Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors

Hans Winderix
hans.winderix@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Marton Bognar
marton.bognar@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Lesly-Ann Daniel
lesly-ann.daniel@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Frank Piessens
frank.piessens@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

ABSTRACT

Control-flow leakage (CFL) attacks enable an attacker to expose control-flow decisions of a victim program via side-channel observations. *Linearization* (i.e., elimination) of secret-dependent control flow is the main countermeasure against these attacks, yet it comes at a non-negligible cost. Conversely, *balancing* secret-dependent branches often incurs a smaller overhead, but is notoriously insecure on high-end processors. Hence, linearization has been widely believed to be *the only* effective countermeasure against CFL attacks. In this paper, we challenge this belief and investigate an unexplored alternative: how to securely balance secret-dependent branches on higher-end processors?

We propose Libra, a generic and principled hardware-software codesign to efficiently address CFL on high-end processors. We perform a systematic classification of hardware primitives leaking control flow from the literature, and provide guidelines to handle them with our design. Importantly, Libra enables secure control-flow balancing without the need to disable performance-critical hardware such as the instruction cache and the prefetcher. We formalize the semantics of Libra and propose a code transformation algorithm for securing programs, which we prove correct and secure. Finally, we implement and evaluate Libra on an out-of-order RISC-V processor, showing performance overhead on par with insecure balanced code, and outperforming state-of-the-art linearized code by 19.3%.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; **Formal security models**; **Information flow control**.

KEYWORDS

Side Channels, Control-Flow Leakage, HW/SW Leakage Contracts, HW/SW Codesign, Secure Compilation, Control-Flow Balancing

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM Reference Format:
Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690319

1 INTRODUCTION

In recent years, software-based microarchitectural attacks [34, 60] have emerged as a critical security threat. When multiple stakeholders run code on the same computing device, this type of side-channel attack makes it possible for an attacker to infer program secrets just by monitoring from software how a victim uses shared hardware such as the cache, branch predictor, or prefetcher.

Of special interest to this work are so-called *control-flow leakage (CFL) attacks* [17, 24, 55, 66, 76, 86, 101] whereby an attacker tries to expose the program counter (PC) trace of a victim program via side-channel observations with the aim of revealing the outcome of conditional control-flow decisions. The program's conditional control flow exposes the outcome of the condition that determines the control flow, which poses a security threat if that condition depends on secret information. In the presence of a microarchitectural attacker, a program's control flow can, in general, be observed in the microarchitectural state of shared hardware or through contention.

A possible software countermeasure against CFL attacks is *control-flow balancing* [4, 18, 28, 53, 75, 94], a program transformation which aims to make the execution of all possible targets of a control-transfer instruction appear the same to an attacker. So far, control-flow balancing has been shown to be secure only for a class of low-power embedded processors [18, 94]. This is because modern superscalar processors feature critical performance-enhancing hardware that maintains state as a function of the PC, thus leaking the PC in an unbalanceable way when this hardware is shared between different security domains. For this reason, it is widely accepted that, to counter CFL attacks on higher-end processors, programs must be PC-secure [66], i.e., their PC should be independent from secret information. PC-secure programs are created by avoiding secret-dependent control flow and the techniques for doing so are well-documented in the literature [19, 66, 78, 87, 95].

Unfortunately, this advice has not been questioned much. Over the years, it has been evolving into a dogma and it has become

an established practice to hardcode it in *constant-time* [8] source code, preventing the adoption of more relaxed policies (for simpler architectures or for weaker attacker models). Furthermore, this trend creates the fallacy that secret-dependent control flow is inherently insecure and, consequently, it discourages the search for novel mechanisms to securely execute PC-insecure programs on higher-end processors.

On the other hand, there still exists a strong desire to keep the secret-dependent control flow for performance reasons, even on high-end processors. Vendors of cryptographic libraries, for instance, sometimes take the risk and do balance secret-dependent branches [101] instead of eliminating them. As another example, numerous offensive research papers have been published that develop new CFL attacks, accompanied by ad-hoc defenses, which are later found to be vulnerable by other offensive research, a trend that has been recently described as *the CFL arms race* [101].

Our Proposal. In this work, we challenge the widely-held belief that secret-dependent control flow is inherently insecure on high-end processors and propose a well-founded hardware-software codesign for secure and efficient balanced execution. In contrast to prior works that target a single vulnerability and propose ad-hoc, incremental defenses, we propose a principled solution that addresses the CFL problem in a generic way with the goal of ending the CFL arms race. Also in contrast to prior works, we do not assume a simple processor pipeline and scheduling but support modern out-of-order processor designs.

We conduct a rigorous analysis of how hardware optimizations leak a program's control flow. A key finding is that hardware optimizations can be partitioned into two categories; those that yield *balanceable observations* and those that yield *unbalanceable observations*. Balanceable observations can be securely balanced by software-only approaches. Unbalanceable observations require hardware support. Based on the findings of our analysis, we propose Libra, a hardware-software security contract that lays the principled foundation for secure balanced execution. We introduce a novel memory layout, called *folded layout*, and an algorithm for *folding* balanced code regions, which makes it possible to keep enabled performance-critical hardware optimizations without compromising security. Additionally, we propose an ISA extension for executing folded regions.

In a nutshell, we make the following contributions:

- A novel hardware-software contract, called Libra, for secure and efficient balanced execution (Section 4).
- A formalization of the ISA-level semantics of Libra and security and correctness proofs of our folding algorithm (Section 5).
- A characterization of hardware optimizations regarding how they leak a program's control flow (Section 6).
- Recommendations for hardware designers wishing to adopt Libra to their designs (Section 6).
- An implementation of Libra on an out-of-order RISC-V core (Section 7.1).
- An experimental evaluation showing that balanced execution is secure and efficient at a low hardware cost (Section 7.2).

```
En: br a0,t,f
 t:    br a1,tt,tf
tt:     add s1,s2,s3
        j Ex
tf:     add s2,s3,s4
        j Ex
 f:    sub s1,s2,s3
       j Ex
Ex: [...]
```

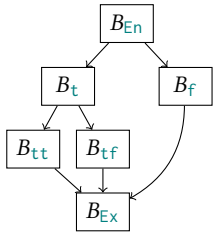

Figure 1: A program and its CFG.

Additional material. Our RISC-V implementation and evaluation are archived on Zenodo [91] and available on GitHub: https://github.com/proteus-core/libra. The proofs of Section 5 are available in the companion technical report [92].

2 TERMINOLOGY AND BACKGROUND

2.1 Terminology

We first define relevant terminology from the fields of graph theory and compiler construction and then introduce some new vocabulary (marked with $*$).

Definition 1 (Basic block)**.** A basic block is a straight-line instruction sequence always entered at the beginning and exited at the end.

Definition 2 (Control-flow graph)**.** A control-flow graph (CFG) is a directed graph that represents all the paths that might be traversed through a program during its execution. The nodes of a CFG represent basic blocks, the edges represent control-flow transfers.

Without loss of generality, we assume that a CFG has a unique *entry* and a unique *exit* block. We also assume that the last instruction in a basic block is a control-transfer instruction, which designates the possible successor blocks. We refer to this instruction as the *terminating instruction* of the basic block. Figure 1 contains an illustration of a CFG with B_{En} the entry basic block and B_{Ex} the exit basic block.

Definition 3 (Distance)**.** The distance between two basic blocks in a CFG is the number of edges in a shortest path connecting them.

In Figure 1, the distance between the basic blocks B_{En} and B_{Ex} is 2 ($B_{En} \rightarrow B_f \rightarrow B_{Ex}$). The distance between two instructions is defined similarly by considering individual instructions as basic blocks.

Definition 4 (Postdominance)**.** A basic block Y postdominates a basic block X (i.e., Y is a postdominator of X) if all paths from X to the exit block go through Y.

The closest postdominator of a basic block is called its *immediate postdominator*. In Figure 1, basic block B_{Ex} postdominates basic block B_{En}. It is also the immediate postdominator of B_{En}.

Definition 5 (Level structure)**.** The level structure of a CFG is a partition of the basic blocks into subsets (levels) that have the same distance from the entry basic block.

The level structure of the CFG in Figure 1 consists of three levels: $L_0 = \{B_{En}\}, L_1 = \{B_t, B_f\}, L_2 = \{B_{tt}, B_{tf}, B_{Ex}\}$.

Definition 6 (∗Level slice)**.** The set of equidistant instructions for a distance δ with respect to basic block B forms the level slice (or simply *slice*) determined by the tuple (B, δ)

In Figure 1, the slice of distance 0 is {`br a0,t,f`} and the slice of distance 1 is {`br a1,tt,tf; sub s1,s2,s3`} (both relative to $B_{\texttt{En}}$).

Definition 7 (∗Secret-dependent region)**.** The set of basic blocks between a secret-dependent control-transfer instruction *inst* and its immediate postdominator form the secret-dependent region determined by *inst*.

We refer to the basic block containing the secret-dependent control-transfer instruction as the *entry block* of the region, and to its immediate postdominator as the *exit block* of the region. In Figure 1, if `a1` is secret (line `t`), then $\{B_{\texttt{tt}}, B_{\texttt{tf}}\}$ is the secret-dependent region determined by the instruction on line `t`. The entry block of the region is $B_{\texttt{t}}$, the exit block $B_{\texttt{Ex}}$. Similar to the level structure of a CFG, we define the *level structure of a secret-dependent region* as the partition of its basic blocks into subsets (levels) that have the same distance from the region's entry block.

2.2 The Control-Flow Leakage Problem

2.2.1 Control-Flow Leakage Attacks. CFL attacks are a type of microarchitectural attack whereby an attacker tries to learn the outcome of a secret-dependent branch by exposing the control flow via microarchitectural side channels. Consider the program in Listing 1a. When the branch on line 1 evaluates to true, the instructions on lines 2-3 are executed and the program exits. When the branch evaluates to false, the instruction on line 4 is executed and the program exits. An attacker that is able to observe the program's execution time will be able to distinguish the two executions, and hence learn if `secret` evaluates to true or false.

Listing 1: Code vulnerable to CFL attacks (Listing 1a) and its balanced version (Listing 1b).

```
1      br secret,t,f
2   t:   add s1,s2,s3
3        j Ex
4   f:   add s2,s3,s4
5
6  Ex: [...]
```

(a)

```
1      br secret,t,f
2   t:   add s1,s2,s3
3        j Ex
4   f:   add s2,s3,s4
5        j Ex
6  Ex: [...]
```

(b)

Besides this start-to-end timing difference, interrupt latency [86], data cache contention [69], structural dependencies [7] or data dependencies stalling the pipeline are other examples of microarchitectural events that can be monitored by an attacker to leak the control flow. Consider Listing 1a again and assume that the addresses of the `add` instructions (lines 2 and 4) map to different instruction cache lines. Monitoring which cache line has been touched (for instance with the Flush+Reload attack [98]) will reveal the control flow.

