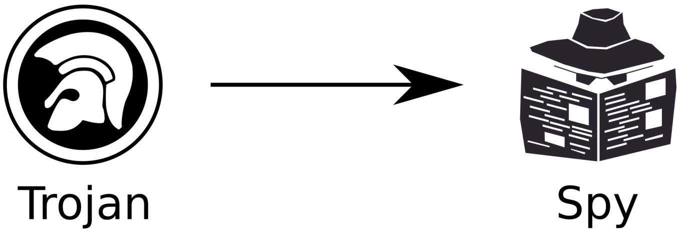


Figure 5.1: Covert and side channels are illegitimate communication channels between a sender, called the Trojan, and a receiver, called the Spy.

Two scenarios are considered:

- **Covert channels** refer to communication channels where the attacker controls both the Trojan and the Spy.
- **Side channels** are communication channels where the attacker controls only the Spy, while the Trojan's role is inadvertently played by the victim.

The characteristics of this communication channel are determined by measuring its **channel capacity**, a theoretical measure of the maximum amount of information that can be transmitted per message (and, by extension, per unit of time). This channel capacity can be evaluated, analytically in simple cases, or empirically in the general case, using the matrix that characterises the channel. This matrix, denoted as $\mathscr{M}_{e,t}$, is constructed by determining the probability that a symbol t sent by the Trojan is received as symbol e. A perfect channel is characterised by a channel matrix equal to the identity matrix. In practice, however, timing channels are rarely perfect due to noise introduced by microarchitectural activity.

Even if the probability of correctly reading a value from the channel is less than one, corresponding to noise in the communication channel, it is still possible to design a reliable communication protocol.

5.1.1 Support for Covert and Side Channels

This abstract definition does not specify which physical quantity is leveraged to transmit information.

Timing-based covert and side channels measure the duration of specific events, such as cache memory access time, as the transmitted symbol. Timing channels are particularly significant because timing measurements can be taken remotely, for instance, over a network, making them a potential threat to any connected device.

On the other hand, power or EM channels use voltage levels as transmitted symbols. Indeed, complementary metal-oxide-semiconductor (CMOS) technologies consume different amounts of power depending on whether a logic gate outputs a '0' or a '1' and whether it switches states or not. By measuring power consumption or EM emissions, an attacker can extract information about the data being processed by a device. These channels typically require the attacker to have physical access to the device[38].

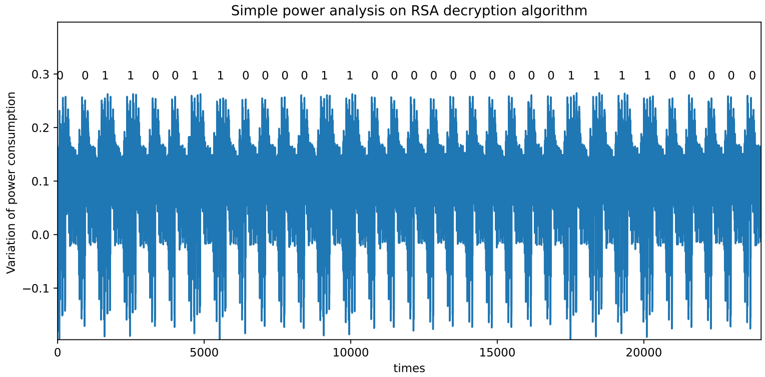


Figure 5.2: Simple Power Analysis against RSA's Square and Multiply algorithm (From Jonathan Amatu, Maël Leproust, Salim Sama Mola et Alexis Prou work at IMT-Atlantique)

5.1.2 Timing Threat Models

An attacker capable of writing a secret to memory and later reading the value can establish a covert channel. However, if the ISA permits such an operation, it would generally be classified as a software vulnerability rather than a covert channel.

Architectural and microarchitectural timing channels are usually distinguished. An **architectural timing channel** is a communication channel that arises from variations in instruction patterns (architectural in the sense of the instruction set architecture (ISA)). A **microarchitectural timing channel** enables communication using the same instruction pattern but exploits variations introduced by the microarchitecture itself.

In this document, I focus on the following threat models:

1. **Architectural timing side channels:** The control flow of a program depends on secret data. Typical examples include PIN verification (illustrated in listing 5.1) and modular exponentiation.

2. **Microarchitectural timing covert and side channels:** The value is transmitted by modifying a microarchitectural state. The attacker may deliberately execute a gadget (a short sequence of instructions) within the victim's security domain to alter this microarchitectural state.

5.2 Architectural Timing Channels

5.2.1 Presenting Architectural Timing Channels

The most well-known architectural timing channel is likely the improper PIN verification that is not executed in constant time.

Listing 5.1 A function that compares two byte arrays. The function returns as soon as it detects a mismatch. This behaviour leaks information through timing side channels, revealing which digit is incorrect.

```
1  // A naive PIN verification function compares two byte arrays:
2  // the secret PIN and the user input.
3  bool compare_array(uint8_t *a, uint8_t *b, size_t len) {
4      for (size_t i = 0; i < len; i++) {
5          if (a[i] != b[i]) {
6              return false; // leaks information
7          }
8      }
9      return true;
10 }
```

In listing 5.1, the function `compare_array` terminates as soon as it encounters a difference between the two arrays line 6. This allows an attacker to measure the function's execution time and infer the index `i` of the first mismatching digit.

With this information, an attacker can attempt all 10 possible values for the first digit and select the one that results in a longer execution time as the correct one. They can then proceed with the second digit, and so on, reducing the number of possible PINs for a 4-digit PIN from $10^4 = 10\,000$ to only $4 \times 10 = 40$. At first glance, 40 may seem sufficiently high, especially when an attacker is limited to only 3 attempts. However, in the context of smartcard smuggling, attackers consider the probability of success over numerous stolen cards. The difference between a 3 in 10 000 chance and a 3 in 40 chance significantly alters the return on investment for criminals.

5.2.2 Current Solutions Against Architectural Timing Channels

Having established the threat, potential solutions are discussed in this section.

5.2.2.1 Constant-Time Code

The most straightforward solution, simple in appearance, is to ensure that the program control flow does not depend on secret data.

The PIN verification function can be rewritten, as in listing 5.2, to maintain a unique control flow, eliminating early returns when a mismatch occurs.

Listing 5.2 A function that compares two byte arrays in constant time.

```c
// A constant-time PIN verification function compares two byte arrays:
// the PIN and the user input.
bool compare_array(uint8_t *a, uint8_t *b, size_t len) {
    uint8_t result = 0;
    for (size_t i = 0; i < len; i++) {
        result |= a[i] ^ b[i];
    }
    return result == 0;
}
```

The exact guarantee sought is **data-independent execution time**. In most cases, this guarantee is provided by constant-time instructions, which inherently ensure execution time does not depend on input data. However, achieving constant-time execution can sometimes be challenging due to timing variations introduced by the microarchitectural state, such as cache contention.

Ensuring the production of constant-time code by compilers remains an active area of research [4, 7]. Cryptographic implementations are often written in hand-optimised assembly to guarantee constant-time execution, particularly because mainstream compilers are still largely unable to enforce this property reliably.

Avoiding Branches The most straightforward countermeasure is to prohibit any branch instruction that takes a secret register as an argument. However, this is easier said than done; how is it determined whether a register holds a secret?

On one hand, there is no hardware concept of a "secret-holding" register; registers are simply storage locations. Then it is necessary, not only to prevent branching based on secret values, but also on any value derived from a secret. This implies the need to track the propagation of values that depend on secrets.

These challenges lead to interesting research topics, particularly in compiler design. However, they are considered beyond the scope of this document.

CMOV Semantics Conditional move (`CMOV`) instructions can replace certain branches, maintaining a linear control flow instead. In the RISC-V ecosystem, these can be implemented using the `Zicond` extension, which is now part of the RISC-V unprivileged specification. This extension defines two instructions:

- `czero.eqz rd, rs1, rs2`, which "moves zero to `rd` if `rs2` is equal to zero; otherwise, it moves `rs1` to `rd`."
- `czero.nez rd, rs1, rs2`, which "moves zero to `rd` if `rs2` is nonzero; otherwise, it moves `rs1` to `rd`."

Using these instructions, the `CMOV` semantics can be implemented with three instructions, as shown in listing 5.3.

Listing 5.3 Conditional selection: if `rc` is zero, then `rd` ← `rs1`; otherwise, `rd` ← `rs2`.

```asm
czero.nez rd, rs1, rc
czero.eqz rtmp, rs2, rc
or rd, rd, rtmp
```

5.2.2.2 Zkt

One issue with purely architectural solutions is that they inherently rely on assumptions about microarchitectural behaviour. For example, the function in listing 5.2 is constant-time, but only under the assumption that an XOR operation between two bytes has a constant execution time. Is this assumption valid? In most cases, yes, but there is no formal guarantee: it depends on the microarchitecture.

The RISC-V ISA addresses this issue by introducing an extension: **Zkt** [98].

The core idea behind this extension is to define a list of instructions that must have data-independent execution durations. In particular, the instructions `czero.eqz` and `czero.nez` fall under this guarantee.

While this marks an improvement over the current state of affairs, I believe this extension provides fewer guarantees than one might expect. Specifically, the extension enforces data-independent execution times for individual instructions, but it makes no guarantees regarding the data-dependent duration of instruction sequences. It remains possible for two constant-time instructions to form a sequence whose execution time varies with data. For instance, microarchitectural optimisations such as special-case handling (e.g. when a register equals zero) can still introduce timing variations.

Therefore, it is not possible in my opinion, to completely dissociate architectural versus microarchitectural timing channels. While it is necessary to prevent architectural timing channels, it is not sufficient.

6. Preventing Microarchitectural Timing Covert and Side Channels

The previous chapter introduced architectural timing channels, where timing variations were due to instructions. But if one prevents architectural timing channels, other possibilities exist: timing variations due to the microarchitecture. This chapter describes this recent threat and how to counter it.

6.1 Microarchitectural Covert Channels

Microarchitectural timing covert and side channels can occur even in the **absence** of architectural channels. In other words, it is possible for the constant-time implementation in listing 5.2 to still leak information through timing variations. For example, if the microarchitectural implementation of the `xor` instruction had timing dependencies based on its inputs, an unlikely scenario in practice.

More realistically, modern microarchitectures inherently introduce timing dependencies, primarily as a byproduct of performance optimisations. These include mechanisms such as cache memories and branch/jump prediction, which can inadvertently create exploitable timing variations. Such variations enable attackers to exfiltrate secrets that should otherwise remain inaccessible.

A concrete example is provided in subsection 6.1.1, where the BHT branch predictor is used as a communication channel.

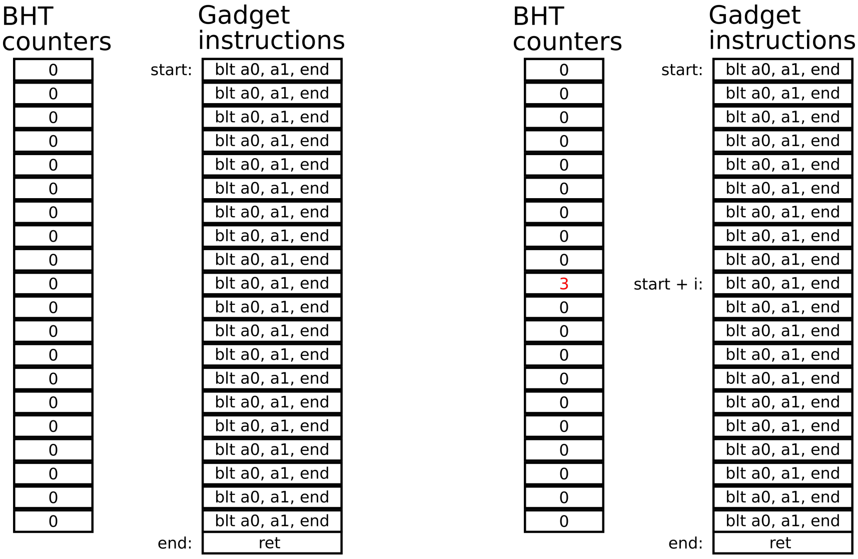
6.1.1 Case Study: Using the BHT as a Covert Channel

Any microarchitectural structure that utilises memory elements can serve as a basis for covert channels. Branch predictors are no exception, and this section demonstrates how to construct one using the BHT.

A BHT is a branch predicting structure that associates a 2-bit counter to the least significant bits of the address of an instruction (e.g. 4 bits for 16 counters). This counter memorises the number of times associated instructions had a taken (T) condition in the last 4 branches. The counter's most significant bit is the next branch direction prediction.

The idea behind the covert channel is to use a gadget that allows forcing the state of the BHT table. This gadget consists of several identical `blt a0, a1, end` instructions, which compare the two registers `a0` and `a1` and branch to the address of the label end if `a0` is strictly lower than `a1` (*branch if lower than*). There are as many branching instructions as there are counters in the BHT. Finally, the last instruction at the labelled address end is a function return (`ret`).

The attack begins with an initialisation step, which consists of clearing all the counters by executing the first branch with a not-taken (N) condition (`a0` $\not<$ `a1`), as illustrated in Figure 6.1a. This operation is repeated three times to set all counters to 0.



(a) Initialising counters to 0.

(b) The Trojan selects a counter to increment by executing branch i with the taken (T) condition.

Figure 6.1: The BHT states at initialisation and after the message emission by the Trojan.

Next, the Trojan selects the value i to encode, corresponding to a unique counter in the BHT. It then executes the corresponding instruction three times with the taken (T) condition (`a0` $<$ `a1`), as shown in Figure 6.1b. Due to the condition, only this branch is executed, as it jumps to `end` for the next instruction.

Finally, the Spy tests the value o, corresponding to a counter, by measuring the execution time of this branch under the T condition. If execution is fast, then $i = o$; if it is slow, the values differ.

By measuring execution times for all combinations of i and o, a pattern as in Figure 6.2 is obtained, highlighting the presence of a covert channel.

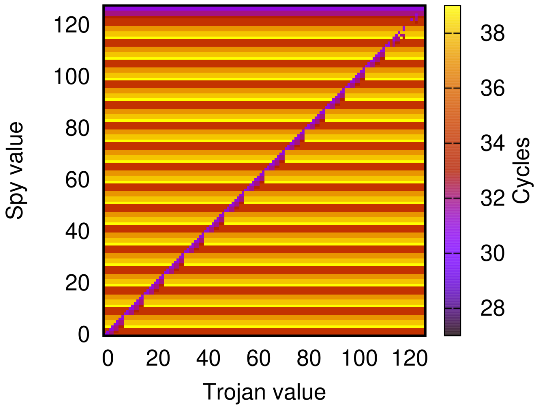


Figure 6.2: The timing matrix on the Aubrac core, highlighting the presence of a covert channel on the BHT. From [20].

From the timing matrix in Figure 6.2, and using a Spy's decision criterion, for example, time < 32 cycles, it is possible to construct the probability matrix characterising the channel and thereby measure its capacity. Specifically, it is sufficient to determine the probability of making a decision, normalised by column (according to the value of the Trojan's message).

The absence of a vulnerability, meaning the impossibility of establishing a channel with this specific timing channel, would imply that execution times are independent of the Trojan's value. This would be reflected in a timing matrix where all rows remain constant. In a more realistic setting, execution times are influenced by independent factors, such as software running in parallel on the same processor but on a different core that shares the last level cache (LLC). Thus, in this case, there is noise in the measurements, making a stochastic approach more appropriate.

6.1.2 Microarchitectural Elements Likely to Create a Covert Channel

Historically, data caches were the first microarchitectural elements to be exploited as covert channels [8, 95]. The fundamental observation is that the time required to access memory depends on whether the data is present in the cache.

However, caches are just one of many potential covert channels. In general, any stateful microarchitectural element can be leveraged as a communication channel.

This includes:

- L1D, L1I, and LLC caches [8, 95],
- Micro-op caches [52],
- Page tables [12],
- Translation lookaside buffer (TLB) structures [28],
- BTB structures [1],
- RSB (Return Stack Buffer),
- Branch predictors [1],
- Prefetchers [62],
- Port contention due to SMT [3],
- State machines such as cache controllers [61],
- Performance counters [49],
- Dynamic voltage and frequency scaling (DVFS) [69],

Each covert channel leverages microarchitectural elements in clever, often subtle, ways. Many of these techniques are complex and rely on attack scenarios that may not always be practical. As a result, assessing the real-world exploitability of these channels is often difficult.

However, the underlying vulnerabilities are real, and attack techniques continue to evolve. **Hardening cache memories alone cannot be considered a sufficient solution for preventing covert channels and, by extension, transient execution attacks.**

As discussed in section 5.1, covert and side channels are distinguished by the nature of the emitter in the communication channel: the attacker for covert channels, the victim for side channels.

But some scenarios can be ambiguous. For example, if the emission of secret data originates from a gadget within the victim's program, but the attacker triggers this gadget (e.g. by influencing branch prediction), the classification can be debated. However, in this document, I classify such cases as **covert channels**, since the attacker's actions are the source of the emission. However, others may argue that this should be considered a side channel.

From a security design perspective, circuit designers focused on hardened security should prioritise countering covert channels. This approach assumes a more powerful attacker model, and a system that effectively prevents covert channels will inherently prevent side channels as well.

6.2 Countermeasures to Microarchitectural Timing Covert Channels

"Under the Dome: Preventing Hardware Timing Information Leakage" Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier, and Jean-Louis Lanet. **Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021** [20]

This section presents results from the PhD Thesis of Mathieu Escouteloup that I advised.

Microarchitectural data sharing, while responsible for covert channels, is a desirable feature. Without it, core performance would suffer significantly. However, depending on the application, selectively disallowing data sharing becomes necessary.

Preventing data sharing is commonly achieved by partitioning state. Partitioning can be categorised into two main types [20]:

- **Temporal partitioning** means that at any given time, the microarchitectural state is exclusively associated with a single security domain. Consequently, switching to a new domain requires flushing the entire state, which can be prohibitively slow in some cases.
- **Spatial partitioning** means that, at the same point in time, different security domains have separate, dedicated microarchitectural resources. This approach prevents direct interference between domains without requiring frequent state flushes. However, it may require additional hardware support and careful resource allocation to avoid inefficiencies. Two variants of spatial partitioning exist:
  - **Tag-based partitioning** assigns a tag value to each microarchitectural resource entry, indicating the associated security domain. Since tag-based partitioning can dynamically adjust the proportion of entries allocated to each security domain, it allows for a high-performance solution. However, it does not protect against covert channels: entry allocation itself can be exploited as a covert channel.
  - **Static partitioning** is required to prevent covert channels. In this approach, resource allocation remains constant for the lifetime of any security domain, ensuring that no unintended information leakage occurs.

A fundamental challenge is deciding **when and across which boundaries** partitioning should be triggered. To minimise ISA modifications, enforcing microarchitectural state isolation can leverage existing architectural concepts already specified by the ISA, despite their original designs not considering security explicitly.

6.2.1 Leveraging Existing Architectural Security Boundaries

The absence of explicit security domains complicates countermeasure implementation. Nevertheless, the ISA specifies concepts naturally mapping to security boundaries:

- **Privilege levels** define microarchitectural states wherein certain features are accessible only with appropriate privilege. RISC-V defines four privilege levels: *User*, *Supervisor*, *Hypervisor*, and *Machine*, ordered from least to most privileged. Certain CSRs are accessible only at supervisor or machine levels. For example, supervisor mode can modify the `satp` register, controlling virtual memory and access to sensitive data.

  Spatial partitioning is feasible by duplicating microarchitectural data structures (e.g., caches, predictors) based on privilege levels. For instance, separating branch predictor states between supervisor and user modes ensures no covert channel exists across this boundary, as they share no state. Intel implemented similar isolation for predictors through mechanisms such as indirect branch restricted speculation (IBRS) (see subsection 8.1.1).

Exemple 6.1 - *The BTB Case*

A typical example illustrating privilege-level partitioning importance is protecting the BTB across privilege boundaries. The BTB is a microarchitectural predictor associating a branch/jump destination with its congruent source address (matching least significant bits). Historically, shared BTB state enabled user-mode attackers to manipulate predictions toward attacker-defined gadgets, causing higher-privileged modes (e.g., supervisor) to execute these gadgets upon privilege-level switches (branch target injection (BTI)).
Modern cores spatially partition the BTB across privilege levels to mitigate BTI. Based on available documentation (though without absolute certainty), this partitioning appears to use tag-based mechanisms. While effective against BTI, tag-based partitioning generally does not prevent covert channels.

- **Address space identifier (ASID)** (**Address Space Identifier**) is a numerical identifier within the `satp` register, uniquely identifying different address spaces. Initially conceived for performance optimisation, it relates closely (though imperfectly) to the notion of a software process. It cannot be assumed that code sharing the same ASID can always trust each other, but different ASIDs typically imply isolation.

  Due to the potentially large number of concurrent processes (thus ASIDs), spatial partitioning becomes impractical. Temporal partitioning is preferred: when the ASID value in `satp` is modified, the microarchitecture triggers a generalised state flush (caches, predictors, etc.), ensuring no residual information leakage across address-space boundaries.
- **Virtual machine identifier (VMID)** (**Virtual Machine Identifier**) is a register field analogous to ASID, distinguishing different virtual machines.

  Similar to ASID, spatial partitioning is generally infeasible due to scalability constraints. Hence, temporal partitioning (generalised flush upon VMID change) is used.
- **Physical memory protection (PMP)** (**Physical Memory Protection**) is a mechanism tied to the machine privilege level restricting read/write access to specific physical memory regions. Although it is a clear hardware boundary, it does not directly isolate microarchitectural state.

However, several challenges emerge when leveraging these existing boundaries:

1. **Originally not security-focused:** ASID and VMID were performance-driven features. Repurposing them for security isolation makes setting these identifiers security-critical, necessitating hardened software implementations.
2. **Limited bit-width:** ASID fields are limited to 9-bit ($\mathrm{SXLEN}=32$) or 16-bit ($\mathrm{SXLEN}=64$), and VMID fields to 7-bit or 14-bit, respectively. Identifier collisions can thus occur. Consequently, flushing decisions should depend on explicit register writes (e.g., to `satp`) rather than solely on identifier values.
3. **Separation of concerns:** ASID and VMID originally identify address spaces or virtual machines. Their incidental use as security boundaries imposes rigid software-level granularity on the security model, restricting flexibility in defining security domains.

To address these challenges, defining a dedicated mechanism explicitly designed to trigger microarchitectural partitioning might ultimately be necessary.

6.2.2 Timing Fences

In [72], the authors propose a new `fence.t` instruction that triggers microarchitectural state flushing. The OS is then responsible for placing these instructions at the appropriate locations, specifically at security domain boundaries, as defined by the OS. They characterise this solution using channel matrices (cf. section 5.1) and demonstrate its effectiveness on a RISC-V CVA6/Ariane core. In this paper, the semantics of `fence.t` are defined as follows: `fence.t` "isolates the timing of any subsequent execution from what happened before."

More concretely, the authors trigger a microarchitectural reset when executing this instruction. In the first, "naïve" version, they performed the following operations:

- The pipeline is flushed.
- The L1 cache, TLB, and BTB are cleared by invalidating their entries.
- BHT saturation counters are reset.

Unfortunately, the authors show that this was insufficient, as they were still able to detect leakage. In a second, "final" version, they additionally performed the following operations:

- The LFSR used for L1 cache replacement is cleared.
- The L1D round-robin arbiter is reset.
- The pseudo-LRU state for TLB replacement is cleared.

With all these modifications, the authors show that covert channels were no longer detected in their tests. The cost of a context switch using `fence.t` was measured at 1 502 clock cycles, compared to 1 180 cycles in the worst case without `fence.t`, representing an overhead of approximately 30%. The most costly operation is flushing the cache, even though the target chip (CVA6) uses a write-through cache. Higher costs can be expected for write-back caches.

This concept is further extended in [73], where the authors specifically study the issue of timer interrupts: the interrupt occurs at a publicly known time during the execution of the targeted process. If this interrupt triggers a process switch where the new destination process is controlled by the attacker, they can measure the timing between the interrupt and the switch. This timing depends on the state of the target process and can be exploited to leak information. To mitigate this, the authors propose applying a padding time scheme, ensuring that process switches occur in constant time from the perspective of the destination process.

6.2.3 Domes

The `fence.t` proposal focuses on temporal partitioning. Since it only marks the boundaries of security domains rather than defining the domains themselves, there is no way to enable spatial partitioning.

In [20], we instead proposed identifying security domains (also referred to as "domes") using a unique identifier. A change in this identifier signals to the hardware that partitioning is necessary. However, the identifier also enables additional hardware optimisations, particularly allowing spatial partitioning. We demonstrate the effectiveness of this approach on our own RISC-V core and introduce a new suite of dedicated benchmarks, timesecbench, which evaluates channel matrices for specific microarchitectural structures (L1D, L1I, BHT, BTB).

The core principle of a correct dome implementation is that microarchitectural resources and states must be statically allocated to a security domain upon its creation and can only be reallocated once the domain terminates. As a result, security domains must be coarse-grained: they should correspond to processes or similar constructs but not to individual routines, as the overhead of switching domains would be too high. We establish design guidelines to mitigate security risks:

1. **Static allocation:** The minimal resources required by a security domain must be allocated at its creation and locked until its deletion.
2. **Release:** When a security domain terminates, all associated resources must be released only after all persistent states have been erased.

3. **Partitioning:** Any resource that serves multiple security domains simultaneously must be able to partition each domain's state into its own isolated compartment. States and data cannot be shared.
4. **Availability split:** A spatially shared resource must ensure that, at any given time, its availability for a security domain is independent of the other domains being served.
5. **Homogeneity:** During execution, all users must be treated equally, with the same types and quantities of allocated resources.

All microarchitectural resources must be explicitly managed as such, with a dedicated lifecycle status finite state machine (FSM), as illustrated in Figure 6.3.

