# Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs

Hans Winderix
*imec-DistriNet*
*KU Leuven*

Marton Bognar
*imec-DistriNet*
*KU Leuven*

Job Noorman
*imec-DistriNet*
*KU Leuven*

Lesly-Ann Daniel
*imec-DistriNet*
*KU Leuven*

Frank Piessens
*imec-DistriNet*
*KU Leuven*

***Abstract*—The control flow of a program can often be observed through side-channel attacks. Hence, when control flow depends on secrets, attackers can learn information about these secrets. Widely used software-based countermeasures ensure that attacker-observable aspects of the control flow do not depend on secrets, relying on techniques like *dummy execution* (for balancing code) or *conditional execution* (for linearizing code). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues. To counter these issues, this paper proposes lightweight hardware extensions for supporting these techniques in a principled way. We propose (1) a novel hardware mechanism (*mimic execution*), that executes an instruction stream only for its attacker-observable effects, and suppresses (most) architectural effects, and (2) ISA support (called AMi, for *Architectural Mimicry*) and programming models to effectively use mimic execution to balance or linearize code. We show the feasibility and benefits of our proposal by implementing mimic execution and AMi for a 32-bit out-of-order RISC-V core that leaks control flow in multiple ways (via e.g., the branch predictor, instruction timings, and the data cache). Our experimental evaluation shows that the hardware cost is low (most importantly, no impact on the processor's critical path), and that AMi enables significant performance improvements. In particular, AMi reduces the overhead of state-of-the-art linearized code by 60% in our benchmarks.***

## 1. Introduction

Control flow that depends on confidential information discloses (parts of) this information to an adversary that can observe the control flow via side channels. For instance, the outcome of a conditional branch might be inferred by measuring the execution time, one of many (micro)architectural timing measurements that expose control flow [1]. Countering this leakage in software typically relies on two classes of countermeasures. First, *code balancing* balances the two sides of a secret-dependent branch to equalize their observable behavior [2]–[6]. If the two different execution paths of a conditional branch exhibit the same observable behavior, an attacker can no longer distinguish them. Unfortunately, for most types of computing platforms this approach is insecure, but, when applicable, it can have performance benefits compared to other approaches [5], [6]. Second, *linearization* [7]–[10] ensures that control flow does not depend on program secrets at all.

Balancing and linearization are important ingredients in state-of-practice software-based countermeasures (such as *constant-time programming* [11]), as well as in recent research prototypes [5]–[7], [10], [12]. They are based on techniques like *dummy execution* (i.e., using architectural no-ops with an appropriate side-channel footprint) and *conditional execution* (e.g., conditional moves). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues.

**Our proposal.** In contrast to the above, this paper investigates how to offer hardware support and a small ISA extension to support control-flow balancing and linearization in a *principled* way instead. We propose a novel hardware mechanism, called *mimic execution*. Mimic execution can be thought of as a *mode* in which the processor executes instructions only for their attacker-observable effects, and suppresses (most) architectural effects: every instruction becomes a no-op, but a side-channel attacker cannot see the difference with a normal execution of the instruction.

Mimic execution is a powerful primitive, but using it correctly in software to obtain secure and correct code is non-trivial. We design and formally specify suitable ISA support (called AMi, for *Architectural Mimicry*) to activate and deactivate mimic execution, and we show how to use it to develop efficient and portable side-channel resistant code. We provide two implementations; for a pipelined in-order, and an out-of-order 32-bit RISC-V processor. We show that AMi enables significant performance improvements for hardened code, while only incurring low hardware costs.

**Contributions.** In summary, we contribute the following:

- *Mimic execution*, a novel and lightweight hardware primitive for imitating computational behavior.
- *Architectural Mimicry (AMi)*, a set of innovative instructions to control mimic execution in an efficient and portable way.
- Programming models showing how to balance and linearize control flow correctly and securely with AMi.
- A simple formal ISA model of AMi and a formal characterization of AMi programming models.

- An implementation of AMi for RISC-V.
- An experimental evaluation showing that the hardware cost is low, and that AMi enables significant performance improvements for hardened code.

We evaluate the benefits of AMi when *manually* writing hardened code, in line with the state-of-practice for writing constant-time cryptographic code. But we see very interesting avenues for future work to build compilers or binary rewriters that automatically (and provably) harden code against side channels by relying on AMi. To support and enable such future work, and to improve reproducibility of our results, our RISC-V implementation of AMi, as well as the full set of benchmarks and experiments are open sourced at https://gitlab.com/hanswinderix/ami.

## 2. Problem Statement

Prior work addressing the problem of control-flow leakage via software-based side channels can broadly be categorized into two classes with the common goal that the trace of observable side effects produced by a program's execution does not depend on secrets. The first approach is based on the insight that if the code is carefully balanced in such a way that all possible targets of a single control-flow transfer induce exactly the same observable behavior, then executing the code does not reveal via side-effect observations which target has been executed [2]–[6]. Unfortunately, this *balanced form* does not prevent control-flow leakage in general as it is not possible on *all* platforms to balance out *all* side effects of a control-flow transfer. For instance, to predict the most likely target of a control flow transfer, modern CPUs are equipped with a branch predictor unit, which maintains a history of recent transfers. The predictor state encodes in a direct manner which target has been selected, and consequently, balanced control flow cannot prevent this shared microarchitectural state from being exposed. For this reason, the second approach avoids secret-dependent control flow altogether and linearizes the control flow using different techniques [7]–[10], [12]–[14]. Control flow in *linearized form* always executes the instructions from all possible targets in a fixed order, but makes sure that architectural state is only modified by the instructions whose associated path condition holds. The linearized form has been adopted by both the security and the architecture community as the de facto standard to prevent applications from leaking confidential data via the control flow. Avoiding secret-dependent branches is a key principle of the constant-time programming discipline [11], which is broadly adhered to for writing security-critical code.

**Performance.** Both the balanced and the linearized form have in common that they rely on clever software tricks to achieve some form of dummy execution. The balanced form relies on the availability of dummy instructions (i.e., no-ops) to compensate for side-effects induced by instructions in alternate execution paths. The linearized form relies on the ability to neutralize the architectural effects of instructions that should not be executed according to program semantics. Implementing these forms of dummy execution incurs a significant performance overhead due to the use of extra instructions and additional registers.

**Security.** More than 25 years after Kocher introduced the concept of timing attacks [15], it is well understood how to systematically harden applications to prevent control flow from exposing secrets: *the timing behavior and the hardware resource utilization due to a control-flow transfer must not depend on confidential data*. Unfortunately, despite this fundamental understanding, vulnerabilities of this kind are being found on a regular basis, even in high-profile code [16]–[18]. This is partially due to the common practice of hardening applications at the level of the source code [19], which is typically written in a high-level programming language. This enables so-called cross-layer vulnerabilities [20], [21], when lower layers such as the compiler or the underlying hardware are not made aware of the security semantics of the application.

**Portability.** It is determined by the underlying hardware implementation what observable side effects are exposed. Since application hardening is typically done at high abstraction levels [19], a comprehensive defense is needed that is effective for all target platforms, ranging from low-cost microcontrollers to high-end servers. Current practice adopts a worst-case adversary model and assumes that the control flow leaks in all situations and on all hardware. This is a secure assumption, but overly conservative. More importantly, it tightly couples the security policy to the source code and leaves no room to adopt more relaxed policies on simpler architectures that leak less information, which could have performance benefits. Furthermore, decoupling the security policy from the source code also improves other software qualities such as readability and maintainability.

## 3. Scope

We argue that there are critical concerns with current countermeasures against control-flow leakage that rely on primitives not designed to solve this problem. We propose a novel approach and, additionally, we show that significant performance improvements for constant-time code are possible (with stronger and future-proof security guarantees).

The purpose of this work is to furnish developers with a new, efficient enforcement mechanism to prevent the disclosure of sensitive information through a program's *control flow*. A developer or a secure compiler [22], [23], such as CompCert [24], FaCT [25], or Jasmin [26], can leverage this mechanism to efficiently *balance* or *linearize* secret-dependent control flow in a principled way (i.e., based on a hardware-software contract for security). Our work can be seen as complementary to OISA [27], which offers a security contract in the ISA expressing information leakage through unsafe instructions, but it does not directly address the control-flow leakage problem.

It is *not* the purpose of this work to propose new techniques that deal with other sources of information leakage

than a program's control flow. To close these other microarchitectural leaks, a developer should use countermeasures described in the rich side-channel attack literature [1], [28]. For instance, to counter cache-based side-channel attacks, a developer can avoid secret-dependent memory accesses (as dictated by the constant-time programming discipline) or rely on Oblivious RAM [29], [30] (and thus hide memory access patterns). Similarly, to mitigate transient execution attacks, a developer can leverage software-based solutions such as fences [31], speculative load hardening [31], [32], and retpolines [33] or hardware-based mechanisms like ProSpeCT [34] and SPT [35]. Furthermore, in this paper we focus on *manually* hardening ISA-level code. How to automatically balance or linearize vulnerable source code is the subject of orthogonal research, which has become practical due to a large body of work [2]–[5], [7]–[10], [12]. Extending these well-documented techniques to automatically harden assembly code that leverages our mechanism, or to support it in a compiler (benefiting from features such as taint analysis) is left for future work (cf. Section 7).

## 4. Assumptions and Security Objectives

**System model.** Our goal is to develop an extension for widely used ISAs, such as the RISC-V RV32IM ISA used by our implementation (cf. Section 6). In our formalization, however, we use a simplified ISA called AMiL. Base AMiL (i.e., without the AMi extensions) is defined in Fig. 1. We assume a set of registers Regs, a set of values $\mathcal{V}$ (including memory addresses), and a set of program locations $Loc \subseteq \mathbb{N}$. We let $Inst$ be the set of instructions. A program $P : Loc \to Inst$ is a mapping from locations to instructions and $P[\ell]$ denotes the instruction at location $\ell$.

(Expr) $e := v \mid x$
(Inst) $i :=$ **add** $x$,$e_1$,$e_2$ | **mul** $x$,$e_1$,$e_2$ | **beqz** $e$,$\ell$ |
**call** $\ell$ | **jmp** $e$ | **load** $x$,$e$ | **store** $e_1$,$e_2$

Figure 1: Syntax of base instructions where $x$ ranges over Regs, $v$ ranges over $\mathcal{V}$ and $\ell$ ranges over $Loc$.

An *architectural configuration* is a tuple $\langle m, r, \mathtt{pc} \rangle \in \mathcal{A}$ where $m : \mathcal{V} \to \mathcal{V}$ is a memory, which maps addresses to values; $r : \text{Regs} \to \mathcal{V}$ is a register file, which maps registers to values; and pc is the program counter, a special register pointing to the next instruction to execute. The semantics of base AMiL can be defined straightforwardly as a transition system over configurations.

**Attacker model.** We consider software that manipulates secrets such as cryptographic keys and that aims to protect these secrets against attackers who can observe microarchitectural timing side-channels [1], revealing access to shared resources such as the instruction cache, data cache, branch predictor and TLB. Physical side-channel attacks [36], and other software-based side-channel attacks, such as fault attacks [37] and power attacks [38], are out of scope for this paper and subject of orthogonal mitigations.

**Leakage model.** We model the observational power of an attacker by defining a *leakage model*, which we integrate in the AMiL semantics. The semantics of base instructions is given by the relation $a \xrightarrow[inst]{o} a'$. It denotes the evaluation of a base instruction $inst$ in an architectural configuration $a$ resulting in configuration $a'$. Additionally, it produces an observation $o \in \mathcal{O}$ defining the architectural information that leaks through microarchitectural side channels (which we abstract from) during the evaluation of the instruction. This is similar to existing work [22]. We parameterize the semantics by a set of leakage functions $\lambda_{inst} : \mathcal{A} \to \mathcal{O}$, which define for each instruction $inst$ what parts of the architectural configuration leak. The observation trace of an $n$-step execution, written $a \xrightarrow[inst]{o} {}^n a'$, is the concatenation of observations produced by individual execution steps.

In this paper, we consider two countermeasures to prevent control-flow leakage: control-flow balancing and linearization. To study these two techniques, we reduce the leakage space to two leakage models by defining two versions of the leakage functions in Fig. 2. For both leakage models, the $\lambda_{add}$ and $\lambda_{mul}$ leakage functions return a fixed (i.e., configuration-independent) observation, such as the instruction latency. The functions $\lambda_{load}$ and $\lambda_{store}$ model the exposure of the accessed memory address (e.g., through the data cache) when executing a `load` and a `store` instruction. Finally, $\lambda_{call}$, $\lambda_{jmp}$ and $\lambda_{beqz}$ model the observations produced by `call`, `jmp` and `beqz` instructions, which are instantiated differently for the two leakage models:

- In the first leakage model, it is possible to avoid exposure of the program counter. An attacker can only infer the value of the program counter when the targets of a control-flow transfer produce different observations. In this model, it is secure to balance secret-dependent branches, i.e., to make sure that the different execution paths produce the same observation trace and thus remain indistinguishable by an attacker. Hence, a developer can choose between balancing and linearizing based on a profitability analysis. This model represents the leakage of low-end microcontrollers, typically not equipped with performance-enhancing hardware.
- In the second leakage model, the program counter is inevitably exposed to an attacker. In this model, it is not secure to balance secret-dependent branches. Branch elimination (by linearizing the branch) is the only secure hardening option. This model corresponds to the constant-time leakage model [11], commonly employed in security analyses. It represents the leakage of high-end processors that typically feature performance-enhancing hardware such as a branch predictor and an instruction cache.