Two common software countermeasures against CFL attacks are *control-flow balancing* and *control-flow linearization*. The former technique keeps the secret-dependent control flow intact while the latter eliminates it completely.

2.2.2 Control-Flow Balancing. Control-flow balancing is based on the idea that if the two sides of a secret-dependent branch induce exactly the same attacker-observable behavior, then executing the code does not reveal via side channels which side of the branch has been executed. Listing 1b gives the balanced form of Listing 1a. The `add` instruction on line 2 is balanced with the `add` instruction on line 4 and a jump instruction is added to the `f` path on line 5 to balance it with the jump on line 3 in the `t` path.

Recent work [18, 94] has demonstrated the security (and efficiency) of control-flow balancing for small, embedded processors with deterministic timing behavior. The authors propose a methodology consisting of three steps. First, by profiling the microarchitecture, the instruction set is classified into a number of *leakage classes* such that executing instructions from the same leakage class induces the same side-channel observations. Second, a dummy (no-op) instruction is composed for every leakage class. Lastly, the secret-dependent branches are algorithmically balanced [94] with respect to the leakage classification, and by inserting dummy instructions when necessary. This approach ensures that the dynamic instruction trace of balanced code always produces the same sequence of leakage classes.

Although control-flow balancing counters attacks exploiting microarchitectural optimizations on low-end devices [64, 86], higher-end devices (the target of our work) typically feature optimizations yielding observations that are unbalanceable in software alone. Yet, for performance reasons, balanced control flow is sometimes found in security-critical libraries targeting these devices [64, 101]. Thus, how to make balanced execution secure on these higher-end devices remains an important research question.

2.2.3 Control-Flow Linearization. Control-flow linearization is a key principle of the widely-established constant-time programming discipline [8]. By eliminating secret-dependent branches, control-flow linearization ensures that the PC does not get tainted (i.e., that the PC trace is independent of secrets). Several linearization techniques have been proposed in the literature [19, 66, 78, 83, 87, 95]. Listing 2 contains the linearized form of the running example from Listing 1a, based on a state-of-the-art method that was first proposed by Molnar et al. [66]. Compared with the balanced form from Listing 1b, the linearized form comes with a higher cost due to the use of additional instructions and registers.

Listing 2: Linearized form of the vulnerable code in Listing 1a.

```
1  seqz t1,secret
2
3  addi t1,t1,-1  # t1 = true mask  (in {0xffff, 0x0000})
4  not  t2,t1     # t2 = false mask (in {0xffff, 0x0000})
5  and  t3,s1,t1  # start of else
6  add  s1,s2,s3
7  and  s1,s1,t2
8  or   s1,s1,t3  # start of then
9  and  t3,s2,t2
10 add  s2,s3,s4
11 and  s2,s3,t1
12 or   s2,s2,t3
```

2.2.4 This paper. The goal of this work is to make sure that executing balanced code (which contains secret-dependent control flow) on high-end processors does not leak more information than executing the equivalent linearized code (which does *not* contain secret-dependent control flow). We demonstrate that, with minimal hardware support, it is possible to securely balance secret-dependent control flow on higher-end platforms, without disabling

performance-critical hardware resources that are shared between different stakeholders.

3 THREAT MODEL

We consider an adversary with the goal to infer secrets (e.g., cryptographic keys) by learning the secret-dependent control flow of a victim application. We consider an adversary with the same capabilities as an adversary under the *classic* constant-time threat model, and thus assume that applications are hardened against transient execution attacks [21]. More specifically, an adversary with the capabilities of this threat model is able to run arbitrary code alongside an architecturally isolated victim (e.g., via process isolation) on the same machine and it shares hardware resources, such as the branch predictor, cache hierarchy and execution units with the victim. This setting enables the adversary to precisely observe the execution time of the victim, and how it uses the shared resources. If these observations depend on the secret control flow, the adversary is able to learn something about the secret.

We consider software-based timing channels, i.e., the adversary monitors the microarchitectural resource usage via timers from software [34, 60]. Side channels that require physical access and physical equipment to measure quantities such as power consumption [52] or EM emissions [77] are out of scope for this paper. Similarly, other types of software-based side-channel attacks, such as software-based fault attacks [68] and software-based power attacks [58] are out of scope and subject of orthogonal mitigations.

We make no further assumptions on the type of (software-based) microarchitectural side-channels attacks that can be mounted by the adversary, ranging from classic cache attacks [69] to more recent contention-based attacks [7].

4 OVERVIEW OF LIBRA

A program's control flow can leak through observations induced by various microarchitectural optimizations. Some of these observations, such as instruction latency, are independent of the value of the PC. We refer to optimizations yielding this type of observation as *sources of balanceable leakage* as their observations can be balanced by software. However, some performance-critical optimizations commonly found in modern hardware (e.g., the instruction cache and the instruction prefetcher) yield observations that are dependent on the value of the PC. They inevitably leak the control flow. We refer to these optimizations as *sources of unbalanceable* leakage as they cannot be dealt with by software alone. In Section 6, we study this distinction further and provide a comprehensive characterization of hardware optimizations regarding how they leak the control flow.

Existing control-flow balancing solutions are ineffective against unbalanceable leakage. It is the goal of Libra to address this gap via a novel hardware-software security contract for secure and efficient balanced execution. On the one hand, the software is responsible for balancing secret-dependent control flow under a *weak observer mode* (accounting for the balanceable leakage) in which the PC does not leak. On the other hand, the hardware provides support to deal with the sources of unbalanceable leakage to ensure that the program remains secure in a *strong observer mode*, representative of our threat model (Section 3) for high-end processors.

4.1 Leakage Contract

Libra requires the hardware to augment the ISA with a *leakage contract* that provides sufficient information on how to balance the control flow. Software, such as a compiler, can then rely on this contract 1) to securely balance secret-dependent control flow (making control-flow balancing a *principled* code transformation) or 2) to verify that secret-dependent control flow is securely balanced. This stands in contrast to prior works [4, 18, 28, 53, 75, 94], where it is the responsibility of the software to empirically figure out *how* to balance corresponding instructions.

The Libra leakage contract classifies an instruction set into two dimensions. First, it partitions instructions into *leakage classes* [18, 94] such that instructions from the same leakage class yield identical side-channel observations. Importantly, any instruction can be used to balance any other instruction from the same leakage class. For every leakage class, the contract additionally designates a canonical *dummy instruction*, which does not produce architectural effects (e.g., `mv x1, x1`). Finally, the hardware provides a blocklist of instructions that are not supported in balanced regions. Blocklisted instructions have to be rewritten in terms of non-blocklisted instructions before performing control-flow balancing.

Second, the leakage contract partitions the instruction set into safe and unsafe instructions [100]. *Safe instructions* are instructions whose timing and shared microarchitectural resource usage are independent of the values of their operands. For instance, an `add` instruction is typically implemented in a safe way, while a `load` typically exposes the value of the address operand on systems with a data cache (making it an *unsafe instruction*). It is insecure to pass secrets to unsafe instructions but it is secure to use unsafe instructions in balanced regions if it can be proven that the operands of any two equidistant unsafe instructions are the same for all possible executions. For instance, the code `if (secret) load x0 a else load x1 a` is secure as the resulting observation is independent of `secret` (under the assumption that the `load` is only unsafe in its address operand).

4.2 ISA Extension

The goal of Libra is to securely execute balanced code regions on high-end CPUs without disabling performance-critical optimizations. In particular, Libra aims at keeping *all* modern hardware optimizations fully enabled when executing security-insensitive code (i.e., the common case), and keeping *as many optimizations as possible* in secret-dependent regions.

To this end, Libra proposes an ISA extension introducing two main novel features:

- A novel memory layout for balanced code, termed *folded layout*, which interleaves the instructions from balanced regions by placing the level slices sequentially in memory.
- A new instruction, the *level-offset branch* (**`lo.br`**), which informs the CPU how to navigate a folded region. Additionally, it signals to the CPU that it is about to execute a secret-dependent region such that it can adapt the behavior of some optimizations.

Importantly, even though folding sequentially lays out instructions of balanced regions in memory (reminiscent of linearization), the *original control flow of the program is preserved*, i.e., only one

side of a folded conditional branch is executed, as prescribed by the original CFG (just like with standard code balancing).

The level-offset branch **lo.br** $c, \texttt{off}_t : \texttt{off}_f : \texttt{bbc}$ specifies how to navigate a folded region:

(1) The level offsets $\texttt{off}_t$ and $\texttt{off}_f$ indicate what instructions of the next level to execute, depending on whether the condition c is true or false;
(2) The basic block count bbc indicates the number of basic blocks of the next level (the slice size of the next level) and is used to increment the PC by the correct value.

Listing 3 illustrates how to fold the balanced code from Listing 1b. First, the two add and the two j instructions are sequentially placed in memory. Second, the conditional branch is rewritten using a **lo.br** with $\texttt{off}_t = 0$, $\texttt{off}_f = 1$ and $\texttt{bbc} = 2$. After the **lo.br**, the CPU will execute the folded region slice by slice, incrementing the PC by 2. If the condition is true, the first (offset $\texttt{off}_t$) instruction of each slice is executed, otherwise the second (offset $\texttt{off}_f$) instruction is executed. Finally, the terminating j instructions are replaced by **lo.br** instructions to reset the level offset and bbc and resume "normal" execution at the Ex label.

Listing 3: Folded form of the balanced code in Listing 1b.

```
      lo.br secret,0:1:2    # offT:offF:bbc
L1:     add s1,s2,s3
        add s2,s3,s4
        lo.br zero,0:0:1    # offT:offF:bbc
        lo.br zero,0:0:1    # offT:offF:bbc
Ex: [...]
```

How does Libra address unbalanceable leakage? The design of Libra is tailored to address unbalanceable leakage in hardware efficiently, i.e., by keeping essential hardware optimizations enabled. Yet, to establish the security guarantees, Libra requires that the PC does not leak at a finer granularity than a slice, possibly requiring adaptations to the behavior of some optimizations.

Importantly, the folded memory layout is crucial to keep enabled performance-critical optimizations of modern hardware (e.g., the instruction cache) without, or with only minimal, adaptations. By virtue of folding (which creates a linear memory layout), the hardware can efficiently implement a data-oblivious instruction memory access pattern by always prefetching all the slices in the same order, effectively making it independent of the outcomes of conditional branch(es).