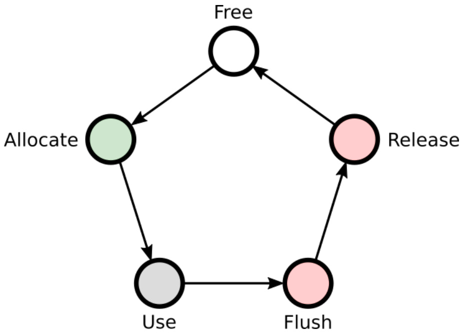
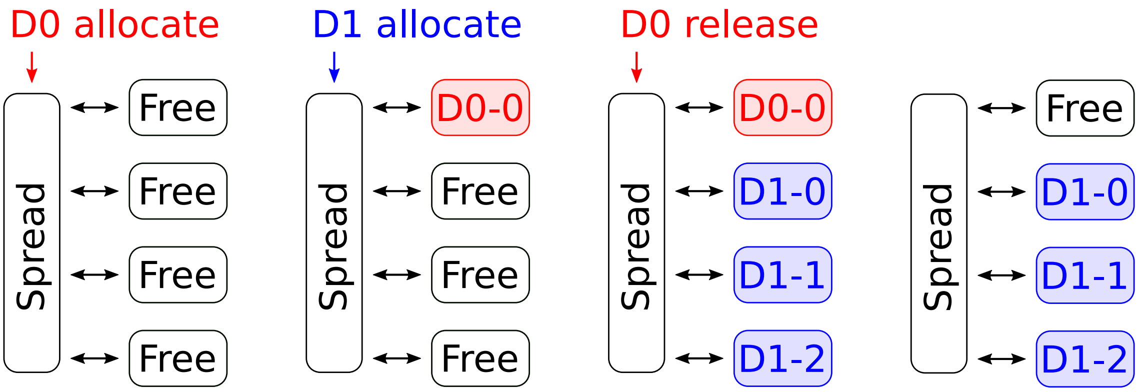


Figure 6.3: The lifecycle of all resources.

A hardware component is responsible for allocating resources based on domain requirements and resource availability. An example is shown in Figure 6.4.



(a) Domain 0 requests minimal resources.

(b) Domain 1 requests maximal resources.

(c) Domain 0 releases all its resources.

(d) Resource allocation for Domain 1 remains unchanged after the initial allocation.

Figure 6.4: Resource allocation process.

Based on these principles, two full RISC-V cores have been designed. The Aubrac core (Figure 6.5) is a classic 5-stage in-order core, enabling comparisons of security and performance with similar designs. The Salers core (Figure 6.6), on the other hand, is more innovative. It supports SMT with two harts in an in-order design. SMT presents the most challenging security concerns in resource sharing, and this core allows us to explore potential solutions.

Finally, we evaluated the security of our cores using a newly developed benchmark suite: timesecbench. This suite assesses channel matrices for common microarchitectural states (L1D, L1I, BHT, BTB) and is extensible. It can generate the corresponding figures, as shown in Figure 6.7, which presents the channel matrix for an unprotected BTB component.

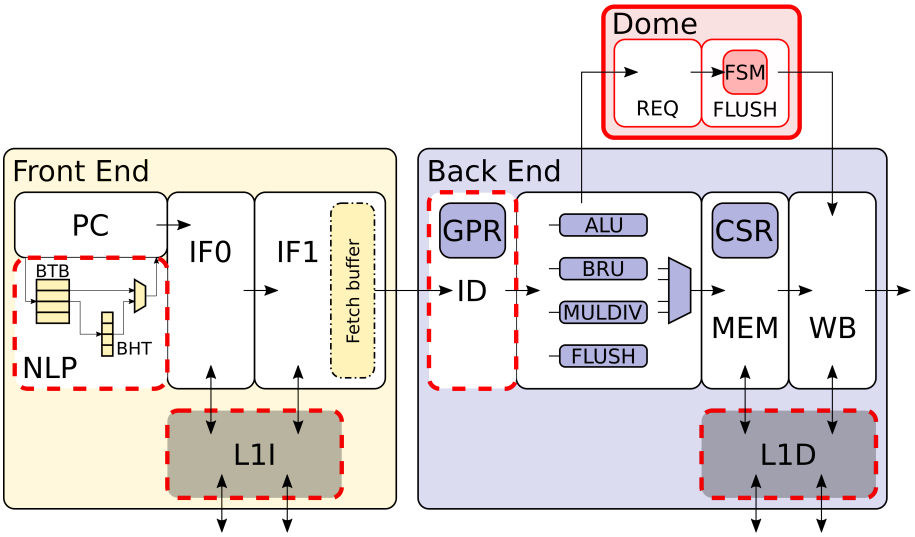


Figure 6.5: The Aubrac core is a classic 5-stage in-order core[20].

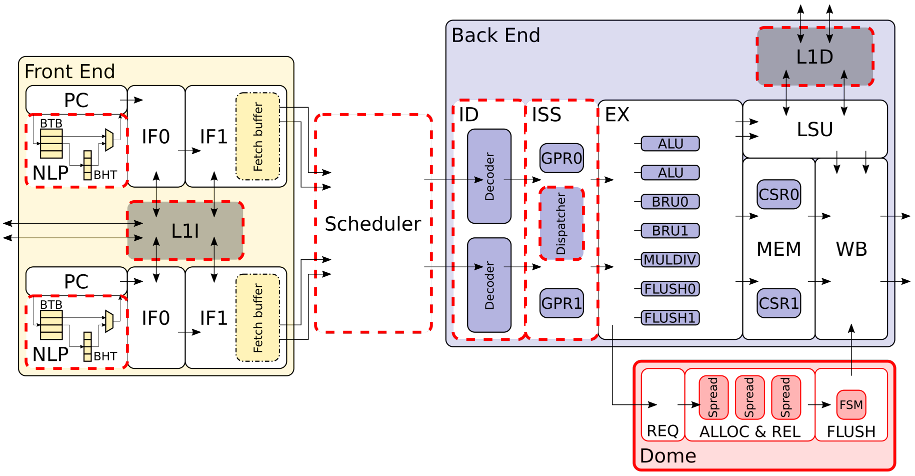


Figure 6.6: The Salers core is an SMT in-order core with two harts, enabling tests on the limits of safe resource sharing[20].

A secure microarchitecture exhibits no horizontal variability in these channel matrices, which can be quantitatively assessed. Using this suite, we demonstrate that both of our cores are secure against leakage from these microarchitectural resources.

The hardware cost of this solution is relatively low, with up to +3% look-up tables (LUTs) utilisation and up to +7% flip-flops (FFs). However, the timing overhead is significant, requiring up to 68 clock cycles for a dome switch. Since flushing can be performed in just a few cycles in our design, the primary source of this overhead is the cold state after each domain switch.

Timing fences and domes provide strong security guarantees but suffer from a substantial performance penalty: microarchitectural state sharing is critical for performance.

6.2.4 `fence.time` as an Official RISC-V Extension in Development

I lead an effort to standardise a solution against timing covert and side channels for the RISC-V Foundation as the chair of the Timing Fences Task Group. There are two existing solutions in the literature, but only one mechanism can be standardised. Which one should be pushed forward? The group's work identified several issues, notably:

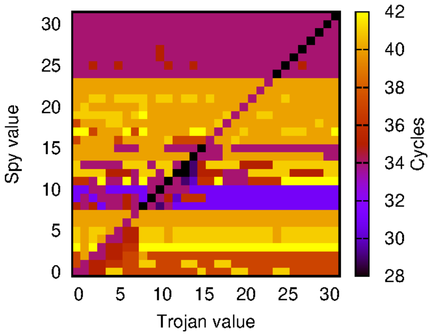


Figure 6.7: The channel matrix of an unprotected BTB.

- `fence.t` clearly lacks a mechanism for spatial partitioning. For example, some cores share predictors across privilege levels (requiring temporal partitioning), while others have dedicated predictors per level (allowing spatial partitioning). The `fence.t` solution would require inserting `fence.t` only for cores that share predictors, but not for others, an approach that lacks portability.
- The domes mechanism is complex to implement in both hardware and software, and adapting it to an existing core is particularly challenging. In our case, we implemented an entire core from scratch.

We concluded with a new solution heavily inspired by `fence.t`, called `fence.time` (definition 6.1).

Definition 6.1 - *Fence.time Semantics*

We define a new `fence.time [flags]` instruction with the following role:

> The timing of any instruction or sequence of instructions executing after the fence must be independent of any microarchitectural state before the fence. The flags may exclude this requirement for specific subsets of microarchitectural state.

In this definition, timing refers to any latency measurable by an attacker: whether it is the actual execution latency of an instruction, time spent in the issue queue, or any other measurable delay.

The defined flags are as follows:

- PRIV_SWITCH: The `fence.time` instruction is associated with a privilege level change.
- AS_SWITCH: The `fence.time` instruction is associated with an address space change.
- VM_SWITCH: The `fence.time` instruction is associated with a virtual machine change.

Any combination of flags is valid.

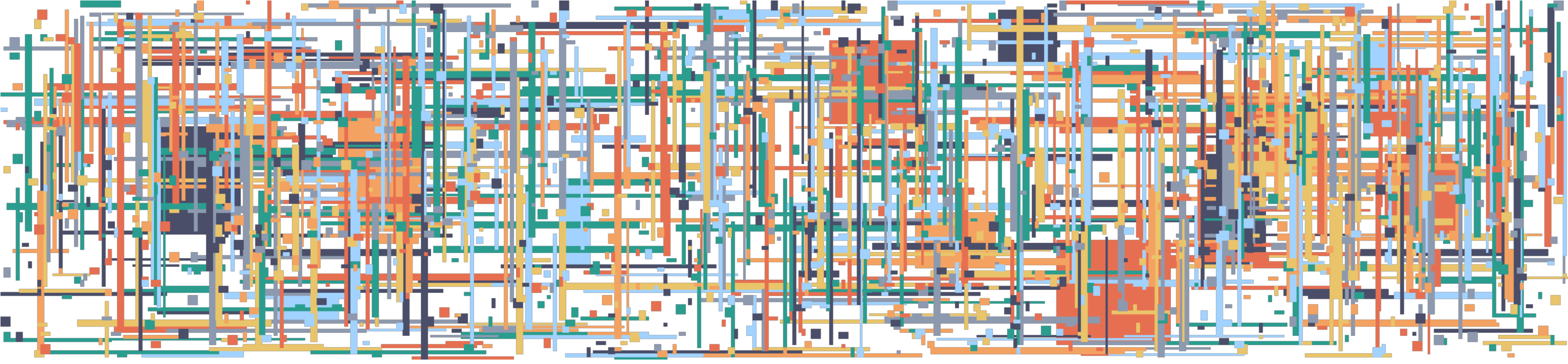
With this definition, we achieve a simple security boundary decision while providing additional information about the nature of the boundary, thereby enabling spatial partitioning. For example, a system call would require a `fence.time PRIV_SWITCH` instruction, which works for both temporal and spatial partitioning, depending on the hardware implementation.

6.2.5 Conclusion on Countermeasures Against Covert Channels

The solutions (`fence.t`, `fence.time`, and domes) presented in this chapter help mitigate covert channels in the microarchitecture. Unfortunately, they still have inherent limitations that must be considered.

1. Flushing the entire microarchitectural state can be prohibitively slow, and is typically only done at a coarse granularity, such as switching to a new process or a new virtual machine. Due to the lack of proof-of-concept implementations, I am unsure about the performance/security trade-offs for finer-grained security domain switches, such as system calls or interrupts.
2. These protections are difficult to implement perfectly in hardware. Any microarchitectural state can potentially support covert channels, including all FSMs. There is an unavoidable trade-off between which states can be reasonably partitioned and which might still leak information. A perfect implementation could only be demonstrated through hardware formal methods, which cannot yet scale to complex cores.
3. Their implementation would be significantly different on large out-of-order cores, where flushing the microarchitectural state could be prohibitively slow.
4. These solutions offer no protection against attackers using physical attacks. They rely on the assumption that the covert channel spy is code running on the same core. However, if an attacker can measure power consumption or infer timing through electromagnetic emissions, these protections become ineffective. A simpler case arises when timing measurements can be performed from a peripheral in the SoC.
5. Only `fence.t` proposes a mechanism to protect against information leakage caused by interrupts.
6. In multicore systems, some microarchitectural states, such as the last level cache (LLC), are typically shared across cores. There is currently no definitive solution for handling this edge case.

Mitigating covert channels must be part of the microarchitecture design process. However, designers must keep in mind that the inherent limitations of current solutions still allow covert channels in specific cases. In particular, covert channels remain possible within the same security domain during speculative execution, as explored in Chapters 7 and 8.



7. The Dangers of Speculation

Covert channels provide a means of transmitting confidential data across security boundaries. However, for an exploit to occur, a secret value must first be accessed. A recent [37] class of attacks, known as **transient attacks**, leverages speculative execution to read secrets: transient attacks combine control flow hijacking during speculative execution to acquire a secret, followed by a disclosure gadget that transmits the secret through a covert channel.

In this chapter, I introduce the fundamental principles behind transient attacks. While some exploits are described, an exhaustive literature review would require extensive technical explanations that are beyond the scope of this document and are not included.

7.1 Microarchitectural Data Sampling

Microarchitectural data sampling (MDS) attacks refer to a set of attacks.

They distinguish themselves from Spectre attacks based on an argument dating back to the early days of transient attacks:

> *“ Unlike other recent attacks such as Spectre, Meltdown, and Foreshadow, which are based on vulnerabilities leaking data from the CPU caches, RIDL and Fallout collect data from internal CPU buffers (Line Fill Buffers, Load Ports, Store Buffers). ”*
>
> **– MDS differentiation from the MDS papers authors**

As discussed in section 7.3, Spectre and Meltdown attacks do not necessarily leak data from caches; they can also exploit internal CPU buffers.

In this chapter, I propose an alternative definition of MDS attacks, focusing on their threat model.

Definition 7.1 - *Microarchitectural Data Sampling Attacks*

MDS attacks exploit the microarchitecture to leak secret data that is legitimately used by an application, using a microarchitectural covert channel. Secrets are moved within the microarchitecture as intended by the application, leaving traces in the microarchitectural state. As a result, an attacker can leak these secrets, including in speculative mode, through an unintended execution path that operates on the secret.

Numerous exploits fall into this category, and I present only the most significant ones here.

RIDL [57] targets line fill buffers (LFBs), data structures that track memory requests leaving the load store unit (LSU), enabling optimisations such as request merging. The authors demonstrate that privileged execution can leak data to a userland application by leaving traces in the LFB.

ZombieLoad [60] is similar to RIDL and also exploits LFBs. The authors show that a misspeculated `load` may introduce data into the LFB, which can subsequently be read, even from a different hart on the same core.

Fallout [14] focuses on *store buffers* within the LSU. In particular, it leverages a faulty store-to-load forwarding mechanism in certain Intel processors to forcibly pass data from a `store` instruction to a subsequent `load`.

7.2 Meltdown and its Variants

Meltdown is an attack first published in 2018 [41]. It exploits poor exception handling in some out-of-order (OoO) execution cores, mostly Intel cores originally.

Although this family of attacks is no longer considered a major threat, since mitigating it is relatively straightforward, a quick review of how it works is presented here.

Listing 7.1 C code for the Meltdown attack.

```c
uint8_t forbidden_secret = *secret_address;
uint8_t trojan = array1[forbidden_secret * 4096];
```

In listing 7.1, the attacker attempts to read a secret by dereferencing a pointer without the necessary privileges (user-level instead of supervisor-level). This instruction triggers a hardware exception.

Since hardware exceptions are expected to be rare, modern processors often optimise for the case where they do not occur. Execution continues normally under this assumption until the speculation is resolved before the instruction commit. If an exception does occur, execution is rolled back, just as in speculative execution.

Thus, in listing 7.1, the second line, which reads data from an address dependent on the secret, is executed before the exception is handled, only to be later discarded. However, the rollback does not clear the caches: the secret remains in the cache line's tag (address). The attacker can then retrieve the secret by measuring access times to determine which array value is in the cache.

The solution is simple: data should not be accessible before the associated exceptions are resolved.

The original Meltdown attack [41] triggers an exception by loading data from a memory page that requires supervisor privileges. In Linux, kernel memory is mapped into the user process's memory space but is protected by page permissions that restrict access to the supervisor level. Thus, any access to protected memory triggers an exception and therein lies the problem.

Other Meltdown variants have been developed. In [13], the authors categorise these variants based on the type of exception used or the specific protection bits in the page table that trigger an exception. Notably, some variants, such as Meltdown Bound-Check Bypass, are effective on AMD processors as well as Intel. This variant exploits the Bound Range Exceeded exception, which can be triggered after the x86 `bound` instruction.

Ultimately, the Meltdown exploit combines a vulnerability related to poor exception handling with a microarchitectural covert channel. While not immediately obvious, careful processor design with robust exception handling can effectively mitigate this exploit.

7.3 A Presentation of Spectre Attacks

Also published in 2018 [37], the Spectre attack exploits the speculative behaviour of the processor core. Unlike Meltdown, which can be seen as an implementation error, Spectre is an attack inherent to the very principle of speculative execution. As a result, it is much more difficult to defend against. Even today, and to be fair, since its publication, it has always been challenging to exploit Spectre attacks in practice. However, the underlying vulnerability is real and remains present, allowing new variations of the attack to emerge regularly.

7.3.1 Basic Principle

A Spectre gadget, the sequence of code/instructions that allow a Spectre attack, is relatively simple from a programming language perspective.

Listing 7.2 C code for the Spectre-PHT attack.

```
1 if (x < array1_size) {
2     y = array2[array1[x] * 4096];
3 }
```

The code in listing 7.2 can be broken down as follows:

- The attacker repeatedly executes this same code, the Spectre gadget, while ensuring that the branch condition (line 1) is met. Alternatively, in some attack scenarios, another process manipulates the branch history so that the branch predictor assumes this condition will always hold.
- A new execution is launched where the branch condition (line 1) is not met (i.e. the index is out of bounds). This introduces the **speculation gadget**.
- Due to branch prediction, the "if" branch is still speculatively executed, even though the index is out of bounds.
- A speculative `LOAD` (line 2) operation is performed at an arbitrary address `array1 + x`, allowing the attacker to read the secret s. This step constitutes the **acquisition gadget**, responsible for acquiring the secret.
- A second speculative `LOAD` (line 2) operation is executed to leak the secret by encoding it into the tag of a cache line. This forms the **disclosure gadget**. The secret is shifted left ($\cdot * 4096$) to ensure that it influences the cache line's tag despite truncation.

The attacker (or "spy") then performs a cache timing attack, such as Flush+Reload, to extract the secret.

7.3.2 Microarchitectural Breakdown

Figure 7.1 illustrates the microarchitectural state when executing a Spectre-PHT gadget on a NaxRiscv core.

This model provides a more detailed view compared to the general description above:

1. Initially, a sequence of `load` (`ld`) and `store` (`sd`) instructions is observed. These serve to increase the speculation window, that is, they delay branch resolution as much as possible.
2. The branch itself is a `beqz` (branch if equal to zero) instruction, which is speculated as NOT TAKEN (N). Incorrectly speculated instructions are shown as hashed-out regions in Figure 7.1.
3. The branch is followed by a critical `load` operation that reads the secret (`lb t2, 0(a0)`). This operation is executed immediately, even though it is never officially committed.
4. The subsequent instructions select a specific bit from the loaded secret, the bit that the attacker intends to extract. This value is then multiplied by an appropriate factor (here,

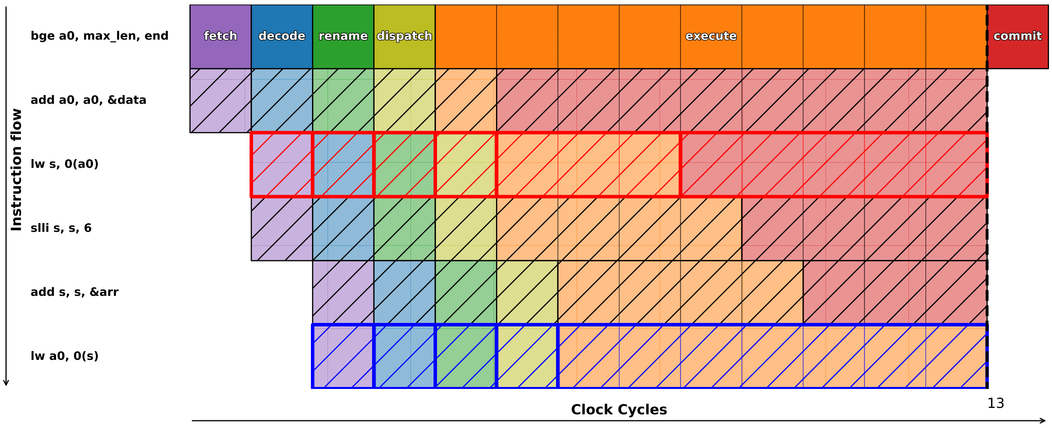


Figure 7.1: The microarchitectural state when executing a Spectre gadget. Colours represent the different pipeline stages. The hashed-out area indicates speculated and uncommitted instructions.

2^6 with `slli t2, t2, 6`) to ensure that it influences the cache line.

5. Finally, a second `load` operation (`ld t2, 0(t2)`) accesses the manipulated cache line, to write the secret in the cache line tag value.
6. The attacker only needs to measure the access time to one of the two targeted cache lines (corresponding to secret bit values 0 or 1) to infer the secret bit.

7.3.3 Exploits

An interesting feature of this attack is that the listing 7.2 gadget can be part of kernel code; the attacker only needs to control two elements:

- The offset x, which, for example, could be an argument of a system call. In this case, a user process can freely choose x.
- The branch history, which determines the predicted execution path. **If the branch predictor is shared between the kernel and user space, the attacker can train it in user mode.** This last point is the main focus of countermeasures introduced in recent processors (see subsection 8.1.1).

This variant is commonly called Spectre-PHT because it relies on corrupting branch direction prediction (influencing the pattern history table (PHT)). This highlights the security risks associated with shared resources, such as the branch history, across different security domains.

However, simply removing the PHT is not sufficient to prevent exploitation; variants exist for all mechanisms that enable speculative execution.

7.3.4 Variants

A Spectre exploit combines a speculation gadget with a covert channel. The speculation gadget is responsible for reading the secret in speculative mode, while the covert channel is used to exfiltrate it. In section 6.1, various types of covert channels that can be leveraged have been reviewed.

The term "variant" typically refers to different implementations of the speculation gadget:

- **Spectre-PHT** [37] exploits branch direction speculation, whether through a pattern history table (PHT) or another mechanism.
- **Spectre-BTB** [37] relies on branch target speculation via a branch target buffer (BTB).
- **Spectre-RSB** [43] exploits branch target speculation using a return stack buffer (RSB).
- **Spectre-STL** [93] abuses alias speculation in the *Load Store Queue*, also known as speculative store-to-load forwarding (STL). It speculates that a `load` following a `store` does

not reference overlapping data (*alias*), thereby assuming no store-to-load forwarding.

- More exotic variants have also been identified [46]:
  - **String Comparison Overrun**, which exploits speculative execution of certain string-handling instructions in x86.
  - **Zero Dividend Injection**, where speculation is applied to the dividend value in a division. If the higher-order bits are unavailable, they are speculated to be zero.
  - **Flop** [35] that exploits load value prediction on recent Apple processors.

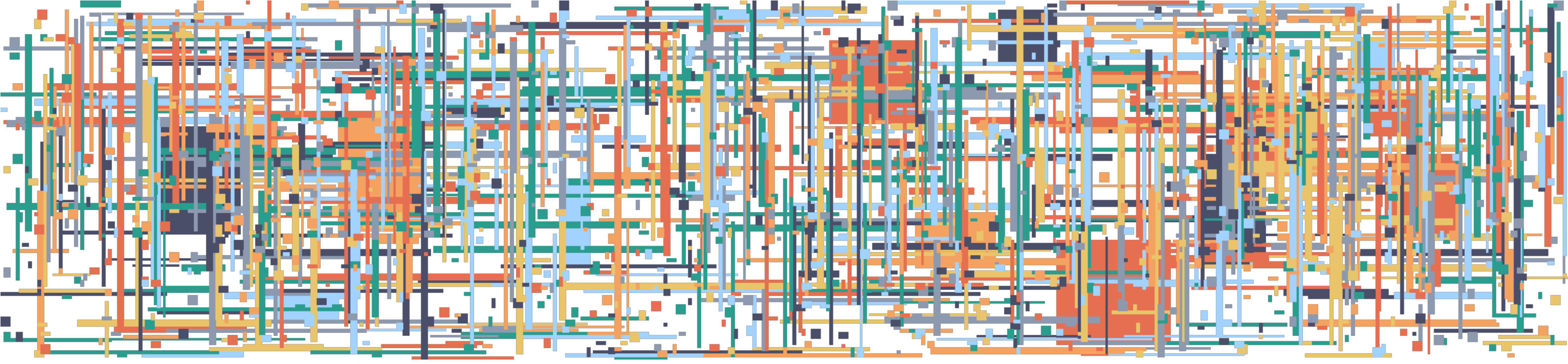
Thus, any speculative operation can serve as the basis for a new Spectre variant. However, attacks involving branch/jump target speculation tend to be particularly dangerous, as they allow the attacker to influence the control flow path.

7.4 Other Transient Attacks

In this document, we classify attacks as Meltdown variants when out-of-order execution enables the execution of acquisition and disclosure gadgets, and as Spectre variants when speculation enables gadget execution. However, there is no universally agreed-upon definition for these attack families. For example, Meltdown attacks can be viewed as a subclass of Spectre attacks, leveraging the speculation that exceptions will not occur.

One thing is certain: even under the broad definitions given to these two categories, some attacks do not fit neatly into either.

