Levioso: Efficient Compiler-Informed Secure Speculation

Ali Hajiabadi[1] Archit Agarwal[2] Andreas Diavastos[1] Trevor E. Carlson[1]
[1]National University of Singapore [2]University of California, San Diego

ABSTRACT

Spectre-type attacks have exposed a major class of vulnerabilities arising from speculative execution of instructions, the main performance enabler of modern CPUs. These attacks speculatively leak secrets that have been either speculatively loaded (seen in sandboxed programs) or non-speculatively loaded (seen in constant-time programs). Various hardware-only defenses have been proposed to mitigate both speculative and non-speculative secrets via all potential transmission channels. However, limited program knowledge is exposed to the hardware and these solutions conservatively restrict the execution of all instructions that can potentially leak.

In this work, we show that not all instructions depend on older unresolved branches and they can safely execute without leaking speculative information. We present Levioso, a novel hardware/software co-design, that provides comprehensive secure speculation guarantees while reducing performance overhead compared to existing defenses. Levioso informs the hardware about true branch dependencies and applies restrictions only when necessary. Our evaluations demonstrate that Levioso is able to significantly reduce the performance overhead compared to two prior defenses from 51% and 43% to just 23%.

1 INTRODUCTION

Speculative execution attacks, like Spectre [18], are a major concern in modern processor designs as they exploit the main enabler of their performance, *speculative execution* [26]. These attacks trick the processor into transiently executing unintended paths of the program and force the victim to access sensitive information and leak it into a microarchitectural covert channel (*e.g.*, data caches [37]). To mitigate speculative execution attacks, comprehensive solutions aim to prevent transmitting secret information through all potential channels by restricting the execution of speculative instructions (*i.e.*, channel-agnostic mitigations). For example, STT [39] deploys a dynamic taint tracking technique to restrict the execution of instructions that are tainted by speculative loads. Following solutions [8, 24, 33] use the same insight, and propose hardware-only mechanisms to detect and restrict the execution of unsafe instructions. These solutions provide secure speculation for *sandboxed* programs that guarantee their memory accesses are within authorized address ranges during the correct execution. However, they fail to provide secure speculation for the secret data that has already been loaded into a register non-speculatively (as seen in *constant-time* and cryptographic programs). To protect non-speculative secrets, speculative execution of all instructions, potentially tainted by secret values, must be restricted. Current solutions for constant-time programs either manually specify secret regions of memory and track the taints in the hardware [10, 12, 33] or assume all memory regions are secret and declassify them only if they leak during non-speculative execution as well [8].

Unfortunately, many of these solutions tend to significantly reduce benefits of speculative execution, as they follow a conservative approach of restricting the execution for majority of instructions after an unresolved branch. However, our studies show that many of the restrictions introduced by these mitigations are unnecessary because not all instructions after a conditional branch are truly dependent on the branch outcome. We show a motivating example in Figure 1. In this example, Inst4 leaks different values, either 1 or 2, depending on the branch outcome (*i.e.*, it is data dependent on branch Inst2). However, Inst5 leaks the same value independently from the path executed after the branch (it always leaks value 1). Hardware-only defenses do not have sufficient information about all possible control flow paths and true branch dependencies of the program, as the processor only views a small window of instructions at any given moment. Hence, they tend to conservatively restrict most instructions. Our key insight is that *a hardware/software co-design can provide a more efficient solution*, where the compiler informs the hardware about true branch dependencies and the hardware allows the execution of instructions once these dependencies are resolved.

```
Inst1│x = 1, y = 1
Inst2│if (condition)
Inst3│  x <- x + 1  //branch-dependent (control)
Inst4│leak(x) //branch-dependent (data)
Inst5│leak(y) //branch-independent
```

Figure 1: Motivating example with branch dependencies.

In this work, we propose Levioso that provides comprehensive security and high performance via hardware/software co-design. To accomplish this, we use a static compiler pass to detect and communicate true branch dependencies to the hardware. Our hardware uses this information to apply restrictions only if necessary, mitigating speculative execution attacks. Levioso provides secure speculation for both sandboxing and constant-time policies.