While some sources of unbalanceable leakage do not require hardware modifications, some will, possibly degrading performance. However, because the hardware is informed when it is executing a folded region, these modifications can be limited to folded regions only. For instance, some hardware structures, such as the branch predictor, must be disabled for the **lo.br** instruction to prevent control-flow exposure to an attacker sharing the branch predictor. However, the linear layout of a folded region makes the branch predictor unnecessary for **lo.br** instructions, because there is no uncertainty (at slice granularity) what address the sequential prefetcher should fetch from, so it can fill the cache with the instructions that are about to be fetched by the CPU.

In Section 6, we present, based on a rigorous study of the attack literature, a characterization of the sources of unbalanceable leakage (with folding in mind), and we provide guidelines about how to handle them.

4.3 Advanced Features

4.3.1 Nested branches. When folding a region with a nested branch (as in Listing 4a), the software must fold the level structure of the entire outer region, as shown in Listing 4b. The slice size grows with the level of nesting. In the example from Listing 4b, each slice of the second level consists of four instructions. Recall that the hardware has to make sure to fetch instructions without exposing their offset within the current level. For instance, if a slice occupies multiple cache lines, the hardware must ensure to always touch all the cache lines in the same order, irrespective of the current instruction's offset.

Listing 4: Region with nested branches (Listing 4a) and its folded version (Listing 4b).

```
    br secret,t,f
 t:  br c,tt,tf
tt:    add r,r,4
       j Ex
tf:    add r,r,8
       j Ex
 f:  br c,ft,ff
ft:    sub r,r,4
       j Ex
ff:    sub r,r,8
       j Ex
Ex: [...]
```

(a)

```
    lo.br secret,0:1:2
L1:   lo.br c,0:1:4
      lo.br c,2:3:4
L2:     add r,r,4
        add r,r,8
        sub r,r,4
        sub r,r,8
        lo.br zero,0:0:1
        lo.br zero,0:0:1
        lo.br zero,0:0:1
        lo.br zero,0:0:1
Ex: [...]
```

(b)

Note that when a nested branch does not depend on secret information (e.g., a loop with a constant trip count), it can be more efficient to keep the branch instead of folding it. In that case, for correctness, the software must ensure that the level offsets of the target instructions are consistent regarding the offsets of the branch instructions. Moreover, for security, the software must ensure that the branch targets of the branches in the source slice all point to targets in the same target slice.

4.3.2 Function calls. To support function calls in balanced code, prior work on control-flow balancing [18, 94] proposed to create a dummy function for each function called from a secret-dependent region. A dummy function is mostly made up of dummy (no-op) instructions designed to mirror the behavior of the real function. These dummy instructions ensure that both the dummy and real functions cause identical changes in the microarchitectural state. As a result, an attacker cannot distinguish between the execution of the dummy function and that of the real function. A call to a function in a secret-dependent region can then be balanced with a call to its dummy version. Libra supports this scheme, yet in order not to expose the control flow on higher-end CPUs (e.g., via the instruction cache), functions must be folded with their dummy counterpart. Libra provides hardware support to efficiently invoke a folded function and extends the ISA with a new instruction, the *level-offset call*: **lo.call** $b\ \ell$. The instruction jumps to the folded function and, according to the boolean immediate b, either executes the real part or the dummy part of the folded function. Additionally, the CPU must save/restore the Libra state (i.e., current offset and bbc) of the caller upon calls/returns. Libra proposes a two-level

hardware stack, used for storing and restoring the Libra state of the caller. For non-leaf functions (i.e., to support more than one level of nesting, including recursion), the software is responsible to save and restore the Libra state on a software-based stack.

4.3.3 Exceptions. Instructions that may throw exceptions are inherently unsafe because whether an exception is thrown depends on the value of their operands and handling an exception impacts both the timing and resource usage of an application. Therefore, such instructions should be treated similarly to other unsafe, balanceable instructions, by balancing the unsafe operands and their dependencies.

4.4 Hardware-Software Security Contract

In summary, with Libra we propose a hardware-software security contract for balanced execution. If both parties fulfill their part of the contract, then executing a balanced code region will not leak more information than the equivalent linearized region.

On the hardware side, Libra imposes the following requirements:

HR1 A leakage contract for control-flow balancing is provided.
HR2 The PC does not leak at a finer granularity than a slice.
HR2a The instruction memory access pattern does not depend on the outcome of the level-offset branch (implied by **HR2**).
HR3 The level-offset branch and the level-offset call are safe instructions.

On the software-side, Libra relies on:

SR1 A correct identification of secret-dependent regions and functions called from secret-dependent regions.
SR2 A secure balancing according to a weak observer mode as prescribed by the leakage contract. In practice, this entails making sure that secrets do not directly flow to unsafe instructions, applying a balancing algorithm (such as the one from [94]), and providing dummy versions for functions called from secret-dependent regions.
SR3 A correct folding of the balanced regions and functions. In Section 5.3, we give a folding algorithm.

5 FORMAL SEMANTICS

5.1 Language and Semantics

5.1.1 Language. The Libra folding transformation transforms programs written in a source assembly language asm[1] to a target language asm (Figure 2).

Source language. In addition to standard ISA instructions, the source language asm is equipped with additional syntactic constructs to: *(1)* identify secret-dependent branches (**SR1**), and *(2)* associate functions that can be called in secret-dependent regions with a dummy version (**SR2**). These constructs should be seen as information derived from source-level annotations. Secret-dependent branches, `s.br` $c\ \ell_t\ \ell_f$, indicate that the condition c is secret and inform the Libra transformation about secret-dependent regions to fold. Their semantics is similar to regular conditional branches. Secret-dependent calls, `s.call` $b\ \ell\ \ell'$, indicate that the function

[1]Following common practice [71], we denote source objects with a blue, sans − serif font and target objects with a red, bold font. Objects common to source and target are written with black normal font.

(Values) $v \in \mathbb{V}$ (Registers) $\mathsf{x} \in \mathbb{R}$ (Labels) $\ell, \ell_t, \cdots \in \mathbb{L}$
$\langle exp \rangle$ ::= $v \mid \mathsf{x}$
$\langle inst \rangle$ ::= $\mathrm{op}_1\ \mathsf{x}\ \langle exp \rangle \mid \mathrm{op}_2\ \mathsf{x}\ \langle exp \rangle\ \langle exp \rangle \mid \mathrm{store}\ \langle exp \rangle\ \langle exp \rangle$
$\mid \mathrm{br}\ \langle exp \rangle\ \ell_t\ \ell_f \mid \mathrm{call}\ \ell \mid \mathrm{ret}$
$\langle \mathrm{inst} \rangle$ ::= `s.br` $\langle exp \rangle\ \ell_t\ \ell_f \mid$ `s.call` $b\ \ell\ \ell' \mid \langle inst \rangle$
$\langle \mathrm{inst} \rangle$ ::= `lo.br` $\langle exp \rangle\ v\ v\ v \mid$ `lo.call` $b\ \ell \mid \langle inst \rangle$

Figure 2: Syntax of asm and asm instructions where $\mathrm{op}_1 \in \{\mathrm{neg}, \mathrm{load} \dots\}$ and $\mathrm{op}_2 \in \{\mathrm{add}, \mathrm{mul}, \dots\}$ are non-control-flow-altering unary and binary instructions and $b \in \{\bot, \top\}$ is an immediate boolean operand. A program P is a partial mapping from locations to instructions and $P[\ell]$ denotes the instruction at location ℓ.

at address ℓ' is the dummy version of the function at address ℓ. If $b = \top$, the original function ℓ is called, whereas if $b = \bot$, the dummy function ℓ' is called. Secret-dependent calls inform the Libra transformation of functions to fold with their dummy version.

Target language. The target language is equipped with a level-offset branch and level-offset call, which are used to navigate folded regions and whose semantics will be detailed later.

5.1.2 Configurations. *Source configurations* are of the form $\langle m, r, \mathsf{pc}, \rho \rangle$ where $m : \mathbb{V} \to \mathbb{V}$ is a memory, mapping addresses to values, $r : \mathbb{R} \to \mathbb{V}$ is a register map, pc is the program counter, and ρ is a stack of return addresses. [2] To execute a folded region slice-by-slice, Libra keeps track of the number of basic blocks in the currently active level (bbc) and the offset of the currently active basic block (off) in a *Libra context*, denoted ctx = (bbc, off). The initial Libra context is $(1, 0)$. A *Libra configuration* σ is a tuple $\langle m, r, \mathsf{pc}, \rho, \lambda \rangle$ where $\langle m, r, \mathsf{pc}, \rho \rangle$ is a source configuration, and λ is a stack of Libra contexts. In the following, we refer to Libra configurations simply as configurations.

Note that handling function calls and exceptions in folded regions requires a stack of (at least) two Libra contexts. In that setting, Libra contexts must be saved and restored by the callee in non-leaf functions. For simplicity, our formalization allows for a stack of unlimited size.

5.1.3 Semantics. The semantics of Libra, given by the relation $\sigma \overset{o}{\Longrightarrow} \sigma'$, defines that the evaluation of an instruction in a configuration σ produces a configuration σ' and an observation o. The semantics is parameterized by a function $obs(\sigma)$, which defines the observation produced in a configuration σ (and will be instantiated in Section 5.2). We give in Figure 3 an excerpt of semantics rules, focusing on the important aspects of Libra i.e., the update of the program counter and the Libra context. The evaluation of an expression e using a register file r is given by $(\!|e|\!)_r$ and the evaluation of a non–control-transfer instruction *inst* (e.g., arithmetic, logic, or memory instruction), is given by a relation $\langle m, r \rangle \xrightarrow{inst} \langle m', r' \rangle$.

[2]For simplicity, our formalization features a stack of return addresses. However, a standard setting with a simple return address register that is correctly saved/restored on the stack would be equivalent, under the assumption that return addresses do not interfere with the rest of the program (i.e., no return address overwrite, no pointer arithmetic on return address, etc).

The program counter always points to the instruction to be executed ($P[\mathtt{pc}]$). To navigate the folded memory layout, we define a function *slice_addr*, returning the address of the current slice, and a function *next_slice*, returning the address directly following the current slice:

$$\mathit{slice\_addr}(\mathtt{pc}, \mathtt{off}) \triangleq \mathtt{pc} - \mathtt{off}$$
$$\mathit{next\_slice}(\mathtt{pc}, \mathtt{bbc}, \mathtt{off}) \triangleq \mathit{slice\_addr}(\mathtt{pc}, \mathtt{off}) + \mathtt{bbc}$$

The rule PC-UPDATE defines the evaluation of a non-control-transfer instruction. It increments the program counter with the basic block count, effectively jumping to current offset in the next slice.