- **Load Value Injection (LVI)** [11]. In this work, the attacker can corrupt the result of a `load` instruction executed speculatively by the victim. During the speculation window, the program continues execution using this incorrect value, which can lead to secret disclosure.
- **ExSpectre** [67]. This attack explores the idea of malware that hides within speculative execution windows. The malware payload appears benign under normal execution but becomes malicious only in a speculative context. Each speculative gadget transmits its results to the ”normal” execution flow, which then determines the next speculative gadget to execute, creating a self-sustaining attack chain. A simple example is malware code that is effectively dead code in a conventional binary analysis but remains active in speculative execution.
- **GhostKnight** [78]. This attack leverages speculative execution to amplify the RowHammer attack (which induces memory bit flips in DRAM through repeated accesses). Speculative `load` instructions are transmitted to DRAM and can serve as the basis for RowHammer. Additionally, these speculative `load` instructions often have elevated privileges, allowing them to target addresses that should be inaccessible.
- **BlindSide** [25]. This attack attempts to bypass address space layout randomization (ASLR) using speculative probing. Typically, probing involves accessing a memory address and observing the system’s response—whether the `load` succeeds, fails due to insufficient privileges, or attempts to access an unmapped memory region. Performing this probing speculatively can provide additional information to the attacker.
- **GhostRace** [50]. This attack exploits speculative execution to trigger transient data races. While the operating system’s synchronisation primitives are architecturally correct, data races can still occur in speculative mode, making them exploitable.



8. Dealing with Transient Attacks

8.1 Current Solutions in AMD, ARM, and Intel Microarchitectures

In this section, how major processor manufacturers (Intel, AMD and ARM) have responded to Spectre attacks is examined. These responses were developed reactively, often in response to newly published exploits. This led to a proliferation of countermeasures and technologies, where a more generic and less complex solution would have been preferable. These measures mix countermeasures against covert channels with those focused on speculative execution.

8.1.1 Intel

Intel's countermeasures against Spectre appropriately focus on branch predictors and the vulnerabilities caused by hardware threads. A webpage [94] lists these technologies and their usage.

- (Enhanced) Indirect branch restricted speculation (IBRS): One of the reasons Spectre vulnerabilities are particularly severe on Intel processors is that branch predictor states were historically shared across privilege levels. This allowed a user process to influence the branch prediction of subsequent kernel code. IBRS provides hardware support for isolating the microarchitectural states of these predictors based on privilege level, ensuring that user-mode predictor entries no longer affect kernel-mode predictions. The precise hardware mechanism is unknown, but I conjecture that Intel now tags BTB entries with the privilege level (easy to do and low performance penalty).
- Indirect branch prediction barrier (IBPB): Although predictor states are now separated between privilege levels, they remain shared across user processes. This enables the creation of covert channels and manipulation of predictions across address spaces. To prevent this, a new barrier instruction, IBPB, was introduced. Instructions executed before the barrier cannot influence branch prediction after the barrier. IBPB specifically targets indirect jumps.
- Single thread indirect branch predictors (STIBP): As noted with IBRS, predictor states were shared between privilege levels. They were also shared between hardware threads, i.e., virtual cores mapped to the same physical core during SMT execution (known as Hyper-Threading on Intel processors). This meant that one core could influence the prediction of another core. Following the model of IBRS, STIBP partitions predictor states between virtual cores.
- `LFENCE`: Originally a memory barrier, this instruction ensures that a `load` does not execute until the address of a preceding `store` has been resolved, preventing Spectre-

STL. More generally, `LFENCE` introduces dependencies between instructions, reducing reordering possibilities and limiting the effectiveness of Spectre gadgets. The semantics of this instruction can act as either a speculation barrier or a serialising instruction, depending on the specific core it is executed on. Documentation on newer processors diverges, but one thing is clear: the semantics of `LFENCE` have evolved toward functioning as a speculation barrier in newer processors.

- BHI_DIS_S: This indirect predictor control mechanism protects against branch history injection (BHI) by enforcing partitioning of indirect jumps across different privilege levels.
- Speculative store bypass disable (SSBD): When `LFENCE` is insufficient, SSBD offers an alternative mitigation. It works by disabling `store` bypass at the hardware level, preventing a `load` from executing before preceding `store`s have been resolved.

8.1.2 AMD

AMD's countermeasures, as required by the x86 architecture, follow Intel's approach, implementing IBRS, IBPB, and STIBP under the collective name indirect branch control (IBC). However, the recommendations [85] for applying these countermeasures differ from those of Intel.

8.1.3 Arm

Arm's solution revolves around speculation barriers, introducing the speculation barrier (SB) instruction in the ARMv8 instruction set. The precise semantics of this instruction are complex:

Definition 8.1 - *ARM's Speculation Barrier Semantics*

The semantics of the Speculation Barrier dictate that, until the barrier completes, any instruction appearing later in program order than the barrier:

- Cannot be executed speculatively if such speculation can be observed through side channels due to control flow speculation or data value speculation.
- Can be speculatively executed based on the prediction that a potentially exception-generating instruction has not caused an exception.

In particular, any instruction appearing later in program order than the barrier cannot trigger a speculative allocation in any caching structure where the allocation of that entry could indicate the presence of specific data values in memory or registers.

The `SB` instruction:

- Cannot be speculatively executed due to control flow speculation or data value speculation.
- Can be speculatively executed based on the prediction that a potentially exception-generating instruction has not caused an exception. The potentially exception-generating instruction can complete once it is confirmed to be non-speculative, and all data values generated by instructions appearing earlier in program order than the `SB` instruction have been verified.

When instruction stream prediction is not influenced by register outputs from the speculative execution of instructions appearing after an uncompleted `SB` instruction, the `SB` instruction has no effect on how prediction resources are used to fetch the instruction stream.

Despite its complex definition, the concept of a speculation barrier is simple: it is an instruction that waits for speculative conditions to resolve before completing its execution. As a result, the next instruction is never executed speculatively. Placing it in the appropriate locations, such as before a `load`, helps prevent Spectre gadgets.

8.2 Propositions from the Academic Literature

InvisiSpec[74], DAWG[36]	CondSpec[40], SafeSpec[34], CleanupSpec[54], EffInvisiSpec[55], MI6[9], NDA[71], SpecShield[6], SpectreGuard[24], STT[76]	IRONHIDE[47], MuonTrap[2], Clearing the Shadows[66], ConTExT[59], InvarSpec[80], SDO[75]	DOLMA[42], SPT[16]	Context-Sensitive Fencing[65]	SpecTerminator[32], SafeBet[29]
2018	2019	2020	2021	2022	2023

Table 8.1: Timeline of Spectre countermeasure papers (2018-2023).

The academic community has extensively explored microarchitectural security, particularly transient attacks and their countermeasures. However, research in this area faces a significant challenge: the absence of open-source processors that match the complexity of commercial products (e.g., Intel, AMD, or Arm). Consequently, results from academic studies often require extrapolation to these commercial architectures, introducing uncertainties in their practical applicability.

This section specifically focuses on countermeasures for transient attacks; solutions addressing covert channels have already been discussed in section 6.2. Relevant surveys (**SoK papers**) on transient execution vulnerabilities and countermeasures include [31, 51].

8.2.1 Invisible Speculation: Reverting Mispeculated State

Spectre attacks exploit transient instructions that leave persistent microarchitectural changes. A theoretically ideal defence involves reverting the microarchitectural state completely after mispeculation, restoring caches, branch predictors, and internal FSMs to their original states. However, practically achieving this is challenging.

Several approaches have emerged to implement this idea:

InvisiSpec [74] proposes using a dedicated speculative buffer (an L0 cache) to handle transient execution. If mispeculation occurs, this buffer is flushed. If speculation is correct, loads must be reissued to maintain coherence, introducing performance penalties. However, as indicated in section 6.1, focusing exclusively on cache-based covert channels is insufficient, given alternative leakage vectors.

CleanupSpec [54] extends InvisiSpec, addressing the overhead and coherence issues by applying state-restoration mechanisms across multiple cache levels (L1, L2, and LLC). Techniques such as address randomisation (L2 cache), eviction restoration (L1 cache), and random replacement policies are introduced.

MuonTrap [2] similarly suggests a dedicated L0 cache specifically for speculative instructions.

DAWG [36] proposes spatial partitioning of caches and predictors according to security domains. While this significantly reduces vulnerability to Spectre attacks, it is not entirely foolproof, as discussed in subsection 6.2.5.

SafeSpec [34] employs shadow structures (speculative buffers similar to InvisiSpec) and extends the concept beyond caches to include TLBs.

8.2.2 Selective Speculation: Delaying Risky Speculation

Rather than reverting mispeculated states, another approach is to delay or selectively permit speculative execution when risks are detected. This strategy, called **selective speculation**, identifies risky situations to suspend or limit speculative behaviour dynamically. Several independent proposals explore this approach, including NDA [71], STT [76], and SpecShield [6], all aiming to identify Spectre gadgets to safely manage speculation.

Selective speculation approaches commonly distinguish three critical events during speculative execution:

1. **Speculation**: Instructions triggering speculative execution.
2. **Acquisition**: Instructions potentially accessing secret data (typically speculative `load` operations).
3. **Disclosure**: Instructions potentially leaking secret data through covert channels.

These approaches often rely on hardware-based taint tracking between acquisition and disclosure events, intervening by preventing speculation, disallowing speculative loads, or blocking covert channels. Earlier intervention improves security but increases performance overhead.

Several notable selective speculation techniques include:

Efficient InvisiSpec [55] reduces InvisiSpec's performance overhead by proposing alternatives such as delaying only speculative loads that miss L1 caches or employing value prediction techniques. It finds that approximately 87% of loads are speculative, with L1 miss rates between 1% and 25%.

Clearing the Shadows [66] further refines this approach by attempting to determine speculative conditions earlier to manage speculation more efficiently.

SpectreGuard [24] proposes annotating memory regions as confidential or non-confidential, thus selectively applying defences only to sensitive data.

CondSpec [40] identifies "security-dependent" memory instructions, marking them as safe or unsafe based on cache behaviour. However, this method exclusively targets cache-based covert channels, missing other leakage vectors.

ConTExT [59] introduces explicit annotations for sensitive memory data and registers (as non-transient). Transient execution replaces these sensitive values with dummy data, requiring source code annotations. Notably, a similar memory-annotation strategy has been patented by NVIDIA [86].

InvarSpec [80] performs compile-time analysis to identify instructions whose execution and operands remain unaffected by speculative conditions, deeming them safe for speculation.

SDO [75] relaxes conditions slightly, allowing speculative execution of instructions whose operands do not influence hardware resource usage.

DOLMA [42] defines *transient non-observability*, allowing transient instructions provided their operands do not influence the timing of committed non-transient instructions. This strategy is a relaxed variation of *speculative non-interference*.

Speculative Privacy Tracking (SPT) [16] generalises earlier delay-on-miss strategies, allowing speculative execution if operands have already been leaked non-speculatively.

While most selective speculation methods focus on hardware mechanisms, software control through speculation barriers is also viable. Our own implementations of speculation barriers are discussed in section 8.4, although earlier solutions exist.

Context-Sensitive Fencing [65] automatically inserts speculation barriers into the micro-op pipeline according to predefined security policies (e.g., barriers placed between control-flow instructions and subsequent loads).

SpecTerminator [32] combines multiple techniques, delaying unsafe speculative execution using hardware-based tainting. Although reporting low overhead (6%), these results have not been independently reproduced [51].

8.2.3 Protecting Secure Enclaves

Some research works focus on protecting secure enclaves, which are distinct security domains with stringent security requirements.

MI6 [9], followed by IRONHIDE [47], explores the design of secure enclaves that are immune to microarchitectural attacks. These works particularly focus on the efficient implementation of microarchitectural cleanup operations during transitions between secure and non-secure worlds. IRONHIDE further addresses the specific challenges associated with multicore designs.

SafeBet [29] does not strictly aim to isolate an enclave, but rather introduces isolation around “trust domains” with dedicated hardware support.

8.2.4 The Use of Formal Methods

Given the complexity of modern microarchitectures, how can it be guaranteed that secrets are never read speculatively or leaked through covert channels?

Some works address this challenge by establishing hardware-software contracts for speculative behaviour [30]. This paper formally models the interaction between information leakage and speculative execution. Notably, it demonstrates that simply delaying speculative `load` instructions is insufficient; instead, speculative behaviour must be controlled comprehensively. The study also proves the correctness of speculative taint tracking solutions for ensuring core security.

ProSpeCT [18] implements this speculative behaviour contract in hardware. The key idea is to ensure that no secret value can be leaked speculatively because it is exclusively used by code that runs in constant time. This implementation is realised on the OoO processor Proteus. The performance overhead depends on the frequency of execution involving secret values and the program’s degree of instruction-level parallelism (ILP), ranging from 0% to 45% depending on the scenario.

A related approach [23] further emphasises the hardware-software contract by formalising microarchitectural behaviour to provide the software with predictable execution properties. In this work, speculation is restricted within a specific security domain, whose scope depends on the implementation.

Formal methods targeting hardware design show great promise. However, they remain difficult to scale to the complexity of a realistic OoO processor.

8.3 Compiler-Based Solutions

8.3.1 Retpoline

Retpoline [103] is a software-based countermeasure that transforms indirect jumps into a gadget that captures speculative execution behaviour.

Listing 8.1 Assembly code for the Retpoline countermeasure, a gadget that replaces an indirect jump.

```
                       jal set_up_target; # ra <- (capture_spec)
                       capture_spec:
                           pause;
jr a0;          →          j capture_spec;
                       set_up_target:
                           mv ra, a0; # ra <- a0
                           ret; # jr ra
```

In listing 8.1, the indirect jump is replaced with a gadget that traps speculative execution in the `capture_spec:` loop using the RSB predictor.

Listing 8.2 Pseudocode for Address-Based SLH.

```
beq a, b, ...
xor diff, a, b // diff = 0 if beq is taken
sltiu c, diff, 1 // c = 1 if diff = 0, 0 otherwise
sub mask, zero, c     // mask = -c

and mask_add, add, mask
load secret, 0(mask_add)
```

The gadget functions as follows:

1. The jump destination is assumed to be stored in the `a0` register.
2. The first instruction, `jal`, jumps to `set_up_target:` and stores the address of `capture_spec:` in the `ra` register (this is the *and link* operation). Consequently, the RSB pushes the address of `capture_spec:` onto the stack.
3. The value of `a0` is moved into `ra`.
4. The `ret` instruction is an alias for `jalr x0, ra`, causing execution to jump to the address stored in `ra` (which now holds `a0`). However, the RSB predictor speculates that this instruction will jump to `capture_spec:`.
5. As a result, speculative execution is "trapped" in `capture_spec:` until speculation resolves and reveals the correct jump to `a0`.

This technique specifically mitigates Spectre-RSB but comes at a significant performance cost. A more security-effective alternative would be to eliminate the RSB entirely, but keeping the performance overhead.

8.3.2 Speculative Load Hardening (SLH)

SLH is a technique implemented in LLVM [87] for x86, designed to prevent the leakage of secrets through speculative `load` instructions associated with branch direction speculation. In other words, SLH is primarily effective against Spectre-BHT.

The key idea is to establish a dependency between the predicate of a branch condition and either the address `rs1` or the result `rd` of a `load rd, 0(rs1)`. Let c be the boolean predicate associated with one or more branching instructions that determine whether a `load` is executed. Since branches can be nested, this predicate must be carried forward and updated for each branching instruction. Thus, the value of c indicates whether execution is speculative or not.

For Spectre-PHT, however, c does not solely determine whether the `load` is executed transiently; an instruction may still be speculatively executed even if $c = false$. Nonetheless, it is possible to create a mask m dependent on the branch predicate:

$$m = \begin{cases} b11...111 & \text{if } c = true \\ b00...000 & \text{if } c = false \end{cases}$$

From this, two countermeasures can be derived:

1. **Address Hardening**: Applying the mask to the `load` address, as shown in listing 8.2, also known as LLVM-aSLH.
2. **Value Hardening**: Applying the mask to the result of the `load`, also known as LLVM-vSLH. In the event of misprediction, the secret value becomes unusable, as demonstrated in listing 8.3.

A variation of this mitigation, SLH-IP, extends the approach inter-procedurally, ensuring that the predicate is maintained across function calls by storing it in the stack. This variation

Listing 8.3 Variant for Value Hardening.

```
//as before...
load secret, 0(add)
and masked_secret, secret, mask
```

improves precision, providing a better security/performance trade-off.

SLH is a crucial countermeasure against Spectre-PHT, but its effectiveness is limited. On one hand, it does not mitigate other Spectre variants. On the other hand, its performance overhead appears prohibitive, with a reported best-case slowdown of 29% [87]. On a RISC-V core, we achieved a ≈ 45% slowdown with our reproduced implementation.

To formalise the approach further, Patrignani et al. [48] introduced strong speculative load hardening (SSLH), which aims to protect all `load` instructions, whereas LLVM-SLH does not safeguard `load` instructions with addresses known at compile time.

This technique has recently been analysed and refined in [79], leading to the development of Ultimate SLH. Ultimate SLH enhances protection by addressing variable-timing instructions, which could be exploited as covert channels.

These studies have identified vulnerabilities in LLVM's SLH implementation while proving the correctness of their own implementations.

A first RISC-V implementation of SLH was proposed by Moein Ghaniyoun [92] and later improved by Thomas Rubiano [97] as part of our work on speculation barriers.

However, these approaches face two critical limitations:

1. **Register Allocation Issues**: The register allocation phase is not adequately considered. Predicate management may introduce additional `load` instructions due to register spilling, a behaviour we have observed in practice that introduces new unprotected `load`.
2. **Microarchitectural Control Flow Complexity**: Proofs concerning the architectural control flow do not directly translate into guarantees for microarchitectural control flow.

Our experiments confirm these limitations. For example, Figure 8.2 shows that the SLH countermeasure still permits numerous Spectre-PHT gadgets, even though the mitigation is explicitly designed to prevent them. In this case, most of the residual gadgets result from register spilling.

8.3.3 Bounds Clipping

An efficient technique to prevent a `load` instruction from reading arbitrary data is to enforce the bounds of the corresponding address. This technique is known as **bounds clipping**.

If the bounds of the address values accessed by a particular `load` are known, it is possible to add bound clipping instructions (saturating min and max) that ensure the address value remains within the bounds. Even if the value is speculated incorrectly, the bounds ensure that no confidential data can be read.

Bounds clipping, by not interfering with speculation, is expected to be performance-efficient. However, as far as I know, there is no dedicated study on the associated performance and security merits. In particular, what proportion of `load` instructions can be efficiently bounded?

8.4 Speculation Barriers for RISC-V

"`fence.spec`: exploring speculation barriers for RISC-V selective speculation" Herinomena Andrianatrehina, Ronan Lashermes, Joseph Paturel, Simon Rokicki, and Thomas Rubiano. **under submission** [5]

This section presents results from the PhD Thesis of Herinomena Andrianatrehina that I advise.

When reviewing the literature, one can only notice discrepancies in the reported figures of merit (also discussed in [51]).

These inconsistencies can be attributed to several factors:

1. The significant impact of test environments on the results of certain mitigation measures (e.g., the benchmarks used). Solutions do not offer protection against the same Spectre variants.
2. The difficulty of openly experimenting on speculative microarchitectures, as researchers must choose between using simplified gem5 models of the most complex x86 cores or precise models of simpler RISC-V cores.
3. Differing threat models and initial assumptions make it difficult to compare solutions that protect against different kinds of threats and assume varying levels of knowledge about which data requires protection.

Thus, although a variety of mitigation approaches have been proposed to address Spectre, the lack of reproducibility has made it challenging to accurately assess and compare the effectiveness of each solution.

To compare solutions, we designed and implemented speculation barriers for RISC-V. They offer a flexible way to compare different solutions by modifying the barrier placement policy. We created an environment for exploring and comparing different implementations of the selective speculation approach in a realistic testing environment: we vary the semantics of the barrier instructions, their placement policies, and their hardware implementations.

To compare the various countermeasures, we introduce a quantitative security metric. A post-analysis module gathers execution traces and detects any **Spectre gadgets**, which refer to sequences of instructions and microarchitectural states that *could* be used to mount Spectre attacks. The number of gadgets found serves as a measure of the difficulty for an attacker to mount a Spectre attack but does not prove that it is impossible to do so.

!

The microarchitectural control flow is arbitrary, so any speculation barrier can be bypassed in speculative mode. **From the start, we know that speculation barriers cannot be a perfect solution!** However, their flexibility makes them well-suited for quickly exploring many combinations of semantics, placement, and hardware implementations. Any promising results could then be directly applied to the microarchitecture, eliminating the need for speculation barrier instructions.

8.4.1 Fences Semantics

The goal of a speculation barrier is to mark a point in the execution flow where architectural and microarchitectural states converge (both register values and PC). To achieve this, we need to resolve the speculative nature of the instructions currently being executed.

The semantics of the x86 speculation barrier are given in definition 8.2.

Definition 8.2 - *x86 `LFENCE` semantics (Felix Cloutier webpage)*

`LFENCE` does not execute until all prior instructions have completed locally, and no later instruction begins execution until `LFENCE` completes. *Personal note*: I believe “prior” and “later” can be interpreted as referring to program order.

We found the `LFENCE` semantics to be too restrictive, as it stalls all instructions, even in cases where it is unnecessary. What if two independent execution paths could proceed, but only one needed to restrict speculation?

To explore whether we could extract more performance, we defined a speculation barrier called `fence.spec` that has explicit dependencies on registers.

Definition 8.3 - *Our own RISC-V `fence.spec rd, rs1` speculation barrier*

We define a new `fence.spec rd, rs1` instruction with the following properties:

1. The barrier cannot terminate execution unless it is certain that it will eventually commit. It cannot continue while speculative.
2. The barrier depends on register `rs1`, as if reading the value. If `rs1` = `x0`, then it depends on all registers.
3. The barrier "touches" register `rd`, as if writing to it, but without modifying the value. Therefore, later instructions that depend on this register also depend on the barrier. If `rd` = `x0`, then it touches all registers.

These semantics explicitly handle dependencies without relying on program order.

We also define a serialisation barrier, `fence.ser rd, rs1`, similar to `fence.spec`, but only with rules 2 and 3. It can execute speculatively.

These semantics are explored in two forms: targeted, using `fence.spec xR, xR` (where `xR` is any register different from `x0`), which targets a relevant register depending on the policy; or global, using `fence.spec.all` as the pseudo-instruction for `fence.spec x0, x0`.

8.4.2 Placement Policies

Defining speculation barriers is the easy part; the challenge lies in determining where and when to apply them. In particular, if a secret exists in the address space, any `load` instruction can access it speculatively.

Therefore, mitigations must be applied broadly to the entire address space containing a secret, **independently of whether the actual program may or may not access this secret**.

Speculation barriers cannot be evaluated independently from their placement policy, which defines a security policy.

8.4.2.1 In the Linux Kernel

The Linux kernel documentation [102] explains the mitigations used in the kernel. The actual mitigations enabled depend on the hardware, as the kernel attempts to rely on hardware countermeasures whenever possible and falls back to compiler-based solutions for older hardware.

The kernel prioritises IBRS, IBPB, and STIBP when available, falling back to Retpoline and placing `LFENCE` instructions if these are unavailable.

Additionally, the following mechanisms may be used:

- **Return stack buffer (RSB) Flushing**: To protect against RSB underflow attacks, the kernel flushes the RSB on context switches and VM exits.
- **Branch history buffer (BHB) Clearing**: To mitigate BHI attacks, a sequence is implemented to clear the branch history buffer (BHB).
- **nospec Accessor Macros**: These macros prevent speculative execution from accessing data pointers influenced by user inputs by enforcing bounds clipping and other security validations during speculative execution. The use of nospec macros ensures that data crossing privilege level boundaries, particularly arguments to system calls (syscalls), are sanitised.

Notably, the use of SLH is not mentioned.

The mitigations for Arm cores are not discussed in this document, but it can be inferred that they are similar, replacing `LFENCE` with `SB`, for example.

These mitigations may effectively prevent the most dangerous exploits that can influence kernel microarchitectural control flow from user space (also known as Branch History Injection). However, these mitigations are not principled; they address specific exploits rather than the underlying vulnerabilities.

8.4.2.2 In our benchmarks

In our experiments, we evaluated three policies:

1. `before_load`: A speculation barrier is placed before all `load` instructions, targeting the address register. This policy prevents all speculative `load` operations.
2. `after_load`: A barrier is placed after all `load` instructions, targeting the loaded value. This policy prevents the use of all speculatively loaded values.
3. `dependency`: A barrier is placed after `branch` instructions, targeting the condition of the branch. The compiler is responsible for optimising unnecessary barriers (e.g., duplicated conditions).

A special policy called `nop` involves placing a `nop` instruction before all `load` instructions. This allows us to evaluate the cost of executing any instruction and compare it with the cost of running speculation barriers.

8.4.3 Hardware implementations

Our implementations are variations of the NaxRiscv OoO core, written in SpinalHDL, a variant of CHISEL.

8.4.3.1 Adding dependencies

The core manages dependencies in the Issue Queue, which contains the instructions waiting for execution. A data structure records, for each instruction, the indices of all other instructions it depends on.

When an instruction has no remaining active dependencies, it becomes eligible for execution, which in turn unblocks other dependent instructions.

Our fences, which have dependencies on `rd` and `rs1` different from `x0`, do not require adaptation to fit into this scheme. However, an additional `fence_all` flag is assigned to all instructions in the Issue Queue, with the flag set to 1 if and only if `rd` $=$ `x0`.

This mechanism applies to both `fence.spec` and `fence.ser`.

8.4.3.2 Stalling while in speculative mode

There are several possibilities for implementing the barriers in hardware to stall execution when in speculative mode. We investigate three implementations.