The main contributions of this work are as follows:

- A novel hardware/software co-design that enables comprehensive defense for Spectre-type attacks (both speculative and non-speculative secrets) with no programmer effort[1];
- Significantly reducing performance overhead compared to prior defenses (reducing 51% and 43% overhead of Dolma [24] and STT [39] to just 23%), with negligible power and area overheads.

2 BACKGROUND

2.1 Speculative Execution Attacks

Speculative execution attacks aim to bypass array bounds checks of sandboxes or speculatively transmit the secret data of constant-time programs (see code snippets in Figure 2 as examples). In these attacks, the attacker (1) trains the branch predictor to mispredict victim branches (lines 5 and 11 in Figure 2), and (2) speculatively transmit the secret to a primed channel (lines 7 and 13). Note, that

[1] Our LLVM compiler is open-sourced at https://github.com/Compiler-Dependency-Analysis/llvm-levioso.

```
1 void leak(x){
2   index <- x * 64
3   temp <- load channel[index]
4 }
```

(a) Leaking/declassifying x

```
5 if (idx < sizeA){
6   x <- load A[idx] //spec. secret       control dependent
7   leak(x)          //unsafe transmit
8 }
9 leak(y) //safe transmit                  independent
```

(b) Sandboxed program (speculative secret)

```
10 //non-spec. secrets: state1, state2
11 for (i = 0; i < num_rounds; i++){
12   state1 <- decrypt_ct(state1, key[i])    control dependent
13 leak(state1) //unsafe transmit            data dependent
14 leak(state2) //safe transmit              independent
```

(c) Constant-time program (non-speculative secret)

Figure 2: (a) leak(x) declassifies and leaks value x by transmitting to a channel. (b) and (c) are motivating examples of sandboxed and constant-time programs indicating the instructions that require restriction (*i.e.*, control/data dependent) and the instructions that are independent and do not require restriction.

the secret is also loaded speculatively a sandboxed program (line 6). The processor then detects the branch misprediction, rolls back the misspeculated state, and resumes the execution from the correct path. However, the transmitted secret leaves persistent changes that the attacker can use to extract the secret. Spectre-v1 [18] was the first attack exploiting this vulnerability, with several follow-up variants [3, 6, 7, 15, 25, 34].

2.2 Existing Defenses

Many initial defenses [5, 17, 27, 28, 31, 32, 36] focused on protecting specific channels, such as caches [18, 19]. However, new Spectre variants have demonstrated that other components in the core can act as a channel as well, such as the Branch Target Buffer (BTB) [35]. As an attempt to build more comprehensive countermeasures, recent works have focused on mitigating the attack at the source [10, 24, 33, 35, 38, 39], by completely restricting speculative execution of instructions that can potentially reveal sensitive information. The focus of these works is to block leaks through all potential channels (*i.e.*, channel-agnostic protection). However, they incur high performance overheads by restricting many instructions even if they are safe and do not leak any speculative data.

3 MOTIVATION: HW/SW CO-DESIGN

A comprehensive and efficient defense for speculative execution attacks restricts execution only for the instructions that can potentially leak misspeculated information and avoids restrictions for the instructions that are guaranteed to be safe and commit.

▶ Motivating Example 1: **sandboxed programs**. Figure 2b shows a sandboxed memory access that can potentially leak confidential values in case of branch misprediction (*i.e.*, idx ≥ sizeA). However, value y leaks during the correct execution of the program (*i.e.*, non-speculative leak, line 9). The desired defense would only restrict the execution for control dependent instructions (highlighted as *control dependent*) and prevent unnecessary restrictions for independent instructions (highlighted as *independent*).

▶ Motivating Example 2: **constant-time programs**. Figure 2c shows a legal constant-time implementation of data decryption. Once all rounds of decryption are completed, state1 contains the plaintext and is declassified (line 13); in other words, it is allowed to leak state1. However, in case of a branch misprediction for the loop branch (line 11), state1 leaks before the decryption is completed and is legally declassified. A comprehensive solution to block such leaks will need to restrict the execution for control dependent instructions (line 12) alongside the instructions that their input data is dependent on the control dependent instructions (highlighted as *data dependent*, line 13). On the other hand, state2 is declassified during non-speculative execution (line 14) and leaks the same state independently of the branch outcome. Hence, state2 can leak even if the loop branch (line 11) is unresolved.