The rule LOB-TRUE defines the evaluation of a level-offset branch $\mathtt{lo.br}\ e\ \mathtt{off}_f\ \mathtt{off}_t\ \mathtt{bbc}'$ when the condition e evaluates to true. It jumps to the next slice at offset $\mathtt{off}_t$ and sets the new basic block count to $\mathtt{bbc}'$. The rule LOB-FALSE is analogous and omitted for brevity.

The rule LO-CALL defines the evaluation of a level-offset call, which calls a function folded with its dummy version, at location ℓ. The rule jumps to the first slice of the function and, according to the boolean b, sets the offset to 0 or 1, to execute the original or the dummy function, respectively. It also sets the basic block count to 2, to account for the folding of the original and dummy functions. Finally, it pushes the return address on the return stack. Normal function calls are similar, but push the initial Libra context $(1, 0)$ to the Libra stack.

The rule RET defines the evaluation of a return instruction. It simply jumps to the return address on the top of the return stack and restores the previous Libra context.

PC-UPDATE
$$\frac{P[\mathtt{pc}] = \mathit{inst} \qquad \mathit{inst} \notin \{q.\mathtt{br}, q.\mathtt{call}, \mathrm{ret}\} \qquad \langle m, r\rangle \xrightarrow{\mathit{inst}} \langle m', r'\rangle \qquad \mathtt{pc}' = \mathtt{pc} + \mathtt{bbc}}{\langle m, r, \mathtt{pc}, \rho, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle \overset{o}{\Longrightarrow} \langle m', r', \mathtt{pc}', \rho, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle}$$

LOB-TRUE
$$\frac{P[\mathtt{pc}] = \mathtt{lo.br}\ e\ \mathtt{off}_t\ \mathtt{off}_f\ \mathtt{bbc}' \qquad (\!|e|\!)_r \neq 0 \qquad \mathtt{pc}' = \mathit{next\_slice}(\mathtt{pc}, \mathtt{bbc}, \mathtt{off}) + \mathtt{off}_t}{\langle m, r, \mathtt{pc}, \rho, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle \overset{o}{\Longrightarrow} \langle m, r, \mathtt{pc}', \rho, \lambda \cdot (\mathtt{bbc}', \mathtt{off}_t)\rangle}$$

LO-CALL
$$\frac{P[\mathtt{pc}] = \mathtt{lo.call}\ b\ \ell \qquad \mathtt{off}' = (\mathit{if}\ b = \top\ \mathit{then}\ 0\ \mathit{else}\ 1) \qquad \mathtt{pc}' = \ell + \mathtt{off}' \qquad \rho' = \rho \cdot \mathtt{pc} + \mathtt{bbc} \qquad \lambda' = \lambda \cdot (\mathtt{bbc}, \mathtt{off}) \cdot (2, \mathtt{off}')}{\langle m, r, \mathtt{pc}, \rho, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle \overset{o}{\Longrightarrow} \langle m, r, \mathtt{pc}', \rho', \lambda'\rangle}$$

RET
$$\frac{P[\mathtt{pc}] = \mathrm{ret} \qquad \mathtt{pc}' = \ell}{\langle m, r, \mathtt{pc}, \rho \cdot \ell, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle \overset{o}{\Longrightarrow} \langle m, r, \mathtt{pc}', \rho, \lambda\rangle}$$

Figure 3: Excerpt of the Libra semantics, where $q \in \{\mathtt{lo}, \mathtt{s}, \varepsilon\}$ and $o = \mathit{obs}(\langle m, r, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle)$.

We additionally equip our source language asm with a source semantics $\overset{o}{\Longrightarrow}$, defined in a standard way and omitted here for brevity. Finally, we let $\sigma \overset{o}{\Longrightarrow}^n \sigma'$ be the n-step evaluation from a configuration σ to a configuration σ', where o is the concatenation of observations produced by individual instructions [13].

5.2 Security Policy

5.2.1 Libra Leakage Model. Side-channel observations are captured in a leakage contract (**HR1**), which partitions the instruction set into leakage classes and safe/unsafe instructions (cf. Section 4.1).

In order to leverage leakage classes in a standard security criterion [13], we associate a unique leakage identifier (*add*, *load*, *br*, etc.) to each leakage class. The leakage identifier of an instruction *inst* is given by $\mathscr{L}(\mathit{inst})$. For instance, if additions and subtractions are indistinguishable to an attacker, a possible instantiation of $\mathscr{L}$ is $\mathscr{L}(\mathtt{add\ x\ x\ x}) = \mathscr{L}(\mathtt{sub\ x\ x\ x}) = \mathit{add}$.

Additionally, the instruction set is partitioned into disjoint sets. Safe unary instructions ($\mathbb{I}^{\checkmark}$) and safe binary instructions ($\mathbb{I}^{\checkmark\text{-}\checkmark}$), do not expose information about the value of their operands. Conversely, unsafe unary instructions ($\mathbb{I}^{\times}$), left-unsafe ($\mathbb{I}^{\times\text{-}\checkmark}$), right-unsafe ($\mathbb{I}^{\checkmark\text{-}\times}$), and left-right-unsafe ($\mathbb{I}^{\times\text{-}\times}$) instructions expose information about the values of their only, left, right, or both source operands, respectively.

Libra leaves freedom to hardware developers regarding the concrete instantiation of leakage classes and safe/unsafe partitioning. It only imposes (**HR3**) that secure branches and level-offset branches do not leak their outcome—i.e., $\{\mathtt{lo}, \mathtt{s}\}\mathtt{.br}\ c\ \_ \in \mathbb{I}^{\checkmark}$—and secure calls and level-offset calls do not reveal whether the original function or the dummy function is actually executed—i.e., $\{\mathtt{lo}, \mathtt{s}\}\mathtt{.call}\ \_ \in \mathbb{I}^{\checkmark}$. For our security policy, we additionally require that normal branches and calls leak their outcome, and that control-flow-altering instructions belong in a distinct leakage class from each other and from non-control-flow-altering instructions. Intuitively, this ensures that low-equivalent source executions are slice-synchronized: at each step, their program counters belong to the same slice.

5.2.2 Weak/Strong Observer Mode. The Libra leakage model is used to instantiate the function *obs*, which, as mentioned earlier, is a parameter of the semantics specifying the observation produced when evaluating an instruction. We define two distinct observer modes (i.e., instantiations of *obs*) that we will apply to asm and asm programs.

The *weak observer mode* (obs^-) exposes all timing and microarchitectural effects that are independent of the program counter (i.e., the *balanceable* leakage). The leakage classes and safe/unsafe partitioning determine the instantiation of obs^-, as defined in Figure 4.

The *strong observer mode* (obs^+) includes observations of the weak mode, plus the observable part of the program counter (i.e., the *unbalanceable* leakage), which, from **HR2**, does not expose more than the address of the current slice:

$$\mathit{obs}^+(\langle m, r, \mathtt{pc}, \rho, \lambda \cdot (\mathtt{bbc}, \mathtt{off})\rangle) = \mathit{slice\_addr}(\mathtt{pc}, \mathtt{off}) \cdot \mathit{obs}^-(\langle m, r, \mathtt{pc}\rangle)$$

5.2.3 Security. Security is defined with respect to a partition of the initial state (memory and registers) into public and secret regions.

SAFE

$$\frac{P[\mathsf{pc}] = \mathsf{op}_2\ \mathsf{x}\ e_1\ e_2 \qquad \mathsf{op}_2\ \mathsf{x}\ e_1\ e_2 \in \mathbb{I}^{\checkmark\text{-}\checkmark} \qquad op_2 = \mathscr{L}(\mathsf{op}_2\ \mathsf{x}\ e_1\ e_2)}{obs^{-}(\langle m, r, \mathsf{pc}\rangle) = op_2}$$

L-UNSAFE

$$\frac{P[\mathsf{pc}] = \mathsf{op}_2\ \mathsf{x}\ e_1\ e_2 \qquad \mathsf{op}_2\ \mathsf{x}\ e_1\ e_2 \in \mathbb{I}^{\times\text{-}\checkmark} \qquad op_2 = \mathscr{L}(\mathsf{op}_2\ \mathsf{x}\ e_1\ e_2) \qquad v = (\!|e_1|\!)_r}{obs^{-}(\langle m, r, \mathsf{pc}\rangle) = op_2\ v}$$

Figure 4: Definition of obs^{-} according to the Libra leakage contract (excerpt). Other rules (R-UNSAFE, LR-UNSAFE, etc.) are analogous.

Definition 8 (Indistinguishability)**.** Two states σ, σ' are indistinguishable, written $\sigma \simeq \sigma'$, if they agree on the value of their public registers and public memory locations.

We define security as (termination-insensitive) *Observational Non-Interference* (ONI) [36] w.r.t. an observation function *obs*:

Definition 9 (*obs*-ONI)**.** A program P, interpreted in a semantics $\Longrightarrow$, is secure under observer mode *obs*, written *obs*-ONI(P) if and only if for any pair of initial configurations σ_0, σ_0', if $\sigma_0 \simeq \sigma_0'$, and $\sigma_0 \overset{o}{\Longrightarrow}{}^{n} \sigma_n$, then $\sigma_0' \overset{o'}{\Longrightarrow}{}^{n} \sigma_n'$ and $o = o'$.

Intuitively, the goal of our Libra transformation is to transform asm programs that are obs^{-}-ONI, to asm programs that are obs^{+}-ONI. In other words, developers should make sure that secrets do not directly flow to insecure instructions and balance secret-dependent branches (**SR2**), while Libra—with compiler (**SR3**) and hardware (**HR2**) support—guarantees that the target program is secure with respect to a strong observer that can observe (parts of) the program counter through microarchitectural side-channels.

5.3 Libra Transformation

To automatically support Libra (**SR3**), we define a folding transformation $\mathcal{F}$ from asm programs—with annotated secret-dependent branches (**SR1**) and dummy functions for functions that can be called in secret-dependent regions (**SR2**)—to asm programs. For clarity, we present the transformation informally, with illustrative examples, and leave the formalization to Appendix A.

5.3.1 Folding secret-dependent regions. For each *balanced* secret-dependent region S—annotated in asm programs by a secret-dependent branch **s.br** $e\ \ell_t\ \ell_f$—the transformation first computes the level structure $L_0 \ldots L_n$ of the region. Next, for all levels L_i, the transformation rewrites each terminating instruction $\{\varepsilon, \mathbf{s}\}$**.br** $e\ \ell_t\ \ell_f$ in the level with a level-offset branch **lo.br** e $\mathtt{off_t}$ $\mathtt{off_f}$ bbc where bbc is the basic block count of the next level (i.e., $|L_{i+1}|$), and $\mathtt{off_t}$ and $\mathtt{off_f}$ are the level offsets corresponding to ℓ_t and ℓ_f, respectively, in the level L_{i+1}. Finally, for each level of the level structure, the transformation folds the corresponding basic blocks by interleaving their instructions.