1. **Execution-stall**: A `fence.spec` instruction can only finish execution when it is the next instruction to be committed.
2. **Dispatch-stall**: A `fence.spec` instruction can only be dispatched to an execution unit when it is sure to commit. This means either no instructions are left to be executed, or they are themselves sure to commit.
3. **Operands-stall**: The `fence.spec rd, rs1` instruction can be dispatched as soon as the instruction producing `rs1` has committed. Note that this behaviour does not strictly follow our semantics, since `fence.spec` may be speculatively executed in this case.

The idea is to compare whether stalling earlier offers different security/performance trade-offs.

8.4.4 Our results

To measure performance, we execute the Embench benchmarking suite for all configurations. The benchmarks are calibrated to execute in approximately $5M$ instructions.

We define a configuration as a specific `[semantics]_[policy]-[hardware]` combination. For example, the `nop_before_load-execute` configuration involves executing our benchmarks by placing a `nop` before all `load` instructions with the `execution stall` hardware implementation.

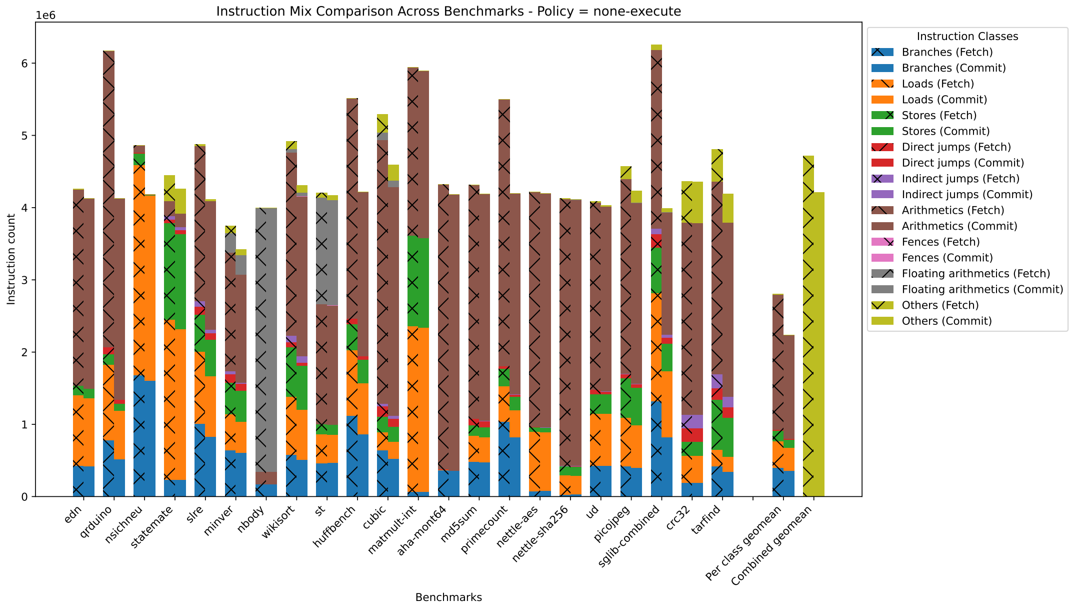


Figure 8.1: Instruction mixes for all benchmarks for the `none-execute` configuration.

We execute our benchmarks using the Verilator simulator. Our simulator generates execution traces in a custom format, which allows for a detailed analysis of execution. For example, Figure 8.1 shows the instruction mix for each benchmark, for both fetched and committed instructions.

8.4.4.1 How to measure security

A major challenge in mitigating transient attacks lies in evaluating the security of the proposed solution. Most works in the literature present a set of attacks and demonstrate that these are effectively thwarted by the solution. However, what happens if a new attack is discovered afterward? What if the specific implementation details of the tested attack mean that the mitigation does not work on other hardware?

In our work, we chose to focus on vulnerabilities rather than exploits. Exploits are implementation-dependent, whereas a vulnerability, i.e., the potential for an exploit, is more general.

More specifically, we identify a **Spectre gadget** as a sequence of three events:

1. A misspeculation window initiated by a **speculation-triggering instruction**,
2. a secret **acquisition** instruction (a `load`) within the window,
3. the **disclosure** of this secret within the same window. Disclosure occurs in our case by taint-tracking the potential secret to one of the following:
   - a `load` using the secret as an address,
   - a `store` using the secret as either an address or a value,
   - a `branch` using the secret as a condition,
   - a `jalr` using the secret as a jump destination.

An example of such a Spectre gadget is shown in Figure 7.1.

Our security measure is therefore ad hoc: we count the number of occurrences of Spectre gadgets in the execution traces of the benchmarks. If no gadgets are found in the traces for a specific *configuration*, we conclude that it will be difficult for an attacker to achieve a Spectre attack. However, this does not guarantee impossibility, as it cannot be formally demonstrated that gadgets are absent.

Since access to the execution traces is available, the gadgets found can be analysed. For instance, in Figure 8.2, we present the number of gadgets for all configurations, categorised by the cause of the misspeculation window.

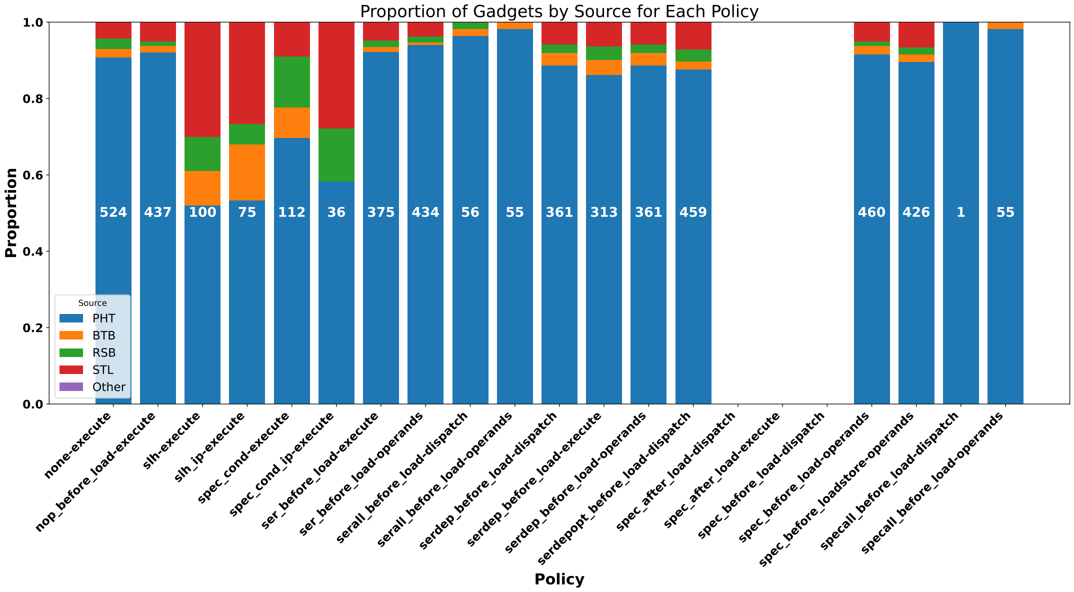


Figure 8.2: The number and proportions of Spectre gadgets categorised by their misspeculation sources per configuration.

Our results indicate that the majority ($\approx 85\%$) of gadgets are caused by a mispredicted branch direction in our benchmarks.

SLH is an interesting case , where most of the PHT variants are removed but not all. Since SLH specifically targets branch direction predictions, and some implementations have been proven correct [48, 79], we would have expected a perfect solution with no PHT gadgets found.

However, our SLH implementation has not been proven correct, even if it was inspired by [79]. We discovered that the predicate is often stored in memory, in the stack, because of the inter-procedural implementation or simply because of register spilling during register allocation. Indeed, it seems that the proofs proposed in the literature being performed on a static single assignment (SSA) representation of the program, the security guarantees cannot be transferred to the final binary after register allocation.

It can be seen in Figure 8.2 that some policies are indeed secure. However, security should be evaluated in light of the corresponding performance drop.

8.4.4.2 Security versus Performance Trade-off

We plot the security (ordinate) and performance (abscissa) results for all configurations in Figure 8.3. The raw results are indicated in Table 8.2, along with a description of the placement policies.

From Figure 8.3, we observe a Pareto front emerging. Unfortunately, the performance of the best secure configuration `spec_after_load-execute`, with 0 gadgets found, is equivalent

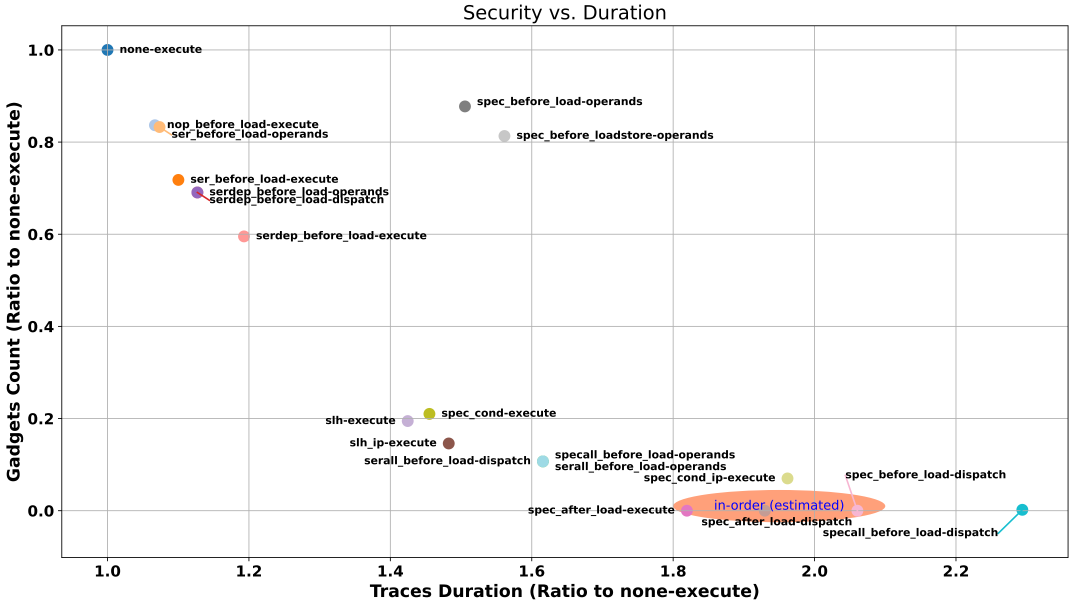


Figure 8.3: Plotting the security/performance trade-off. Security is the ratio to the `none-execute` gadget count, performance is the ratio to the `none-execute` geomean trace duration. The corresponding raw data is shown in Table 8.2.

to the performance of an in-order core, according to [44], with a $\approx 80\%$ performance overhead.

It appears that the ability to speculatively execute `load` instructions justifies most of the out-of-order advantage in our case. However, even allowing speculative `load` with the `spec_after_load` policy, blocking the speculative usage of the value read, is not sufficient.

Table 8.2: List of evaluated policies and results

		Gadget counts			Benchmark duration geomean (hot cycles)		
Policy name	**Description**	`execute`	`dispatch`	`operands`	`execute`	`dispatch`	`operands`
`none`	No policy applied (baseline).	514			3.6 M		
`nop_before_load`	Insert `nop` instruction **before** each `load`.	430			3.8 M		
`ser_before_load`	Insert `fence.ser rA, rA` instruction **before** each `load rB, offset(rA)`.	369	428	428	3.9 M	3.8 M	3.8 M
`serall_before_load`	Insert `fence.ser x0, x0` **before** each `load`.	16	55	55	7.0 M	5.8 M	5.8 M
`serdep_before_load`	Insert `fence.ser rB, rA` with `rB` used as address by the following `load` and `rA` register from the "most dominant branching instruction operands". It is a naive approach to add serialization between `load`s and most dominant branching instruction if there is one.	306	355	355	4.3 M	4.0 M	4.0 M
`slh`	SLH implementation for RISCV.	100			5.1 M		
`slh_ip`	SLH implementation with inter-procedural predicate transfer via stack pointer.	75			5.3 M		
`spec_after_load`	Insert `fence.spec rB, rB` instruction **after** each `load rB, offset(rA)`.	0	0	0	6.5 M	6.9 M	6.6 M
`spec_before_load`	Insert `fence.spec rA, rA` instruction **before** each `load rB, offset(rA)`.	0	0	451	7.4 M	7.4 M	5.4 M
`spec_before_loadstore`	Insert `fence.spec rA, rA` instruction **before** each `load rB, offset(rA)` or `store rB, offset(rA)`.	3	3	418	8.1 M	8.0 M	5.6 M
`spec_cond`	Insert `fence.cond rA` at beginning of each basic block that contains `load`. It takes as operand `rA`, an always updated predicate computed as in `slh` policy.	108			5.2 M		
`spec_cond_ip`	As `spec_cond` with inter-procedural predicate transfer.	36			7.0 M		
`specall_before_load`	Insert `fence.spec x0, x0` **before** each `load`.	0	1	55	8.4 M	8.2 M	5.8 M

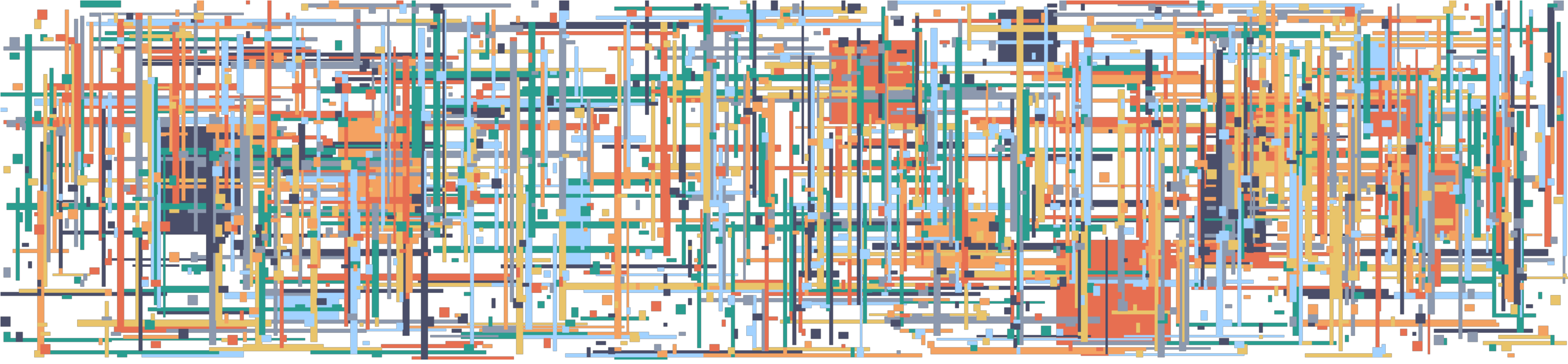
Gadget counts in Table 8.2 are the same ones as used to generate Figure 8.3 but different than in Figure 8.2. Indeed, in Figure 8.2 two gadgets occuring at the same acquistion address are counted twice if they have differente speculation sources, but are counted once in Table 8.2 and Figure 8.3. Performance is measured as the geometric mean of the number of hot cycles (after the warmup) for the benchmark suite.

8.4.5 Concluding on Speculation Barriers and Spectre Countermeasures

Our results are mostly negative: we were unable to prevent all Spectre gadgets without an unacceptable performance overhead. The fact that a Pareto front emerges, despite varying our countermeasures in widely different directions (semantics, placement, and hardware implementation), seems to indicate a hard limit of speculation barriers.

However, our results only apply to our implementations, variants of the same NaxRiscv core. It is possible that we would get different results with a different core. I conjecture that a wider core, able to issue 4 instructions or more per cycle (instead of 2 in our case) would offer better results with our speculation barriers by providing more reordering opportunities.

I do not believe that choosing between security and the performance of an OoO execution is an acceptable choice. Therefore, we may need to revisit our assumptions. In particular, we assume that the microarchitecture has no way to determine if data is confidential and thus must not be executed speculatively. This reflects the capabilities of today's microarchitectures. However, it seems that the way forward is to change this assumption and support confidential data in the microarchitecture, with a modification of the ISA. But this approach would be radical, as it would make developers responsible for the annotation of confidential data, which is not an easy task. This direction of reasoning is explored in chapter 10.



9. Concluding on Microarchitectural Attacks

Covert channels and transient attacks have been described in chapters 5 and 7 respectively, and corresponding countermeasures have been proposed in chapters 6 and 8.

Covert channels are inherent to the design of microarchitectures: any microarchitectural state can potentially support a covert channel. Mitigating covert channels is achieved both at the microarchitectural level, through state splitting, and at the architectural level, with dedicated instructions hinting at security domain switches.

The cost of covert channel mitigation makes it suitable only for coarse-grained security domains. In particular, speculative execution can still pose a threat through transient attacks. Our exploration of mitigation strategies against transient attacks has been fruitless. Unfortunately, as of today, solutions that can be implemented without causing an unacceptable performance drop remain far from perfect.

My recommendations for designers

To design more secure cores today:

- Design a microarchitecture that limits the possibilities for covert channels: partition state where relevant. In particular, you **must** split state per privilege level for branch predictors to prevent BTI and BHI. Additionally, optimise for a deep and fast flush: for example, write-through caches should be quicker to flush than write-back caches.
- Speculation barriers seem to be a dead end. Instead, focus on hardening your program with bounds clipping. Compilers should be improved to insert these automatically in most cases.

These recommendations will not make your core perfectly secure, but they will make it significantly harder for attackers to develop actual exploits. The real performance overhead on realistic cores is largely unknown since the protection on commercial cores is minimal. In the long term, designers should consider more radical approaches, such as those discussed in Part III.

III

Radical New Core Designs for Security

In this part, I explore how to design secure cores without being constrained by existing microarchitectural designs. Ideas that are not feasible for today's cores are freely explored to assess their potential for enhancing security.

This exploratory part discusses unproven ideas, with designs that may not yet have reached the proof-of-concept stage. As a consequence, most of these propositions have not attained the scientific maturity required for publication and should be considered with the corresponding precautions. I take this opportunity to present a personal vision of future-proof designs, contributing to a scientific debate that I believe is necessary.

Chapter 10 is the continuation of Part II: if we are not limited by following an existing ISA, what could we do to improve the security of our microarchitecture, particularly concerning covert channels, transient attacks, and other security threats? For that purpose, we propose modifying the ISA to explicitly introduce the notion of confidential data. //

Chapters 11, 12 and 13 focus on the security of microcontrollers with the following characteristics:

- Compared to Part II, they must withstand a stronger threat model that includes physical attacks.
- They lack an MMU, reducing the role and protection offered by OSs through the MMU. However, such chips typically include an memory protection unit (MPU).
- These devices typically operate with a fixed software image: no additional software is installed after deployment, except through a *firmware update*. I consider that over-the-air (OTA) updates are not possible. This is an editorial simplification to avoid delving into the complexities of secure boot and OTA firmware update implementations, which are beyond the scope of this discussion.
- The threat model primarily focuses on physical attacks and fault injection in particular, as described in subsection 3.2.2.

In accordance with Kerckhoffs' principle [81], which states that a secure system must be designed under the assumption that the adversary knows its inner workings, I aim to ensure the following.

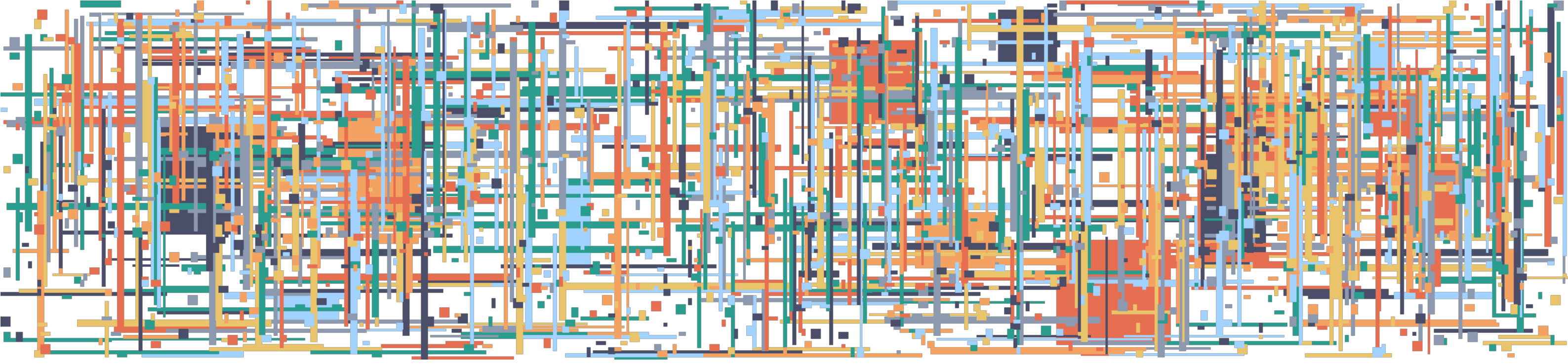
- The **integrity** of the program and its execution: the adversary must not be able to tamper with the program while it is running. This implies that we should also strive to ensure the
- **availability** of the system whenever possible. However, neither the confidentiality of the core nor the program should be a requirement.
- The **integrity** and **confidentiality** of data. In certain scenarios, data confidentiality takes precedence over system availability. For example, a banking smartcard should erase its cryptographic keys upon detecting an attack.

Our core ISA will be based on the RV32I RISC-V specification but will diverge from it on key aspects.

In detail, chapter 11 discusses the possibility of restricting the ISA expressiveness to improve security by forbidding forward indirect jumps.

Chapter 12 presents instruction-set randomization (ISR) schemes and explains how they can enhance integrity guarantees compared to lockstep processors.

Finally, chapter 13 develops these ideas to envision what a microarchitecture built upon the concepts from the previous chapters would look like.



10. Architectural Secret Values

This section presents ideas from the PhD Thesis of Herinomena Andrianatrehina that I advise.

Confidentiality is the property of keeping data accessible only to legitimate entities. Symmetric encryption provides a simple (though somewhat circular) answer to the question of legitimacy: legitimate entities are those who legitimately know the secret key. However, in the microarchitecture, distinguishing between legitimate and illegitimate entities is not straightforward. Another way to frame the issue is: against whom are we trying to protect our data?

There is no concept of secrecy within the RISC-V microarchitecture. In state-of-the-art designs, confidential data such as cryptographic keys should be handled by a dedicated coprocessor.

1. To isolate the keys from the rest of the system. If the key leaves the coprocessor, it must be encrypted to prevent any information leak.
2. To use dedicated and optimised circuits for key operations, preventing leakage (e.g., through timing or power consumption analysis).

This chapter explores how to design a chip that can natively handle confidential data without requiring a coprocessor.

To achieve this, the concept of **architectural secret values** must be introduced.

Architectural secret values are data that must remain confidential within a security domain, here understood as a combination of address space, privilege level, and virtual machine. In particular, these values must not leak to another application executing on the same chip or to an attacker capable of measuring execution timings via observation attacks (timing, electromagnetic emissions, power consumption, etc.).

Architectural secret values are mapped to hardware registers that can now be either confidential or public.

The semantics of architectural secret values is explored in section 10.1, whereas how the confidentiality is propagated to memory is discussed in section 10.2.

10.1 The Semantics of Architectural Secret Values

10.1.1 Hardware Confidential Registers

The concept of hardware confidential registers is central to the proposed scheme for managing architectural secret values. These registers are designed to inherently manage 'static' secrets, since the confidentiality status is encoded directly in the instruction in the register index. By designating certain registers as confidential, the hardware can enforce strict rules on how these secrets are accessed and manipulated, thereby mitigating the risk of unauthorised access or leakage.

Definition 10.1 - *Confidential Registers*

The 32 integer general purpose registers (GPRs) can now be either `public` or `confidential`. We define a new CSR called `confidential.regs`, where the lowest 32 bits designate the confidentiality status of the corresponding register. For example, the i-th bit is set to 1 if register `xi` is `confidential` and 0 if it is `public`.

Hardware behaviour must adapt to the confidentiality status of a register, considering both cache-based and cacheless architectures. Confidential registers must not be used in the following cases:

- As an operand of a `branch`.
- As an operand of an indirect jump (`jalr`).
- As an operand of a variable-time instruction (this depends on the hardware implementation).
- As the address of a `load`.
- As the address of a `store`.
- `load` instructions that write to a confidential register must be non-speculative (i.e., guaranteed to commit).

Failure to comply with these conditions must trigger a hardware security exception. Compilers should generate only valid programs that adhere to these new semantics.

Additional possibilities exist (some discussed in section 10.2):

- If the value being `store`d is confidential, it must be encrypted (requiring a key management mechanism).
- If a `load` destination register is confidential, the data must be decrypted accordingly.
- Writing to a confidential register may require an extra clock cycle, during which a random value is first written into the register before the actual value is stored. This prevents the mask overwriting issue that could leak secrets in masked schemes.
- Dedicated execution units could handle confidential register values, implementing masking techniques to reduce physical leakage.

In this scheme, the hardware is not responsible for tracking the confidentiality property dynamically. It is solely the compiler's responsibility to assign confidential registers for confidential data.

This approach allows static analysis to easily track declassification, defined as any instruction that has at least one confidential operand but a non-confidential destination register. The compiler may ask for explicit declassification in the source code, to be able to prove that no accidental declassification is possible.

Writing to the `confidential.regs` register should be restricted to machine or supervisor mode and remain coarse-grained. For example, if each function begins with a preamble writing to `confidential.regs`, an attacker could potentially manipulate microarchitectural control flow to untaint a register. One possible solution is to make `confidential.regs` read-only, with confidential registers fixed and determined by hardware.

10.1.2 Imagining the Workflow

I assume that source code is annotated with confidentiality semantics. For example, a `confidential` keyword could be added during variable declaration: `confidential int key`. The `confidential` keyword serves as syntactic sugar for the corresponding variable attribute.

In any application, it is the developer's responsibility to annotate variables and memory, determining which values are secret and which are not.