**Security objectives.** The developer identifies what parts of the program state should remain secret, and the security objective of the developer is to avoid that these secrets leak to the attacker. We model this in the classic way, using a lattice with two security levels: public (low) and secret (high). A security policy $\mathcal{P}$ is a mapping from registers

**Common leakage**

$$\lambda_{add}(\langle m, r, \mathtt{pc}\rangle) = add \quad \text{if } P[\mathtt{pc}] = \mathtt{add\ x, e1, e2}$$
$$\lambda_{mul}(\langle m, r, \mathtt{pc}\rangle) = mul \quad \text{if } P[\mathtt{pc}] = \mathtt{mul\ x, e1, e2}$$
$$\lambda_{load}(\langle m, r, \mathtt{pc}\rangle) = load\ a \quad \text{if } P[\mathtt{pc}] = \mathtt{load\ x, e} \text{ and } a = [\![\mathrm{e}]\!]_r$$
$$\lambda_{store}(\langle m, r, \mathtt{pc}\rangle) = store\ a \quad \text{if } P[\mathtt{pc}] = \mathtt{store\ e1, e2} \text{ and } a = [\![\mathrm{e2}]\!]_r$$

**Leakage model 1** (Control flow exposure can be avoided)

$$\lambda_{call}(\langle m, r, \mathtt{pc}\rangle) = call \quad \text{if } P[\mathtt{pc}] = \mathtt{call}\ \ell$$
$$\lambda_{jmp}(\langle m, r, \mathtt{pc}\rangle) = jmp \quad \text{if } P[\mathtt{pc}] = \mathtt{jmp}\ e$$
$$\lambda_{beqz}(\langle m, r, \mathtt{pc}\rangle) = br \quad \text{if } P[\mathtt{pc}] = \mathtt{beqz\ e}, \ell$$

**Leakage model 2** (Control flow is inevitably exposed)

$$\lambda_{call}(\langle m, r, \mathtt{pc}\rangle) = call\ \ell \quad \text{if } P[\mathtt{pc}] = \mathtt{call}\ \ell$$
$$\lambda_{jmp}(\langle m, r, \mathtt{pc}\rangle) = jmp\ \ell \quad \text{if } P[\mathtt{pc}] = \mathtt{jmp}\ e \text{ and } \ell = [\![\mathrm{e}]\!]_r$$
$$\lambda_{beqz}(\langle m, r, \mathtt{pc}\rangle) = br\ \ell' \quad \text{if } P[\mathtt{pc}] = \mathtt{beqz\ e}, \ell \text{ and } \ell' = \begin{cases} \ell & [\![\mathrm{e}]\!]_r = 0 \\ \mathtt{pc}+1 & [\![\mathrm{e}]\!]_r \neq 0 \end{cases}$$

Figure 2: Leakage functions for AMiL where *add*, *mul*, *load*, *store*, *call*, *jmp*, *br* are (fixed) observations. Notice that in leakage model 1 the locations $\ell$ are absent (i.e., only the fixed cost leaks). The expression evaluation function $[\![e]\!]_r$ evaluates expression $e$ using register file $r$.

and memory locations to security levels, identifying them as secret or public. Two configurations $\sigma$ and $\sigma'$ are low-equivalent with respect to a policy $\mathcal{P}$, written $\sigma{=}_{\mathcal{P}}\sigma'$, if they agree on the public part of their register file and memory as defined by $\mathcal{P}$. A program is *secure* if two executions starting from low-equivalent initial configurations produce the same observation trace.

**Definition 1** (Secure program)**.** *A program $P$ is secure w.r.t. a security policy $\mathcal{P}$ if for all initial configurations $\sigma_0$ and $\sigma_0'$, and for all $n$ such that $\sigma_0 =_{\mathcal{P}} \sigma_0'$, $\sigma_0 \xrightarrow{o}{}^{n} \sigma_n$, and $\sigma_0' \xrightarrow{o'}{}^{n} \sigma_n'$, then we have $o = o'$.*

## 5. Architectural Mimicry

We now present Architectural Mimicry (AMi). Recall that on the hardware side, we propose a new primitive, called *mimic execution*. Mimic execution imitates instructions in terms of their timing and microarchitectural behavior, but suppresses (most of) their architectural effects. It is left to the hardware designer how to mimic an instruction since this heavily depends on the implementation.

To control mimic execution in software, AMi extends the base ISA from Section 4 with qualifiers $q \in \mathcal{Q} = \{\mathtt{s}, \mathtt{m}, \mathtt{a}, \mathtt{g}, \mathtt{p}\}$ that can be associated with base instructions, denoted $q.i$. At the assembly level, each instruction has an instruction-dependent default qualifier (discussed in detail later), which can be omitted. For instance, the `add` instruction has the *standard* qualifier (**s**) by default. In machine code, the qualifier is always present. Additionally, we propose a number of programming models that rely on AMi to balance and linearize control-flow in a correct and secure way.

First, we informally introduce the basics of AMi and the programming models by example in Section 5.1. Then we discuss more advanced features, such as how to handle intricate control flow, in Section 5.2. We end this section with a formalization of AMi in Section 5.3 and 5.4.

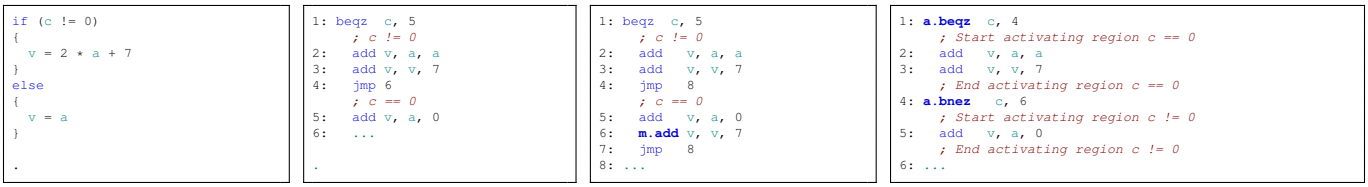
### 5.1. Basic AMi Features

**Balancing branches.** We assume leakage model 1, where we can securely balance secret-dependent control flow. Consider the insecure program in Listing 1b. This produces an observation trace that depends on the secret condition. When the branch is not taken (lines 2-4), the observation trace is $[br \cdot add \cdot add \cdot jmp]$. When the branch is taken (line 5), the observation trace is $[br \cdot add]$. As a consequence, an attacker is able to infer the outcome of the secret-dependent branch.

A solution to harden this program is to insert instructions to *balance* the two sides of the branch so that they produce the same observation trace. AMi provides hardware support to do so using *mimic instructions*, which are prefixed with the mimic qualifier **m**. A mimic instruction $\mathtt{m}.inst$ produces the same observations as the instruction $inst$ but does not update the architectural state. Therefore, mimic instructions can be used to securely balance branches (with hardware guarantees), instead of relying on ad-hoc techniques using existing instructions.

The program in Listing 1c illustrates how AMi can be used to build a (secure) balanced version of the code in Listing 1b. First, notice that instructions on lines 2 and 5 are already balanced as they produce the same observation. Second, the `add` instruction on line 3 is balanced with a mimic `add` on line 6: it produces the observation *add* but does not change the value of `v`. Finally, the `jmp` instruction on line 4 is also balanced with a jump on line 7. In this version, both sides of the branch produce the same observation trace $[br \cdot add \cdot add \cdot jmp]$, while the functional behavior of the program is equivalent to the one in Listing 1b.

**Linearizing branches.** We now assume leakage model 2. Under this leakage model, the balanced program in Listing 1c is insecure because the conditional jump on line 1 leaks its target, resulting in an observation trace starting with $[br\ 2]$ if $\mathtt{c} \neq 0$, and $[br\ 5]$ otherwise. This way, an attacker can gain information on the secret `c`. *Linearizing* the secret-dependent region by eliminating the branch on line 1 makes the program secure. AMi provides hardware support for linearization through *activating branches*. An activating branch is a branch instruction prefixed with the activating qualifier **a**. An activating branch always falls through to the next instruction, but if the branch condition evaluates to true (i.e., the branch should be taken), the processor enables *mimicry mode* for the duration of the branch (i.e., until the branch target is reached). When in




(a) C code (b) Vulnerable code (c) Balanced form (d) Linearized form

Listing 1: Balancing and linearizing a secret-dependent region with AMi instructions.

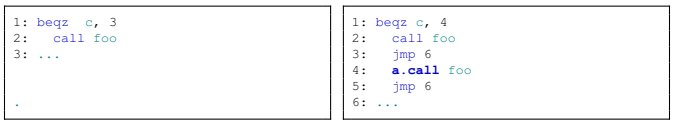
mimicry mode, the CPU mimics standard instructions. It is important to understand that the activating branch is not a branch per se but an instruction to efficiently linearize secret-dependent control flow, which, just like any other linearization technique, turns (insecure) control dependencies into (secure) data dependencies. Importantly, the activating branch instruction does not introduce extra sources of overhead compared to other linearization techniques. In particular, because activating branches deterministically fall through to the next instruction, there is no uncertainty about what instructions to fetch after an activating branch and the CPU can fetch and issue subsequent instructions without any delay (the code is effectively linear). Hence, it is not necessary for security reasons to delay the fetch (and stall the pipeline) until the outcome of the activating branch condition is known.

The program in Listing 1d illustrates how to eliminate a branch leveraging AMi. If $c = 0$, mimicry mode is enabled on line 1, the instructions on line 2 and 3 are mimicked, and line 5 is executed normally. If $c \neq 0$, lines 2 and 3 are executed normally, but the activating branch on line 4 activates mimicry mode and line 5 is mimicked. In both cases, the observation trace produced by the execution of the program is $[br\ 2 \cdot add \cdot add \cdot br\ 5 \cdot add]$, thus no secret information is leaked.

## 5.2. Advanced AMi Features

**Mimic functions.** It is sometimes necessary to mimic the execution of an entire function. For instance, consider the program in Listing 2a where function `foo` is called only when $c \neq 0$. Balancing this branch requires mimicking the execution of `foo`. While this could be done by duplicating the function and prefixing all its instructions with **`m`**, AMi actually provides a more efficient way to do this using *activating calls* (**`a.call`**), as shown in Listing 2b. The activating call on line 4 activates mimicry mode for the duration of the call. If the callee satisfies conditions that we formalize in Section 5.4, the **`a.call`** produces the same observation trace as the regular call on line 2, without affecting the architectural state. Therefore, the two sides of the branch in Listing 2b are balanced.

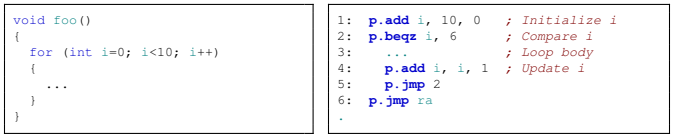
**Persistent instructions.** The same code can sometimes be executed in standard mode, and sometimes in mimicry mode. For instance, the same function can be invoked




(a) Vulnerable code (b) Balanced form

Listing 2: Example of an activating call.

through a standard call or through an activating call. Similarly, the outcome of an activating branch determines if the instructions in the branch shadow are executed or mimicked. Mimicry mode is designed with security and correctness in mind. To be *secure*, the observation trace of such code must be independent of the processor mode. To be *correct*, such code cannot have an effect on the program result in mimicry mode (i.e., it must effectively behave as a no-op). Hence, (most) architectural effects must be suppressed in mimicry mode. However, some architectural effects may not be suppressed. For example, architectural updates that influence the control flow within an activating region may not be ignored. Otherwise the control flow would depend on the processor mode, which would be insecure. Likewise, instructions that compute the address of a later memory access may not be suppressed. More generally, *if a value can leak to the microarchitectural state, any architectural update that this value depends on should always be executed, regardless of the processor mode*. AMi deals with this architectural/microarchitectural entanglement in two ways.




(a) Function with loop (b) Persistent loop using AMi

Listing 3: Example of an activating function with a loop.