▶ Design Choice: We deploy a hardware/software co-design to inform the hardware about guaranteed-to-be-safe speculation. We start with a secure baseline (*i.e.*, restricting execution for all speculative instructions) and lift restrictions only if the static compiler analysis guarantees that the execution is safe. This approach provides comprehensive security while improving performance.

4 THREAT MODEL

We cover all Spectre-type attacks exploiting known sources of speculation (*e.g.*, branch predictions and memory dependence speculation) [3, 6, 15, 18, 25, 34]. The attacker can use any channel to misspeculatively transmit data. We cover attacks targeting both speculative secrets (referred to as sandboxing throughout the paper) and non-speculative secrets (referred to as constant-time programs).

Out of Scope. We do not consider Meltdown-type attacks (exploiting the delays arising from CPU exceptions and faults, e.g., [23], including MDS attacks). New processors are already resistant to them [11]. In addition, attacks exploiting leaks during non-speculative execution are out of scope of this work.

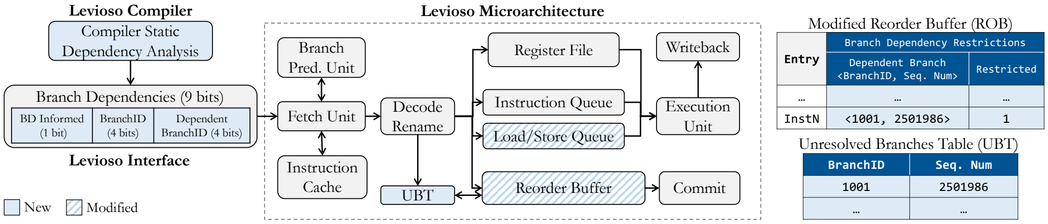
5 LEVIOSO DESIGN

Levioso design deploys a hardware/software co-design solution that efficiently applies minimal restrictions to guarantee strong protections. It provides a compiler and interface (Section 5.1) that communicates true branch dependencies. Hardware uses this information to apply proper restrictions (Section 5.2). Figure 3a shows the Levioso design, highlighting the modifications over existing processor designs and compilers. Levioso has always-on security, even for legacy binaries not compiled by our compiler. But, if needed, Levioso protections can be disabled by simply instrumenting the binary and manipulating the restriction bits through Levioso interface. We built and verified the end-to-end and automated design of Levioso using the LLVM compiler [20] and gem5 simulation [4].

5.1 Levioso Compiler and Interface

Current hardware designs cannot reason about true branch dependencies, as they do not have information about the entire program and its control flow [13]. However, static compiler analysis shows that not all instructions are truly dependent on the branch outcome. Algorithm 1 shows how we detect all the dependent instructions of a conditional branch using the control flow graph (CFG) and data flow graph (DFG).

First, we detect control dependent instructions by visiting all the instructions between the branch and its reconvergence point (*i.e.*,



Modified Reorder Buffer (ROB)

Entry	Branch Dependency Restrictions	
	Dependent Branch <BranchID, Seq. Num>	Restricted
…	…	…
InstN	<1001, 2501986>	1

Unresolved Branches Table (UBT)

BranchID	Seq. Num
1001	2501986
…	…

(a) Levioso End-to-End Design

(b) New/Modified Hardware Structures

Figure 3: (a) Levioso design and (b) the structure of the modified ROB and the new UBT that stores unresolved branches and maps the compiler-specified `BranchID` to the dynamic sequence number. Note, that the Branch Dependencies are set per instruction through prefix bytes of x86 ISA (9 bits required for each instruction). The Load/Store Queue is hashed, similarly to prior work [10, 24, 35, 39], we updated it to prevent speculative store-to-load forwarding attacks [15].