With this information, the compiler ensures that cryptographic keys and certain intermediate values are only assigned to confidential registers. Register spilling must be handled separately for confidential and non-confidential registers, with additional constraints for confidential registers, as outlined in section 10.2.

10.2 Dynamic Tracking of Architectural Secrets in Memory

10.2.1 Issues with Current Mechanisms

Registers alone cannot be the sole target of the confidentiality attribute. Sometimes, secrets must be stored in random-access memory (RAM):

- For secrets in a program that persist longer than a few instructions.
- For register spilling when handling a large amount of confidential data.
- For sharing secrets between programs.

In the RISC-V ecosystem, there are already simple modifications that allow declaring a memory range as confidential.

1. The MMU can be used to add a new `confidential` property to a memory page. This solution has been explored in [59], for example.
2. The PMP can also be modified to add the `confidential` tag to a physical memory range.

The issue with both of these solutions is that the `confidential` property is not inherently tied to the secret data but rather to a subjective view of memory. Nothing prevents the OS from creating a new page mapping to secret data without the `confidential` property. In other words, data confidentiality depends on the memory mapping configuration, an access control mechanism, rather than on the data itself. PMP offers a slight improvement since it is tied to physical memory. However, since the PMP configuration is unique per core, another core in a multicore system could access the same data without confidentiality restrictions.

In my opinion, a new approach is therefore required. In [53], the authors identify this same issue and propose a memory tracking table (MTT), a structure tied to physical memory that tags memory regions as `confidential`. The MMU must then check the MTT to propagate memory attributes. In this model, only a hart in confidential mode can access `confidential` data.

While this is an improvement, it introduces significant hardware complexity. Moreover, `confidential` remains an access control property rather than an inherent data property.

Another issue with architectural secret values arises from transient attacks. If a `load` instruction speculatively addresses a secret value in memory with a public destination register, the secret must not be revealed. This scenario may violate our new ISA, but since the microarchitectural control flow is arbitrary, it can produce such cases.

10.2.2 Inline Memory Encryption

Instead of relying on access control, confidentiality can be enforced through encryption. The ideal approach for this is inline memory encryption: a `store` instruction with a confidential data register as the store value should automatically encrypt the data. Similarly, a `load` instruction that writes to a confidential register should decrypt the corresponding data.

Inline memory encryption is not trivial. Few encryption modes allow direct access to and decryption of arbitrary locations in ciphertext [96]. The NIST recommends the XTS-AES scheme for this use case [90], as illustrated in Figure 10.1. In some ways, XTS is similar to ECB mode in that it allows independent encryption of data blocks, but without the same security vulnerabilities.

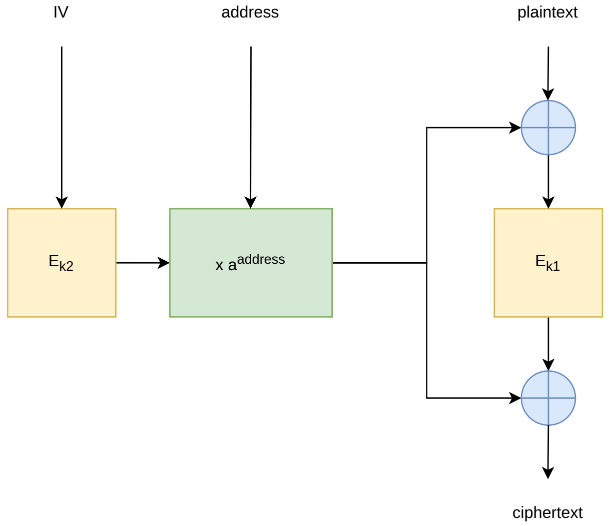


Figure 10.1: The XTS-AES scheme:. The complete key is split into two 128-bit parts, $k1$ and $k2$, used by the two AES ciphers E_{k1} and E_{k2}. A "location-dependent" intermediate value is computed from a random initialisation vector IV, the address of the data to encrypt, and a constant a in $GF(2^{128})$.

XTS mode is not perfect. In particular, encrypting the same 128-bit block at the same address twice results in identical ciphertext.

Implementing inline memory encryption requires integration into the memory hierarchy, presenting several challenges:

- Since only 128-bit blocks can be encrypted at a time, a `sb` (store byte) instruction cannot be independently encrypted. The corresponding block must first be decrypted, modified, and re-encrypted. Alternatively, more complex strategies may be employed. For example, a different encryption scheme could be used in the L1D cache, while inline memory encryption is applied only when data exits or enters the cache.
- What is the scope of the XTS key? It should not be shared across cores or across processes. However, this necessitates a key management scheme to load and unload keys at the appropriate times.

Inline memory encryption is particularly useful for handling a speculative `load` without confidential registers and could speculatively address confidential data. In this case, since the destination register is not confidential, the data would not be decrypted. Using the speculatively loaded value would only leak the encrypted value, which does not reveal the secret itself.

10.3 Limitation: Against Hardware Secrets

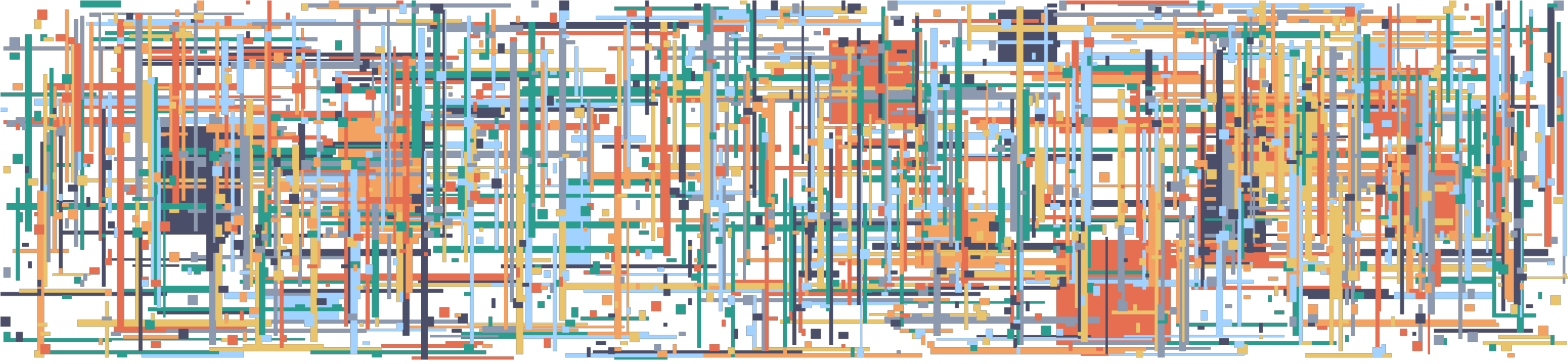
It is unrealistic to expect that software execution of ciphers would be preferable in most cases, since it gives less opportunity for the hardware to provide performance optimisations and security hardening. For such use cases, a dedicated hardened coprocessor remains the more secure and practical solution. However, many confidential data types (e.g., passwords, sensitive personal data) are currently processed without sufficient hardware protection, which must be improved.

The strongest argument against architectural secrets is not technical but economic. For this approach to be effective, developers must be willing and able to annotate confidential variables in their applications. Can they be trusted to do so when failing to annotate a secret yields better performance, and the resulting security vulnerability may remain undiscovered for years? I believe that the complexity of annotation would lead to the compartmentalisation of secret-handling libraries, vetted by security professionals. Just as today's mantra is "never write your own cryptographic library," in the future, it may become "never write your own secret-handling library." Ultimately, I would consider this a positive step for security.

The `confidential.regs` register provides an easy transition to cores supporting architectural secrets. Writing 0 to this register effectively disables architectural secrets, ensuring backward compatibility with traditional cores.

10.4 Conclusion

Architectural secrets would mitigate architectural and microarchitectural timing channels and prevent transient attacks such as Spectre from speculatively loading confidential data. They allow efficient countermeasures but require modifications to both hardware and compilers for proper support. By shifting the responsibility of annotating secrets to developers, architectural secrets would reshape the software ecosystem. Fortunately, a writable `confidential.regs` register ensures backward compatibility.



11. Forbidding Forward Indirect Jumps

To enhance security in microarchitecture design, this chapter delves into the concept of forbidding forward indirect jumps. Indirect jumps, while versatile, introduce significant security risks due to the possibility of unpredictable control flow. By replacing forward indirect jumps with statically knowable dispatch mechanisms, I aim to mitigate these risks.

This chapter explores the rationale behind this approach, examines the use cases of indirect jumps, and proposes alternative mechanisms to achieve secure and efficient control flow management.

> **!** In this chapter, "jumps" are considered unconditional unless stated otherwise for readability. Conditional jumps are referred to as "branches." The RISC-V terminology is followed in this regard.

Jumps consist of only two types of instructions.

- **Direct jumps** use the `jal rd, offset` instruction. The control flow *jumps* to a new instruction at the address given by the PC plus the offset, which is encoded in the instruction itself. The `rd` register is set to the address of the instruction following the jump.
- **Indirect jumps** use the `jalr rd, rs1` instruction. The control flow *jumps* to a new instruction at the address given by the PC plus the value stored in the `rs1` register. The `rd` register is set to the address of the instruction following the jump.

Branches, which execute only when a condition is met, are conditional direct jumps because the offset is encoded in the instruction. For example, `beq rs1, rs2, offset` jumps to the instruction designated by the offset if the values in `rs1` and `rs2` are **eq**ual; otherwise, execution continues with the next instruction. The RISC-V specification supports the following branch conditions: `beq` (**eq**ual), `bne` (**n**ot **e**qual), `bge` (**g**reater or **e**qual), `blt` (strictly **l**ess **t**han), and the unsigned variants `bgeu` and `bltu`.
The RISC-V specification does not support conditional indirect jumps.

In this chapter, I argue that to enhance security, forward indirect jumps should be replaced with dispatch mechanisms. The following sections provide a detailed justification for this approach.

Listing 11.1 Indirect jumps for calling and returning from a procedure.

```
call t0 // Call a procedure at the address stored in t0
// Pseudo-instruction for
// jalr ra, t0

ret // Return from a procedure
// Pseudo-instruction for
// jalr x0, ra
```

11.1 Forward and Backward Indirect Jumps

To justify eliminating forward indirect jumps, I must first define them and examine their use cases.

11.1.1 Use Cases of Indirect Jumps

Indirect jumps serve different purposes.

Calls and Returns Their primary use is for calling and returning from procedures. A procedure is typically a small sequence of instructions that may be reused in multiple contexts. This use case is so common that the RISC-V ISA provides pseudo-instructions to simplify it.

Jumps that move forward in the CFG are referred to as forward jumps (or forward-edge jumps) and those that move backward as backward jumps (or backward-edge jumps). In other words, backward jumps return to a previous point in the execution sequence.

In practice, `jalr x0, ra` and `jalr x0, t0` should be considered backward jumps according to the RISC-V specification [104, section 2.5, which discusses when to update the RSB]. However, the notion of forward and backward jumps is not formally defined in the ISA. The CFG only defines a partial order, meaning that forward and backward directions are not always strictly determined. Additionally, it is possible (but unusual) to use `x2` instead of `x1` (= `ra`) as the return address register, in which case the same edge in the CFG would not be classified as a backward jump according to the RISC-V specification.

Dynamic Dispatch Another common use case is efficiently dispatching control flow to a list of different destinations. A typical example is the use of virtual tables (vtables) in C++ as illustrated in listing 11.2.

The method `bPtr->show()` calls the method of the `Derived` instance because it overrides `Base::show()`. Under the hood, each class has a vtable, a table containing method addresses corresponding to `show()` and `tell()` in each case.

11.2 The Case for Forbidding Forward Indirect Jumps

“A case against indirect jumps for secure programs” Alexandre Gonzalvez and Ronan Lashermes. **Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering 2019** [26]

“Recommendations for a radically secure ISA” Mathieu Escouteloup, Ronan Lashermes, Jean-Louis Lanet, and Jacques Fournier. **CARRV 2020-Workshop on Computer Architecture Research with RISC-V** [21]

Listing 11.2 Illustration of vtable usage in C++.

```
#include <iostream>
using namespace std;

class Base {
public:
    virtual void show() { cout << "Base show" << endl; }
    virtual void tell() { cout << "Base tell" << endl; }
};

class Derived : public Base { // Derived inherits from Base
public:
    void show() override { cout << "Derived show" << endl; }
    // 'tell' is not overridden and will use Base's implementation
};

int main() {
    Base *bPtr;
    Derived dObj;

    bPtr = &dObj;
    bPtr->show();  // Outputs: Derived show
    bPtr->tell();  // Outputs: Base tell

    return 0;
}
```

11.2.1 Indirect Jumps to Statically Unknown Destinations

Any `call rs1` instruction (an alias for `jalr ra, 0(rs1)`) can be replaced with a direct jump `jal ra, address` if the destination is known at compile time (statically). Indirect jumps are only necessary when this is not the case.

Definition 11.1 - *Opaque Predicate*

An opaque predicate is a function that returns True or False. Only the function's author knows the inputs that yield True. Knowledge of the function's software implementation does not help others determine a True-yielding input, and the function must be evaluated at runtime to determine the True/False output for any given input.
Intuitively, the function author knows the solution in advance, but it cannot be evaluated by the compiler.

Exemple 11.1 - *Unresolvable Jump*

Opaque predicates can be used to make the destination of an indirect jump unresolvable at compile time.
Let h be a cryptographically secure hash function (e.g., sha-256). Let pa be the public jump destination address, known to all. The adversary selects a random key k large enough to prevent brute-force attacks and computes $\texttt{hk} = h(k)$. They also choose a

secret jump destination address sa and conceal it by defining $s = h(k+1) \oplus sa \oplus pa$. Now consider the following program:

```
int predicate(bigint x) {
    return (x == hk)? 1 : 0;
}

bigint source = /* user input */;
bigint delta = predicate(source) * (h(source + 1) ^ s);
// opaque jump
goto pa ^ delta;
```

For all values of source not equal to k, the opaque jump goes to the public address pa. However, if source $= k$, then

$$\delta = 1 \cdot (h(k+1) \oplus s) = h(k+1) \oplus h(k+1) \oplus sa \oplus pa = sa \oplus pa.$$

Thus, the opaque jump redirects to the secret address.
Crucially, examining the program alone does not reveal k or the secret destination address.

Example 11.1 demonstrates that an ISA allowing forward indirect jumps is not equivalent to one that does not.

In practice, disallowing programs that contain unresolvable jumps seems beneficial for security. Moreover, all jumps are always resolvable if the destination is known at compile time and opaque predicates are not used.

There are, however, legitimate use cases for unresolvable jumps. For example, an OS launching a new application must jump to its first instruction, which is dynamically placed in memory. Similarly, executing a function from a plugin added to a base application requires an unresolvable jump.

These use cases are generally irrelevant when considering typical microcontroller usage. In this context, the firmware is compiled as a whole and then stored in the chip's memory. Adding a new application requires a firmware update, the alternative is usually a security nightmare (e.g., the bootloader must ensure the authenticity of the loaded image). This is also a key reason for the absence of an MMU in microcontrollers.

In conclusion, forward indirect jumps are unnecessary for most microcontroller use cases. On the contrary, they introduce security concerns. Forbidding them would enable more precise static analysis of the firmware.

11.2.2 Dispatch Gadgets Are Inefficient

Another common use of forward indirect jumps is dynamic dispatch. Since all possible dispatch destinations are known at compile time, they can be replaced with direct jumps.

Indirect jumps can theoretically be replaced by direct jumps, as shown in listing 11.3, but this approach is inefficient. A more optimised gadget than the one in listing 11.3 can be constructed. Using a tree structure, the best gadgets require $\mathcal{O}(n)$ space (for n destinations) and $\mathcal{O}(\log n)$ instructions to execute, compared to $\mathcal{O}(1)$ for both space and execution time with indirect jumps.

Listing 11.3 An inefficient dispatch gadget that avoids indirect jumps.

```
jalr x0, a0
// could be replaced with

// destination [d0]
d0:
    li t0, 0
    bneq t0, a0, d1:
    jal x0, d0
// destination [d1]
d1:
    li t0, 1
    bneq t0, a0, d2:
    jal x0, d1
d2:
    //...
```

11.2.3 A Dedicated Dispatch Instruction

Since all possible jump destinations must be encoded in an instruction, the best achievable space complexity is $\mathcal{O}(n)$ for n destinations. However, the number of executed instructions can be reduced to $\mathcal{O}(1)$ by introducing a dedicated `dispatch rs1, bound` instruction.

Definition 11.2 - *New Dispatch Instruction*

We define the `dispatch rs1, bound` instruction, where `bound` is an immediate value. The `dispatch` instruction is followed by `bound`$+1$ words, each representing a jump destination address. The `rs1` register selects the destination address, using zero-based indexing. For example, if `rs1` equals 1, execution jumps to the second address following the dispatch instruction. The value [`rs1`] must satisfy $0 \leq$ [`rs1`] $<$ `bound`; otherwise, the dispatch raises an exception.

```
dispatch a0, 5
.word address0
.word address1
.word address2
.word address3
.word address4

// rest of code
```

Each `.word` is SXLEN bits in size (typically 32 or 64 bits, cf subsection 2.1.2). The jump addresses must be treated as instructions rather than data, meaning this memory must be non-writable.

11.2.4 Considerations on the Hardware Implementation of Dispatch

The proposed `dispatch` instruction is less efficient than an indirect jump and more challenging to implement in hardware. As far as I know, efficiently implementing such a dispatch instruction in hardware remains an open problem. If a BTB is used, the destination address must be associated with the `dispatch` instruction itself rather than the corresponding following address.

Theoretically, it is possible (though not recommended) to have many destination addresses, which could fill up the instruction cache, fetch buffer, and other resources.

A simpler approach would be to use direct jump instructions `jal` instead of `.word` addresses following the dispatch. However, this would eliminate the ability to perform long jumps (where destinations are far from the jump instruction) since `jal` has only a 21-bit offset.

The main hardware challenge is that the `dispatch` instruction effectively becomes variable-sized: the interpretation of `.word address1` depends on the preceding `dispatch` instruction.

11.2.5 Backward Indirect Jumps Are Necessary for Efficient Designs

According to [26], removing backward jumps results in significant performance and memory overhead. Procedures typically have numerous call sites, leading to large return dispatches. Therefore, the goal is to allow backward jumps while disallowing forward jumps to obtain a good performance/security trade-off.

11.3 Stronger Security Guarantees for Backward Indirect Jumps

Since backward indirect jumps are essential for performance, their restricted semantics (*return to a previous location in the execution flow*) must be strictly enforced, even against a strong attacker. The goal is to guarantee the integrity of the return stack.

I propose a solution that leverages a dedicated hardware register, `rsdepth`, which tracks the return stack depth as an unsigned value initialised to 0 at reset. When executing a forward direct jump `jal rd, offset`, the return address is not written directly to `rd`. Instead, a *token* is generated, containing the return address, the stack depth, and an integrity tag. Let k_1 and k_2 be cryptographic keys initialised randomly at boot and Enc_{k_2} a symmetric encryption algorithm using key k_2. The token is computed as

$$\text{token} = MAC_{k_1}(\text{return address}||rsdepth)||Enc_{k_2}(\text{return address}), \tag{11.1}$$

and the return stack counter `rsdepth` is incremented.

To fit within a machine word (32 or 64 bits), I assume that the most significant bits of pointers can be repurposed to store the integrity tag. However, due to this constraint, the tag size would be relatively small (≈ 8 bits), which presents a security challenge. To prevent attackers from forging new return addresses by modifying tokens, the return address is encrypted using a symmetric block cipher Enc with a device-specific key. Thus, even if an attacker alters a token while maintaining a valid tag by chance, they cannot control the destination of the indirect jump.

A token is the only valid argument for an indirect jump. Therefore, the `jalr rd, rs1` indirect jumps are replaced with a more restricted `jbck rs1` ("jump back") instruction. The `rs1` argument must contain a token. Upon executing this instruction:

- the microarchitecture decrements the internal `rsdepth` register,
- the return address is obtained by decrypting the token payload,
- the tag is recomputed and verified by comparing it with the token's higher bits.

If the tag is incorrect, the system triggers an alarm and responds accordingly.

This scheme is not yet finalised, and the cryptographic protocol must be further refined based on what can be efficiently implemented in the microarchitecture.

Interrupts in microcontrollers can also be managed using this scheme, as an interrupt would trigger the creation of the corresponding token.

11.4 Forward Indirect Jumps as a Security Boundary

In subsection 11.2.1, certain patterns (e.g., an OS launching a new process or an application executing a plugin) have been shown to require forward indirect jumps. These use cases inherently represent a transition to a different security domain, as they involve jumping to code unknown to the previous domain.

Therefore, forward indirect jumps must be restricted, and the security domain switch must be enforced, for example, by ensuring the isolation of the microarchitectural state (cf. chapter 6). As a result, such forward jumps are inherently costly, resembling privilege-level switch mechanisms more than conventional indirect jumps.

11.5 Compiler Support

Modifying the ISA is one challenge, but this proposal also requires adapting compilers to eliminate all forward indirect jumps, which remain essential for certain patterns (cf. listing 11.2). Furthermore, since an ISA that supports forward indirect jumps is not equivalent to one that does not, some programs (specifically, less secure ones) can no longer be compiled. Building such a compiler or more realistically, modifying an existing one to achieve this goal, remains an open problem.

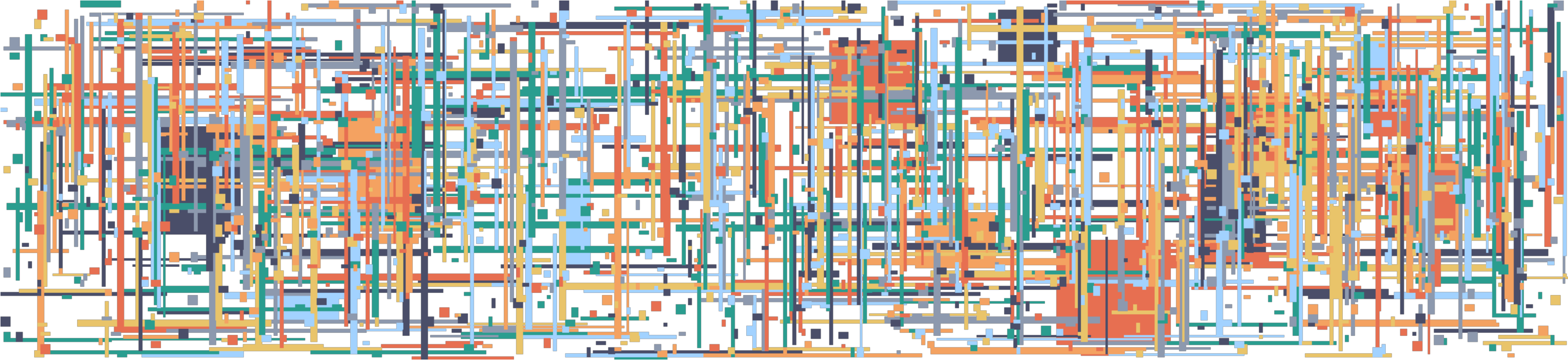
A key advantage of our proposal is that the CFG can now be computed efficiently (both quickly and precisely) by the compiler itself, enabling new static analysis passes that enforce stronger guarantees about the code being written.

The schemes presented in this chapter will be the subject of a future PhD thesis, expected to begin in Fall 2025.

11.6 Other Implications

The approach proposed in this chapter 11 is radical, as it involves modifying the ISA, which in turn affects both the microarchitecture and the compiler. Existing programs, when recompiled with a newly adapted compiler, may no longer function correctly, requiring modifications to ensure compatibility.

Interestingly, an inherent property of our proposal is that the programs rendered invalid are precisely the less secure ones. However, this does not mean that such programs do not exist or are not widely deployed. For instance, consider computer music software, which heavily relies on plugins to simulate various musical effects, instruments and more, where performance is critical. As a result, prohibiting indirect jumps would pose significant challenges for computer music applications in the current landscape.



12. Instruction Set Randomization for Execution Integrity

If an attacker gains physical access to the targeted chip and injects faults such as glitches, electromagnetic fault injection, or laser fault injection, they may compromise the integrity of the executing program. Integrity, in this context, can refer to any of the five types of integrity defined in section 3.4: data integrity, instruction integrity, control-flow integrity, architectural state integrity, and microarchitectural state integrity.

To mitigate this threat, effective countermeasures must be proposed.

12.1 Lockstep Processors

12.1.1 General Working Principle

The solution commonly adopted by industrial designers is **lockstep processors**.

In a lockstep processor, as illustrated in Figure 12.1, the core is duplicated (or triplicated, etc.), and both cores receive the same inputs. Their outputs are then compared. If a discrepancy is detected, an alarm is triggered. In the case of triplication, a voting mechanism can be used to determine the correct output.

An attacker able to inject the same fault into both cores could potentially bypass this protection. To make such an attack more difficult, the two cores are typically desynchronized, with one executing a few cycles ahead of the other.

Lockstep processors are primarily used in safety-critical applications [63], such as avionics, to mitigate the effects of radiation-induced faults, which are more prevalent at high altitudes. They are also common in security-oriented smartcards, which typically embed small microcontrollers with cryptographic accelerators. Defence-related devices may also use these processors for the benefits of both safety and security.

12.1.2 Security

Lockstep processors ensure all five types of integrity defined in section 3.4, through duplication.

In particular, control-flow integrity (CFI) is guaranteed by the redundant execution of control-flow instructions, such as branches and jumps. Data integrity is preserved by duplicating register files; however, it is not guaranteed if the memory itself is not duplicated.

12.1.3 Economics

An important consideration is the relative cost of core duplication compared to memory costs, with both being proportional to the required silicon area.