First, an instruction can be made *persistent* by associating it with the **`p`** qualifier. Persistent instructions are always executed, even in mimicry mode. For instance, consider an activating call to function `foo` in Listing 3a. This function contains a loop. To be secure, the loop trip count must be independent of the processor mode. To be correct (and secure), the loop induction variable must be updated to avoid mimic execution getting stuck in an infinite loop. Listing 3b

illustrates how to lower function `foo` with persistent instructions. The instructions that operate on the loop induction variable `i` are associated with the **`p`** qualifier, as is the instruction that jumps to the return address on line 6. (In fact, `jmp` and `beqz` have **`p`** as default qualifier, so it would be fine to omit it.) As another example, consider the activating branch in Listing 4a. To be secure, the `load` address at line 3 must be independent of the processor mode. Because `a` is assigned a value by a standard instruction at line 2, the program leaks `0x200` in standard mode and `0x100` in mimicry mode (loads always fetch from memory, regardless of the mode), making the code insecure. Listing 4b illustrates how to make use of persistent instructions to make this region secure. Maintaining the stack pointer is another use case for persistent instructions. Indeed, the stack pointer is typically used to access memory and therefore it will be attacker observable. Consequently, the value of the stack pointer should be independent of the processor mode.

```
   ; a = 0x100
1: a.beqz c, 4
2:   add  a, 0x200, 0
3:   load b, a ; leak 0x100 or 0x200
4: ; a is not live here
```

(a) Insecure activating branch.

```
   ; a = 0x100
1: a.beqz c, 4
2:   p.add a, 0x200, 0
3:   load  b, a  ; leak 0x200
4: ; a is not live here
```

(b) Secure activating branch.

Listing 4: Example of a load where `x` and `y` are public.

Second, an ISA designer can decide *not* to suppress some of the architectural effects in mimicry mode. For instance, it is reasonable for a call instruction to always store the return address, so that the control flow always returns to the correct program location after a function invocation. As for AMiL, to ensure that control-flow instructions always have the desired architectural effects, a `jmp` can *only* be associated with the **`p`** qualifier and a `beqz` and `call` can only be associated with the **`a`** and **`p`** qualifiers.

**Nested and recursive activators.** Recall that activating instructions (e.g., **`a.`**`call` or **`a.`**`beqz`) (un)conditionally activate mimicry mode. Such activations can be *nested* or *recursive*. To correctly maintain the mimicry context, AMi introduces three registers. First, the processor keeps track of the depth of recursive activations in the *activation counter* register (`AC`). The processor is in standard mode if $\texttt{AC} = 0$, otherwise the processor is in mimicry mode. Second, the *mimicry entry address* register (`En`) holds the address of the *entry instruction* (i.e., the instruction responsible for the current activation). Third, the *mimicry exit address* register (`Ex`) holds the address where mimicry mode must be disabled. A pair $[\texttt{En}, \texttt{Ex}]$ corresponds to what we call an *activating region*. For an activating call **`a.`**`call` $\ell$, the corresponding activating region is $[\texttt{pc}, \texttt{pc} + 1]$ and for a branch **`a.`**`beqz` e, $\ell$ it is $[\texttt{pc}, \ell]$. Recursive activations are illustrated in Listing 5a, with an activating branch inside a recursive function. When the processor first executes the activating instruction on line 3, it sets `AC` to 1, `En` to 3 and `Ex` to 6. For each recursive call, `AC` is incremented on line 3, and decremented at the start of line 6.

In case of nested activations, as illustrated in Listing 5b, the inner activating branch does not update `AC`, `En`, or `Ex`, if the outer one activates mimicry mode.

In Section 5.4 we state conditions how to write secure activating regions and prove that, if these conditions hold, 1) mimicry mode is not deactivated prematurely and 2) `AC` is restored at the end of the activating region.

```
1: p.beqz n, 7
2:   p.add n, n, -1
3:   a.beqz c, 6
4:     ...
5:     p.call foo
6:     ...
7: jmp ra
```

(a) Recursive activation

```
1: a.beqz c1, 6
2:  ...
3:  a.beqz c2, 5
4:    ...
5:  ...
6: ...
.
```

(b) Nested activating branches

Listing 5: Example of recursive and nested activations.

**Non-mimicable instructions.** Some instructions cannot be mimicked. For instance, unless the memory subsystem is mimicry-aware (something we discuss in Section 7) it is not possible to mimic store instructions. Table 1 classifies AMiL instructions into mimicable, activating, and always-persistent instructions. The default qualifier for each instruction class is listed first in the list of qualifiers (column 3). Always-persistent instructions can be a problem when present in activating regions. To illustrate this, the program in Listing 6a contains an always-persistent store in the branch shadow of the activating branch. In mimicry mode, the memory will be modified, violating program correctness since mimic execution should not affect the program result.

TABLE 1: AMiL classes of qualified instructions.

| Class | Instructions | Qualifiers |
|---|---|---|
| Mimicable | `add`, `mul`, `load` | **`s`**, **`g`**, **`m`**, **`p`** |
| Activating | `call`, `beqz` | **`p`**, **`a`** |
| Always persistent | `jmp`, `store` | **`p`** |

Disallowing non-mimicable instructions in activating regions is an acceptable approach for some non-mimicable instructions (e.g., some system calls), but it is too restrictive for other instructions, such as stores. AMi introduces *ghost instructions* (prefixed with the ghost qualifier **`g`**) to deal with this situation. Ghost instructions are only executed when the processor is in mimicry mode. They are mimicked in standard mode. In this sense, they are the dual of standard instructions. For instance, a ghost load can be used to nullify an always-persistent store, as illustrated in Listing 6b. When the processor is in mimicry mode, the value at address `v` is loaded in the register `x` on line 3, before being stored again on line 4, which is functionally equivalent to a no-op, Conversely, when the processor is in standard mode, `x` is set to 42 on line 2, the `load` on line 3 is mimicked (but necessary actions such as touching cache lines and producing events on the memory bus still happen), and the `store` on line 4 writes 42 at address `v`.

```
1: a.beqz c, 4
2:   add     x, 42, 0
3:   p.store x, v
4: ...
.
```

(a) Store in mimicry mode

```
1: a.beqz c, 5
2:   add     x, 42, 0
3:   g.load  x, v
4:   p.store x, v
5: ...
```

(b) Ghost load nullifying the store

Listing 6: Example using a ghost instruction.

**Exceptions.** We distinguish between terminating and non-terminating exceptions. Terminating exceptions indicate a program error, such as a memory protection violation, and cause program termination. They should not occur in correct programs. Non-terminating exceptions can be handled and normally resumed, such as a page fault. To be secure, non-terminating exceptions must be accepted and processed independent of the processor mode. To be correct and secure, terminating exceptions must be ignored in mimicry mode.

The exception handler must use persistent instructions and save AC, En, and Ex before clearing AC to disable mimicry mode. Once the AC is cleared, the exception handler can use standard instructions, and even activate mimicry mode. Finally, the exception handler must restore the values of AC, En, and Ex before resuming execution of the interrupted task.

## 5.3. Formal Semantics

Now that all AMi features have been introduced informally, we can define the semantics of the AMiL ISA formally. An *AMi configuration* $\sigma$ is a tuple $\langle m, r, pc, \mathtt{AC}, \mathtt{En}, \mathtt{Ex}\rangle$ where $\langle m, r, \mathtt{pc}\rangle$ is an architectural configuration (cf. Section 4), AC is the activation counter, En is the mimicry entry address, and Ex is the mimicry exit address. In the following, we refer to AMi configurations simply as *configurations*. The semantics of *AMi*, given by the relation $\sigma \overset{o}{\Longrightarrow} \sigma'$, defines how qualified instructions are evaluated in a configuration $\sigma$. More precisely, it defines when instructions are executed or mimicked, and how the activation counter is updated.

**Non-activating instructions.** The semantics of non-activating instructions (i.e., qualified instructions with $q \in \{\mathtt{s}, \mathtt{m}, \mathtt{p}, \mathtt{g}\}$) is given in Fig. 3 (the *decrAC* function can be ignored for now). The EXECUTE rule *executes* the corresponding base instruction ($\langle m, r, \mathtt{pc}\rangle \xrightarrow[inst]{o} \langle m', r', \mathtt{pc}'\rangle$), produces an observation $o$, and updates the AMi configuration with the new architectural configuration $\langle m', r', \mathtt{pc}'\rangle$. The MIMIC rule *mimics* the execution of the base instruction: it executes the instruction but does not commit architectural changes to the AMi configuration ($\langle m, r, \mathtt{pc}\rangle \xrightarrow[inst]{o} \cdot$). However, it produces the corresponding observation $o$, and increments the program counter. Which rule is applied depends on the value of the qualifier $q$ and the activation counter AC. The EXECUTE rule applies when the instruction is persistent (p), the instruction is standard (s) and the processor is in standard mode, or the instruction is ghost (g) and the processor is in mimicry mode. Conversely, the MIMIC rule applies when the instruction is mimic (m), the instruction is standard (s) and the processor is in mimicry mode, or the instruction is ghost (g) and the processor is in standard mode. Keep in mind when reading the semantics that not all qualifiers are allowed for all instructions (Table 1). For instance, jmp instructions are *always* persistent and hence always handled by the EXECUTE rule.

$$\textsc{execute}\quad \frac{P[\mathtt{pc}] = q\,.\,inst \qquad \mathtt{AC}' = \mathit{decrAC}(\mathtt{AC}, \mathtt{Ex}, \mathtt{pc}) \qquad \mathit{execute}(q, \mathtt{AC}') \qquad \langle m, r, \mathtt{pc}\rangle \xrightarrow[inst]{o} \langle m', r', \mathtt{pc}'\rangle}{\langle m, r, \mathtt{pc}, \mathtt{AC}, \mathtt{En}, \mathtt{Ex}\rangle \overset{o}{\Longrightarrow} \langle m', r', \mathtt{pc}', \mathtt{AC}', \mathtt{En}, \mathtt{Ex}\rangle}$$

$$\textsc{mimic}\quad \frac{P[\mathtt{pc}] = q\,.\,inst \qquad \mathtt{AC}' = \mathit{decrAC}(\mathtt{AC}, \mathtt{Ex}, \mathtt{pc}) \qquad \mathit{mimic}(q, \mathtt{AC}') \qquad \langle m, r, \mathtt{pc}\rangle \xrightarrow[inst]{o} \cdot \qquad \mathtt{pc}' \triangleq \mathtt{pc} + 1}{\langle m, r, \mathtt{pc}, \mathtt{AC}, \mathtt{En}, \mathtt{Ex}\rangle \overset{o}{\Longrightarrow} \langle m, r, \mathtt{pc}', \mathtt{AC}', \mathtt{En}, \mathtt{Ex}\rangle}$$

where

$$\mathit{execute}(q, \mathtt{AC}) \triangleq q = \mathtt{p} \vee (q = \mathtt{s} \wedge \mathtt{AC} = 0) \vee (q = \mathtt{g} \wedge \mathtt{AC} > 0)$$

$$\mathit{mimic}(q, \mathtt{AC}) \triangleq q = \mathtt{m} \vee (q = \mathtt{s} \wedge \mathtt{AC} > 0) \vee (q = \mathtt{g} \wedge \mathtt{AC} = 0)$$

Figure 3: Evaluation rules for non-activating instructions.

**Activating instructions.** The semantics of activating instructions (i.e., in our small language a.call and a.beqz) is given in Fig. 4. The rules use the functions *incrAC* and *decrAC* respectively to activate and deactivate mimicry mode. The function *decrAC*(AC, Ex, pc), used during the evaluation of every instruction, decrements the activation counter AC whenever the program counter reaches the exit address Ex. The function *incrAC*(AC, En, Ex, pc, Ex′, $c$) is only used during the evaluation of activating instructions. This function takes as parameters the previous activation configuration AC, En, Ex, the address of the current activating instruction pc, the corresponding exit address Ex′, and a condition $c$ to conditionally activate mimicry mode. If mimicry mode is not enabled and $c$ is true, the function enables mimicry mode and returns the new activation counter, together with the corresponding entry and exit addresses (i.e., pc and Ex′). If mimicry mode is already enabled and in case of recursive activation (pc = En), the activation counter is incremented. In other cases, *incrAC* simply returns the old configuration.