Algorithm 1: Branch Dependents Traversal

```
   Input: Conditional Branch $BR$, $CFG$, $DFG$
   Output: Determining all the dependents of $BR$ ($BR.dependents$)
1  $RecPoint \leftarrow BR$.immediatePostDominator()
2  $control\_dependets \leftarrow$ DFS($CFG$, src: $BR$, dest: $RecPoint$)
3  $data\_dependents$.clear()
4  $processed\_insts$.clear()
5  $working\_set \leftarrow control\_dependents$
6  while $\neg working\_set.empty()$ do
7      $inst \leftarrow working\_set$.pop()
8      if $inst \notin processed\_insts$ then
9          foreach $dep \in inst.direct\_dependents$ do
10             $data\_dependents$.insert($dep$)
11             $working\_set$.push($dep$)
12         end
13         $processed\_insts$.insert($inst$)
14     end
15 end
16 $BR.dependents \leftarrow control\_dependents \cup data\_dependents$
```

the first point after the branch that control flow will reach regardless of the branch outcome) using depth-first traversal (line 2). The reconvergence point of a branch is the immediate post-dominator of the branch in CFG [9]. Next, we detect data dependent instructions. We initialize the `working_set` with control dependent instructions (line 5) and visit all the direct dependencies of the `working_set` (line 9). Direct dependencies are determined through the compiler def-use and alias analysis (*i.e.*, analyzing whether two memory objects point to the same location). If two instructions have a may-alias dependency we consider it as a dependency. Finally, we determine the dependents of the input branch by the union of `control_-dependents` and `data_dependents` (line 16). This compiler pass statically considers all possible paths of the program between a branch and its reconvergence point and considers all instructions that their data can possibly change based on the branch outcome in order to conservatively block all potential speculative data leaks.

Marking dependencies. First, we assign a static identifier to each branch (see `BranchID` in Figure 3a). Second, we mark each instruction with the `BranchID` of the most recent dependent branch detected in the previous step (see `Dependent BranchID` in Figure 3)[2]. Note, that an instruction can depend on multiple branches (*e.g.*, in nested loops), however, we only mark the most inner branch and leverage the chain of dependencies to correctly restrict execution for all dependent instructions (*e.g.*, `Inst` depends on `BR2`, and `BR2` depends on `BR1`; `Inst` will not execute until `BR2` is resolved, and `BR2` does not resolve until prior dependent branches are resolved, *i.e.*, `BR1`; see Section 5.2 for the details of our implementation).

We embed dependency information in the prefix bytes of the x86 ISA instructions. Other ISAs like RISC-V can define new instructions to communicate the compiler information but with the cost of extra instruction decoding. Our experiments show that 4 bits are enough to represent the `BranchID` (see Section 7.2.2). In addition, we set the `BDInformed` bit to 1 for the branches that our compiler is providing dependency information. This bit is used to (1) support both legacy binaries and (2) allow disabling protections if desired (we refer to branches and instructions with `BDInformed=1` as BD_{valid}, and $BD_{invalid}$ otherwise).

5.2 Levioso Microarchitecture

The requirements for Levioso's microarchitecture are: (1) to collect the compiler information communicated by the Levioso interface, and (2) to prevent speculative execution of the instructions that are dependent on at least one unresolved branch in the ROB. Note, that we refer to branch `BR` as resolved only if it is guaranteed to be on the correct path of the program; for unprotected processor and hardware-only defenses, this means that all prior branches are resolved, and in Levioso, it means that all prior branches that `BR` depends on are resolved (*i.e.*, it does not need to wait for independent branches to resolve).

To correctly apply branch dependency restrictions, we introduce a new hardware structure, called the Unresolved Branches Table (UBT). The UBT stores all live unresolved branches (see Figure 3b). The ROB is also modified to: (1) indicate if the execution of an instruction is restricted due to branch dependency (`Restricted` bit in Figure 3b) and (2) store the `BranchID` of the most recent dependent branch (`Dependent Branch` in Figure 3b).