Example 1 (Folding branches)**.** Consider the balanced secret-dependent region in Listing 5a and let $B_{\mathtt{En}}, B_{\mathtt{t}}, B_{\mathtt{tt}} \ldots B_{\mathtt{Ex}}$ be the basic blocks corresponding to labels `En`, `t`, `tt`, ..., `Ex`. The compiler first computes the level structure of the region: $L_0 = \{B_{\mathtt{En}}\}, L_1 = \{B_{\mathtt{t}}, B_{\mathtt{f}}\}, L_2 = \{B_{\mathtt{tt}}, B_{\mathtt{tf}}, B_{\mathtt{ft}}, B_{\mathtt{ff}}\}, L_3 = \{B_{\mathtt{Ex}}\}$. Next, the transformation rewrites the terminating instruction in each level (except L_3) with level-offset branches. For instance, terminating instructions of L_1 are replaced with **lo.br** c $\mathtt{off_t}$ $\mathtt{off_f}$ $|L_2|$ where $\mathtt{off_t}, \mathtt{off_f}$ are computed according to the mapping $\{\mathtt{tt} \mapsto 0, \mathtt{tf} \mapsto 1, \mathtt{ft} \mapsto 2, \mathtt{ff} \mapsto 3\}$. Finally, the transformation interleaves the basic blocks in each level, giving the program in Listing 5b.

Listing 5: Libra transformation (Listing 5b) of a balanced secret-dependent branch (Listing 5a) where `j Ex` is syntactic sugar for `br 0,Ex,Ex`; `lo.j` is syntactic sugar for `lo.br 0,0:0:1`; and $i_t, i_t', \ldots$ are arbitrary non-terminating instructions.

```
En: s.br c,t,f
 t:  it; it'
     br d,tt,tf
tt:    itt; j Ex
tf:    itf; j Ex
 f:  if; if'
     br e,ft,ff
ft:    ift; j Ex
ff:    iff; j Ex
Ex: [...]
```

(a)

```
En: lo.br c,0:1:2
L1:  it; if
     it'; if'
     lo.br d,0:1:4
     lo.br e,2:3:4
L2:    itt; itf; ift; iff;
       lo.j; lo.j; lo.j; lo.j

Ex: [...]
```

(b)

5.3.2 Folding functions. First, the algorithm computes the union of the level structures of the functions (the original and the dummy function) to fold. Then, similarly as for secret-dependent branches, it replaces branches with level-offset branches, and interleaves instructions according to the level structure. Finally, it replaces the call with a level-offset call **lo.call** $b\ \ell$, where ℓ is the (fresh) label of the folded function.

Example 2 (Folding functions)**.** Consider the program in Listing 6a and let $B_{\mathtt{foo}}, B_{\mathtt{t}}, B_{\mathtt{f}} \ldots B_{\mathtt{Ex'}}$ be the basic blocks corresponding to labels `foo`, `t`, `f`, ..., `Ex'`. The compiler first computes the union of the level structure of the functions: $L_0 = \{B_{\mathtt{foo}}, B_{\mathtt{foo'}}\}, L_1 = \{B_{\mathtt{t}}, B_{\mathtt{f}}, B_{\mathtt{t'}}, B_{\mathtt{f'}}\}, L_2 = \{B_{\mathtt{Ex}}, B_{\mathtt{Ex'}}\}$. The transformation then rewrites the terminating instructions and interleaves the basic blocks in each level, giving the program in Listing 6b.

Listing 6: Libra transformation (Listing 6b) of a call inside a balanced secret dependent region (Listing 6a) where `j Ex` is syntactic sugar for `br 0,Ex,Ex`; `lo.j n` is syntactic sugar for `lo.br n,0:1:2`; and $i_0, i_0', \ldots$ are arbitrary non-terminating instructions.

```
[...]
s.call T,foo,foo'
[...]
foo: i0
     br c,t,f
  t:   i1; i2; j Ex
  f:   i3; i4; j Ex
 Ex: ret
foo': i0'
     br c',t',f'
 t':   i1'; i2'; j Ex'
 f':   i3'; i4'; j Ex'
 Ex': ret
```

(a)

```
[...]
lo.call T,ffoo
[...]
ffoo: i0; i0'
      lo.br c 0:1:4;
      lo.br c' 2:3:4
  L2:   i1; i3; i1'; i3'
        i2; i4; i2'; i4';
        lo.j 0; lo.j 0;
        lo.j 1; lo.j 1


Ex'': ret; ret
```

(b)

5.4 Correctness and Security

This section states the correctness and security of our Libra transformation $\mathcal{F}$. First, we establish a correspondence relation between source and target configurations. Intuitively, this relates source and program configurations that are at the same point of execution and have the same memory and register states.

Definition 10 ($\sigma \overset{\mathsf{P}}{\sim} \boldsymbol{\sigma}$)**.** A source configuration $\sigma = \langle \mathsf{m}, \mathsf{r}, \mathsf{pc}, \rho \rangle$ for a program P is related to a target configuration $\boldsymbol{\sigma} = \langle \mathbf{m}, \mathbf{r}, \mathsf{pc}, \boldsymbol{\rho}, \boldsymbol{\lambda} \rangle$ for a program $\mathbf{P}$, denoted $\sigma \overset{\mathsf{P}}{\sim} \boldsymbol{\sigma}$, if and only if the following holds: (1) $\mathsf{m} = \mathbf{m}$, (2) $\mathsf{r} = \mathbf{r}$, and (3) $\mathsf{pc} \overset{\mathsf{P}}{\sim}_{\ell} \mathsf{pc}$, where $\overset{\mathsf{P}}{\sim}_{\ell}$ relates program locations in the source program to their corresponding location in the target program.

Libra is a correct program transformation, preserving program semantics, as established by the following proposition:

Proposition 1 (Correctness). *For any* asm *program* P*, number of steps* n*, and initial source and target configurations* σ *and* $\boldsymbol{\sigma}$ *such that* $\sigma \overset{\mathsf{P}}{\sim} \boldsymbol{\sigma}$*, if* $\sigma \Longrightarrow^{n} \sigma'$ *then* $\boldsymbol{\sigma} \Longrightarrow^{n} \boldsymbol{\sigma}'$ *and* $\sigma' \overset{\mathsf{P}}{\sim} \boldsymbol{\sigma}'$*, where* $\Longrightarrow$ *is parameterized by* P *and* $\Longrightarrow$ *is parameterized by* $\mathcal{F}(\mathsf{P})$*.*

Libra is a program transformation that hardens programs secure against a weak attacker, to programs secure against a strong attacker, as established by the following proposition:

Proposition 2 (Security). *For any* asm *program* P*,*

$$obs^{-}\text{-ONI}(\mathsf{P}) \implies obs^{+}\text{-ONI}(\mathcal{F}(\mathsf{P}))$$

Proof sketches are given in the companion technical report [92].

6 CFL CHARACTERIZATION

Based on a rigorous analysis of the microarchitectural attack literature (cf. Table 1 for references), we now present a characterization of hardware optimizations regarding how they have been exploited to leak the control flow of applications. The importance of this characterization is twofold. First, it provides a mental framework for improving the understanding of CFL, which also guided the design of Libra. Second, it provides the basis to establish recommendations for hardware designers wishing to adopt Libra. The results of our CFL attack analysis, i.e., the raw data for our characterization, are presented in Table 1. Each row in this table corresponds to a microarchitectural optimization. The first column names the optimization and points to representative papers exploiting it for CFL attacks. The second column indicates if the hardware optimization yields balanceable observations (i.e., if they can be balanced without Libra support). The third column lists our recommendation on how to handle the leakage using Libra. The last column contains additional notes.

We start by dividing the optimizations into two top-level classes: those that yield balanceable observations (class C1), and those that yield unbalanceable observations (class C2).

C1 - Balanceable observations

For optimizations yielding balanceable observations, the hardware can rely on the software to balance these observations according to the Libra leakage contract (**SR2**).

C2 - Unbalanceable observations

Optimizations yielding unbalanceable observations inevitably leak the control flow when the processor executes weakly balanced code. One of objectives of Libra is to keep the optimizations in this category enabled as much as possible. We further break down this category into two subcategories.

C2.1 - Inhibiting dummy composition. The first subcategory groups optimizations that inhibit the composition of a dummy instruction. Consider for example the silent-store optimization [56, 81]. A silent store writes a value to memory that is already present at the specified address. A silent-store optimization skips writes to memory for silent stores. This behavior turns a store instruction from a right-unsafe into a full-unsafe instruction since its timing and resource usage will depend not only on the memory address operand, as before, but also on the value to store. To securely balance an unsafe instruction, both of its operands must be balanced as well. Yet, since a store affects architectural state, a silent store is the only possible dummy instruction to balance a store, which would leak the control flow in the presence of a silent-store optimization.

Guideline: Disable instances from this optimization class in folded regions. In case that the composition of a dummy instruction is inhibited by the combination of multiple optimizations, it sometimes suffices to disable only one of them. An alternative solution to disabling the optimization is to blocklist the affected instruction(s) in the hardware-software security contract (**HR1**).

C2.2 - Observations as a function of the instruction address. The second subcategory concerns optimizations yielding observations that are a function of the instruction address. We further divide this subcategory into four optimization classes.

C2.2.1 - Observations that reveal the level offset. Some optimizations yield observations that are inherently different for each instruction within a slice. Hence, they *inevitably* reveal the level offset of an executed instruction. Take the branch predictor for instance. The possible targets of a `lo.br` instruction are different for each `lo.br` of the same slice. Hence, if `lo.br` targets are encoded in the branch predictor, an attacker sharing the predictor state could distinguish `lo.br` instructions within a slice and learn the level offset.

Guideline: Disable optimizations of this type in folded regions. For some optimizations, it is necessary to completely disable them (e.g., cache banking [99]), for others this might be unnecessary, such as in the example of the directional predictor we gave, which must only be disabled for a `lo.br` instruction.

C2.2.2 - Libra-safe optimizations. Some optimizations, such as the instruction cache, directly benefit from **HR2a**, which imposes a data-oblivious access pattern to the instruction memory. If the processor frontend follows **HR2**, by implementing slice-granular fetch/decode, these optimizations do not leak at a finer granularity than a slice.

Guideline: No hardware modifications are required.

C2.2.3 - PC-dependent mappings. Some optimizations map instruction addresses to instruction-specific information. The BTB and the PC-based strided data prefetcher, for instance, are typically

Table 1: Control-flow leakage attack landscape.