The rule ACTIVATING-CALL unconditionally activates mimicry mode for the duration of the call, updates the return address, and jumps to the call target. The rule ACTIVATING-BRANCH activates mimicry mode for the duration of the branch, only if the branch condition is true; and increments the program counter, regardless of the value of the condition. Consequently, if the branch condition is true, the following

code is mimicked, but if the condition is false, the following code is actually executed.

$$\textsc{activating-call}\quad \frac{\begin{array}{c} P[\mathtt{pc}] = \mathtt{a.call}\ \ell \qquad \mathtt{AC}' = \mathit{decrAC}(\mathtt{AC}, \mathtt{Ex}, \mathtt{pc}) \\ \mathtt{AC}'', \mathtt{En}', \mathtt{Ex} = \mathit{incrAC}(\mathtt{AC}', \mathtt{En}, \mathtt{Ex}, \mathtt{pc}, \mathtt{pc}+1, \mathit{true}) \\ r' = r[\mathtt{ra} \mapsto \mathtt{pc}+1] \qquad \mathtt{pc}' = \ell \end{array}}{\langle m, r, \mathtt{pc}, \mathtt{AC}, \mathtt{En}, \mathtt{Ex}\rangle \overset{\varepsilon}{\Longrightarrow} \langle m, r', \mathtt{pc}', \mathtt{AC}'', \mathtt{En}', \mathtt{Ex}'\rangle}$$

$$\textsc{activating-branch}\quad \frac{\begin{array}{c} P[\mathtt{pc}] = \mathtt{a.beqz}\ \mathtt{e}, \ell \qquad \mathtt{AC}' = \mathit{decrAC}(\mathtt{AC}, \mathtt{Ex}, \mathtt{pc}) \\ c = (\llbracket \mathtt{e} \rrbracket_r = 0) \qquad \mathtt{pc}' = \mathtt{pc}+1 \\ \mathtt{AC}'', \mathtt{En}', \mathtt{Ex} = \mathit{incrAC}(\mathtt{AC}', \mathtt{En}, \mathtt{Ex}, \mathtt{pc}, \ell, c) \end{array}}{\langle m, r, \mathtt{pc}, \mathtt{AC}, \mathtt{En}, \mathtt{Ex}\rangle \overset{\varepsilon}{\Longrightarrow} \langle m, r, \mathtt{pc}', \mathtt{AC}'', \mathtt{En}', \mathtt{Ex}'\rangle}$$

where

$$\mathit{decrAC}(\mathtt{AC}, \mathtt{Ex}, \mathtt{pc}) \triangleq \begin{cases} \mathtt{AC}-1 & \text{if } \mathtt{AC} > 0 \wedge \mathtt{pc} = \mathtt{Ex} \\ \mathtt{AC} & \text{otherwise} \end{cases}$$

$$\mathit{incrAC}(\mathtt{AC}, \mathtt{En}, \mathtt{Ex}, \mathtt{pc}, \mathtt{Ex}', c) \triangleq \begin{cases} \mathtt{AC}+1, \mathtt{pc}, \mathtt{Ex}' & \text{if } \mathtt{AC} = 0 \wedge c = \mathit{true} \\ \mathtt{AC}+1, \mathtt{En}, \mathtt{Ex} & \text{if } \mathtt{AC} > 0 \wedge \mathtt{pc} = \mathtt{En} \\ \mathtt{AC}, \mathtt{En}, \mathtt{Ex} & \text{otherwise} \end{cases}$$

Figure 4: Evaluation rules for activating instructions.

### 5.4. Formalizing the Programming Models

AMi is a hardware/software co-design, intended to be used for linearization and balancing in the way we informally described in Section 5.1. We now formalize the conditions under which the use of AMi is sound. There are three important properties: *well-behavedness* ensures that nested and recursive activation work as intended, *security* defines conditions under which the programming models for linearization and balancing lead to secure programs, and *correctness* defines conditions under which code has no effect on the program result when running in mimicry mode. We let $[\ell, \ell']\,\sigma \Downarrow_o \sigma'$ denote the big-step evaluation of a program region $[\ell, \ell']$. Intuitively, it executes the program from $\ell$ to $\ell'$ (included), and includes all recursive function calls. The definition is given in Appendix A. In an initial configuration, denoted $\sigma_0$, $\mathtt{pc}$ evaluates to the entry point of the program, and $\mathtt{AC} = 0$. We say that a configuration is *valid* if it can be reached via $\Longrightarrow$ from an initial configuration.

**Well-behaved activating regions.** An *activating region* is a pair $[\ell, \ell']$ where $\ell$ is the address of an activating instruction and $\ell'$ is the corresponding exit address. In the remainder of this section, we give sufficient conditions to build *well-behaved activating regions*. More specifically, an activating region is well-behaved if mimicry mode is not deactivated prematurely and the activation counter is eventually restored to its initial value when reaching the end of the activating region. We assume that the program $P$ can be partitioned into a set of functions $f_1 \ldots f_n : \mathit{Loc} \rightharpoonup \mathit{Inst}$ defined on disjoint subsets of $\mathit{Loc}$. We also assume that the control flow cannot be redirected arbitrarily between these functions:

**Hypothesis 1** (Control flow integrity)**.** *A function cannot directly jump to another function (a function can only enter another function via a call to its entry point). Additionally, a function call at address $\ell$ always returns to its return address $\ell + 1$ or does not terminate.*

Additionally, we define the control-flow graph (CFG) of a function, the concept of (post-)dominance relations between labels, and ultimately, single-entry single-exit (SESE) regions [39], on which we rely to implement well-behaved activating regions:

**Definition 2** (Intra-procedural CFG)**.** *The CFG of a function $f$, denoted $\mathit{cfg}(f)$, is a tuple $(\ell_{en}, \ell_{ex}, L, E)$ where $\ell_{en} \in L$ is the entry point of the function, $\ell_{ex} \in L$ is the exit point of the function*[1]*, vertexes in $L$ are program locations defined by the function, and edges in $E$ link locations together such that there is an edge from $\ell$ to $\ell'$ if and only if one of the three conditions hold: 1) $\ell$ can immediately follow $\ell'$ in program execution, 2) there is a call at address $\ell$ with return address set to $\ell'$, or 3) there is an activating branch at address $\ell$ with target $\ell'$.*

**Definition 3** ((Post-)domination)**.** *In a CFG $(\ell_{en}, \ell_{ex}, L, E)$, a label $\ell$ dominates a label $\ell'$ if every path from the entry node $\ell_{en}$ to $\ell'$ includes $\ell$. Conversely, a node $\ell$ post-dominates a node $\ell'$ if every path from $\ell'$ to the exit node $\ell_{ex}$ includes $\ell$. By convention, a node dominates and postdominates itself.*

**Definition 4** (SESE region)**.** *In a CFG, a SESE region is a pair of labels $[\ell, \ell']$ where: 1) $\ell$ dominates $\ell'$, 2) $\ell'$ post-dominates $\ell$, 3) every cycle containing $\ell$ also contains $\ell'$ and vice versa.*

Intuitively, the two first conditions guarantee that an SESE region $[\ell, \ell']$ cannot be entered without executing the instruction at label $\ell$ and cannot be exited without executing the instruction at label $\ell'$. The last condition guarantees that when the SESE region is executed it only goes through $\ell$ and $\ell'$ once.

**Hypothesis 2.** *All activating regions of the program are SESE regions.*

Notice that this hypothesis usually holds for function calls and for branches that are compiled from high-level `if` statements, which are typically the target of mimic execution.

Finally, the following proposition expresses that under Hypothesis 1 and 2, activating regions are well-behaved:

**Proposition 1** (Well-behaved activating regions)**.** *For any activating region $[\ell, \ell']$ and valid configuration $\sigma$ such that $[\ell, \ell']\,\sigma \Downarrow_o \sigma'$, if $\mathtt{AC}$ is set after executing the instruction at location $\ell$:*

1. Any function can trivially be converted to a function with a single exit point.

1) *it remains set during the execution of* $[\ell, \ell']$ *(including recursive function calls but excluding* $\ell'$*), and*
2) *if* `AC` *is incremented at location* $\ell$*, it is restored to its initial value right before the evaluation of the instruction at location* $\ell'$*.*

A sketch of proof is given in Appendix C.

**Secure activating regions.** We now formalize the conditions under which our programming models for linearization are secure. A security criterion for balancing is given in Appendix B. We focus on the linearization pattern in Listing 7. Other patterns are similar (but differ in the details). We assume $[\ell_0, \ell_n]$ to be a well-behaved and terminating SESE region, and we assume a fixed security policy $\mathcal{P}$.

```
ℓ0 :  beqz  c ℓn
ℓ1 :  B
ℓn :
```

(a) Before linearization

```
ℓ0 :  a.beqz  c ℓn
ℓ1 :  B
ℓn :
```

(b) After linearization

Listing 7: Linearization programming model

$B$ could be a complex region, with e.g., loops and function calls. What are the conditions on $B$ for the pattern to work securely? Informally, we want $B$ to (1) be secure itself (if $B$ already leaks secrets, then this pattern will not eliminate that leakage), and (2) have the same attacker-observable behavior in mimicry mode and standard mode (then the attacker cannot determine what the branch condition was). This is formalized in the definition below.

**Definition 5.** *A terminating SESE region* $B = [\ell_1, \ell_n]$ *is* secure *for the linearization pattern in Listing 7 if the following holds for any low-equivalent* $(m, r) \simeq_{\mathcal{P}} (m', r')$*, and configurations* $\sigma_0 = \langle m, r, \ell_1, 0, -, - \rangle, \sigma_0' = \langle m', r', \ell_1, 0, -, - \rangle, \sigma_0'' = \langle m', r', \ell_1, 1, \ell_0, \ell_n \rangle$*:*

1) *if* $[\ell_1, \ell_n] \, \sigma_0 \Downarrow_o \sigma_1$ *and* $[\ell_1, \ell_n] \, \sigma_0' \Downarrow_{o'} \sigma_1'$ *then* $o = o'$ *and* $\sigma_1 \simeq_{\mathcal{P}} \sigma_1'$
2) *if* $[\ell_1, \ell_n] \, \sigma_0 \Downarrow_o \sigma_1$ *and* $[\ell_1, \ell_n] \, \sigma_0'' \Downarrow_{o''} \sigma_1''$ *then* $o = o''$ *and* $\sigma_1 \simeq_{\mathcal{P}} \sigma_1''$

Note that in the definition above, $\sigma_0$ and $\sigma_0'$ are two low-equivalent configurations that will execute $B$ in standard mode. So the first condition says that $B$ is secure itself: it does not leak secrets into observations or into the public part of the end configuration. $\sigma_0''$ is like $\sigma_0'$ except that it will execute in mimicry mode. So the second condition formalizes that $B$ has the same observable behavior in standard mode and in mimicry mode. For this paper, we assume that it is up to the developer to make sure that $B$ complies with this property, but building tool support for this is an obvious area for future work (Section 7). The first condition is a classic non-interference property. The second condition should guide the developer on when to insert persistent and/or ghost instructions. For example, forgetting persistent qualifiers on loops with a public iteration count or when computing load addresses (cf. Listing 3 and Listing 4) would lead to a violation of the second condition. However, note that mimic execution is designed to make the second condition easy to achieve. For instance, a linear region of mimicable instructions satisfies the condition by design.

**Correct activating regions.** We now formalize the conditions under which our programming models for linearization are correct. A correctness criterion for balancing is given in Appendix B.

To define correctness, we assume a partition of the architectural state into *live* and *dead* state. Intuitively, the live part of the state can influence the result of the program, whereas the dead part of the state cannot. For instance, in a program compiled from a C-like language which does not use a heap, the live state can be the global variables and anything between the base of the stack and the stack pointer. For programs that additionally use the heap, the live state also includes reachable objects on the heap.

Given such a liveness partition, we define a notion of live-equivalence of states where two configurations are live-equivalent, written $\sigma =_{\mathcal{L}} \sigma'$ if they agree on the value of the live part of their register file and memory as defined by $\mathcal{L}$.

Informally, correctness says that $B$ should have no effect on the live state if executed in mimicry mode. This is formalized in the definition below.

**Definition 6.** *A terminating SESE region* $B = [\ell_1, \ell_n]$ *is* correct *for the linearization pattern in Listing 7 if the following holds for any* $(m, r)$*:*

$$[\ell_1, \ell_n] \, \langle m, r, \ell_1, 1, \ell_0, \ell_n \rangle \Downarrow_o \langle m', r', -, -, -, - \rangle \implies (m, r) =_{\mathcal{L}} (m', r')$$

Correctness, like security, can guide the developer on when to insert persistent/ghost instructions. For instance, without the ghost load, Listing 6 would not be correct.

Again, the hardware is designed to help achieve correctness: linear regions of mimicable instructions are correct by the design of mimic execution.

**Discussion.** For now, the formalizations of our programming models are just precise guidance for the developer. But we believe they can be the basis for a formal theory to compositionally compile source code to AMiL. We conjecture that our programming models make it possible to build bigger correct and secure regions from smaller ones. For instance, if $B$ is correct and secure, then (under some conditions) the code (including the activating branch) in Listing 7b will also be correct and secure. Or, if $B_1$ and $B_2$ are both correct and secure, then (under some conditions) their sequential composition is also correct and secure.

## 6. Implementation and Evaluation

To evaluate the costs and benefits of our proposal, we instantiate AMi for RISC-V and develop a hardware prototype. We perform an empirical security and performance evaluation on a set of benchmark programs taken from related work.

### 6.1. AMi for RISC-V

In this subsection, we propose a RISC-V extension for AMi. We limit ourselves to the RV32IM instruction set and to RISC-V machine-mode but it should be straightforward to extend this to other instructions and to other privilege levels.

**AMi extension.** Table 2 contains an overview of the valid associations for the AMi qualifiers. Mimic stores are not supported. Conditional branches and unconditional jumps are the only activating instructions. Memory ordering instructions, environment calls and breakpoints are always executed. To encode the AMi instruction qualifiers, several alternative encoding schemes are possible. In our proposal, we repurpose the two prefix bits in the fixed-width 32-bit encoding, giving us the ability to associate each instruction with up to four instruction qualifiers. We propose three CSRs to store the activation counter, the address of the entry instruction, and the exit address of the current activation.