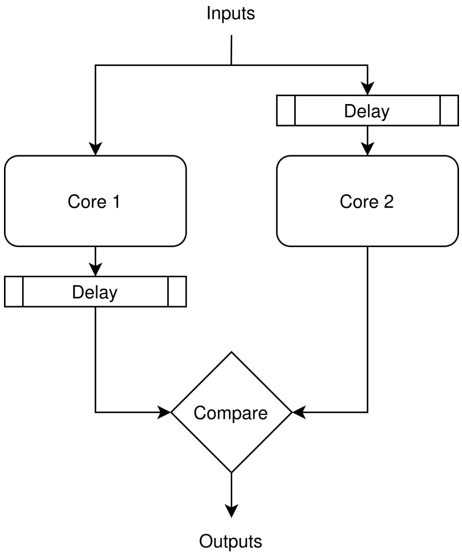


Figure 12.1: Generic DCLS processor scheme.

In the case of SEs, a key product requiring protection, according to [99], a Cortex-M0 microcontroller core occupies 0.01 mm^2 in a 40 nm process. Meanwhile, according to [101], a single static random-access memory (SRAM) cell in the same technology occupies 0.242 µm^2.

This means that for the area of a single core, only $\frac{10}{0.242} \approx 41,322$ SRAM cells can fit, equivalent to approximately 5 kB. Although these numbers may vary depending on the technology, the order of magnitude should remain.

These numbers indicate that any alternative solution ensuring data integrity must not introduce a memory overhead exceeding 5 kB.

For more complex chips (such as those used in safety-critical applications), the economic considerations may shift, as different types of memory are used in these cases.

12.1.4 Limitations

Lockstep processors are used for security primarily due to the lack of economic alternatives for very small chips. These processors have proven effective over time, demonstrating their ability to protect against various fault injection techniques.

However, injection techniques continue to evolve, and if an attacker were to successfully target both cores simultaneously in a precise manner [84], the security provided by lockstep processors could become obsolete overnight. Indeed, there is no inherent security guarantee ensuring the integrity of the system. Unlike cryptographic integrity schemes, which offer formal assurances, lockstep processors do not provide such guarantees.

Consequently, even though lockstep processors are effective today, they are not future-proof.

12.2 Instruction Set Randomization

“Hardware-Assisted Program Execution Integrity: HAPEI” Ronan Lashermes, Hélène Le Bouder, and Gaël Thomas. **Secure IT Systems - 23rd Nordic Conference, NordSec 2018** [39]

Having explored the limitations of lockstep processors, this section introduces ISR schemes as an alternative security mechanism. ISR schemes aim to enhance execution integrity by encoding the CFG in the program binary, thereby preventing attackers from altering the control-flow. This approach offers a robust defence against fault injection attacks and other threats that compromise the integrity of program execution. The following discussion delves into the principles of ISR, its implementation, and its implications for microarchitecture security.

12.2.1 Working Principle

This is a simplified description inspired by [39]. I omit implementation details to focus on the core principles.

To ensure execution integrity (i.e., instruction integrity and control-flow integrity), instruction-set randomization (ISR) techniques can be employed [19, 33]. The core idea is to modify the program by encoding instructions, control-flow information, and integrity tags in a way that allows on-the-fly verification during execution.

12.2.1.1 1-Predecessor Case

Encoding: Let k be a device-specific cryptographic key. An accumulator value, acc, is defined which represents the computation history between instruction executions. Initially,

$$acc_0 = MAC_k(IV), \tag{12.1}$$

where IV is a random initialization vector.

For a program consisting of an ordered sequence of instructions $[i_1, i_2, \cdots, i_n, \cdots]$:
Instructions with only one predecessor can be encoded as

$$i'_n = C(acc_n) \oplus i_n, \tag{12.2}$$

where C is a compression function (effectively a hash) ensuring that $C(acc_n)$ has the same bit width as an instruction. The accumulator is updated in the single-predecessor case as follows:

$$acc_{n+1} = MAC_k(acc_n || i_n). \tag{12.3}$$

The accumulator, representing the execution history at a given point in time, is updated based on the previous history and the last executed instruction.

Decoding is performed similarly.

During execution, the accumulator is updated on-the-fly, enabling the correct decoding of the next instruction:

$$\begin{aligned} acc_n &= MAC_k(acc_{n-1} || i_{n-1}), & (12.4) \\ i_n &= C(acc_n) \oplus i'_n. & (12.5) \end{aligned}$$

This construction ensures that the correct decoding of the next instruction is only possible if both the encoded instruction and execution history are valid. The scheme provides strong control-flow integrity guarantees, backed by cryptographic constructs.

12.2.1.2 2-Predecessor Case

A challenge arises when an instruction has two or more predecessors. *The two-predecessor case can be considered without loss of generality.*

In this scenario, an instruction has two possible execution histories. Therefore, a scheme that allows computing a unique new execution history from two possible values is needed. The typical solution is to store metadata alongside multi-predecessor instructions to patch the accumulator based on the previous value.

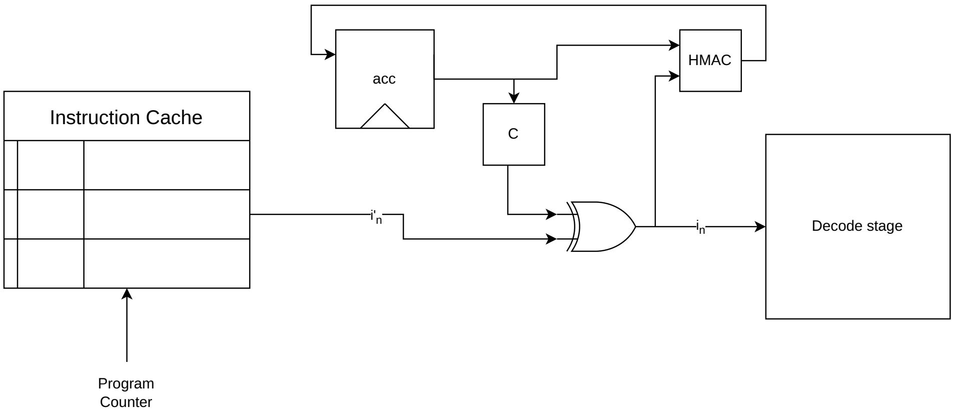


Figure 12.2: The fetch stage must be modified to decode (ISR decode, not microarchitectural decode) instructions on-the-fly using the ISR scheme. Only the 1-predecessor circuitry is shown.

A more general polynomial-based solution was developed in [39], but it incurs significant computational and memory overhead, making it impractical. However, most solutions (e.g., [15]) directly embed patch values within the executable file.

12.2.2 Forward Indirect Jumps

As discussed in chapter 11, forward indirect jumps may redirect control flow in a way that is not determinable at compile time. As a result, ISR schemes struggle to handle these jumps effectively.

Some approaches exist: Confidaent [56] supports forward indirect jumps but only to a limited set of locations. The key idea is that the implementation allows access to the accumulator value through dedicated instructions. Thus, it is possible to dynamically patch the accumulator for indirect jumps, but at a high cost: either storing patches for all possible destinations or computing them on-the-fly, requiring additional instructions for each indirect jump.

In my opinion, this issue has already been extensively discussed in chapter 11: forward indirect jumps should be prohibited. Efficiently handling backward indirect jumps remains an open problem, as far as I know.

12.2.3 (Micro)Architectural State Integrity

ISR schemes are not equivalent to lockstep processors, as they do not guarantee architectural and microarchitectural state integrity. MAFIA [15] or Gousselot et al [27] propose a mechanism to address this limitation.

Without delving too deeply into details, the core idea is to include microarchitectural signal values as part of the execution history:

$$acc_{n+1} = MAC_k(acc_n||i_n||\mu_n), \tag{12.6}$$

where μ_n is a bit vector containing the signals whose integrity must be protected at that clock cycle.

This technique requires a microarchitectural model of the target chip during encoding, necessitating a new application lifecycle (cf. section 12.3). It also imposes the constraint that the microarchitecture must be deterministic, meaning two executions of the same program must produce identical signal values. While this may be feasible for simple microcontrollers, it is generally not true for complex out-of-order cores.

12.2.4 Data Integrity

Data integrity is often considered separately from other integrity concerns, as it requires distinct techniques. To ensure data integrity in RAM, cryptographic integrity schemes can be employed, such as storing integrity tags alongside the data. These schemes are typically combined with confidentiality mechanisms. Recent cryptographic algorithms, such as ASCON [100], are authenticated ciphers that ensure both integrity and confidentiality within a common process.

The main challenge with data integrity lies at the interface: when should data be encrypted and decrypted? Should data in cache memories be encrypted for protection, or should it remain in plaintext for performance reasons?

12.2.5 Results from Instruction Set Randomization Techniques

ISR schemes introduce computational, area, and memory overheads. Computational overhead arises from the need to compute the execution history value, decode instructions, and perform related operations. MAFIA [15], one of the most recent and well-designed solutions, reports area overheads of 6% and 24%, depending on the hardware implementation, an 18% execution time overhead, and a 29% memory overhead.

By applying these results to the computed economic values (cf. subsection 12.1.3), an area overhead budget of

$$0.76 \times 5\,\text{kB} \approx 3.8\,\text{kB},$$

is obtained, converted into SRAM cells. This memory budget must accommodate the 29% memory overhead. Thus, the application size must not exceed

$$\frac{3.8\,\text{kB}}{0.29} \approx 13.1\,\text{kB}, \tag{12.7}$$

to be economically viable with MAFIA. For larger applications, lockstep processors are more cost-effective.

12.3 The Application Lifecycle

ISR schemes require an encoding phase that transforms the binary into its encoded equivalent. Since this step relies on a device-specific key, it must be performed on the same chip that executes the software.

I refer to this step as **installation**: from the program binary, which is the output of the compilation process, the CFG is extracted, and the binary is encoded using the CFG information and the device-specific key. Thus, installation is a program that runs on the chip and generates a new executable program that can only run on that specific chip. This step must be performed whenever the binary changes or the device key is updated. Installation may also enable device-specific optimizations.

This lifecycle implies that, at some point, the unprotected program is present on the device. To adhere to the threat model, installation must therefore be performed in a secure environment.

A straightforward approach is to enforce installation as a final step in the device fabrication process. Once the device key is generated, it is used to perform the installation, and a fuse is blown to prevent further installations. Only then can the device be shipped to its final destination. However, in this scenario, firmware updates are impossible, which may be problematic depending on the use case.

An alternative approach is to allow in-situ installation, requiring a more complex process. In this case, the installation process must verify the integrity and authenticity of the installed application using a signature from the device manufacturer. This is similar to secure firmware updates, a well-studied but complex topic [88]. Interestingly, since installation is performed on the hardened core, it is reasonable to assume that an attacker cannot easily disrupt the process through physical attacks. The installation program must be set up during the device fabrication process, similar to pre-installed bootloaders commonly found in microcontrollers.

Finally, a challenge specific to our "no indirect jumps" proposal arises: if installation is a program running on the chip, how can execution transition to the installed application, given that our modified ISA explicitly prevents such jumps? This transition requires dedicated support mechanisms:

- **Hardware support:** a mechanical switch is used to remap addresses to a different physical memory location. The installation program resides at `0x1XXXXXX` and writes the encoded application to `0x2XXXXXXX`. The mechanical switch determines the upper four bits to be either `0x1` or `0x2`, allowing the application to run after a simple reboot and switch position change.
- **Configuration support:** similarly, a CSR can be used to store the upper four bits of the memory mapping. This CSR could be backed by non-volatile memory to ensure that it is applied at the next boot.

12.4 The Limits of Control-Flow Integrity

When discussing CFI, it is essential to reflect on the actual protection offered by CFI techniques. In general, ensuring CFI is a way to prevent an attacker from accessing data they should not have access to. A typical attack scenario involves diverting the control flow to read and exfiltrate a secret using the application's privileges: CFI is required to maintain data confidentiality.

However, our expectations must be adjusted because of the **control-flow integrity impossibility conjecture** (definition 12.1).

Definition 12.1 - *Control-Flow Integrity Impossibility Conjecture*

In sufficiently large and complex programs, enforcing CFI guarantees that the program's execution is restricted to paths present in its CFG. However, because such CFGs inherently contain Turing-complete substructures (i.e., virtual machines), an attacker who can manipulate the data that determines which valid CFG edge is taken can simulate arbitrary computations.
As a result, CFI alone cannot be transposed as a strong security property regarding data. For example, CFI does not guarantee that sensitive data will remain inaccessible in sufficiently large programs.

To illustrate this conjecture, consider the CFG of a program without forward indirect jumps, as shown in Figure 12.4. This CFG is precise, and the edges between instructions represent valid execution sequences.

When a node in the CFG, representing an instruction (or block of instructions), has multiple outgoing edges, it means that the instruction can be followed by any of the pointed instructions, depending on data. The CFG highlights the dichotomy between the world of instructions and the world of data. Instructions are represented as nodes in the CFG, but data influence execution by determining which edges actually define the program's progress. In this representation, CFI ensures that the program execution adheres strictly to the CFG, but it usually cannot guarantee that the correct edge is taken.

An attacker may attempt to execute their own code snippets, for example, by following new edges in the CFG that are not normally present. When CFI is enforced, this is not possible.

```
x += y

if x > 10:
    while x < 20:
        x += 2
else:
    x *= 2

if x == 25:
    x = 5

return x
```

Figure 12.3: The pseudo-source code for the CFG example in Figure 12.4.

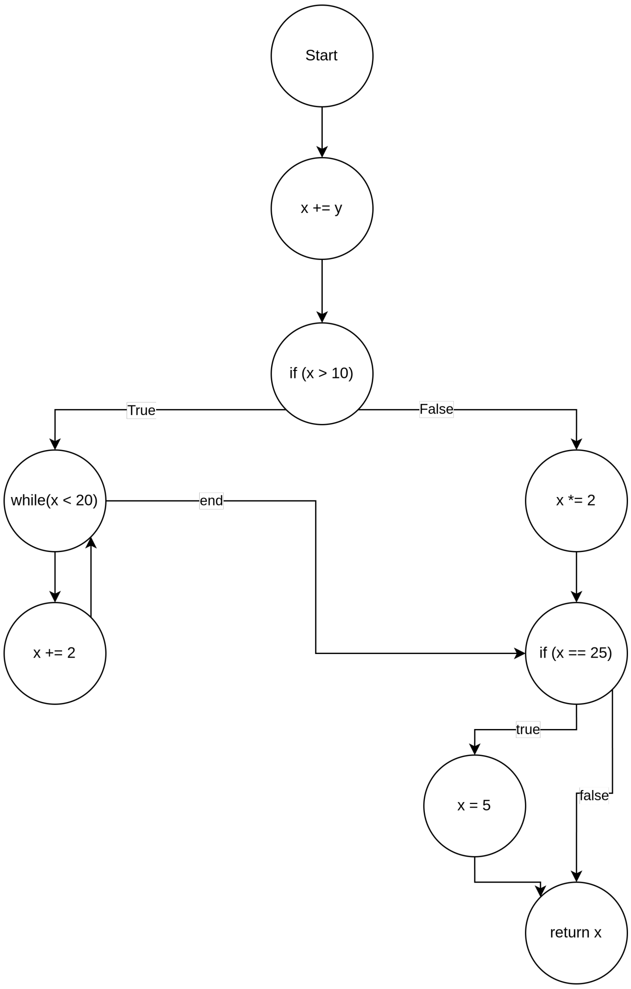


Figure 12.4: A CFG example.

However, if a program (and thus its CFG) is large enough, VMs begin to emerge. In this context, a VM is a subset of the CFG that is Turing complete. With such a VM, an attacker only needs to corrupt or control data to gain arbitrary access to the system! VMs can be extremely small (as exemplified in listing 12.1), meaning that CFI alone cannot prevent arbitrary execution by an attacker.

This impossibility result is related to the concept of weird machines [10], which describes how Turing-complete machines can emerge from sufficiently complex systems.

The CFI impossibility conjecture can be used as an argument in favour of lockstep processors: there is little value in cryptographically guaranteeing CFI if it does not translate into actual security for the data. A counterargument is that ensuring CFI may allow the compiler to prove security properties. In particular, I suspect that proving the absence of a VM within the CFG will eventually become the responsibility of the compiler. In this case, CFI enforcement, combined with proof of VM absence, would serve as a significant deterrent against attackers.

12.5 Conclusion

In this chapter, the security of lockstep processors has been shown to be at risk from emerging fault injection techniques, and ISR has been proposed as an alternative solution. ISR provides a cryptographically-based integrity guarantee by encoding both instructions and control-flow information.

Listing 12.1 Pseudo-assembly code for a small SUBLEQ VM, from [26].

```
// Data memory is filled with user-defined values
// Initialization
// Virtual program counter
// VPC(x13) = 0
load x13, #0

// Execute one subleq instruction
subleq:
// Read operands from data memory
move x1, x13
load x15, #1
add x13, x13, x15
move x2, x13
add x13, x13, x15
move x3, x13
// Increment for next instruction if no jump
add x13, x13, x15

// SUBLEQ execution
load x4, 0(x1)
load x5, 0(x2)
sub x6, x5, x4
store x6, 0(x2)
ble ijump, x6, x0

// Start next instruction
jump subleq

// Virtual indirect jump
ijump:
move x13, x3
// Start next instruction
jump subleq
```

Both lockstep processors and ISR-based processors have their merits and limitations. Currently, only lockstep processors are economically viable, but their security guarantees are too limited to not actively search for a replacement technology. However, the implementation of ISR introduces challenges related to performance and memory overhead, as well as the need for a secure application lifecycle. While ISR schemes are more secure, their practical implementation remains an active area of research.

13. Security Validation in Hardware

This chapter describes some implementation ideas that have not yet been rigorously tested, nor fully implemented, and certainly not validated through a peer-reviewed publication. As a result, there are numerous blind spots in these descriptions. However, I believe that considering actual implementations is a way to put theoretical schemes to the test and allows us to think about the bigger picture: are these schemes realistic?

When an integrity mechanism detects an issue, that could signal an ongoing attack, how should the system respond? One possible approach, focused on ensuring the confidentiality of secret data in the system, is to shut down the system: erase memory to remove secrets, blow a fuse to indicate a compromise, and prevent rebooting.

However, in most contexts, maintaining execution is preferable to ensure device availability. Thus, the system must be capable of recovering from a compromised state.

In this chapter, there are two different types of decoders: the ISR decoder, responsible for computing the instruction opcode from the encoded one as part of the ISR scheme, and the standard pipeline stage decoder, which extracts information from an instruction opcode.

In particular, consider the specific case of an ISR scheme: a fault injection corrupts an instruction to be executed, and the ISR decoder detects the fault. Papers [15, 19, 27, 33, 39, 56] on ISR leave the response to such faults as an open question, or simply assume that the infection property of the scheme will corrupt all subsequent instructions and lead to a system crash. After all, at this point, the countermeasure has demonstrated its effectiveness. In my view, recovering from a compromised state is closely tied to how the ISR decoder is implemented. In this chapter, I explore possible microarchitectures for ISR.

13.1 Fetch and Decode

All ISR schemes require ISR decoding instructions on-the-fly, a process that depends on the execution history. This step introduces a heavy computational load on the decoding path, increasing fetch and decode latency. As far as I know, no literature evaluates different microarchitectural approaches for handling this challenge.

In this section, the microarchitecture of the fetch and decode stages of an ISR-supporting microarchitecture is discussed.

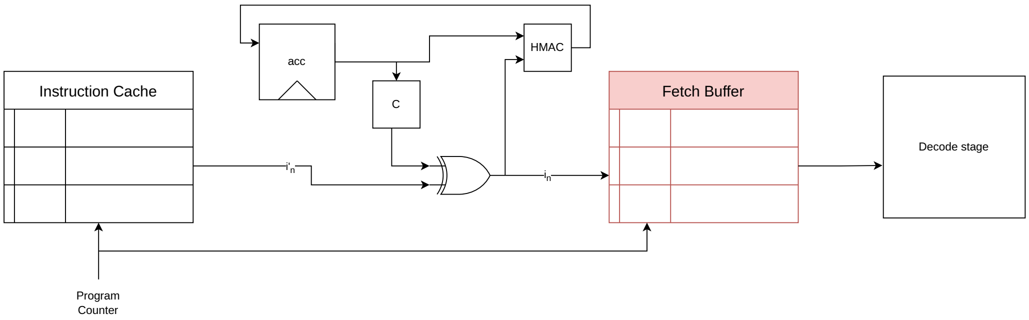


Figure 13.1: Adding a fetch buffer (in red) to the ISR scheme from Figure 12.2.

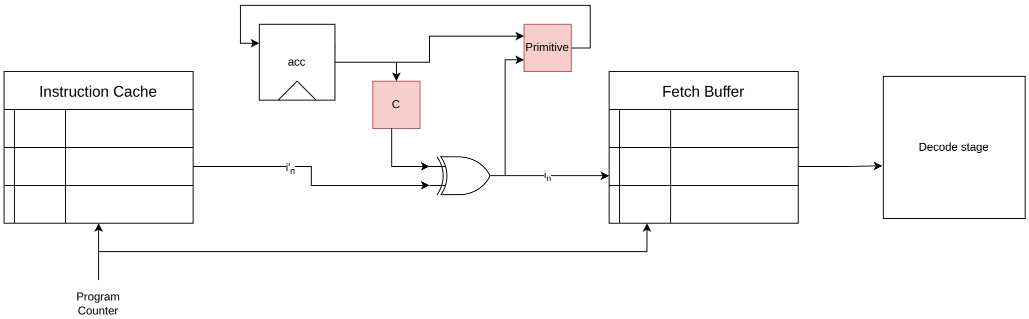


Figure 13.2: Highlighting in red the primitives to optimise.

13.1.1 Dedicated Buffer

A potential solution to amortise the cost of ISR decoding is to buffer decoded instructions in a dedicated buffer (typically in a trace buffer or a fetch buffer, as shown in Figure 13.1). Thus, if the next instruction to be executed is already in the buffer, there is no need to wait for ISR decoding. However, the accumulator value must still be updated to verify that the ISR-decoded instruction matches the executed one. The buffer does not exempt from verifying that the instruction in the buffer has been correctly ISR-decoded; however, it does decouple the data paths: verification can be done in parallel without a latency penalty.

13.1.2 Optimised Primitives

In HAPEI [39], we used an HMAC to update the accumulator, as it provides the desired security properties. However, an HMAC is too large to be directly implemented as a hardware block in the pipeline, as it would typically lie on the critical path, as shown in Figure 13.2.

In MAFIA [15], the authors use ASCON [100] for this purpose. Interestingly, although ASCON is an authenticated cipher that provides integrity guarantees, it is not used in this way. Instead, the implementation exploits the cipher's *infection property*: if a fault occurs, the ISR-decoded instruction is randomised, preventing the attacker from controlling its value. Designing dedicated primitives and modes for this specific use case should allow better performance and lower memory overhead.

13.1.3 Integrity Tags

Two techniques are commonly used to enforce and verify integrity:

1. **Infection:** the cryptographic properties of the cipher ensure that a corrupted instruction appears pseudo-random once ISR decoded. This error propagates to the accumulator and affects all subsequent instructions. There is a high probability that a core executing

random instructions will eventually crash, typically by executing invalid instructions or accessing memory out of bounds. Depending on the ISA, some opcodes are undefined, which will also cause the processor to halt. However, most ISAs are quite dense, making this less reliable.

2. **Tag-checking instructions:** a new instruction is introduced that embeds a large immediate value. This value serves as a tag (or part of a tag) corresponding to the current accumulator state. These instructions can be strategically placed within the program, typically before sensitive operations, to verify execution integrity. Adding more tag-checking instructions improves security but at the cost of increased memory usage and execution overhead.

A third option may be feasible in the case of the RISC-V ISA:

3. **2-bit immediate in all opcodes:** In RISC-V, the two least significant bits of an instruction opcode specify the instruction's size. The value "11" indicates a 32-bit instruction, while other values are reserved for the **compressed extension (C extension)**, which supports 16-bit instructions. In systems without the C extension, these two bits could be repurposed as tags. For example, a 2-bit tag could represent the accumulator value modulo 4 ($acc \mod 4$). This tag could be checked at every clock cycle. Of course, there is a 25% probability that a faulty instruction will coincidentally have a valid tag. However, due to the infection property, subsequent tags would also be affected by the fault. Thus, to achieve a given security level s at instruction i, it is sufficient to verify the tags of the $\frac{s}{2}$ instructions following i. The C extension improves code density by 25% [70], thus the memory cost of this approach.

13.2 Security Validation is Speculative

Validating integrity tags introduces a new challenge: it takes time. Waiting for tag validation before executing each instruction is not feasible from a performance perspective.

However, continuing execution without waiting for validation is precisely the purpose of speculative microarchitectures, as discussed in subsection 2.2.2. I argue that a properly designed ISR-supporting microarchitecture should take inspiration from speculative microarchitectures.

In this approach, the commit operation can only proceed once a sufficient number of tag bits have been consumed to validate one or more instructions. This method allows for a combination of 2-bit immediate tags and dedicated tag-checking instructions, achieving an optimal balance between security, memory overhead, and execution time overhead, as illustrated in Figure 13.3.

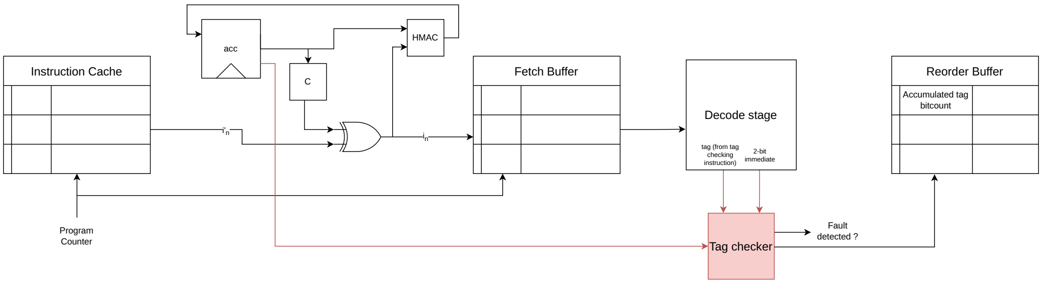


Figure 13.3: A tag checker component, in conjunction with a tag bit counter in the ROB, accumulates tag bits up to a predefined security level.