When a *conditional branch* is decoded it will enter the ROB and update the UBT with its compiler-specified `BranchID` and unique dynamic sequence number (dynamic identifier of instructions). In case of a full UBT, we stall inserting new instructions to the ROB (similar to ROB full event) until at least one of the branches in the UBT resolves and is removed from the table (see Section 7.2.2 for the impacts of UBT size). For every decoded instruction entering

[2] In the rare case of an instruction depending on multiple independent branches, we either need to communicate multiple branches or conservatively assume that the instruction depends on all prior unresolved branches (similar to state-of-the-art hardware-only solutions [24]).

the ROB, the processor checks if the instruction's most recent dependent branch exists in the UBT. In case of a hit in the UBT, the instruction's `Restricted` bit is set to 1 in the ROB and execution of the instruction will be restricted. The ROB is also updated with the `BranchID` and sequence number of the branch that the instruction depends on. Note, that if an instruction is marked as $BD_{invalid}$ (seen in legacy binaries) then it will depend on the youngest branch in the ROB and its execution is allowed only if the youngest branch and its prior branches are resolved (similar to hardware-only solutions). Finally, when a branch resolves and the correct path is determined, the `Restricted` bit of all dependent instructions is set to 0 and they can execute (*i.e.,* they are guaranteed to be on the correct path).

For *indirect jumps* (*e.g.,* calls), all instructions after the jump are assumed to be dependent and they execute only if the jump resolves, addressing Branch Target Injection (BTI) attacks, *e.g.*, Spectre-v2. In BTI attacks, indirect jumps are poisoned to execute attacker-chosen targets, which makes it infeasible for static compiler analysis to mark any instruction as independent. Hence, our approach for indirect jumps is similar to hardware-only and other channel-agnostic solutions like Dolma [24]. However, our hardware/software co-design provides an extra level of flexibility to disable the restrictions for indirect jumps in a fine-grained way if the system desires to use other solutions of the CPU, like Intel eIBRS/AutoIBRS [16, 30], while Levioso is still protecting other sources of speculation (*e.g.*, conditional branches). This is possible through the use of binary instrumentation; marking the jump as BD_{valid} but not specifying any dependent branches (*i.e.,* no branch dependency).

In addition, prior works [24, 39] prevent all speculative fetch redirects to mitigate channels relying on transient conditional execution [34] or port contention in SMT processors [3]. However, Levioso does not need to prevent redirects upon branches that do not depend on any of the branches present in the UBT, i.e., unresolved branches. The reason is that the compiler information guarantees that the branch direction does not depend on any existing transient data (*i.e.*, the data that will be squashed), and as a result, it does not reveal any information about transient data.

To prevent memory dependency misspeculation (*e.g.*, speculative store bypass [15]), we deploy a similar approach to prior work [8, 24, 35, 39] by always sending requests for loads even if they match with an older, unresolved store (Load/Store Queue in Figure 3a).

5.3 Levioso Example

Figure 4 shows an example and a snapshot of the ROB where the compiler information helps Levioso to avoid unnecessary restrictions posed by hardware-only defenses like Dolma [24]. There are two branches `BR10` and `BR11`. The condition of `BR10` depends on the value of `Addr` that misses in the cache; hence, facing a long latency to resolve. Dolma restricts all the instructions after `BR10` until it resolves (*i.e.,* `Addr` arrives from memory). However, Levioso's compiler information shows that only three instructions are truly dependent and the rest are independent and can safely execute; increasing instruction and memory level parallelism. The benefits will be even more significant when the following branches resolve faster than `BR10` (*e.g.*, the condition of branch `BR11` depends on the value of `ValueX` which hits in the cache) that allows for parallelization of `C[r4]` and `Addr` memory requests. Our results show that load stalls are the main reason of performance overhead which Levioso improves over Dolma (see Figure 6).

Levioso Markings			Micro-ops in ROB	Levioso Restrictions	Dolma Restrictions
BDI	BR	DepBR			
1	-	9	`load r0 <- Addr //Miss`	no restriction	no restriction
1	-	9	`cmp r0, 10`	no restriction	no restriction
1	10	9	`BR10: jg REC1`	no restriction	no restriction
1	-	10	`load r1 <- A[r0]`	until `BR10` resolves	until `BR10` resolves
1	-	10	`load r2 <- B[r1]`	until `BR10` resolves	until `BR10` resolves
1	-	10	`REC1: add r3 <- r2 + 5`	until `BR10` resolves	until `BR10` resolves
1	-	9	`xor r4 <- r4 ^ r4`	no restriction	until `BR10` resolves
1	-	9	`load r5 <- ValueX //Hit`	no restriction	until `BR10` resolves
1	-	9	`cmp r5, 20`	no restriction	until `BR10` resolves
1	11	9	`BR11: jne REC2`	no restriction	until `BR10` resolves
1	-	11	`add r4 <- r4 + 1`	until `BR11` resolves	until `BR10` resolves
1	-	11	`REC2: load r6 <- C[r4] //Miss`	until `BR11` resolves	until `BR10` resolves