TABLE 2: RV32IM instruction qualifiers (default first).

| Instruction type | Supported instruction qualifiers |
|---|---|
| integer computational / load | standard, persistent, mimic, ghost |
| store | persistent |
| control transfer | persistent, activating |
| ecall / ordering / breakpoints | persistent |

**Toolchain.** Adding full compiler support for AMi is considered future work. However, to make it possible to manually harden applications at assembly level, we do implement an assembler and a disassembler. To this end, we add the file `RISCVInstrInfoXAMi.td` to the RISC-V backend of the LLVM compiler framework [40]. This file contains the description of the AMi instructions that we present in this paper. Based on this description, the TableGen program [41] generates C++ code to (dis)assemble the new instructions.

### 6.2. Hardware Prototype

Our implementation is based on Proteus [42], a research CPU designed for extensibility. Proteus comes with two separate pipeline implementations; a classic 5-stage in-order, and an out-of-order pipeline with parallel execution units. Both pipelines can be configured with a branch predictor and a data cache. For our evaluation, we extend both pipelines to support AMi. These baseline implementations leak information via timing in multiple ways. For instance, it takes 32 cycles to perform a multiplication (other instructions take a single cycle), data hazards slow down certain instructions while others can continue without stalling due to forward logic. The data cache is also a source of timing leaks. Since we configure Proteus with a branch predictor, the execution time of a conditional branch instruction varies (making the balanced hardened form insecure). All other instructions behave in a data-oblivious fashion i.e., their execution time and resource usage do not depend on the value of their operands. The memory subsystem is not mimicry-aware (i.e., stores cannot be mimicked) so a store must be preceded by a (mimic) load in the balanced form and by a ghost load in the linearized form.

**In-order pipeline.** We implement mimic execution on the in-order pipeline by having a mimicked instruction visit the pipeline stages in the same way as its standard counterpart would do, except that the register file is not updated in the writeback stage. To prevent leaking information via a pipeline channel, we implement mode-independent stalling (i.e., pipeline stalls are independent of the processor mode) but values are *not* forwarded between an instruction that is mimicked and one that is normally executed. On the in-order pipeline, we want to support both the balanced and the linearized hardened forms. However, we also want to configure Proteus with a simple (stateless) always-predict-not-taken branch predictor in order not to negatively impact the performance of code that is not security critical. To reconcile these two conflicting requirements, we introduce a new instruction, the data-oblivious conditional branch (i.e., `ct.beq`). A data-oblivious branch disables the branch predictor and stalls until the branch outcome is known for the branch latency to become independent of the branch condition. It can thus be securely used to balance secret-dependent branches.

**Out-of-order pipeline.** We implement mimic execution on the out-of-order pipeline in a similar fashion as on the in-order pipeline, except that more microarchitectural structures, such as the reorder buffer (ROB), have to be visited by the mimicked instructions. Analogously to the in-order pipeline, the register file is not updated in the retirement stage for instructions that are being mimicked. For the out-of-order pipeline, there are three additional tasks to perform. First, to be able to track mimic dependencies (instructions are mimic-dependent on activating instructions whose shadow they reside in), we introduce a new microarchitectural structure, a double-ended queue, for the bookkeeping of in-flight activating instructions. Second, to be secure and correct, we need to broadcast the outcome of activating branches on the common data bus (CDB), which connects the parallel execution units. Finally, we need to track write-after-write (WAW) dependencies between instructions that are mimic-dependent on different activating branches. Making sure that this information is always correctly calculated and propagated required minor changes to the ROB, the execution units, and some of the internal data buses.

**Hardware cost.** We measure the hardware overhead of our implementation by synthesizing to a Xilinx XC7A35TI-CSG324 FPGA using Xilinx Vivado 2022.2. Table 3 lists the hardware measurements for both the in-order and the out-of-order implementation, showing that the hardware overhead is reasonable. Importantly, the critical path did not increase due to our modifications, which shows the practicality of our proposal on low-end and high-end processors.

TABLE 3: Hardware overhead factors.

| Pipeline | LUTs | FFs | CP (ns) | LUTs Δ | FFs Δ | CP Δ |
|---|---|---|---|---|---|---|
| In-order | 2621 | 1411 | 17.0 | + 512 (19.5%) | + 313 (22.2%) | +0.1 (0.6%) |
| Out-of-order | 16185 | 11943 | 29.2 | +4264 (26.3%) | +1102 ( 9.2%) | -0.3 (1.0%) |

## 6.3. Experimental Evaluation

**Methodology.** To establish a vulnerable baseline, we start from third-party benchmark programs (written in C) that all contain *annotated* secret-dependent control flow. Each benchmark is placed in a dedicated function which is invoked several times with different but carefully chosen input parameters to execute all the secret-dependent execution paths. We compile (`-O3`) these benchmarks to RISC-V assembly code and then manually balance and linearize the secret-dependent control flow at assembly level with and without leveraging the new instructions for mimic execution (i.e., we create four hardened versions for each benchmark). We balance the code using dummy instructions [5] and we linearize the code based on a state-of-practice technique that was first proposed by Molnar et al. [9]. We then compile and link the different versions of the programs. We further perform and automate the following steps. First, to guarantee that the manually hardened programs preserve program semantics, we perform a correctness evaluation. Second, we perform a security evaluation to confirm that 1) all benchmark programs leak secret information in their control flow and that 2) the hardened versions effectively close these leaks. Finally, we compare the performance of the hardened forms with the vulnerable baseline. To obtain cycle-accurate measurements, we execute the programs in a simulator that produces a waveform file which represents the complete state of the CPU during simulation. Since this file contains the values of all the signals exactly as on real hardware, our automated evaluations can rely on it for the data they need about the executions.

**Benchmark suite.** To the best of our knowledge, there is no established benchmark suite of vulnerable (i.e., non constant-time) programs to evaluate our defense. In our evaluation, we use the benchmark suite from [5] as it is specifically created for evaluating an algorithm to balance secret-dependent control flow. It consists of 1) synthetic programs that push the limits of code-transformation techniques (it features a wide range of control-flow patterns such as nested conditionals and loops), and of 2) realistic programs that have been used before in the evaluation of other related work [2], [43], [44]. Having no compiler support, hardening vulnerable programs is a manual effort. For this reason we make a (random) selection of the benchmark programs from [5] for our evaluation.

**Experimental setup.** We conduct our experiments on an off-the-shelf laptop running Ubuntu 22.04. We compile and assemble the benchmarks with LLVM 15.0.0 with our changes to the assembler incorporated. We link the programs with version 11.1.0 of the GCC toolchain. The Verilog code of Proteus is generated with version 1.6.4 of SpinalHDL. We use Verilator 4.028 to simulate the hardware.

TABLE 4: Binary size overhead factors.

| Benchmark | Baseline | Balanced | | Linearized | |
|---|---|---|---|---|---|
| | Size (bytes) | No-AMi | AMi | Molnar | AMi |
| bsl | 336 | 1.04x | 1.04x | 1.08x | 1.00x |
| kruskal | 452 | 1.05x | 1.05x | 1.16x | 1.02x |
| keypad | 460 | 1.07x | 1.07x | 1.11x | 1.02x |
| modexp2 | 324 | 1.02x | 1.02x | 1.09x | 1.00x |
| mulmod16 | 276 | 1.01x | 1.01x | 1.16x | 0.97x |
| sharevalue | 500 | 1.02x | 1.02x | 1.15x | 1.01x |
| triangle | 132 | 1.06x | 1.06x | 1.15x | 1.00x |
| fork | 136 | 1.00x | 1.00x | 1.12x | 0.97x |
| switch | 500 | 1.41x | 1.41x | 1.92x | 1.02x |
| diamond | 212 | 1.00x | 1.00x | 1.11x | 0.94x |
| ifthenloop | 200 | 1.20x | 1.10x | 1.20x | 1.00x |
| **mean** | | **1.08x** | **1.07x** | **1.19x** | **1.00x** |

**Security evaluation.** We empirically evaluate at gate level the timing and resource usage behavior of the benchmark programs with respect to our hardware implementation (instead of relying on a hardware model). To obtain cycle-accurate measurements, we use a hardware simulator that makes the complete cycle-level behavior visible (in the waveform file), which we use to detect attacker-visible secret-dependent behavior. With our automated approach, we exercise all possible secret-dependent execution paths on the hardware simulator and verify that attacker-observable signals behave identically for identical public, but different secret inputs. By configuring our approach with the set of attacker-observable signals, we use it to validate the control-flow security properties of our AMi implementation under two attacker models (materialized by our in-order and our out-of-order pipeline implementations).

Our security evaluation confirms that all the benchmark programs leak secret information in their control flow and that the hardened versions effectively close these leaks.

**Binary size.** Table 4 presents the binary size overhead. We discuss the balanced form first. According to these results, it seems that AMi only minimally reduces the overhead of the balanced form. There are three reasons for this.

First, the leakage behavior of our RISC-V core makes it possible for each instruction to craft a no-op that leaks information in exactly the same way. This makes it possible to balance secret-dependent control flow without having to blacklist instructions. Therefore, no expensive rewrites in terms of non-blacklisted instructions are necessary. In other words, on our core it is possible to compensate the leakage of every instruction with a suitable no-op. Note that this is most likely going to be the case with AMi because most instructions can be turned into mimic instructions.

Second, although balancing control flow without AMi instructions increases register pressure in general (in contrast to our approach), for the benchmark programs that we use in our experiments it does not lead to additional (and expensive) spill and reload operations. Consider for instance

the program with a secret-dependent branch in Figure 8a. Assume that the code is compiled for a platform that provides eight general purpose registers which are all live during the execution of the code of the example. Figure 8b contains the balanced form without AMi instructions. Due to the high register pressure, registers need to be spilled to the stack first (line 2 and line 7) before using them in the no-op instructions (line 3 and line 8). Afterwards, they have to be loaded from the stack again (line 5 and line 10). Figure 8c contains the balanced form that leverages the AMi instructions which does not need to spill any registers. Note that a high register pressure might also prevent other optimizations such as function inlining.

Third, the benchmark programs typically feature short branches without function calls. While AMi provides the `a.call` instruction to invoke a function and execute it in mimicry mode, balancing without AMi must invoke a *dummy function*. Constructing such a dummy function considerably increases code size. The ifthenloop benchmark demonstrates this. It is the only program in the benchmark suite that contains a function call in one of its secret-dependent regions,

```
1: beq x1, x2, 5
2:  mul x3, x4, x5
3:  jmp 5
4:  add x6, x7, x8
5: ...
   ; x1-x8 are live here
.
```

(a) Vulnerable

```
1: beq   x1, x2, 12
2:  store x6, sp
3:  mul   x3, x4, x5
4:  add   x6, x7, x8
5:  load  x6, sp
6:  jmp   12
7:  store x3, sp
8:  mul   x3, x4, x5
9:  add   x6, x7, x8
10: load  x3, sp
11: jmp   12
12: ...
```

(b) Balanced

```
1: beq   x1, x2, 8
2:  mul   x3, x4, x5
3:  m.add x6, x7, x8
4:  jmp   8
5:  m.mul x3, x4, x5
6:  add   x6, x7, x8
7:  jmp   8
8: ...
.
```

(c) Balanced AMi

Listing 8: Balancing branches under high register pressure.

For the linearized form, the positive impact of AMi is very significant. On average, the state-of-the-art Molnar-based technique increases the binary size by 19%. With AMi the average increase is zero. In other words, the overhead is reduced by **100%**. In our implementation, only stores contribute to a code size increase. Regions without stores that are linearized with AMi do not increase the size of the binary at all, and manual optimizations can even lead to decreases in the binary size.

**Execution time.** Tables 5 and 6 present the execution time overhead for the in-order and out-of-order pipelines. For the same reasons as for the binary size, AMi does not reduce the execution time overhead for the balanced form. Also in line with the binary size results, the impact of AMi on the execution time is very significant for the linearized form. On average, AMi reduces the overhead in our benchmarks compared to the state-of-the-art by **58%** on the in-order pipeline and by **60%** on the out-of-order pipeline. It is important to note that linearized code inherently introduces an overhead that cannot be avoided. Indeed, to make sure that different executions cannot be distinguished, at the very least, *all* instructions of *all* possible targets of a control-flow

TABLE 5: Execution time overhead factors (*in order*).

| **Benchmark** | **Baseline** | **Balanced** | | **Linearized** | |
|---|---|---|---|---|---|
| | Time (cycles) | No-AMi | AMi | Molnar | AMi |
| bsl | 1678 | 1.51x | 1.51x | 1.31x | 0.95x |
| kruskal | 1280 | 1.12x | 1.12x | 1.28x | 1.08x |
| keypad | 3672 | 2.43x | 2.43x | 1.99x | 1.47x |
| modexp2 | 12773 | 1.71x | 1.71x | 1.79x | 1.69x |
| mulmod16 | 295 | 1.37x | 1.37x | 1.59x | 1.41x |
| sharevalue | 1472 | 1.79x | 1.79x | 1.94x | 1.64x |
| triangle | 80 | 1.23x | 1.23x | 1.11x | 0.99x |
| fork | 80 | 1.06x | 1.06x | 1.11x | 0.99x |
| switch | 863 | 4.73x | 4.73x | 3.75x | 1.54x |
| diamond | 174 | 1.07x | 1.07x | 1.09x | 0.89x |
| ifthenloop | 186 | 1.48x | 1.48x | 1.49x | 1.36x |
| **mean** | | **1.59x** | **1.59x** | **1.57x** | **1.24x** |

TABLE 6: Execution time overhead factors (*out of order*).

| **Benchmark** | **Baseline** | **Balanced** | | **Linearized** | |
|---|---|---|---|---|---|
| | Time (cycles) | No-AMi | AMi | Molnar | AMi |
| bsl | 1655 | - | - | 1.27x | 0.92x |
| kruskal | 1291 | - | - | 1.26x | 1.06x |
| keypad | 3414 | - | - | 2.06x | 1.77x |
| modexp2 | 12196 | - | - | 1.79x | 1.69x |
| mulmod16 | 323 | - | - | 1.48x | 1.30x |
| sharevalue | 1250 | - | - | 1.83x | 1.59x |
| triangle | 97 | - | - | 0.99x | 0.89x |
| fork | 94 | - | - | 1.02x | 0.89x |
| switch | 992 | - | - | 3.33x | 1.41x |
| diamond | 210 | - | - | 0.95x | 0.78x |
| ifthenloop | 192 | - | - | 1.45x | 1.28x |
| **mean** | | | | **1.48x** | **1.19x** |

transfer must *always* be executed while the corresponding vulnerable code only needs to execute a single target. AMi, however, makes it possible to limit the cost of linearization to this unavoidable overhead and is thus optimal in this sense.