As with misspeculation, if an error is detected in the tags, the microarchitectural state can be reverted to the last valid point. Thus, this microarchitecture can ensure availability even in the presence of fault injection.

Out-of-order execution can also help ensure data integrity. To detect faults, each instruction can be executed multiple times, preferably not within the same clock cycle. A configurable register could determine how many times an instruction must be executed before it is committed. The first execution writes the result into the corresponding ROB entry, as illustrated in Figure 13.4. Subsequent executions compare their results against the stored ROB entry. If a mismatch occurs, the pipeline is reverted to the last valid committed instruction.

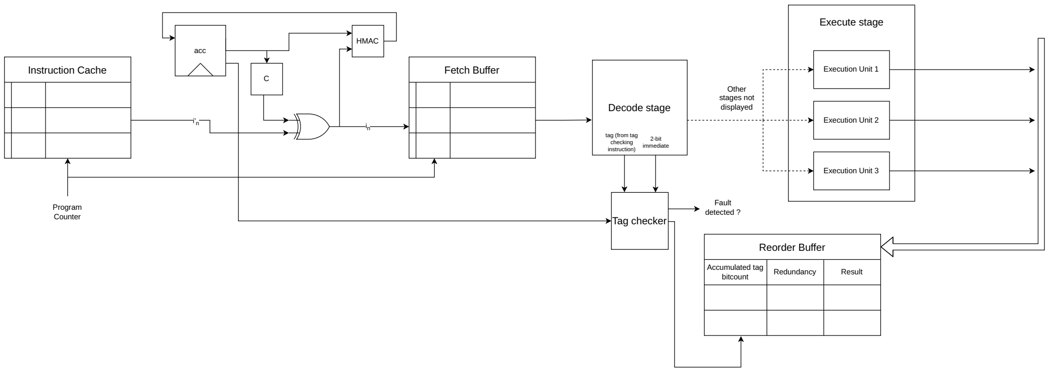
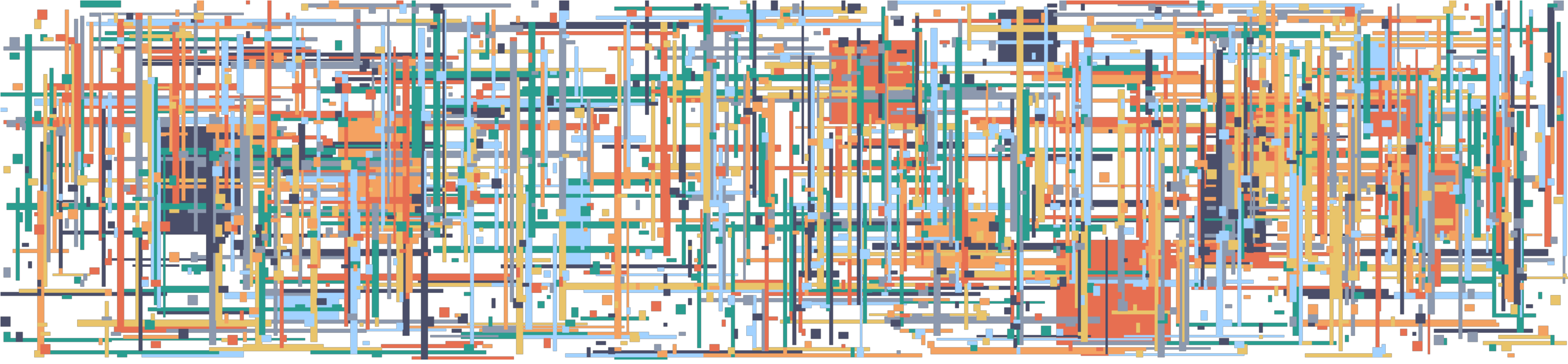


Figure 13.4: Each instruction can be executed multiple times to ensure that the computation remains fault-free.

Interestingly, increasing the number of execution units to accommodate redundant computation should minimise execution time overhead.

The proposed solution demonstrates that out-of-order microarchitectures are strong candidates for ensuring core availability, even in the presence of fault injection. However, numerous details must be refined to validate this scheme: how should faults in memories, such as the register file or the ROB, be handled? Additionally, out-of-order microarchitectures are vulnerable to transient attacks. To address this, the microarchitecture must prevent speculative loads from accessing data. In other words, it must wait to commit all instructions preceding each `load` instruction, ensuring that only non-speculative loads are executed.



14. Concluding on Radical Designs

In this part, techniques to ensure data integrity and confidentiality, particularly against physical attacks, have been proposed.

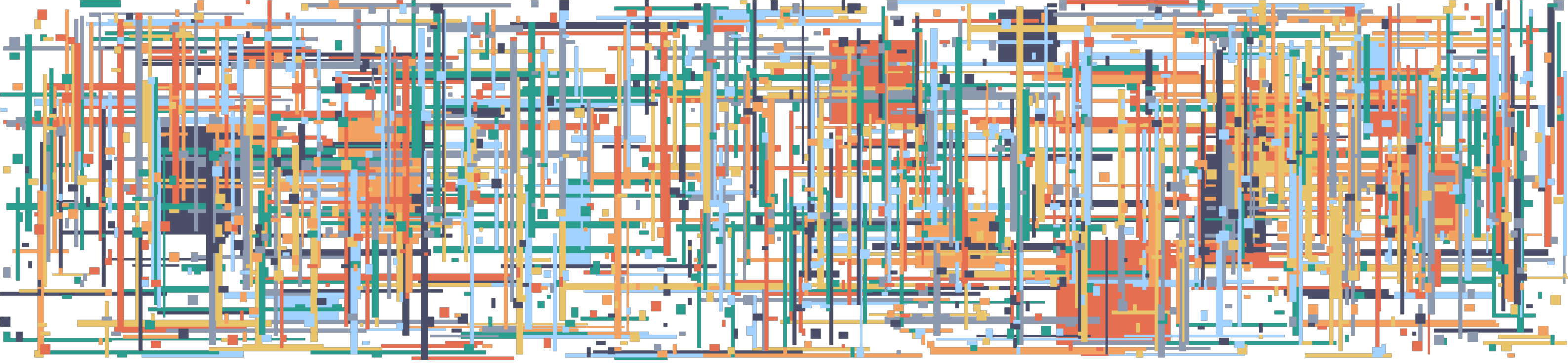
In particular, lockstep processors are the standard solution for enhancing computational integrity, but their long-term viability is uncertain. Indeed, with advances in fault injection techniques, lockstep processors may become vulnerable in the future. Therefore, it is crucial to anticipate a future without lockstep processors now.

My Recommendations for Radical Designs

Future-proof designs must ensure integrity through cryptographic primitives embedded within the microarchitecture. To lay the groundwork for more secure chips, here are my recommendations.

- The ISA must be modified in two ways:
  1. To enhance data confidentiality while maintaining acceptable performance, the microarchitecture must be aware of secret data, which can be achieved through confidential registers (chapter 10).
  2. To strengthen computational integrity, forward indirect jumps should be eliminated. This not only prevents certain control flow hijacking attacks but also lays the foundation for ISR schemes (chapter 11).
- To replace lockstep processors, instruction-set randomization (ISR) schemes should be employed. These schemes provide cryptographic guarantees for execution integrity. However, their efficiency is currently insufficient, preventing their adoption in commercial chips today (chapter 12). Research efforts should be pursued to improve the performance of these schemes.
- Microarchitectural integrity should be improved by designing a core microarchitecture capable of handling speculation for integrity validation. Redundancy can then be verified speculatively to maintain performance (chapter 13).

IV Conclusion



15. Conclusion

15.1 Reflecting on Past Works

This manuscript was heavily influenced by the joint work done with the PhD students that I advise or supervise:

- **Sébanjila Kevin Bukasa** defended his thesis "Analyse de vulnérabilité des systèmes embarqués face aux attaques physiques" in 2019.
- **Mathieu Escouteloup** defended "Garantir l'isolation microarchitecturale des processeurs" in 2021.
- **Amélie Marotta** will defend her thesis "Characterising and Modelling Synchronous Clock-Glitch Fault Injection" in 2025.
- **Herinomena Andrianatrehina** will defend his thesis "Ensuring Confidentiality in Modern Out-of-Order Cores" in 2025.

These past works were critical in developing my personal view of hardware security, centred around the necessity of a holistic design process. This view is what I try to share in this manuscript.

15.2 Future-Proof Designs

In this document, several security issues concerning modern microarchitectures have been examined. These problems are inherent to the nature of microarchitectural design, and there is no quick fix for them.

For application processors, there are currently no efficient solutions to covert channels and transient attacks. A trade-off must be made between security and performance, as achieving perfect security would require formal methods that have not yet been scaled effectively. In my opinion, vendors have too heavily favoured performance in response to demand, leaving numerous exploitable security vulnerabilities, as evidenced by the constant stream of publications detailing new attacks on these processors.

Additionally, the microarchitecture of fault-hardened microcontrollers has been described. Lockstep processors, the standard solution today, are effective in practice but do not provide an integrity guarantee in the cryptographic sense. Consequently, as fault injection techniques continue to advance, lockstep processors may eventually become vulnerable.

Current designs are not future-proof: they are adopted because they offer a favourable performance-to-cost trade-off while operating at the edge of acceptable security. New attacks continue to expose weaknesses to transient attacks, and lockstep processors may eventually succumb to emerging fault injection techniques, such as dual-beam laser fault injection.

The current vulnerabilities are not inevitable; more secure designs are possible.

If you are a hardware designer today, my recommendations depend on the device class. For application processors (chapter 9):

- In the short term, microarchitectural designers should minimise covert channels by partitioning microarchitectural state whenever possible, selecting structures that are easily flushable, and leveraging dedicated extensions such as `fence.time`. Speculation barriers currently do not offer a favourable performance/security trade-off and should likely be deferred until more robust implementations and effective fence placement strategies are developed.
- In the long term, the architecture should define confidential registers. These registers would enable efficient security enhancements such as selective speculation and provide guarantees against both architectural and microarchitectural timing channels.

For microcontrollers (chapter 14):

- In the short term, use lockstep processors.
- In the long term, define and adopt a restricted ISA that, for example, forbids forward indirect jumps in the general case.
- Instruction-set randomization (ISR) schemes should be employed to provide integrity guarantees in place of lockstep processors. However, current ISR-based cores do not yet offer acceptable efficiency.

For both types of cores, architectural confidential registers can enhance security by making the microarchitecture aware of the data's security constraints.

15.3 Perspectives: How to Get There?

Among the proposed solutions, some are readily applicable (e.g., fence.time), while others are not. In particular, some research questions still need to be addressed.

1. **Efficient instruction-set randomization (ISR) schemes**: Current ISR schemes are not performant enough to replace lockstep processors. New schemes must be developed with better performance and lower memory overhead. A promising approach would be to design schemes based on cryptographic primitives for integrity rather than repurposing confidentiality primitives.
2. **Security-aware compilers**: Mainstream compilers such as LLVM and GCC do not even guarantee correctness, let alone security! To enhance security, compilers must be security-aware: they should track security properties (e.g., confidential variables) from the input language through to instruction generation. This capability could enable security guarantees or, at the very least, ensure non-interference from optimization passes.
3. **Security-aware microarchitectures**: The hardware counterpart of better compilers is microarchitectures that can leverage the knowledge of security properties associated with data and instructions to achieve better performance/security trade-offs. In my opinion, the long-term solution to transient attacks necessitates security-aware microarchitectures.
4. **Modern hardware description languages (HDLs)**: Designing a complex out-of-order microarchitecture is challenging, made even harder by the lack of modern (public) tools. Much of hardware development still relies on VHDL or Verilog HDLs, which are outdated for this purpose, akin to developing a complex operating system in assembly language!
   More recently, a new generation of hardware description languages, such as Chisel (2012) and SpinalHDL (2015), has gained traction by leveraging modern programming paradigms and type systems (they are domain-specific languages embedded in Scala 2). These languages have demonstrated their effectiveness, as evidenced by the development of fully synthesizable out-of-order cores like BOOM (in Chisel) and Nax (in SpinalHDL), achieved by very small teams or even single individuals. Notably, leveraging strong type

systems enables early error detection, reducing the need for extensive testing.

However, Scala itself (first released in 2004) is now over twenty years old! By now, we should have better languages built on more modern foundations, featuring advanced type systems with sum types, built-in support for invariant guarantees, and more. Integrating such features into Scala would be challenging. The next generation of HDLs should be embedded in a more powerful language. In my opinion, the Lean language is a strong candidate due to its focus on formal methods.

These research topics are pursued in the research projects I am involved in: the PEPR Cyber ARSENE project, the PTCC Forward project, the Grand Défi Cyber Cyberpros project. I am part of a team of researchers trying to improve ISR schemes through a new ANR project called FAIR, currently in the proposal evaluation stage.

A cornerstone of a holistic approach to security is, in my opinion, the ISA. Hence my active involvement in the RISC-V Foundation work, as a member of the Security HC and chair of the Timing Fences Task Group. The technical debates in these groups are very interesting as they represent a variety of opinions from members representing both industry and academic points of view.

Ultimately, addressing microarchitectural security challenges requires a collective effort across academia, industry, and the open-source community. The directions outlined in this manuscript offer concrete pathways toward more secure, future-proof hardware designs, but realising these advancements depends critically on fostering open collaboration, prioritising security alongside performance, and adopting rigorous, security-aware development methodologies. While achieving secure and efficient architectures is ambitious, more research is essential to move the ecosystem in the right direction.

IV Annexes

Bibliography

Articles

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. "Predicting Secret Keys Via Branch Prediction". In: **Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings**. Edited by Masayuki Abe. Volume 4377. Lecture Notes in Computer Science. Springer, 2007, pages 225–242. DOI: 10.1007/11967668_15. URL: https://doi.org/10.1007/11967668%5C_15 (cited on page 36).

[2] Sam Ainsworth and Timothy M. Jones. "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State". In: **47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020**. IEEE, 2020, pages 132–144. DOI: 10.1109/ISCA45697.2020.00022. URL: https://doi.org/10.1109/ISCA45697.2020.00022 (cited on page 51).

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. "Port Contention for Fun and Profit". In: **2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019**. IEEE, 2019, pages 870–887. DOI: 10.1109/SP.2019.00066. URL: https://doi.org/10.1109/SP.2019.00066 (cited on page 36).

[4] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations". In: **25th USENIX Security Symposium (USENIX Security 16)**. Austin, TX: USENIX Association, Aug. 2016, pages 53–70. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida (cited on page 32).

[5] Herinomena Andrianatrehina, Ronan Lashermes, Joseph Paturel, Simon Rokicki, and Thomas Rubiano. "`fence.spec`: exploring speculation barriers for RISC-V selective speculation". In: **under submission** (2025). URL: https://ronan.lashermes.0nline.fr/papers/fence-spec.pdf (cited on page 55).

[6] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels". In: **28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019**. IEEE, 2019, pages 151–164. DOI: 10.1109/PACT.2019.00020. URL: https://doi.org/10.1109/PACT.2019.00020 (cited on pages 51, 52).

[7] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: **2018 IEEE 31st Computer Security Foundations Symposium (CSF)**. 2018, pages 328–343. DOI: 10.1109/CSF.2018.00031 (cited on page 32).

[8] Daniel J Bernstein. "Cache-timing attacks on AES". In: (2005) (cited on page 36).

[9] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. "MI6: Secure Enclaves in a Speculative Out-of-Order Processor". In: **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019**. ACM, 2019, pages 42–56. DOI: 10.1145/3352460.3358310. URL: https://doi.org/10.1145/3352460.3358310 (cited on pages 51, 53).

[10] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation". In: **login Usenix Mag.** 36.6 (2011). URL: https://www.usenix.org/publications/login/december-2011-volume-36-number-6/exploit-programming-buffer-overflows-weird (cited on page 85).

[11] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: **2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020**. IEEE, 2020, pages 54–72. DOI: 10.1109/SP40000.2020.00089. URL: https://doi.org/10.1109/SP40000.2020.00089 (cited on page 48).

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: **26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017**. Edited by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pages 1041–1056. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck (cited on page 36).

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: **28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019**. Edited by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pages 249–266. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella (cited on page 45).

[14] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: **Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019**. Edited by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pages 769–784. DOI: 10.1145/3319535.3363219. URL: https://doi.org/10.1145/3319535.3363219 (cited on page 45).

[15] Thomas Chamelot, Damien Couroussé, and Karine Heydemann. "MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks". In: **IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.** 42.12 (2023), pages 4555–4568. DOI: 10.1109/TCAD.2023.3276507. URL: https://doi.org/10.1109/TCAD.2023.3276507 (cited on pages 82, 83, 88, 89).

[16] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. "Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy". In: **MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021**.

ACM, 2021, pages 607–622. DOI: 10.1145/3466752.3480068. URL: https://doi.org/10.1145/3466752.3480068 (cited on pages 51, 52).

[17] A D'Amato, S Dancel, J Pilutti, L Tellis, E Frascaroli, and JC Gerdes. "Exceptional driving principles for autonomous vehicles". In: **JL & Mobility** (2022), page 1 (cited on page 24).

[18] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. "ProSpeCT: Provably Secure Speculation for the Constant-Time Policy". In: **32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023**. Edited by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pages 7161–7178. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/daniel (cited on page 53).

[19] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. "SOFIA: Software and control flow integrity architecture". In: **Computers & Security** 68 (2017), pages 16–35. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2017.03.013. URL: https://www.sciencedirect.com/science/article/pii/S0167404817300664 (cited on pages 81, 88).

[20] Mathieu Escouteloup, Ronan Lashermes, Jacques Fournier, and Jean-Louis Lanet. "Under the Dome: Preventing Hardware Timing Information Leakage". In: **Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021**. Edited by Vincent Grosso and Thomas Pöppelmann. Volume 13173. Lecture Notes in Computer Science. Springer, 2021, pages 233–253. DOI: 10.1007/978-3-030-97348-3_13. URL: https://ronan.lashermes.0nline.fr/papers/CARDIS2021.pdf (cited on pages 11, 35, 37, 39, 41).

[21] Mathieu Escouteloup, Ronan Lashermes, Jean-Louis Lanet, and Jacques Fournier. "Recommendations for a radically secure ISA". In: **CARRV 2020-Workshop on Computer Architecture Research with RISC-V**. ACM. 2020, pages 1–22. URL: https://ronan.lashermes.0nline.fr/papers/CARRV2020.pdf (cited on page 73).

[22] Clément Fanjas, Driss Aboulkassimi, Simon Pontié, and Jessy Clédière. "Exploration of System-on-Chip Secure-Boot Vulnerability to Fault-Injection by Side-Channel Analysis". In: **IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2023, Juan-Les-Pins, France, October 3-5, 2023**. Edited by Luca Cassano, Mihalis Psarakis, Marcello Traiola, and Alberto Bosio. IEEE, 2023, pages 1–6. DOI: 10.1109/DFT59622.2023.10313346. URL: https://doi.org/10.1109/DFT59622.2023.10313346 (cited on pages 24, 25).

[23] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Marno van der Maas, Alexandre Joannou, Alexander Richardson, Jessica Clarke, Nathaniel Wesley Filardo, Brooks Davis, John Baldwin, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. "Architectural Contracts for Safe Speculation". In: **41st IEEE International Conference on Computer Design, ICCD 2023, Washington, DC, USA, November 6-8, 2023**. IEEE, 2023, pages 578–586. DOI: 10.1109/ICCD58817.2023.00093. URL: https://doi.org/10.1109/ICCD58817.2023.00093 (cited on page 53).

[24] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks". In: **Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019**. ACM, 2019, page 61. DOI: 10.1145/3316781.3317914. URL: https://doi.org/10.1145/3316781.3317914 (cited on pages 51, 52).

[25] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era". In: **CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020**. Edited by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pages 1871–1885. DOI: `10.1145/3372297.3417289`. URL: `https://doi.org/10.1145/3372297.3417289` (cited on page 48).

[26] Alexandre Gonzalvez and Ronan Lashermes. "A case against indirect jumps for secure programs". In: **Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering 2019**. 2019, pages 1–10. URL: `https://ronan.lashermes.0nline.fr/papers/SSPREW2019.pdf` (cited on pages 73, 77, 86).

[27] Théophile Gousselot, Jean-Max Dutertre, Olivier Potin, and Jean-Baptiste Rigaud. "Code Encryption for Confidentiality and Execution Integrity down to Control Signals". In: **IEEE International Symposium on Hardware Oriented Security and Trust (HOST)**. 2025 (cited on pages 82, 88).

[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "TLBleed: When Protecting Your CPU Caches is not Enough". In: **Black Hat USA**. Aug. 2018. URL: `Slides=https://i.blackhat.com/us-18/Thu-August-9/us-18-Gras-TLBleed-When-Protecting-Your-CPU-Caches-is-Not-Enough.pdf%20Web=https://vusec.net/projects/tlbleed` (cited on page 36).

[29] Conor Green, Cole Nelson, Mithuna Thottethodi, and T. N. Vijaykumar. "SafeBet: Secure, Simple, and Fast Speculative Execution". In: **CoRR** abs/2306.07785 (2023). DOI: `10.48550/ARXIV.2306.07785`. arXiv: `2306.07785`. URL: `https://doi.org/10.48550/arXiv.2306.07785` (cited on pages 51, 53).

[30] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: **42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021**. IEEE, 2021, pages 1868–1883. DOI: `10.1109/SP40001.2021.00036`. URL: `https://doi.org/10.1109/SP40001.2021.00036` (cited on page 53).

[31] Guangyuan Hu, Zecheng He, and Ruby B. Lee. "SoK: Hardware Defenses Against Speculative Execution Attacks". In: **2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021**. IEEE, 2021, pages 108–120. DOI: `10.1109/SEED51797.2021.00023`. URL: `https://doi.org/10.1109/SEED51797.2021.00023` (cited on page 51).

[32] Hai Jin, Zhuo He, and Weizhong Qiang. "SpecTerminator: Blocking Speculative Side Channels Based on Instruction Classes on RISC-V". In: **ACM Trans. Archit. Code Optim.** 20.1 (2023), 15:1–15:26. DOI: `10.1145/3566053`. URL: `https://doi.org/10.1145/3566053` (cited on pages 51, 52).

[33] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: **Proceedings of the 10th ACM Conference on Computer and Communications Security**. CCS '03. Washington D.C., USA: Association for Computing Machinery, 2003, pages 272–280. ISBN: 1581137389. DOI: `10.1145/948109.948146`. URL: `https://doi.org/10.1145/948109.948146` (cited on pages 81, 88).

[34] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". In: **Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA,**

June 02-06, 2019. ACM, 2019, page 60. DOI: 10.1145/3316781.3317903. URL: https://doi.org/10.1145/3316781.3317903 (cited on page 51).

[35] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. "FLOP: Breaking the Apple M3 CPU via False Load Output Predictions". In: **USENIX Security**. 2025 (cited on page 48).

[36] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: **51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018**. IEEE Computer Society, 2018, pages 974–987. DOI: 10.1109/MICRO.2018.00083. URL: https://doi.org/10.1109/MICRO.2018.00083 (cited on page 51).

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: **40th IEEE Symposium on Security and Privacy (S&P'19)**. 2019 (cited on pages 44, 46, 47).

[38] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: **Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings**. Edited by Neal Koblitz. Volume 1109. Lecture Notes in Computer Science. Springer, 1996, pages 104–113. DOI: 10.1007/3-540-68697-5_9. URL: https://doi.org/10.1007/3-540-68697-5%5C_9 (cited on page 30).

[39] Ronan Lashermes, Hélène Le Bouder, and Gaël Thomas. "Hardware-Assisted Program Execution Integrity: HAPEI". In: **Secure IT Systems - 23rd Nordic Conference, NordSec 2018**. Edited by Nils Gruschka. Volume 11252. Lecture Notes in Computer Science. Springer, 2018, pages 405–420. DOI: 10.1007/978-3-030-03638-6_25. URL: https://doi.org/10.1007/978-3-030-03638-6%5C_25 (cited on pages 80–82, 88, 89).

[40] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks". In: **25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019**. IEEE, 2019, pages 264–276. DOI: 10.1109/HPCA.2019.00043. URL: https://doi.org/10.1109/HPCA.2019.00043 (cited on pages 51, 52).

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: **27th USENIX Security Symposium (USENIX Security 18)**. 2018 (cited on page 45).

[42] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. "DOLMA: Securing Speculation with the Principle of Transient Non-Observability". In: **30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021**. Edited by Michael D. Bailey and Rachel Greenstadt. USENIX Association, 2021, pages 1397–1414. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin (cited on pages 51, 52).

[43] Giorgi Maisuradze and Christian Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018**. Edited by David Lie, Mohammad Mannan, Michael Backes, and

XiaoFeng Wang. ACM, 2018, pages 2109–2122. DOI: 10.1145/3243734.3243761. URL: https://doi.org/10.1145/3243734.3243761 (cited on page 47).

[44] Daniel S. McFarlin, Charles Tucker, and Craig B. Zilles. "Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?" In: **Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013**. Edited by Vivek Sarkar and Rastislav Bodík. ACM, 2013, pages 241–252. DOI: 10.1145/2451116.2451143. URL: https://doi.org/10.1145/2451116.2451143 (cited on page 61).

[45] Matt Miller. "Trends, Challenges, and Shifts in Software Vulnerability Mitigation". In: **Proceedings of BlueHat IL 2019**. 2019. URL: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf (cited on page 28).

[46] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing". In: **44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023**. IEEE, 2023, pages 1737–1752. DOI: 10.1109/SP46215.2023.10179391. URL: https://doi.org/10.1109/SP46215.2023.10179391 (cited on page 48).