🟩 No Restriction 🟦 Short Latency Restriction 🟥 Long Latency Restriction

Figure 4: Example of Levioso and Dolma. *BDI*: `BDInformed`, *BR*: `BranchID`, *DepBR*: `Dependent BranchID`. Memory requests for `Addr` and `C[r4]` are parallelized in Levioso while serialized in Dolma. `BR9` has already committed in this example.

6 SECURITY ANALYSIS

We define two policies; one for *sandboxing* and another for *constant-time*, and analyze Levioso's secure speculation w.r.t. each policy.

Definition 1 (Sandboxing policy). *The sandboxing policy requires the program to ensure that all memory accesses are within the authorized address range.*

To provide secure speculation for the sandboxing policy, hardware must restrict the execution of branch control dependent instructions and their dependents. Levioso supports sandboxed programs by accurately detecting all control dependent instructions through the compiler and communicating them with the hardware for restriction. Since the static compiler analysis can accurately detect the branch reconvergence point and all possible paths between the branch and its reconvergence point, it eliminates the possibility of potential errors [9].

Definition 2 (Constant-time policy). *The constant-time policy requires that all non-speculative observations of the program (e.g., program counter and memory addresses) are independent and not affected by secret values.*

To provide secure speculation for constant-time programs, hardware must avoid speculative execution of the instructions that leak the values that are loaded in all registers non-speculatively (see Figure 2c as an example of non-speculative secrets). Levioso provides secure speculation for the constant-time policy by conservatively detecting all branch control dependent instructions and their data dependent instructions. In other words, Levioso restricts execution of all the instructions where their leaking values can be different based on the branch outcomes. The performance benefits of Levioso come from allowing execution of the instructions that are guaranteed to be independent of the speculation sources and speculative instructions. While our static compiler analysis might over-approximate data dependencies (due to conservative static alias analysis), it is sound by design and will not declassify any true branch dependency.

Table 1: System configuration for simulation.

L1d Cache	32KB, 8-way	F/D/I/C width	8/8/8/8	RF (INT/FP) size	280/332
L1i Cache	32KB, 8-way	LQ/SQ size	192/114	Branch Predictor	TAGE-SC-L-64KB
L2 Cache	256KB, 8-way	ROB size	512	Data Prefetcher	Stride
L3 Cache	1MB, 16-way	IQ size	97	UBT size	16

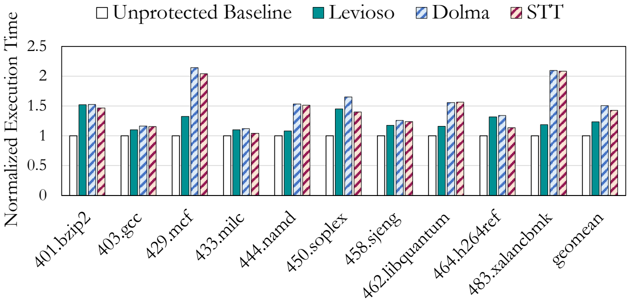


Figure 5: Execution time of evaluated SPEC CPU2006 [14] applications normalized to Unprotected Baseline.

7 EVALUATION

7.1 Experimental Setup

Simulation. We implemented Levioso in gem5 [4] and run simulations in syscall emulation mode. We modify McPAT [22] version 1.3 for power analysis. Table 1 shows the gem5 configuration (a Golden-Cove-like microarchitecture in Intel Alder Lake processors).

Compiler Implementation. We implemented our compiler pass in LLVM 10.0 [20] to detect and mark branch dependencies. Our pass is built at the machine level for the x86 architecture, but it is not architecture-specific and can be ported to other architectures.

Benchmarks. We use a subset of C/C++ applications from SPEC2006 [14] and the ELFie [29] methodology to generate representative (SimPoint) executables with a region size of 1 billion instructions.