Also notice that AMi sometimes performs better than the vulnerable baseline. The vulnerable baseline can induce pipeline flushes as a result of mispredicted sensitive control-flow instructions. In the linearized form, however, activating branches always fall through to the next instruction, and as a result are never mispredicted. This avoids the performance penalty of pipeline flushes, sometimes resulting in significant performance gains in our small benchmark programs. Thus, it is important to compare the execution times of AMi to that of the linearized form that we created by applying Molnar's method, which also experiences no branch penalty.

## 7. Discussion and Future Work

We have shown in this paper that the idea of mimic execution and Architectural Mimicry is feasible and useful for improving the code size and execution time of software hardened against control-flow leaks. But we believe there is very rich potential for further exploration of the idea.

First, it would clearly be beneficial to take the concern of side-channel protection out of the hands of the developer. Rather than manually writing low-level code and painstak-

ingly taking care that it complies with the programming models we defined, this should be handled by tools like compilers, binary rewriters or verifiers. Researchers have already developed impressive tool support to develop, compile and verify data-oblivious code [19]. Tools for compliance with our programming models could be developed along the same lines. The most interesting area to explore is *compilation* towards platforms that support AMi. AMi is a security-aware ISA-extension specifying indistinguishability between real execution and mimic execution in current *and* future microarchitectures. Following the idea of hardware-software security contracts [45], the leakage model can also be considered to be part of the ISA specification, and hence compilers can generate code that relies on these contracts for security, following the recent idea of Contract-Aware Secure Compilation (CASCO) [23]. For instance, an architecture that complies with our Version 1 leakage model can then securely rely on balancing for security, potentially leading to substantial performance benefits over constant-time code [5]. The long-term goal should be to give the compiler: (1) source code with security annotations (what is secret?), and (2) a security contract for the ISA. The compiler can then generate the best possible secure code for that ISA, possibly relying on advanced code analysis and heuristics. We believe that AMi can be an important enabler for the development of CASCO compilers, and that the further development of these ideas is a very fertile area for future work.

Second, it would be interesting to generalize our design. There are different dimensions to explore. On the one hand, more hardware components can be made mimicry-aware. An obvious candidate is the memory subsystem: in mimicry mode, the memory subsystem should suppress actual storing of values, but should mimic all buffering/caching side-effects. This would, for instance, make the store instruction mimicable too. On the other hand, generalization towards other kinds of microarchitectural features with corresponding side-channel attacks should be investigated. An interesting direction is to consider transient execution attacks [28].

Finally, we believe the formal model and the formalizations of the AMi programming models deserve further exploration. For instance, we believe it should be feasible to develop provably correct compositional compilation schemes from high-level structured control flow to correct and secure assembly code in an AMi-supporting ISA.

## 8. Related Work

**Control-flow leakage.** To the best of our knowledge, hardware mechanisms specifically designed to prevent control-flow leakage have not been proposed before. Some countermeasures do leverage hardware primitives designed with performance in mind (e.g., predicated execution [7], [8], [10], [12] and Intel TSX [46]). Using these primitives to harden applications provides brittle security guarantees as it relies on behavior not guaranteed in future hardware.

Software-only countermeasures to prevent leaking secret-dependent control flow have been the subject of research by a large body of work since Kocher's seminal work on timing attacks [15]. Transforming out timing leaks by balancing conditional branches was first proposed by Agat in [2], where he describes a transformational security type system based on cross-copying skip instructions. Köpf and Mantel [47] propose a unification-based improvement to this approach. Dewald et al. [3] apply these ideas in practice with a (non-transformational) security type system to detect unbalanced branches on the AVR platform. Pouyanrad et al. [4] implement this approach for the MSP430 architecture. Winderix et al. [5] present, implement and evaluate an algorithm to balance secret-dependent control flow during compilation for lightweight embedded platforms.

Balancing secret-dependent control flow does not offer strong security guarantees on high-end computing platforms. Molnar et al. [9] propose an improvement in the form of the program counter security model. Under this model, the trace of executed instructions is independent of confidential information. Additionally, the authors propose a bit-masking based scheme to linearize secret-dependent control flow. This scheme is still representative as it is being incorporated in more recent work, for instance to lower language abstractions to platforms that do not have conditional execution [7], [10]. Coppens et al. [8] were the first to present a linearization technique for the x86 architecture that leverages predicated execution (`cmov` instructions).

Linearizing branches has been adopted as the de facto standard technique to solve the control-flow leakage problem as it is an essential part of the widespread constant-time programming policy [11], [22], [25], [48], [49]. Recently, it has been shown that in the context of speculative execution the classic notion of constant-time is not secure [50], and the notion of speculative constant-time appeared [51]. Spectre mitigations are covered in complementary work [28].

**ISA support.** There is some related work that is situated at the level of the hardware-software interface to support some aspects of the classic constant-time programming model. For instance, many processors nowadays have constant-time support for the Advanced Encryption Standard (AES) to improve the speed and security of applications that rely on AES (e.g., AES-NI [52]). As an another example, CPU manufacturers such as Arm and Intel have extended their ISAs to make the execution time of instructions independent of the values of their operands [53], [54].

Yu et al. [27], propose ISA design principles for Data-oblivious ISAs (OISAs) to perform side-channel resistant and high-performance computations. Their work is complementary to ours as it focuses on leakage through unsafe instructions (whose execution induces operand-dependent timing or hardware resource usage). It does not propose new techniques to deal with the control-flow leakage problem.

**Contract-aware secure compilation.** To improve source code portability, with this work we encourage decoupling the security policy from the source code. This idea is related to recent work on hardware-software contracts for security [45], [55]–[58], where the ISA specifies how hardware

leaks information. This information can then be leveraged by a contract-aware secure compiler [23] to decide if it is secure (and profitable) to balance a secret-dependent branch instead of eliminating it. Dinesh et al. [59] advocate a similar direction for other data-oblivious aspects by synthesizing translations for unsafe instructions using only instructions from a safe set (specified in a security contract in the ISA).

## 9. Conclusions

We propose and evaluate a new hardware-software co-design to support the development of efficient and portable side-channel resistant code. On the hardware side, we introduce mimic execution, and an ISA design (AMi) to work with it. On the software side, we propose and formalize programming models that use AMi to perform correct and secure control-flow balancing and linearization. Our implementation and experiments show substantial performance benefits at low hardware cost, and hence we believe it would be very interesting to explore in future work how AMi can be leveraged in compilers and binary verification tools. We believe AMi can be an important enabler for the idea of contract-aware secure compilation, where a compiler produces *efficient* and *secure* code for a platform for which it has not only the functional ISA specification, but also a security specification.

## Acknowledgments

This research was partially funded by the ORSHIN project (Horizon Europe grant agreement No. 101070008), by the Research Foundation Flanders (FWO), and by the Flemish Research Programme Cybersecurity. We thank the anonymous reviewers for their valuable feedback.

## References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptogr. Eng. 8.1*, 2018.

[2] J. Agat, "Transforming out timing leaks," in *POPL*, 2000.

[3] F. Dewald, H. Mantel, and A. Weber, "AVR processors as a platform for language-based security," in *ESORICS*, 2017.

[4] S. Pouyanrad, J. T. Mühlberg, and W. Joosen, "SCF-MSP: Static detection of side channels in MSP430 programs," in *ARES*, 2020.

[5] H. Winderix, J. T. Mühlberg, and F. Piessens, "Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks," in *EuroS&P*, 2021.

[6] M. Bognar, H. Winderix, J. Van Bulck, and F. Piessens, "Microprofiler: Principled side-channel mitigation through microarchitectural profiling," in *EuroS&P*, 2023.

[7] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *CCS*, 2021.

[8] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P*, 2009.

[9] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2005.

[10] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*, 2018.

[11] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time implementations," in *USENIX Security*, 2016.

[12] L. Soares, M. Canesche, and F. M. Q. a. Pereira, "Side-channel elimination via partial control-flow linearization," *TOPLAS*, 2023.

[13] G. Barthe, T. Rezk, and M. Warnier, "Preventing timing leaks through transactional branching instructions," *Electr. Notes in Theoretical Comp. Science 153.2*, 2006.

[14] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital Side-Channels through obfuscated execution," in *USENIX Security*, 2015.

[15] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*, 1996.

[16] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *S&P*, 2013.

[17] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on amazon's s2n implementation of tls," in *EUROCRYPT*, 2016.

[18] D. Moghimi, J. V. Bulck, N. Heninger, F. Piessens, and B. Sunar, "CopyCat: Controlled Instruction-Level attacks on enclaves," in *USENIX Security*, 2020.

[19] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *S&P*, 2022.

[20] F. Piessens, "Security across abstraction layers: old and new examples," in *EuroS&PW*, 2020.

[21] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you c: Controlling side effects in mainstream c compilers," in *EuroS&P*, 2018.

[22] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *CSF*, 2018.

[23] M. Guarnieri and M. Patrignani, "Contract-aware secure compilation," *arXiv*, 2020.

[24] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 7:1–7:30, 2020.

[25] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A Flexible, Constant-Time Programming Language," in *SecDev*, 2017.

[26] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *CCS*. ACM, 2017, pp. 1807–1823.

[27] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *NDSS*, 2019.

[28] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019.

[29] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *STOC*, 1987.

[30] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *CCS*, 2013.

[31] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *CCS*, 2021.

[32] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Breaking and fixing speculative load hardening," *IACR Cryptol. ePrint Arch.*, 2022.

[33] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018.

[34] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "ProSpeCT: Provably Secure Speculation for the Constant-Time Policy," in *USENIX Security*, 2023.

[35] R. Choudhary, J. Yu, C. W. Fletcher, and A. Morrison, "Speculative privacy tracking (SPT): leaking information from speculative execution without compromising privacy," in *MICRO*, 2021.

[36] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*, 1999.

[37] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *S&P*, 2020.

[38] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *S&P*, 2021.

[39] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *PLDI*, 1994.

[40] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[41] "TableGen overview," https://llvm.org/docs/TableGen/.

[42] M. Bognar, J. Noorman, and F. Piessens, "Proteus: An extensible risc-v core for hardware extensions," in *RISC-V Summit Europe '23*, Jun. 2023.

[43] H. Mantel and A. Starostin, "Transforming out timing leaks, more or less," in *ESORICS*, 2015.

[44] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *CCS*, 2018.

[45] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *S&P*, 2021.

[46] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache Side-Channel protection using hardware transactional memory," in *USENIX Security*, 2017.

[47] B. Köpf and H. Mantel, "Transformational typing and unification for automatically correcting insecure programs," *Int. J. of Inf. Security 6.2*, 2007.

[48] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *LATINCRYPT*, 2012.

[49] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *DATE*, 2017.

[50] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *S&P*, 2020.

[51] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *PLDI*, 2020.

[52] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough aes performance with intel aes new instructions," *White paper 12*, 2010.

[53] "Arm Armv8-A Architecture Registers: DIT, Data Independent Timing," https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing.

[54] "Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[55] G. Heiser, "For safety's sake: We need a new hardware-software contract!" *IEEE Des. Test 35.2*, 2018.

[56] J. Lowe-Power, V. Akella, M. K. Farrens, S. T. King, and C. J. Nitta, "Position paper: A case for exposing extra-architectural state in the isa," in *HASP*, 2018.

[57] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *ISCA*, 2022.

[58] H. Ponce-de León and J. Kinder, "Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks," in *S&P*, 2022.

[59] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "Synthct: Towards portable constant-time code," in *NDSS*, 2022.