[47] Hamza Omar and Omer Khan. "IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications". In: **IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020**. IEEE, 2020, pages 111–122. DOI: 10.1109/HPCA47549.2020.00019. URL: https://doi.org/10.1109/HPCA47549.2020.00019 (cited on pages 51, 53).

[48] Marco Patrignani and Marco Guarnieri. "Exorcising Spectres with Secure Compilers". In: **CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021**. Edited by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pages 445–461. DOI: 10.1145/3460120.3484534. URL: https://doi.org/10.1145/3460120.3484534 (cited on pages 55, 60).

[49] Pengfei Qiu, Qiang Gao, Dongsheng Wang, Yongqiang Lyu, Chunlu Wang, Chang Liu, Rihui Sun, and Gang Qu. "PMU-Leaker: Performance Monitor Unit-Based Realization of Cache Side-Channel Attacks". In: **Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC 2023, Tokyo, Japan, January 16-19, 2023**. Edited by Atsushi Takahashi. ACM, 2023, pages 664–669. DOI: 10.1145/3566097.3567917. URL: https://doi.org/10.1145/3566097.3567917 (cited on page 36).

[50] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. "GhostRace: Exploiting and Mitigating Speculative Race Conditions". In: **USENIX Security**. Aug. 2024. URL: Paper=https://download.vusec.net/papers/ghostrace_sec24.pdf%20Web=https://www.vusec.net/projects/ghostrace%20Code=https://github.com/vusec/ghostrace (cited on page 48).

[51] Allison Randal. "This is How You Lose the Transient Execution War". In: **CoRR** abs/2309.03376 (2023). DOI: 10.48550/ARXIV.2309.03376. arXiv: 2309.03376. URL: https://doi.org/10.48550/arXiv.2309.03376 (cited on pages 51, 52, 56).

[52] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. "I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches". In: **48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021**. IEEE, 2021, pages 361–374. DOI: 10.1109/ISCA52012.2021.00036. URL: https://doi.org/10.1109/ISCA52012.2021.00036 (cited on page 36).

[53] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. "CoVE: Towards Confidential Computing on RISC-V Platforms". In: **Proceedings of the 20th ACM International Conference on Computing Frontiers, CF 2023, Bologna, Italy, May 9-11, 2023**. Edited by Andrea Bartolini, Kristian F. D. Rietveld, Catherine D. Schuman, and Jose Moreira. ACM, 2023, pages 315–321. DOI: 10.1145/3587135.3592168. URL: https://doi.org/10.1145/3587135.3592168 (cited on page 69).

[54] Gururaj Saileshwar and Moinuddin K. Qureshi. "CleanupSpec: An "Undo" Approach to Safe Speculation". In: **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019**. ACM, 2019, pages 73–86. DOI: 10.1145/3352460.3358314. URL: https://doi.org/10.1145/3352460.3358314 (cited on page 51).

[55] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. "Efficient invisible speculative execution through selective delay and value prediction". In: **Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019**. Edited by Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman. ACM, 2019, pages 723–735. DOI: 10.1145/3307650.3322216. URL: https://doi.org/10.1145/3307650.3322216 (cited on pages 51, 52).

[56] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. "Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption". In: **23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020**. IEEE, 2020, pages 246–253. DOI: 10.1109/DSD51259.2020.00048. URL: https://doi.org/10.1109/DSD51259.2020.00048 (cited on pages 82, 88).

[57] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue In-Flight Data Load". In: **2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019**. IEEE, 2019, pages 88–105. DOI: 10.1109/SP.2019.00087. URL: https://doi.org/10.1109/SP.2019.00087 (cited on page 45).

[58] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM errors in the wild: a large-scale field study". In: **Commun. ACM** 54.2 (2011), pages 100–107. DOI: 10.1145/1897816.1897844. URL: https://doi.org/10.1145/1897816.1897844 (cited on page 25).

[59] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. "ConTExT: A Generic Approach for Mitigating Spectre". In: **27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020**. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/context-a-generic-approach-for-mitigating-spectre/ (cited on pages 51, 52, 69).

[60] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: **Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019**. Edited by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM, 2019, pages 753–768. DOI: 10.1145/3319535.3354252. URL: https://doi.org/10.1145/3319535.3354252 (cited on page 45).

[61] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. “Leaky Controller: Cross-VM Memory Controller Covert Channel on Multi-core Systems”. In: **ICT Systems Security and Privacy Protection - 35th IFIP TC 11 International Conference, SEC 2020, Maribor, Slovenia, September 21-23, 2020, Proceedings**. Edited by Marko Hölbl, Kai Rannenberg, and Tatjana Welzer. Volume 580. IFIP Advances in Information and Communication Technology. Springer, 2020, pages 3–16. DOI: 10.1007/978-3-030-58201-2_1. URL: https://doi.org/10.1007/978-3-030-58201-2%5C_1 (cited on page 36).

[62] Young-joo Shin, Hyung Chan Kim, Dokeun Kwon, Ji-Hoon Jeong, and Junbeom Hur. “Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage”. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018**. Edited by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pages 131–145. DOI: 10.1145/3243734.3243736. URL: https://doi.org/10.1145/3243734.3243736 (cited on page 36).

[63] Mong Tee Sim and Yanyan Zhuang. “A Dual Lockstep Processor System-on-a-Chip for Fast Error Recovery in Safety-Critical Applications”. In: **The 46th Annual Conference of the IEEE Industrial Electronics Society, IECON 2020, Singapore, October 18-21, 2020**. IEEE, 2020, pages 2231–2238. DOI: 10.1109/IECON43393.2020.9255188. URL: https://doi.org/10.1109/IECON43393.2020.9255188 (cited on page 79).

[64] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: **26th USENIX Security Symposium (USENIX Security 17)**. Vancouver, BC: USENIX Association, Aug. 2017, pages 1057–1074. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang (cited on page 21).

[65] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. “Mitigating Speculative Execution Attacks via Context-Sensitive Fencing”. In: **IEEE Des. Test** 39.4 (2022), pages 49–57. DOI: 10.1109/MDAT.2022.3152633. URL: https://doi.org/10.1109/MDAT.2022.3152633 (cited on pages 51, 52).

[66] Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. “Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design”. In: **PACT ’20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020**. Edited by Vivek Sarkar and Hyesoon Kim. ACM, 2020, pages 241–254. DOI: 10.1145/3410463.3414640. URL: https://doi.org/10.1145/3410463.3414640 (cited on pages 51, 52).

[67] Jack Wampler, Ian Martiny, and Eric Wustrow. “ExSpectre: Hiding Malware in Speculative Execution”. In: **26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019**. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/exspectre-hiding-malware-in-speculative-execution/ (cited on page 48).

[68] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: **31st USENIX Security Symposium (USENIX Security 22)**. Boston, MA: USENIX Association, Aug. 2022, pages 679–697. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen (cited on page 21).

[69] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: **31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022**. Edited by Kevin R. B. Butler and Kurt Thomas. USENIX Association, 2022, pages 679–697. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen (cited on page 36).

[70] Andrew Waterman. "Improving energy efficiency and reducing code size with RISC-V compressed". In: **Master's thesis** (2011) (cited on page 90).

[71] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. "NDA: Preventing Speculative Execution Attacks at Their Source". In: **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019**. ACM, 2019, pages 572–586. DOI: 10.1145/3352460.3358306. URL: https://doi.org/10.1145/3352460.3358306 (cited on pages 51, 52).

[72] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. "Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core". In: **Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021**. IEEE, 2021, pages 627–632. DOI: 10.23919/DATE51398.2021.9474214. URL: https://doi.org/10.23919/DATE51398.2021.9474214 (cited on page 39).

[73] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Gernot Heiser, and Luca Benini. "Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning". In: **IEEE Trans. Computers** 72.5 (2023), pages 1420–1430. DOI: 10.1109/TC.2022.3212636. URL: https://doi.org/10.1109/TC.2022.3212636 (cited on page 39).

[74] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy". In: **51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018**. IEEE Computer Society, 2018, pages 428–441. DOI: 10.1109/MICRO.2018.00042. URL: https://doi.org/10.1109/MICRO.2018.00042 (cited on page 51).

[75] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. "Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution". In: **47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020**. IEEE, 2020, pages 707–720. DOI: 10.1109/ISCA45697.2020.00064. URL: https://doi.org/10.1109/ISCA45697.2020.00064 (cited on pages 51, 52).

[76] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data". In: **IEEE Micro** 40.3 (2020), pages 81–90. DOI:

10.1109/MM.2020.2985359. URL: https://doi.org/10.1109/MM.2020.2985359 (cited on pages 51, 52).

[77] Hao Zhan and Dan Wan. “Ethical Considerations of the Trolley Problem in Autonomous Driving: A Philosophical and Technological Analysis”. In: **World Electric Vehicle Journal** 15.9 (2024). ISSN: 2032-6653. DOI: 10.3390/wevj15090404. URL: https://www.mdpi.com/2032-6653/15/9/404 (cited on page 24).

[78] Zhi Zhang, Yueqiang Cheng, Yinqian Zhang, and Surya Nepal. “GhostKnight: Breaching Data Integrity via Speculative Execution”. In: **CoRR** abs/2002.00524 (2020). arXiv: 2002.00524. URL: https://arxiv.org/abs/2002.00524 (cited on page 48).

[79] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. “Ultimate SLH: Taking Speculative Load Hardening to the Next Level”. In: **32nd USENIX Security Symposium (USENIX Security 23)**. Anaheim, CA: USENIX Association, Aug. 2023, pages 7125–7142. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh (cited on pages 55, 60).

[80] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. “Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis”. In: **53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020**. IEEE, 2020, pages 1138–1152. DOI: 10.1109/MICRO50266.2020.00094. URL: https://doi.org/10.1109/MICRO50266.2020.00094 (cited on pages 51, 52).

Livres

[81] Auguste Kerckhoffs. **La cryptographie militaire**. Journal des Sciences Militaires, 1883 (cited on page 66).

[82] Donald A. Norman. **The Design of Everyday Things**. 1988 (cited on page 25).

[83] David A Patterson and John L Hennessy. **Computer organization and Design**. Morgan Kaufmann, 1994 (cited on page 17).

Autres publications

[84] AlphaNov. **Injection de fautes laser double**. Accessed: 2025-03-25. 2025. URL: https://archive.is/kYEcC (cited on page 80).

[85] AMD. **Software Techniques for Managing Speculation on AMD Processors**. Technical Document. Available online at https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/software-techniques-for-managing-speculation.pdf. AMD, 2024. URL: %7Bhttps://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/software-techniques-for-managing-speculation.pdf%7D (cited on page 50).

[86] Darrell D Boggs, Ross Segelken, Mike Cornaby, Nick Fortino, Shailender Chaudhry, Denis Khartikov, Alok Mooley, Nathan Tuck, and Gordon Vreugdenhil. **Memory type which is cacheable yet inaccessible by speculative instructions**. US Patent 10,642,744. 2020 (cited on page 52).

[87] Chandler Carruth. **Speculative Load Hardening: A Spectre Variant #1 Mitigation Technique**. https://llvm.org/docs/SpeculativeLoadHardening.html. Accessed: 2024-03-12. Mar. 2024 (cited on pages 54, 55).

[88] Cloud Security Industry Summit Supply Chain Technical Working Group. **Secure Firmware Development Best Practices**. Technical report. July 2019. URL: https://www.opencompute.org/documents/csis-firmware-security-best-practices-position-paper-version-1-0-pdf (cited on page 84).

[89] Cory Doctorow. **The Coming War on General Computation**. Accessed: 2024-11-15. 2011. URL: http://opentranscripts.org/transcript/coming-war-general-computation/ (cited on page 24).

[90] Morris Dworkin. **Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices**. Special Publication 800-38E. National Institute of Standards and Technology, Jan. 2010. DOI: 10.6028/NIST.SP.800-38E. URL: https://csrc.nist.gov/publications/detail/sp/800-38e/final (cited on page 69).

[91] Andreas Galauner. **Glitching the Switch**. Online video. Accessed: 2024-10-29. 2018. URL: https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch (cited on pages 24, 25).

[92] Moein Ghaniyoun. **RISC-V LLVM-SLH implementation**. https://github.com/MoeinGhaniyoun/LLVM-SLH-RISCV. Accessed: 2024-07-19. 2024 (cited on page 55).

[93] J. Horn. **Speculative Execution, Variant 4: Speculative Store Bypass**. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. 2024/02/16. Feb. 2018 (cited on page 47).

[94] Intel Corporation. **Speculative Execution Side Channel Mitigations**. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html. Accessed: 2024-02-27. Feb. 2024 (cited on page 49).

[95] Colin Percival. **Cache missing for fun and profit**. 2005 (cited on page 36).

[96] Phillip Rogaway. **Evaluation of Some Blockcipher Modes of Operation**. Technical report. Evaluation carried out for the Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan. University of California, Davis, Feb. 2011. URL: https://www.cs.ucdavis.edu/~rogaway/papers/modes.pdf (cited on page 69).

[97] Thomas Rubiano. **RISC-V LLVM-SLH Inria's implementation**. https://gitlab.inria.fr/arsene-pepr/llvm-fence-spec. Accessed: 2024-07-19. 2024 (cited on page 55).

[98] rvkrypto contributors. **RISC-V Zkt Extension**. https://github.com/rvkrypto/riscv-zkt-list/blob/main/zkt-list.adoc. Accessed: 2024-05-14. 2021 (cited on page 33).

[99] Anand Lal Shimpi. **ARM's Cortex M: Even Smaller and Lower Power CPU Cores**. Accessed: 2025-01-09. Aug. 2014. URL: https://www.anandtech.com/show/8400/arms-cortex-m-even-smaller-and-lower-power-cpu-cores (cited on page 80).

[100] National Institute of Standards and Technology (NIST). **Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions**. Initial Public Draft NIST SP 800-232. Accessed: 2025-03-26. NIST Computer Security Resource Center, Nov. 2024. URL: https://csrc.nist.gov/pubs/sp/800/232/ipd (cited on pages 83, 89).

[101] Taiwan Semiconductor Manufacturing Company Limited (TSMC). **40nm Technology**. Accessed: 2025-01-10. 2025. URL: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_40nm (cited on page 80).

[102] Linux Kernel Documentation Team. **Spectre Side Channels**. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html. Accessed: 2024-07-26. 2024. URL: https://docs.kernel.org/admin-guide/hw-vuln/spectre.html (cited on page 57).

[103] Paul Turner. **Retpoline: a software construct for preventing branch-target-injection**. 2018 (cited on page 53).

[104] Andrew Waterman and Krste Asanović. **RISC-V Unprivileged Specification**. Version 20240411. 2024. URL: https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?pli=1 (cited on pages 15, 16, 73).

[105] Wikipedia contributors. **AACS encryption key controversy**. https://en.wikipedia.org/wiki/AACS_encryption_key_controversy. Accessed: 2025-03-20. 2025 (cited on pages 24, 25).

Acronyms

ABI application binary interface. 16, **Glossary:** application binary interface
API application programming interface. 26
ASIC application-specific integrated circuit. **Glossary:** application-specific integrated circuit
ASID address space identifier. 38
ASLR address space layout randomization. 48

BHB branch history buffer. 57
BHI branch history injection. 50, 57, 64
BHT branch history table. 7, 34, 35, 39, 40
BTB branch target buffer. 7, 36, 38–40, 42, 47, 49, 76
BTI branch target injection. 38, 64

CFG control-flow graph. 8, 73, 78, 81, 83–85
CFI control-flow integrity. 79, 84, 85, 115
CMOS complementary metal-oxide-semiconductor. 30
CoT chain-of-trust. 23, 24
CPU central processing unit. 20, 21, 44
CSR control and status register. 16, 37, 68, 84

DCLS dual-core lockstep. 7, 80
DMA direct memory access. **Glossary:** direct memory access
DRAM dynamic random-access memory. 23, 25, 114
DVFS dynamic voltage and frequency scaling. 36

ECC error correcting code. 25
EM electromagnetic. 14, 20–22, 30

FF flip-flop. 41, **Glossary:** flip-flop
FPGA field programmable gate array. 113
FSM finite state machine. 40, 43, 51

GPIO general purpose input/output. **Glossary:** general purpose input/output
GPR general purpose register. 68

HDD hard disk drive. **Glossary:** hard disk drive

HDL hardware description language. 95, 96
HDR habilitation à diriger des recherches. 13, 115
HSM hardware security module. 23–26

IBC indirect branch control. 50
IBPB indirect branch prediction barrier. 49, 50, 57
IBRS indirect branch restricted speculation. 38, 49, 50, 57
ILP instruction-level parallelism. 53
IPC instructions per cycle. **Glossary:** instructions per cycle
ISA instruction set architecture. 13, 15–17, 23, 30, 33, 37, 63, 66, 69, 73, 75, 78, 84, 90, 92, 95, 96, 113–115, **Glossary:** instruction set architecture
ISR instruction-set randomization. 8, 66, 80–83, 85, 87–90, 92, 95, 96

LFB line fill buffer. 45
LFSR linear feedback shift register. 39
LLC last level cache. 36, 43, 51
LRU least recently used. 39
LSU load store unit. 45
LUT look-up table. 41, **Glossary:** look-up table

MDS microarchitectural data sampling. 44
MMU memory management unit. 21, 22, 66, 69, 75, 114, **Glossary:** memory management unit
MPU memory protection unit. 66
MTT memory tracking table. 69

OoO out-of-order. 45, 53, 58, 63, **Glossary:** out-of-order
OS operating system. 13, 23, 24, 39, 66, 69, 75, 77, 95, 114
OTA over-the-air. 66, **Glossary:** over-the-air

PC program counter. 17, 56, 72, **Glossary:** program counter
PHT pattern history table. 47, 60
PIN personal identification number. 22
PMP physical memory protection. 38, 69

RAII resource acquisition is initialization. **Glossary:** resource acquisition is initialization
RAM random-access memory. 16, 69, **Glossary:** random-access memory
RNG random number generator. 26
ROB reorder buffer. 8, 90, 91, **Glossary:** reorder buffer
RoT root-of-trust. 23, 24
RSB return stack buffer. 16, 36, 47, 53, 54, 57, 73, 113, 114

SB speculation barrier. 50
SE secure element. 23, 80
SLH speculative load hardening. 10, 54, 55, 57, 60
SMT simultaneous multithreading. 7, 36, 40, 41, 49, 113
SoC System-on-Chip. 21, 23–26, 43
SRAM static random-access memory. 80
SSA static single assignment. 60
SSBD speculative store bypass disable. 50
SSLH strong speculative load hardening. 55
STIBP single thread indirect branch predictors. 49, 50, 57
STL store-to-load forwarding. 47, 48

syscall system call. 57, **Glossary:** system call

TEE trusted execution environment. 23
TLB translation lookaside buffer. 36, 39, 51, **Glossary:** translation lookaside buffer

UART universal asynchronous receiver-transmitter. **Glossary:** universal asynchronous receiver-transmitter

VM virtual machine. 10, 85, 86
VMID virtual machine identifier. 38

WCET worst case execution time. **Glossary:** worst case execution time

Glossary

application binary interface The application binary interface is the convention on the role of registers (e.g. $x1$ is the return address), how to call functions (e.g. returned value is in register $a0$), ...It allows optimization (e.g. RSB) upon a more specific model of instruction sequences than the one dictated by the ISA.. 16

flip-flop Flip-flops, or flip-flop registers are elementary memory elements in a circuit.. 41

function A function is a block of code that performs a specific task and returns a value. Functions are used to compute and return a result, making them essential in both procedural and functional programming. They can take input parameters and produce an output based on those inputs. . 74, 113, 114

hart A *hart* is a *hardware thread*. Cores generally support a single hart, but some implementing SMT can have several.. 45, 69

instruction set architecture The instruction set architecture (ISA) is the interface between software and hardware. It includes the semantics of instructions supported by the processor, as well as the description of the architecture: number and size of registers, possible processor configurations, etc. What can be controlled by the instruction set pertains to the architecture. Hardware elements that cannot be controlled by the instruction set pertain to the microarchitecture.. 13, 15–17, 23, 30, 33, 37, 63, 66, 69, 73, 75, 78, 84, 90, 92, 95, 96, 113, 115

look-up table Look-up tables are the elementary logic components in an field programmable gate array (FPGA). Their count is a measure of the size of the combinational logic circuit.. 41

memory management unit The memory management unit (MMU) is the hardware component responsible for the translation of virtual addresses into physical addresses.. 21, 22, 66, 69, 75, 114

method A method is a function that is associated with an object or a class. Methods define the behavior of objects in object-oriented programming. They can operate on the data (attributes) of the object they belong to and can return values. Methods are invoked on instances of classes and can modify the state of the object. . 73

out-of-order An out-of-order execution core is a core that allows independent instruction to be executed out-of-order, without waiting unnecessarily for previous instructions.. 45, 53, 58, 63

over-the-air Over-the-air updates are modification of the running firmware in an embedded system after it has been deployed in its final location. Over-the-air means that the update is performed with radio signals, but it can be more generally understand as a communication with the vendor.. 66

procedure A procedure is a block of code that performs a specific task. It does not return a value and is used to execute a series of statements. Procedures are typically used in procedural programming to encapsulate reusable code. In assembly language, a procedure is a subroutine that can be called using instructions like `call` and may or may not return a value. The focus is on executing a sequence of instructions, and the term is often used interchangeably with "function" depending on whether a value is returned. . 10, 73, 114

program counter The program counter is the register containing the address of the next instruction to be executed. Depending on the instruction set architecture, this register is directly accessible (e.g., ARM) or not (e.g., RISC-V).. 17, 56, 72

random-access memory RAM memory is fast and volatile memory. This term is often used in place of DRAM which function is to store programs instructions and data.. 69

reorder buffer The reorder buffer is a data structure in the microarchitecture responsible for reordering instructions after execution. A reorder buffer entry is reserved when an instruction is decoded, thus preserving program order. Instruction commits are performed according to the reorder buffer, which reestablishes the program order for committed instructions.. 8, 90, 91

return stack A return stack is a data structure used to store the return addresses of subroutines (procedures or functions) in a program. When a subroutine is called, the address of the instruction following the call is pushed onto the return stack. Upon completion of the subroutine, the return address is popped from the stack, allowing the program to resume execution at the correct location. The return stack is essential for managing control flow in programs, particularly in languages that support recursion and nested subroutine calls. The return stack can be placed in main memory or in a dedicated hardware structure called a return stack buffer (RSB). . 77

system call A system call is a particular function that requires privileged access from the operating system. A system call allows code in user mode to interact with the OS, for specific operations.. 57

translation lookaside buffer The translation lookaside buffer (TLB) is a cache memory for the MMU dedicated to the translation of virtual addresses to physical addresses. In some systems, there is a hierarchy of TLBs.. 36, 39, 51

Summary

This habilitation à diriger des recherches (HDR) manuscript discusses the design of secure microarchitectures, specifically focusing on RISC-V cores. The discussion is organized around two central questions.

The first question addresses how to design a security-conscious applicative out-of-order processor in 2025. The challenges posed by covert channels and transient attacks are explored. Immediately applicable solutions, such as timing fences, domes, and speculation barriers, are proposed to enhance security. However, designing such a secure core without significant performance trade-offs or radical modifications to the entire design process (including software source code, compilers, instruction set architectures (ISAs), and microarchitecture) remains challenging.

Thus, the second question reimagines the design of RISC-V cores from the ground up, exploring radical changes and their potential to improve security. The focus is on microcontroller cores that must be resilient to physical attacks. Key considerations include: how can an ISA be designed to accommodate registers containing confidential data? Should forward indirect jumps be prohibited to enhance control-flow integrity (CFI)? Why might lockstep processors be vulnerable against upcoming fault injection techniques, and how can they be replaced with cores offering cryptographic guarantees of integrity?

This manuscript provides an opportunity to discuss the design choices necessary for secure microarchitectures, choices that are not always based solely on technical merits. Improving the security of modern computing systems is a complex process, and by investigating the security of today's microarchitectures, future-proof designs can be advocated for.

Résumé

Ce manuscrit d'habilitation à diriger des recherches (HDR) traite de la conception de microarchitectures sécurisées, en se concentrant spécifiquement sur les cœurs RISC-V. La discussion est organisée autour de deux questions centrales.

La première question aborde la manière de concevoir un processeur applicatif à exécution dans le désordre axé sur la sécurité en 2025. Les défis posés par les canaux cachés et les attaques microarchitecturales sont explorés. Des solutions immédiatement applicables, telles que les barrières temporelles, les dômes et les barrières de spéculation, sont proposées pour améliorer la sécurité. Cependant, concevoir un tel cœur sécurisé sans compromis significatifs sur les performances ou sans modifications radicales de l'ensemble du processus de conception (y compris le code source logiciel, les compilateurs, les jeux d'instructions et la microarchitecture) reste un défi.

Ainsi, la deuxième question réinvente la conception des cœurs RISC-V à partir de zéro, en explorant des changements radicaux et leur potentiel pour améliorer la sécurité. L'accent est mis sur les cœurs de microcontrôleurs qui doivent être résilients face aux attaques physiques. Les questions abordées incluent : comment concevoir un jeu d'instructions pour prendre en compte les registres contenant des données confidentielles ? Les sauts indirects vers l'avant doivent-ils être interdits pour améliorer l'intégrité du flot de contrôle ? Pourquoi les processeurs lockstep pourraient-ils être vulnérables face aux techniques d'injection de fautes modernes, et comment les remplacer par des cœurs offrant des garanties cryptographiques d'intégrité ?

Ce manuscrit offre l'occasion de discuter des choix de conception nécessaires pour des microarchitectures sécurisées, des choix qui ne sont pas toujours uniquement basés sur des mérites techniques. Améliorer la sécurité des systèmes informatiques modernes est un processus complexe, et en examinant la sécurité des microarchitectures actuelles, il est possible d'entrevoir des systèmes pérennes.