Exploited Optimization	Balanceable	Guideline	Notes
Computation simplification [9]	✓	**C1**	Alternatives: reject program, DIT [10, 49]
Data TLB [37, 88]	✓	**C1**	Balance address operands (page granular)
Data cache [40, 59, 69, 72, 98]	✓	**C1**	Balance address operands (cache-line granular)
Data cache bank [99]	✓	**C1**	Balance address operands (byte granular)
DRAM row buffer (data) [73]	✓	**C1**	Balance address operands
Data-dependent data prefetcher [22, 81]	✓	**C1**	Balance address operands (loads/stores) and value operands (stores)
Load/store buffers	✓	**C1**	
Pipeline interlock [63, 84, 99]	✓	**C1**	Balance stalling data dependencies
µop fusion [79]	✓	**C1**	
Execution engine [7, 15, 32, 33, 80, 90]	✓	**C1**	Balance structural dependencies
Interrupt controller [64, 86]	✓	**C1**	Balance interrupt latencies
Reorder buffer (ROB) [5]	✓	**C1**	Balance instruction types
Memory bus / controller [17, 18, 89, 96]	✓	**C1**	Balance memory bus(es) usage
Computation reuse [81]	✓	**C1**	Balance operands
Branch order buffer (BOB) [48]	✓	**C1**	
Interconnect [70]	✓	**C1**	
Frontend [76]	✓	**HR2**	Slice-granular fetch/decode
Instruction cache [1, 23, 43, 59, 98]	✓	**C2.2.2**	Balancing (confining region inside a single cache line) is more limited
MMU / Page tables [20, 64, 88, 97]	✓	**C2.2.2**	Balancing (confining region inside a single page) is more limited
DRAM row buffer (instructions) [73]		**C2.2.2**	
Instruction prefetcher [57, 102]		**C2.2.2**	
Instruction TLB [37, 88]		**C2.2.2**	
PC-dep data prefetcher [14, 23, 24, 39, 82]		**C2.2.3**	
Directional predictor [2, 3, 31, 47]		**C2.2.3**	Only for public branches, disable for `lo.br`
BTB [30, 55, 101]		**C2.2.3**	Care must be taken not to leak the target transiently
Value prediction [27, 81]		**C2.2.3**	
µop cache (DSB) [26, 51, 79]		**C2.2.4**	Alternative: disable in folded regions
Silent stores [81]		**C2.1**	Disable in folded regions
Instruction cache bank [99]		Disable	Disable in folded regions (violates **HR2**)

implemented using table-based structures indexed by instruction address.

Guideline: Thanks to folding, it becomes possible to represent instruction-specific information as slice-specific information. Mappings from instruction address to instruction-granular information can be changed into mappings from slice address to slice-granular information (per **HR2**). This usually requires minimal hardware modifications such as indexing hardware structures by slice address instead of by instruction address.

C2.2.4 - Instruction-specific optimizations. Some optimizations perform different operations depending on the instruction and are not generalizable to the slice, contrary to optimization class **C2.2.3**. An example is the µop cache, where the operations, *decode*, *insert* and *evict*, depend on the specific instruction.

Guideline: Instead of disabling these optimizations, it might be more beneficial to keep them enabled and always perform the operation on every instruction of the slice. Keeping optimizations enabled for instances for this optimization class will typically be more expensive compared to optimization class **C2.2.3**.

7 IMPLEMENTATION AND EVALUATION

7.1 Implementation

Following the requirements from Section 4.4 and the guidelines from Section 6, we implemented Libra on Proteus [16] (version 2024.01-O), a RISC-V out-of-order core designed to experiment with hardware security extensions.

***HR1**: Leakage contract.* We partitioned the RISC-V instruction set into leakage classes and validated the correctness of this classification via our automated security evaluation (cf. Subsection 7.2). In particular, load instructions leak their address via the data cache and are balanced in software.

***HR2**: Slice-granular PC leakage.* Based on our analysis, the sources of precise PC leakage on Proteus were the branch target predictor, the instruction cache, and the instruction prefetcher. For security reasons, we completely disable the branch predictor in folded regions. Yet, thanks to the linear layout of folded regions, the performance impact of this is limited: the next slice—where the execution will continue—will be prefetched by the time the branch condition is resolved. The other hardware structures did not have to be altered, as explained next.

***HR2a**: Data-oblivious instruction memory access pattern.* The instruction fetch unit has been made Libra-aware. In a secret-dependent folded region, the level offset of the currently executing instruction needs to be invisible to the memory subsystem. In our implementation, this is achieved by fetching in a fixed order all cache lines including instructions from the current slice. This also results in the state of the instruction cache being independent from the level offset. As the prefetcher's behavior in Proteus only depends on the instruction cache state, it also observes the same access patterns and does not require any additional changes. Thanks to the folded layout in memory, the prefetching remains very effective during the execution of folded regions.

Table 2: Overhead factors: execution time (cycles) / binary size (bytes).

Benchmark	Baseline	Balanced	Linearized	Folded
fork	110 c / 136 B	1.00x / 1.00x	1.11x / 1.12x	1.00x / 0.94x
triangle	116 c / 132 B	1.03x / 1.06x	1.05x / 1.15x	0.98x / 1.00x
bsl	1415 c / 336 B	1.20x / 1.04x	1.54x / 1.08x	1.24x / 1.01x
diamond	186 c / 192 B	1.07x / 1.10x	1.18x / 1.23x	1.06x / 1.04x
kruskal	1573 c / 452 B	1.09x / 1.05x	1.21x / 1.16x	1.16x / 1.04x
ifthenloop	407 c / 200 B	1.35x / 1.20x	1.28x / 1.20x	1.56x / 1.16x
switch	1402 c / 500 B	2.11x / 1.41x	2.70x / 1.92x	1.90x / 1.15x
sharevalue	1410 c / 500 B	1.38x / 1.02x	1.76x / 1.15x	1.77x / 1.01x
mulmod16	339 c / 276 B	1.23x / 1.01x	1.47x / 1.16x	1.32x / 0.96x
keypad	3490 c / 416 B	2.86x / 1.08x	3.48x / 1.12x	3.61x / 1.06x
modexp2	11716 c / 324 B	1.72x / 1.02x	1.79x / 1.09x	1.78x / 1.01x
mean		**1.38x / 1.09x**	**1.57x / 1.20x**	**1.46x / 1.03x**

***HR3**: Level-offset branch.* For our implementation, we introduced a variant of the `lo.br` instruction, the *terminating level-offset branch* `tlo.br`. This instruction behaves similarly as a regular `lo.br`, but additionally encodes the number of slices in the next level, an optimization that makes the `lo.br` instructions of the last level of a folded region unnecessary. Our prototype encodes the `lo.br` and `tlo.br` variants for each RISC-V branch instruction by repurposing the two prefix bits in the fixed-width 32-bit RISC-V encoding, but other implementations could use the free opcode slots as defined by the RISC-V specification. The current and previous Libra contexts are stored in a two-level hardware stack. We support folded regions with up to 16 basic blocks per level (8 for a terminating level).

7.2 Evaluation

We evaluated our implementation by measuring the binary size and execution time overheads using a benchmark suite from related work [18, 75, 85, 93, 94]; measuring the hardware overhead; and performing RTL-level noninterference testing to validate security.

Binary size. The results on binary size can be found in Table 2, which shows the binary size of the original benchmark, the overhead of balancing the secret-dependent branches (which still leaks information through unbalanceable observations), the overhead of linearizing the secret-dependent branches with Molnar's method [66], and finally, folding the secret-dependent branches with Libra. The benchmarks show that the overhead is small compared to state-of-the-art linearized (constant-time) code. In certain cases, the folded program can even be expressed more succinctly due to the characteristics of folded regions; after the last slice, the next instruction will be executed regardless of which branch was taken, making additional jump instructions, such as in Listing 1a, unnecessary.

Execution time. We evaluate the execution time overhead using the same extended benchmark suite, shown in Table 2. Even though our prototype implementation is not optimal, the benchmarks clearly show an advantage of Libra over linearized code. The mean performance overhead of Libra is 46% compared to 57% of the linearized code (a relative overhead reduction of 19.3%), and for certain benchmarks it not only performs much better than linearized code, but also outperforms insecure balanced code. For example, the `switch` and `triangle` benchmarks clearly show the power of Libra over alternative approaches.

Hardware cost. We evaluate the hardware cost of implementing Libra on Proteus by synthesizing the design to the Xilinx XC7A35TICSG324-1L FPGA in Xilinx Vivado 2022.2. According to our measurements, the Libra additions increase the number of look-up tables by 11.4% (from 16,531 to 18,414), the number of registers by 9.5% (from 13,566 to 14,850), while keeping the critical path unchanged (37.4 ns).

Security. To evaluate the security of our implementation, we adopt a methodology from related work [18, 93, 94]: noninterference-based testing in a cycle-accurate Verilog simulator. For each benchmark, we manually ensure that all possible code paths are explored, which is feasible due to the relatively small size of the benchmarks. We verify that, for executions with identical public inputs but varying secret inputs, the processor's internal signals associated with side channels remain consistent. Any variation would indicate a leak of secret information. The signals we focus on include the state of the branch predictor, addresses in the instruction and data caches, the state of the instruction prefetcher, and the occupancy of the execution units. Each simulation is run independently, starting from a cold microarchitectural state.

Interestingly, our security evaluation revealed that the hardened `kruskal` benchmark (originally introduced in [62]) contains a recursive function with a secret-dependent number of iterations, as hypothesized by the original authors. As a linearized implementation of Kruskal's algorithm is not a trivial effort and out of scope for our paper, we only transformed the secret-dependent branch in the main function of the benchmark.

8 DISCUSSION

Intra-cache-line attacks. **HR2** requires that the PC does not leak at a finer granularity than a level slice. This implies that executing folded regions is secure only if an attacker is unable to observe intra-cache-line instruction memory accesses in folded regions. To the best of our knowledge, only two published attacks expose intra-cache-line accesses: cache-bank conflicts [63] and false dependencies [99]. To comply with **HR2**, the optimizations exploited by these attacks must be disabled in folded regions. However, we do not consider them to be performance-critical. In more recent microarchitectures these leakages have been closed, confirming our assumption that **HR2** will not significantly affect the performance.