# Appendix A. Big-step Evaluation of a Region

We define the big-step evaluation of a program region $[l, l']$, denoted $[l, l']\,\sigma \Downarrow_o \sigma'$, in Fig. 5. Intuitively, it executes the program from a configuration $\sigma$, starting at location $l$ and until the execution reaches location $l'$ (included). Notice that if a function is called, rule CALL ensures that the function returns before the big-step evaluation is terminated in order to avoid early termination due to recursive function calls.

$$\textsc{inst}\quad \frac{l = l' \qquad \sigma.\mathtt{pc} = l \qquad P[l] \neq q\texttt{.call } f \qquad \sigma \overset{o}{\Longrightarrow} \sigma'}{[l, l']\,\sigma \Downarrow_o \sigma'}$$

$$\textsc{call}\quad \frac{l = l' \qquad \sigma.\mathtt{pc} = l \qquad P[l] = q\texttt{.call } f \qquad \sigma \overset{o_c}{\Longrightarrow} \sigma_f \qquad [\mathit{entry}(f), \mathit{exit}(f)]\,\sigma_f \Downarrow_{o_f} \sigma' \qquad o = o_c \cdot o_f}{[l, l']\,\sigma \Downarrow_o \sigma'}$$

$$\textsc{step}\quad \frac{l \neq l' \qquad \sigma.\mathtt{pc} = l \qquad [l, l]\,\sigma \Downarrow_{o''} \sigma'' \qquad l'' = \sigma''.\mathtt{pc} \qquad [l'', l']\,\sigma \Downarrow_{o'} \sigma' \qquad o = o'' \cdot o'}{[l, l']\,\sigma \Downarrow_o \sigma'}$$

Figure 5: Big step evaluation of a region $[l, l']$ where $\mathit{entry}(f)$ (resp. $\mathit{exit}(f)$) denotes the entrypoint (resp. exit-point) of the function $f$ and $\sigma.\mathtt{pc}$ denotes the value of the program counter in the configuration $\sigma$.

# Appendix B. Secure and Correct Balanced Regions

We focus on the balancing pattern in Listing 9. Other patterns are similar (but differ in the details). We assume $B_1$ and $B_2$ to be disjoint terminating SESE regions such that the only successor of their exit node is $\ell_n$. We assume a fixed security policy $\mathcal{P}$ and liveness partition $\mathcal{L}$.

## B.1. Secure Balanced Regions

Analogous to the linearization pattern defined in Section 5.4, we need two conditions for the balancing pattern

| | |
|---|---|
| $\ell_0$ : beqz $c$ $\ell_3$ | $\ell'_0$ : beqz $c$ $\ell'_3$ |
| $\ell_1$ : $B_1$ | $\ell'_1$ : $B'_1$ |
| $\ell_3$ : $B_3$ | $\ell'_3$ : $B'_3$ |
| $\ell_n$ : | $\ell'_n$ : |

(a) Before balancing (b) After balancing

Listing 9: Balancing programming model

to work securely. First, we want $B'_1$ and $B'_3$ to be secure by themselves, i.e., they should not leak secrets. Second, $B'_1$ and $B'_3$ should have the same attacker-observable behavior (then the attacker cannot determine whether $B'_1$ or $B'_3$ was executed). This is formalized in the definition below.

**Definition 7.** *Terminating SESE regions $B'_1 = [\ell'_1, \ell'_2]$ and $B'_3 = [\ell'_3, \ell'_4]$ are* secure *for the balancing pattern in Listing 9 if the following holds for any configurations $\sigma_0, \sigma'_0$ with low-equivalent architectural configurations $(m, r) \simeq_{\mathcal{P}} (m', r')$:*

1) *if $[\ell'_1, \ell'_2]\,\sigma_0 \Downarrow_o \sigma_1$ and $[\ell'_1, \ell'_2]\,\sigma'_0 \Downarrow_{o'} \sigma'_1$ then $o = o'$ and $\sigma_1 \simeq_{\mathcal{P}} \sigma'_1$*
2) *if $[\ell'_3, \ell'_4]\,\sigma_0 \Downarrow_o \sigma_2$ and $[\ell'_3, \ell'_4]\,\sigma'_0 \Downarrow_{o'} \sigma'_2$ then $o = o'$ and $\sigma_2 \simeq_{\mathcal{P}} \sigma'_2$*
3) *if $[\ell'_1, \ell'_2]\,\sigma_0 \Downarrow_o \sigma_1$ and $[\ell'_3, \ell'_4]\,\sigma'_0 \Downarrow_{o'} \sigma'_2$ then $o = o'$ and $\sigma_1 \simeq_{\mathcal{P}} \sigma'_2$*

In the definition above, the first (resp. second) condition says that $B'_1$ (resp. $B'_3$) is secure itself: it does not leak secrets into observations or into the public part of the end configuration. The third condition formalizes that $B'_1$ and $B'_3$ have the same observable behavior.

### B.2. Correct Balanced Regions

We now formalize the conditions under which our programming models for balancing are correct. Informally, correctness says that $B'_1$ (resp. $B'_3$) should behave like $B_1$ (resp. $B_3$). This is formalized in the definition below.

**Definition 8.** *A terminating SESE region $B'_1 = [\ell'_1, \ell'_2]$ is a* correct *balanced version of $B_1 = [\ell_1, \ell_2]$ for the balancing pattern in Listing 9 if the following holds for any live-equivalent AMi configurations $\sigma_0 =_{\mathcal{L}} \sigma'_0$:*

$$\text{if } [\ell_1, \ell_2]\,\sigma_0 \Downarrow \sigma_1 \text{ and } [\ell'_1, \ell'_2]\,\sigma'_0 \Downarrow \sigma'_1 \text{ then } \sigma_1 =_{\mathcal{L}} \sigma'_1$$

*where two AMi configurations $\sigma, \sigma'$ are live-equivalent (i.e., $\sigma =_{\mathcal{L}} \sigma'$) if and only if their architectural configurations are live-equivalent.*

## Appendix C. Proof of Proposition 2

In this section, we provide a sketch of proof that activating regions are well-behaved in the sense that they are not deactivated prematurely and the activation counter is restored to its initial value when reaching the end of the region:

**Proposition 2** (Well-behaved activating regions)**.** *For any activating region $[\ell, \ell']$ and valid configuration $\sigma$ such that $[\ell, \ell']\,\sigma \Downarrow_o \sigma'$, if* AC *is set after executing the instruction at location $\ell$:*

1) *it remains set during the execution of $[\ell, \ell']$ (including recursive function calls but excluding $\ell'$), and*
2) *if* AC *is incremented at location $\ell$, it is restored to its initial value right before the evaluation of the instruction at location $\ell'$.*

We let $\sigma.f$ denote the field $f$ in the configuration $\sigma$ and use the term *activation configuration* to refer to the triplet AC, En, Ex in a configuration. Before proving Proposition 2 in Appendix C.2, we show intermediate lemmas about the structure of activating regions in Appendix C.1.

### C.1. Control-flow Graphs and Activating Regions

**Lemma 1.** *Given an activating region $[l, l']$ in a function $f$, there is always an edge between $l$ and $l'$ in cfg($f$).*

*Proof.* For an activating call, $l$: **a.call** $f$ the corresponding activating region is $[l, l+1]$. From Hypothesis 1, the function call returns to $l+1$, hence from Definition 2 there is an edge between $l$ and $l+1$. For an activating branch $l$: **a.beqz** c $l'$ the corresponding activating region is $[l, l']$. It directly follows from Definition 2 that there is an edge between $l$ and $l'$. □

**Definition 9.** *A label $l$ is contained in a SESE region $[l_1, l'_1]$ if $l_1$ dominates $l$ and $l'_1$ postdominates $l$.*

Intuitively, the location $l$ is on a path from $l_1$ to $l'_1$. We say that an SESE region $[l_1, l'_1]$ is contained in another SESE region $[l_2, l'_2]$ if $l_1$ and $l'_1$ are contained in $[l_2, l'_2]$. Additionally, $[l_1, l'_1]$ is adjacent to $[l_2, l'_2]$ if and only if both regions are disjoint except from $l'_1 = l_2$ or $l'_2 = l_1$.

The following lemma expresses that activating regions are properly nested. In particular, it means that activating regions cannot partially overlap.

**Lemma 2** (Properly nested activating regions)**.** *For all disjoint activating regions $[l_1, l'_1]$, $[l_2, l'_2]$ that belong to the same function, either*

1) *$[l_1, l'_1]$ and $[l_2, l'_2]$ are disjoint or adjacent, or*
2) *$[l_1, l'_1]$ is contained in $[l_2, l'_2]$ or vice versa.*

*Proof.* The proof is an adaptation of a proof from [39] (simplified to fit our case). Suppose two distinct activating regions $[l_1, l'_1]$ and $[l_2, l'_2]$ that are not disjoint nor adjacent (they contain a shared instruction $l$ that is not $l_1$ nor $l_2$). First, we have from Hypothesis 2 that $[l_1, l'_1]$ and $[l_2, l'_2]$ are SESE regions. Since $l_1$ and $l_2$ both dominate $l$ (cf. Definition 9), we have either $l_1$ dominates $l_2$ or vice versa. Without loss of generality, suppose $l_1$ dominates $l_2$. Similarly, $l'_1$ and $l'_2$ both postdominate $l$. If $l'_1$ postdominates $l'_2$ then $[l_2, l'_2]$ is contained in $[l_1, l'_1]$. Otherwise $l'_2$ postdominates $l'_1$. There are two cases to consider, which both lead to a contradiction.

- $l_2$ dominates $l'_1$. Note that from Lemma 1 there is an edge $l_1 \to l'_1$ so, we have that $l_2$ must dominate

$l_1$ to dominate $l'_1$. Because $l_1$ dominates $l_2$ this can only happen if $l_1 = l_2$. However, from the evaluation rules of activating instructions (Fig. 4), there cannot be two distinct activating regions starting with the same instruction, meaning that $[l_1, l'_1] = [l_2, l'_2]$, which is a contradiction.

- $l_2$ does not dominate $l'_1$. In this case, we either have: 1) there exists a path from the beginning of the function to $l'_1$, which does not contain $l_2$ or, 2) $l'_1 = l_2$. In the first case, because $l'_2$ postdominates $l'_1$, this path also goes through $l'_2$. Hence, because $l_2$ dominates $l'_2$, this path must also go through $l_2$, meaning that $l_2$ must postdominate $l'_1$. In the second case, we have $l_2$ postdominates $l'_1$ by definition. Finally, because $l'_1$ postdominates $l$, we have that $l_2$ also postdominates $l$. However, because $l_2$ also dominates $l$, this can only happen if $l = l_2$ (both regions are adjacent), which is a contradiction.

□

## C.2. Well-behavedness of Activating Regions

It follows from the evaluation rules of AMi in Figs. 3 and 4 that:

**Proposition 3.** *A step from a configuration where* $\texttt{AC} > 0$ *decrements* AC *only if* $\texttt{pc} = \texttt{Ex}$ *and increments* AC *only if* $\texttt{pc} = \texttt{En}$. *Additionally,* En *and* Ex *are only modified if* AC *is set to 0.*

Remember that a configuration is *valid* if it can be reached from an initial configuration. It follows from the evaluation rules that:

**Proposition 4.** *For any valid configuration* $\sigma$, *if* $\sigma.\texttt{AC} > 0$, *then* $[\sigma.\texttt{En}, \sigma.\texttt{Ex}]$ *is an activating region.*

We first show a base case stating that nested SESE regions behave as intended assuming function calls preserve activation configurations:

**Lemma 3.** *During the evaluation of an SESE region* $[l_1, l'_1]$ *in a configuration* $\sigma$ *that has mimicry mode activated for* $[l_2, l'_2]$ *and where* $[l_2, l'_2]$ *is contained in* $[l_1, l'_1]$, *assuming function calls in* $[l_1, l'_1]$ *preserve the activation configuration and do not deactivate mimicry mode, we have that if* $[l_1, l'_1]\,\sigma \Downarrow_o \sigma'$, *then* $\sigma.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle = \sigma'.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle$ *and mimicry mode is not deactivated during the evaluation of* $[l_1, l'_1]$.

*Proof sketch.* Assume two SESE regions $[l_1, l'_1]$ and $[l_2, l'_2]$ that belong to a function $f$, such that $[l_2, l'_2]$ is contained in $[l_1, l'_1]$, and a configuration $\sigma$ that has mimicry mode activated for $[l_2, l'_2]$.
Assuming the following hypothesis,

Function calls in $[l_1, l'_1]$ preserve the activating configuration and do not deactivate mimicry mode (H)

we want to show that the big-step evaluation of $[l_1, l'_1]$ preserve the activation configuration and does not deactivate mimicry mode. From Proposition 3, En and Ex can only be modified when $\texttt{AC} = 0$ meaning that is sufficient to show that the big-step evaluation of $[l_1, l'_1]$ restores AC and that AC does not reach 0.