Future work. There are some open questions that should be addressed in future work. First, a limitation with the implementation of our prototype is that the pipeline stalls after fetching a `lo.br` (until its condition is resolved). As described in Section 4, the linearity of folded regions removes the uncertainty of what instructions to *fetch* after a `lo.br`. However, the uncertainty of what instruction to *execute* (i.e., what is the level offset of the next instruction in the next prefetched slice?) still remains. For security reasons, the CPU cannot proceed based on a prediction of the direction of the `lo.br` as this would induce a timing signal exposing the control flow. On the RISC-V processor we used for our implementation, the `lo.br` penalty is generally only a few cycles. However, the penalty on superscalar CPUs with deeper pipelines (capable of fetching and executing multiple instructions in a single cycle) is much higher. How to deal with the `lo.br` penalty on this class of CPUs (up to 10-15 cycles on some CPUs [45]) remains an open design question.

We believe that exploiting the regularity and the linearity of folded regions is key to solving this challenge.

Second, we informally argue (Table 1) that many optimizations either comply with **HR2**, or can be adapted to do so. Our empirical evaluation on an implementation featuring instruction and data caches, branch predictor and prefetcher, supports this argument. Formally verifying that these optimizations comply with **HR2** would be an interesting avenue for future work.

Third, there is no compiler support for Libra. We manually identify, balance and fold secret-dependent regions at assembly level, restricting the size of our benchmark programs. Compiler support for Libra to be able to conduct more extensive performance measurements on real-world programs is future work.

Fourth, we put software-based fault attacks [68] and software-based power attacks [58] out of scope. It is interesting future work to extend the leakage contract to cover these attacks.

Fifth, we only considered fixed-length instructions. Variable-length instructions, as found in the popular x86 ISA, might require some changes to the current Libra design. To increase the chances of adoption on architectures with variable-length instructions, it would be interesting to investigate how to support them.

Finally, creating the leakage contract is a manual effort. A very interesting avenue for future work is to investigate how to generate the leakage contract from the hardware description (RTL level), as done in recent work [29, 46, 65], and how to express contracts such that they can be consumed by a compiler [42].

9 RELATED WORK

CFL Hardening. The literature contains a vast amount of prior work on software-only countermeasures against CFL attacks. Almost 25 years ago, Agat [4] already proposed a transformational security type system to balance conditional branches, later refined by Köpf and Mantel [53]. Non-transformational type systems to detect unbalanced branches have been implemented for the AVR [28] and MSP430 [75] architectures. Winderix et al. [94] proposed an algorithm for control-flow balancing during compilation and implemented and evaluated it using the LLVM compiler infrastructure [54]. Prior work considers secret-dependent control flow inherently insecure and strongly discourages control-flow balancing as a defense against CFL attacks. This view is incorporated in the well-established constant-time programming discipline [8], which disallows programmers from writing secret-dependent branches. There is a rich literature to automatically detect [35, 50] and eliminate [19, 66, 78, 83, 87, 95] secret-dependent control flow.

Architectural Support. Many existing software-only countermeasures leverage hardware primitives designed with performance in mind. The resulting security guarantees are brittle, as these countermeasures rely on undocumented behavior that is not guaranteed in future versions of the hardware. For instance, conditional execution (a.k.a. predicated execution or predication) is supported in some form by the x86, Arm and RISC-V ISAs to accelerate some hard-to-predict branches, yet is sometimes used to eliminate secret-dependent control flow [19, 25, 95], critically relying on the (current) data-oblivious behavior. Another example is Intel TSX, used as a primitive for a countermeasure proposed by Gruss et al. [38]. In contrast, Libra provides principled support by augmenting the ISA with a security contract representing its security guarantees. Many modern CPUs provide safe instructions, making explicit security guarantees part of the ISA. An example is constant-time support for AES to improve speed and security of applications relying on it (e.g., [6]). As another example, x86, Arm and RISC-V ISAs have extended their ISAs with facilities to turn unsafe instructions into safe instruction via a feature called Data (Operand) Independent Timing [10, 49]. To the best of our knowledge, architectural support to securely execute balanced code on high-end processors has not been proposed before. Recently, Winderix et al. [93] proposed architectural support for control-flow balancing and control-flow linearization to efficiently counter CFL attacks. Unfortunately, their solution for control-flow balancing is only targeted towards processors with a microcontroller profile featuring simple processor pipelines. In contrast, our proposal for control-flow balancing is designed to securely execute balanced regions on high-end systems.

Hardware-Software Leakage Contracts. Recent work on hardware-software leakage contracts [41, 44, 61, 67, 74] proposes to augment the ISA with a specification of how the hardware leaks information. Libra also specifies such a hardware-software leakage contract and partitions the instruction set into leakage classes. Instructions of the same leakage class leak the same information and thus are indistinguishable to an attacker. The first idea for classifying the instruction set this way was proposed by Winderix et al. [94] under the form of *latency classes*, a concept that was later generalized into *leakage classes* by Bognar et al. [18]. Yu et al. [100], propose ISA design principles for data Oblivious ISAs (OISAs) to perform side-channel resistant and high-performance computations. The authors proposes an ISA-level data oblivious abstraction, which partitions the instruction set into safe and unsafe instructions. In contrast to Libra, their work does not include ISA-level principles to make control-transfer instructions data oblivious, and hence is complementary to ours.

Secure Compilation for Side-Channel Defenses. Our secure compilation proof is inspired by existing proof techniques for preservation of side channel defenses by compilers [11–13] with some adaptations to account for important differences. Compared to the constant-time policy [11], balancing allows program counters of low-equivalent executions to diverge, and compared to constant-resource transformation [12], our transformation is not entirely leakage preserving. In this work, we assume a non-canceling leakage model (i.e., $o_1 \cdot o_2 = o_1' \cdot o_2' \implies o_1 = o_1' \wedge o_2 = o_2'$). Secure compilation for relaxed policies based on canceling leakage (e.g., program cost in terms of clock cycles) have been proposed [12]. However, it remains unclear whether there exist a concrete threat model (attacker model and microarchitecture) to which these policies securely apply. In particular, such relaxed policies are insecure against the strong attacker that we consider in this paper [86].

10 CONCLUSION

In this paper, we challenged the widely-held belief that control-flow balancing is either insecure or inefficient on modern out-of-order CPUs. We proposed Libra, a novel hardware-software codesign for principled, secure and efficient balanced execution. We gave evidence that it is possible with minimal hardware support to securely

balance secret-dependent control flow while keeping performance-critical hardware optimizations enabled. A key feature of Libra is the specification of a hardware-software security contract that software can rely on to harden applications in a principled way, similar to how software relies on an ISA specification for the functional correctness of programs. Libra minimally extends the instruction set to make balanced execution secure and efficient on high-end systems, mainly by virtue of folding. We formalized the Libra semantics and the folding transformation, which we proved correct and secure. We also presented a characterization of how microarchitectural optimizations can leak a program's control flow, the basis for our recommendations for hardware designers wanting to adopt Libra to their designs. Our implementation and evaluation show significant performance benefits compared to state-of-the-art control-flow linearization at low hardware cost.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was partially funded by the ORSHIN project (Horizon Europe grant agreement #101070008), the Research Foundation – Flanders (FWO) via grants #G081322N and #12B2A24N, and the Flemish Research Programme Cybersecurity.

REFERENCES

[1] Onur Aciicmez. 2007. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *ACM Workshop on Computer Security Architecture*.
[2] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology – CT-RSA 2007*.
[3] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *ASIACCS*.
[4] Johan Agat. 2000. Transforming out Timing Leaks. In *POPL*.
[5] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. 2021. Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions. *IEEE Computer Architecture Letters* (2021).
[6] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. Breakthrough AES performance with intel AES new instructions. *White paper 12* (2010).
[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *S&P*.
[8] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*.
[9] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *S&P*.
[10] Arm. 2021. Arm Armv8-A Architecture Registers: DIT, Data Independent Timing. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Registers/DIT--Data-Independent-Timing.
[11] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30.
[12] Gilles Barthe, Sandrine Blazy, Rémi Hutin, and David Pichardie. 2021. Secure Compilation of Constant-Resource Programs. In *CSF*. IEEE, 1–12.
[13] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *CSF*.
[14] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. 2012. Hardware Prefetchers Leak: A Revisit of SVF for Cache-Timing Attacks. In *IEEE/ACM Intern. Symp. on Microarch. Workshops*.
[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*.
[16] Marton Bognar, Job Noorman, and Frank Piessens. 2023. Proteus: An Extensible RISC-V Core for Hardware Extensions. In *RISC-V Summit Europe '23*.
[17] Marton Bognar, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *S&P*.
[18] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. 2023. MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling. In *EuroS&P*.
[19] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS*.
[20] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*.
[21] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*.
[22] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*.
[23] Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E Carlson. 2024. PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks. In *HPCA*.
[24] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. 2023. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *ASPLOS*.
[25] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *S&P*.
[26] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends. In *HPCA*.
[27] Shuwen Deng and Jakub Szefer. 2021. New Predictor-Based Attacks in Processors. In *DAC*.
[28] Florian Dewald, Heiko Mantel, and Alexandra Weber. 2017. AVR Processors as a Platform for Language-Based Security. In *ESORICS*.
[29] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher Fletcher. 2024. CONJUNCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety Against Microarchitectural Timing Attacks. In *S&P*.
[30] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*.
[31] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*.
[32] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. 2024. Remote Scheduler Contention Attacks. In *FC*.
[33] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. 2023. SQUIP: Exploiting the Scheduler Queue Contention Side Channel. In *S&P*.
[34] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2018).
[35] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *CCS*.
[36] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *S&P*. IEEE Computer Society, 11–20.
[37] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.
[38] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*.
[39] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.
[40] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[41] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *S&P*.
[42] Marco Guarnieri and Marco Patrignani. 2020. Contract-Aware Secure Compilation. *CoRR* abs/2012.14205 (2020).
[43] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX ATC*.
[44] Gernot Heiser. 2018. For Safety's Sake: We Need a New Hardware-Software Contract! *IEEE Des. Test 35.2* (2018).
[45] John L Hennessy and David A Patterson. 2019. *Computer architecture: a quantitative approach*.
[46] Yao Hsiao, Dominic P Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2022. Scalable assurance via verifiable hardware-software contracts. In *OSCAR*.

[47] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2019. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *CHES* (2019).

[48] Intel. 2017. *Intel® 64 and IA32 Architectures Performance Monitoring Events.*

[49] Intel. 2022. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance.

[50] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *S&P*.

[51] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. 2021. UC-Check: Characterizing Micro-Operation Caches in X86 Processors and Implications in Security and Performance. In *MICRO*.

[52] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*.

[53] Boris Köpf and Heiko Mantel. 2007. Transformational typing and unification for automatically correcting insecure programs. *Int. J. of Inf. Security 6.2* (2007).

[54] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[55] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*.

[56] Kevin M Lepak and Mikko H Lipasti. 2000. On the value locality of store instructions. In *ISCA*.

[57] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security*.

[58] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*.

[59] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *S&P*.

[60] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography. *ACM Comput. Surv.* (2021).

[61] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. 2018. Position Paper: A Case for Exposing Extra-Architectural State in the ISA. In *HASP*.

[62] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *Computer Security – ESORICS 2015*. 447–467.

[63] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* (2019).

[64] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*.

[65] Gideon Mohr, Marco Guarnieri, and Jan Reineke. 2024. Synthesizing Hardware-Software Leakage Contracts for RISC-V Open-Source Processors. *arXiv* (2024).

[66] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*.

[67] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *ISCA*.

[68] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA*.

[70] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security*.

[71] Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. *CoRR* abs/2001.11334 (2020).

[72] Colin Percival. 2005. Cache missing for fun and profit.

[73] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. {DRAMA}: Exploiting {DRAM} Addressing for {Cross-CPU} Attacks. In *USENIX security*.

[74] Hernán Ponce-de León and Johannes Kinder. 2022. Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In *S&P*.

[75] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. 2020. SCF-MSP: Static Detection of Side Channels in MSP430 Programs. In *ARES*.

[76] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *USENIX Security*.

[77] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In *Smart Card Programming and Security*.

[78] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital {Side-Channels} through obfuscated execution. In *USENIX Security*.

[79] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *ISCA*.

[80] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. 2022. Port Contention Goes Portable: Port Contention Side Channels in Web Browsers. In *Asia CCS*.

[81] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. 2021. Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data. In *ISCA*.

[82] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS*.

[83] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-channel Elimination via Partial Control-flow Linearization. *ACM Trans. Program. Lang. Syst.* (2023).

[84] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. 2018. Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in Iaas Clouds.. In *NDSS*.

[85] Rodothea Myrsini Tsoupidi, Elena Troubitsyna, and Panagiotis Papadimitratos. 2023. Thwarting code-reuse and side-channel attacks in embedded systems. *arXiv* (2023).

[86] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*.

[87] Daan Vanoverloop, Hans Winderix, Lesly-Ann Daniel, and Frank Piessens. 2024. Compiler Support for Control-Flow Linearization Using Architectural Mimicry. (2024).

[88] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*.

[89] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014. Timing channel protection for a shared memory controller. In *HPCA*.

[90] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *ACSAC*.

[91] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. *Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High- End Processors.* https://doi.org/10.5281/zenodo.12786159

[92] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors (Extended Version). arXiv:2409.03743 [cs.CR]

[93] Hans Winderix, Marton Bognar, Job Noorman, Lesly-Ann Daniel, and Frank Piessens. 2024. Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs. In *S&P*.

[94] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. 2021. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks. In *EuroS&P*.

[95] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *ISSTA*.

[96] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Transactions on Networking* 23, 2 (2014), 603–615.

[97] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*.

[98] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[99] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. of Cryptographic Engineering* (2017).

[100] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS*.

[101] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. 2023. All Your PC Are Belong to Us: Exploiting Non-Control-Transfer Instruction BTB Updates for Dynamic PC Extraction. In *ISCA*.

[102] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. {BunnyHop}: Exploiting the Instruction Prefetcher. In *USENIX Security*.

A FOLDING TRANSFORMATION

This section details our folding transformation $\mathcal{F}$ from asm programs to asm programs.

Notations. In the following, we let $B[i]$ denote the i^{th} instruction in basic block B. We also let B_ϵ denote the empty basic block. By

definition of a CFG, all program labels point to the beginning of a basic block and we write B_ℓ to denote the basic block corresponding to label ℓ. In a level L, we assume that basic blocks are indexed, and we let $idx(L, B)$ denote the index of basic block B in level L.

We let $S = (B_{entry}, B_{exit}, \{B_1, \ldots, B_n\})$ denote a secret dependent region $\{B_1, \ldots, B_n\}$, with entry block B_{entry} and exit block B_{exit}. We let $entry(S)$ and $exit(S)$ return the entry and exit basic block of the region, respectively. We let $F = (\ell_f, \{B_1, \ldots, B_n\})$ denote a function, defined by its entry label ℓ_f and its sequence of basic blocks $\{B_1, \ldots, B_n\}$. Finally, we let $level\_struct(S)$ return the level structure of a secret dependent region S (excluding B_{exit} and B_{exit}) and $level\_struct(F)$ return the level structure of a function.

Secret-dependent branches. We define a function $RewriteTerminator(L_i, L_{i+1})$ that rewrites the terminating instructions from a level L_i (Algorithm 1). It replace all branches in the level with a level offset branch `lo.br` e $\texttt{off}_\texttt{t}$ $\texttt{off}_\texttt{f}$ bbc where bbc is the basic block count of the next level ($|L_{i+1}|$), and $\texttt{off}_\texttt{t}$ (resp. $\texttt{off}_\texttt{f}$) is the basic block number corresponding to ℓ_t (resp. ℓ_f) in L_{i+1}.

def $RewriteTerminator(L_i, L_{i+1})$**:**
  Input: Level to modify $L_i = \{B_0 \ldots B_n\}$
  Result: Modified level $L'_i = \{B'_0 \ldots B'_n\}$
  bbc $\leftarrow |L_{i+1}|$; $len \leftarrow |B_0|$; $\{B'_0 \ldots B'_n\} \leftarrow \{B_0 \ldots B_n\}$;
  for $B_i \in L_i$ **:**
    assert $|B_i| = len$;
    switch $B_i[len-1]$ **:**
      case $\{\varepsilon, \texttt{s}\}$`.br` e ℓ_t ℓ_f **:**
        assert $B_{\ell_t}, B_{\ell_f} \in L_{i+1}$;
        $\texttt{off}_\texttt{t}, \texttt{off}_\texttt{f} \leftarrow idx(L_{i+1}, B_{\ell_t}), idx(L_{i+1}, B_{\ell_f})$;
        $B'_i[len-1] \leftarrow$ `lo.br` e bbc $\texttt{off}_\texttt{t}$ $\texttt{off}_\texttt{f}$;
      case ret **:** $B'_i[len-1] \leftarrow$ ret;
      otherwise : assert *False*;
  return $L'_i = \{B'_0 \ldots B'_n\}$

Algorithm 1: Rewriting of terminating instructions.

Next, we define a function $FoldLevel(L)$, which takes a level $L = \{B_0 \ldots B_n\}$ and returns a single folded basic block B' interleaving the instructions of $B_0 \ldots B_n$ (Algorithm 2).

def $FoldLevel(L)$**:**
  Input: Level to fold $L = \{B_0 \ldots B_n\}$
  Result: Basic block B' (folded level)
  bbc $\leftarrow |L|$; $len \leftarrow |B_0|$; $B' \leftarrow B_\varepsilon$;
  for $B_i \in L$ **:**
    assert $|B_i| = len$;
    for $j \in [0, len)$ **:** $B'[j \times \text{bbc} + i] \leftarrow B_i[j]$;
  return B'

Algorithm 2: Folding of the basic blocks of a level L.

To fold secret-dependent branches, we define a function $FoldRegion(S)$ that returns the folded version of a secret dependent region S (Algorithm 3). The function first computes the level structure of S, rewrite the terminators, and fold the level structure.

Function calls. To fold a function F with its dummy function F', we define a function $FoldFunction(F, F', \ell_{ff'})$ (Algorithm 4) that

def $FoldRegion(S)$**:**
  Input: Secret-dependent region S
  Result: Folded secret-dependent region S'
  $L_1 \ldots L_n \leftarrow lvl\_struct(S)$; $L_0 \leftarrow entry(S)$;
  $L_{n+1} \leftarrow exit(S)$;
  for $L_i \in L_0 \ldots L_n$ **:**
    $L'_i \leftarrow RewriteTerminator(L_i, L_{i+1}, False)$;
    $B'_i \leftarrow FoldLevel(L'_i)$;
  $S' \leftarrow (B'_0, exit(S), \{B'_1 \ldots B'_n\})$;
  return S'

Algorithm 3: Folding of a secret dependent region S.

returns a folded function with entry label $\ell_{ff'}$. First, the algorithm computes the union of the level structure of F and F'. Then, it replaces branches with level-offset branches. Finally, it folds basic blocks according to the level structure and returns the final function, defined by the set of folded basic blocks and entry label $\ell_{ff'}$.

def $FoldFunction(F, F', \ell_{ff'})$**:**
  Input: Functions F, F
  Result: Folded function F''
  $L_0 \ldots L_n \leftarrow level\_struct(F) \cup level\_struct(F')$;
  $L_{n+1} \leftarrow B_\varepsilon$;
  for $L_i \in L_0 \ldots L_n$ **:**
    $L''_i \leftarrow RewriteTerminator(L_i, L_{i+1}, \text{True})$;
    $B''_i \leftarrow FoldLevel(L''_i)$;
  return $F'' = (\ell_{ff'}, \{B''_0 \ldots B''_n)\}$

Algorithm 4: Folding of a function F with its dummy version F'. The union of level structures is defined as the componentwise union of basic blocks.

Finally, we define a function $RewriteCall(B)$, which replaces all secret dependent calls `s.call` b ℓ_f $\ell_{f'}$ with a level-offset call `lo.call` b $\ell_{ff'}$ where $\ell_{ff'}$ is the label of the folded function.

Final folding transformation. The final folding transformation $\mathsf{P} = \mathcal{F}(\mathsf{P})$ performs the following steps: (1) For each pair of function/dummy $F_{\ell_f}/F'_{\ell_{f'}}$ that can be called from a secret-dependent region (annotated in asm with a secret dependent call), $\mathcal{F}$ computes the folded function $F_{\ell_{ff'}} = FoldFunction(F_{\ell_f}, F'_{\ell_{f'}}, \ell_{ff'})$ and places it at location $\ell_{ff'}$ in P. Functions $F_{\ell_f}, F'_{\ell_{f'}}$ are also included in P if they can be called with "normal" calls. (2) For each (outermost) secret-dependent region S (annotated in asm programs by a secret-dependent branch), $\mathcal{F}$ computes the folded region $\mathsf{S} = FoldRegion(\mathsf{S})$, and replaces the original region S with S. Note that for a given secret-dependent region, our algorithm folds its entire level structure (which includes nested branches). Hence, for nested secret-dependent branches, only the outermost branch need to be considered and the nested branches will be automatically folded. (3) All other basic blocks are directly copied from P to P; (4) Secret dependent calls in P are replaced by $RewriteCall(B)$; (5) In the final code memory layout of P, the folded levels are placed adjacent to each other, in level order: the basic block corresponding to $FoldLevel(L_{i+1})$ directly follows the basic block corresponding to $FoldLevel(L_i)$. The exit block of a secret dependent region is also placed just after the last folded level.