Because of (H), we can ignore the evaluation of functions and, because we assume a sound CFG (cf. Definition 2), we can reason about the big-step evaluation of $[l_1, l'_1]$, as a sequence of small steps following a path in $\mathit{cfg}(f)$. From Proposition 3, we have that AC can only be incremented when the evaluation reaches $l_2$ or decremented when it reaches $l'_2$. Hence we have to show that every path from $l_1$ to $l'_1$ in $\mathit{cfg}(f)$:

1) must contain $l'_2$ if it contains $l_2$,
2) must go through $l_2$ before going through $l'_2$,
3) cannot have any cycle containing $l_2$ that does not contain $l'_2$ and vice versa.

These three points ensure that the activation counter is decremented exactly the same number of time it is incremented (hence the value of AC is restored) and is always incremented before it is incremented (hence AC is never reset). We show each point separately:

1) We show by contradiction that every path that contains $l_2$ in $[l_1, l'_1]$, also contains $l'_2$. Assume that there is a path from $l_1$ to $l'_1$ that contains $l_2$ but does not contain $l'_2$, which means $l'_1 \neq l'_2$. Because $l'_2$ postdominates $l_2$ we must have $l'_2$ postdominates $l'_1$, but because $l'_2$ is contained in $[l_1, l'_1]$ we also have that $l'_1$ postdominates $l'_2$. This can only happen if $l'_1 = l'_2$ which is a contradiction;
2) We show by contradiction that every path from $l_1$ to $l'_2$ in $[l_1, l'_1]$ also goes through $l_2$. Assume that there is a path from $l_1$ to $l'_2$ in $[l_1, l'_1]$ that does not contain $l_2$, which means $l_1 \neq l_2$. Because $l_2$ dominates $l'_2$ we must have $l_2$ dominates $l_1$, but because $l_2$ is contained in $[l_1, l'_1]$, we also have that $l_1$ dominates $l_2$. This can only happen if $l_1 = l_2$ which is a contradiction;
3) Finally, there cannot be any cycle in $[l_1, l'_1]$ containing $l_2$ that does not contain $l'_2$ (and vice versa) because $[l_2, l'_2]$ is an SESE region, and therefore any cycle containing $l_2$ also contains $l'_2$ (and vice versa).

□

Now, we show that big-step evaluation of functions preserve activation configurations:

**Lemma 4.** *For any function* $f$, *activating region* $[l, l']$, *and configuration* $\sigma$ *that has mimicry mode enabled for* $[l, l']$, *if* $[l_f, l'_f]\,\sigma \Downarrow \sigma'$ *then* $\sigma.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle = \sigma'.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle$ *and mimicry mode is not disabled during the evaluation of* $f$.

*Proof sketch.* Let $f$ be a function where $l_f = \mathit{entry}(f)$ and $l'_f = \mathit{exit}(f)$, $[l, l']$ an activating region, and $\sigma$ be a configuration such that $\texttt{AC} > 0$, $\texttt{En} = l$ and $\texttt{Ex} = l'$. We show that $[l_f, l'_f]\,\sigma \Downarrow \sigma'$ preserves the activation configuration and does not reset AC. The proof goes by induction on the depth of function calls.

**Base case.** First, consider the case where a function $f$ does not contain any function call (the call depth of $f$ is

0). Notice that by Hypothesis 2, we have that $[l, l']$ is an SESE region; which means that it is either contained in $f$ or belongs to another function.

- $[l, l']$ does not belong to $f$: from Hypothesis 1, we have that $[l_f, l'_f]\,\sigma \Downarrow_o \sigma'$ does not reach $l$ or $l'$. Hence, from Proposition 3 we have that AC, En, and Ex are not modified.
- $[l, l']$ is contained in $f$: in this case, Lemma 3 can be applied because $[l_f, l'_f]$ in an SESE region that does not contain function calls, and $[l, l']$ is contained in $[l_f, l'_f]$. Hence, it follows from the application of Lemma 3 that if $[l_f, l'_f]\,\sigma \Downarrow_o \sigma'$, then $\sigma.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle = \sigma'.\langle \texttt{AC}, \texttt{En}, \texttt{Ex}\rangle$ and mimicry mode is not disabled during the evaluation of $f$.

To conclude, the activation configuration is restored to its initial value after $[l_f, l'_f]\,\sigma_f \Downarrow_o \sigma'$ and mimicry mode is not disabled.

**Inductive case.** Assume that functions with call depth $n-1$ or below preserve the activation configuration and do not reset AC. We show that any function with call depth at most $n$ also preserve the activation configuration and do not reset AC. Assume a function $f$ with call depth at most $n$; it means that the depth of function calls in $f$ is at most $n-1$ or below. It follows from the induction hypothesis that:

Function calls in $f$ preserve the activation configuration and do not disable mimicry mode. (IH)

The rest of the proof is similar to the base case: if $f \neq g$ then $[l_f, l'_f]\,\sigma_f \Downarrow_o \sigma'$ does not reach $[l, l']$ and the activation configuration is not modified; otherwise, the proof follows by application of Lemma 3 (which applies with (IH)). □

Finally, we show Proposition 2:

**Proposition 2** (Well-behaved activating regions)**.** *For any activating region $[\ell, \ell']$ and valid configuration $\sigma$ such that $[\ell, \ell']\,\sigma \Downarrow_o \sigma'$, if AC is set after executing the instruction at location $\ell$:*

1) *it remains set during the execution of $[\ell, \ell']$ (including recursive function calls but excluding $\ell'$), and*
2) *if AC is incremented at location $\ell$, it is restored to its initial value right before the evaluation of the instruction at location $\ell'$.*

*Proof sketch.* Consider a valid configuration $\sigma$ such that $[l, l']\,\sigma \Downarrow_o \sigma'$ and mimicry mode is set after executing the instruction at location $l$. Let $\langle \texttt{AC}', \texttt{En}', \texttt{Ex}'\rangle$ be the corresponding activation configuration where $\texttt{AC}' > 0$. From Hypothesis 2, we have that $[l, l']$ is a SESE region; let $f$ be the function that contains $[l, l']$. From Lemma 4, we know that evaluation of nested functions during the evaluation of $[l, l']$ (including recursive function calls) do not reset AC and restore its value before returning. Therefore, we can ignore the evaluation of nested functions and, because we assume a sound CFG (cf. Definition 2), we can reason about the big-step evaluation of $[l, l']$ as a sequence of small steps following a path in $cfg(f)$, starting at $l$ and ending as soon as $l$ is reached. We need to show that: 1) AC is not reset during the whole execution of $[l, l']$ ($l'$ excluded), 2) it is restored to its previous value when executing the instructions at location $l'$. Notice that, if we show that AC is not reset during the whole execution of $[l, l']$, we can also consider than En and Ex are not modified and remain set to $\texttt{En}'$ and $\texttt{Ex}'$ (cf. Proposition 3). We first consider the case where $\texttt{En}' = l$ and then the case where $\texttt{En}' \neq l$.

**Case $\texttt{En} = l$.** Because $\sigma$ is a valid configuration (and so is its successor), we know from Proposition 4 that $[\texttt{En}', \texttt{Ex}']$ is an activating region and therefore $[\texttt{En}', \texttt{Ex}'] = [l, l']$. From AMi semantics (cf. Figs. 4 and 5), it follows that $\texttt{AC}' = \sigma.\texttt{AC} + 1$. Thus, it suffices to show that:

1) there is no path in the region $[l, l']$ that goes through $l$ twice, hence AC is not incremented again,
2) the evaluation of the instruction at location $l'$ decrements AC to its previous value, $\sigma.\texttt{AC}$.

The first point follows from the fact that $[l, l']$ is a SESE region and thus the program has no cycle containing $l$ which does not also contains $l'$; hence the evaluation of $[l, l']$ cannot go through $l$ twice before ending in $l'$. The second point follows from the fact that during the final step, (i.e., the evaluation of the instruction at location $l$), because $\texttt{Ex}' = l'$, AC is decremented and therefore restored to its previous value (cf. AMi evaluation rules in Figs. 4 and 5).

**Case $\texttt{En}' \neq l$.** In this case, AC is already set before entering $[l, l']$ then we have from Proposition 4 that $[\texttt{En}', \texttt{Ex}']$ corresponds to another (distinct) activating region. From Lemma 2, we can consider the following cases:

1) They belong to distinct functions or are disjoint: in this case, evaluating the region $[l, l']$ cannot go through $\texttt{En}'$ or $\texttt{Ex}'$, meaning AC is not modified (Proposition 3);
2) Regions are adjacent and $l = \texttt{Ex}'$: in this case, the initial AC is decremented to $\texttt{AC}' > 0$ during the evaluation of the instruction at location $l$ and we show that it cannot be modified during the remaining execution of $[l, l']$. Indeed AC can only be modified when reaching $\texttt{En}'$ or $\texttt{Ex}'$ (cf. Proposition 3) but $\texttt{En}'$ is not contained in $[l, l']$ and, because $[l, l']$ is an SESE region and there is no cycle containing $l$ that does not contain $l'$, the evaluation of $[l, l']$ cannot go through $\texttt{Ex}'$ twice;
3) Regions are adjacent and $l' = \texttt{En}'$: AC cannot be modified until the evaluation of the instruction at location $l'$. This follows from the fact that, because regions are adjacent with $l' = \texttt{En}'$, $\texttt{Ex}'$ is not contained in $[l, l']$.
4) $[l, l']$ is contained in $[\texttt{En}', \texttt{Ex}']$ and $l' \neq \texttt{Ex}'$: in this case, $[l, l']$ does not contain $\texttt{En}'$ nor $\texttt{Ex}'$, meaning (cf. Proposition 3) that AC is not modified;
5) $[l, l']$ is contained in $[\texttt{En}', \texttt{Ex}']$ and $l' = \texttt{Ex}'$: in this case, the evaluation of $[l, l']$ does not contain $\texttt{En}'$ and cannot modify AC before $l'$. Notice that at location $l'$, AC is decremented but this does not violate our definition of well-behavedness as this is also the end of the activating region $[l, l']$;
6) $[\texttt{En}', \texttt{Ex}']$ is contained in $[l, l']$: this can happen in case of a recursive call to $f$. The fact that AC is restored to its previous value after $[l, l']$ follows from Lemma 3.

□

## Appendix D.
## Meta-Review

### D.1. Summary

The paper proposes hardware support, an ISA extension, and a programming model to achieve resilience to control-flow based side-channels. The core novelty are so-called mimic instructions, which have the same timing behaviour as their standard counterparts but do not have any architectural effects. While different software-level side channel mitigation methods are proposed, they were not extensively adopted in reality; often, only patching some manually discovered critical code components. This work, with its software-hardware co-design approach, may establish new and practical research directions in this field. The paper provides a formalization of the core ideas for providing resilience to control-flow side channel leakages and attacks.

### D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Establishes a New Research Direction
- Independent Confirmation of Important Results with Limited Prior Research
- Provides a Valuable Step Forward in an Established Field

### D.3. Reasons for Acceptance

1) The paper inches us forward on the problem of branch balancing.
2) The paper fleshes out the entire design from idea to semantics to implementation.
3) The paper describes how to handle a number of interesting program constructs (e.g., function calls, recursion).
4) The paper provides a basis to improve upon (and likely composes with) state-of-the-art. For example, it may enable Raccoon-style transactions [14] where only a subset of transaction instructions need to have their architectural state rolled back.

### D.4. Noteworthy Concerns

1) There were some concerns around the performance measurement numbers.
2) The security evaluation lacks details and has a lot of scope for improvement.
3) The paper only deals with control-flow leakage. This is on the narrow side. Further, the line between control-flow leakage and other forms of leakage (e.g., see Listing 4) is subtle because microarchitectural state updates and architectural state updates are often entangled.
4) Compiling code to use the proposed ISA extension seems difficult to automate.

## Appendix E.
## Response to the Meta-Review

In this section, we address the concerns outlined in the Meta-Review above.

**Performance Evaluation (1).** Because compiler support for AMi is missing, the benchmark programs are manually hardened at assembly level, inherently restricting their size. More extensive performance measurements should be conducted once a compiler is available. Moreover, the size of the benchmarks sometimes causes performance gains, the reason for which we explain in the last paragraphs of Section 6.3. To facilitate the reproduction and extension of our performance evaluation, we open source both our hardware implementation and the benchmarks used in the paper.

**Security Evaluation (2).** Verifying hardware security is a challenging problem and it is also an active area of research. With this paper, it is not our goal to make a contribution to this area. Our automated evaluation just provides evidence that the benchmark programs secured with our hardware implementation of AMi indeed close the vulnerabilities that were present (as explained in 6.3). We also open source our security evaluation.

**Scope (3).** We agree with the observation that the paper only deals with control-flow leakage. We also acknowledge that the architectural/microarchitectural entanglement issue is indeed present. However, this is not only an issue for AMi but for all competing control-flow linearization approaches. It is something that the developer should address based on an implementation-specific leakage specification. AMi, however, makes it possible to efficiently deal with this entanglement using the persistent instructions. We refer to Section 5.2 for more details.

**Compiler Support (4).** We agree that automating compilation to AMi is a very interesting challenge, and it will be an exciting topic to explore in future work.