7.2 Experimental Results

7.2.1 Performance of SPEC CPU2006. Figure 5 shows the performance results for SPEC CPU2006 applications for four different designs: (1) Unprotected Baseline, (2) Dolma [24], (3) STT [39][3], and (4) Levioso. Dolma and STT show an average performance overhead of 51% and 43% compared to the Unprotected Baseline, respectively. Levioso's performance overhead is just 23% on average. This means that the Levioso methodology reduces the performance loss by 2.22× over Dolma and 1.87× over STT designs on average. In some cases STT shows better performance compared to Levioso (*e.g.*, `464.h264ref`). The reason is that STT assumes that secret transmissions only happen through loads and does not restrict the execution of tainted stores, hence, restricting fewer instructions compared to Levioso and Dolma in some cases. However, prior work [24] demonstrates the vulnerability of STT against Spectre attacks transmitting data via stores.

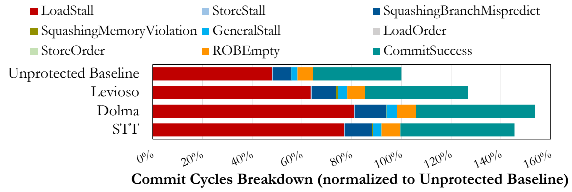


Figure 6: Commit cycle breakdown for different designs normalized to total commit cycles of the Unprotected Baseline.

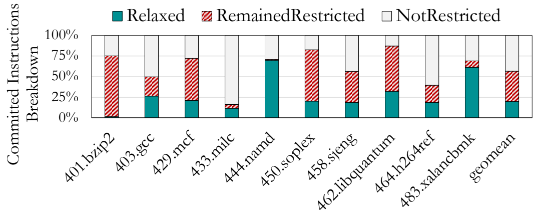


Figure 7: Committed instructions breakdown in Levioso based on their restrictions: (1) Relaxed: restrictions relaxed via compiler information, (2) RemainedRestricted: restrictions remain despite compiler information, (3) NotRestricted: no restrictions. Note, that the instruction counts do not equate to performance.

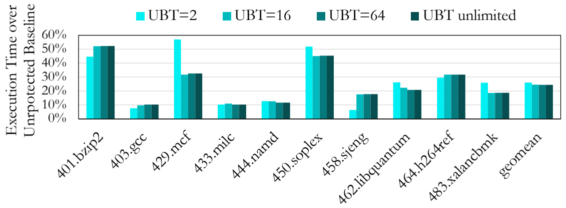


Figure 8: Execution time of Levioso over the Unprotected Baseline with different UBT sizes.

Figure 6 depicts the commit cycles breakdown (*i.e.*, the cycles that at least one instruction is committed) of different designs (bars are normalized to the total commit cycles of the Unprotected Baseline). *LoadStall* is the main reason for blocking the head of the ROB as a result of speculation restrictions; this occurs because restricting execution of instructions will delay memory requests and lead to more cycles waiting for data. Levioso is able to bring down the 81% and 77% of LoadStall in Dolma and STT to 63%, due to relaxing unnecessary restrictions informed by the compiler. Figure 7 shows the percentage of committed instructions in Levioso for different applications that were initially restricted (dependent on an unresolved branch) but Levioso was able to relax the restriction later (when the true dependent branches resolved). Levioso relaxes the restriction of 20% of committed instructions on average based on the compiler information. Some applications benefit from the compiler information significantly; 61% of the restricted instructions in `483.xalancbmk` are relaxed and Levioso shows just an 18% performance overhead, compared to 110% in Dolma and STT.

7.2.2 Impacts of UBT Size. Figure 8 shows the performance impacts for different UBT sizes. The expectation is higher overhead for smaller UBT sizes since it will block inserting instructions to the ROB more frequently (like `429.mcf`, and `450.soplex`). However,

[3] Dolma and STT implementations are adopted from [24] to support both sandboxing and constant-time policies, as defined STT-Spectre (M+R) and Dolma-Default (M+R) in [24]. Moreover, we disable the delay-on-miss optimization of Dolma as it is shown to be vulnerable to speculative interference attacks [2].

we observe that in some cases the Levioso overhead decreases when using a smaller UBT (*e.g.*, `458.sjeng`). More in-depth investigation shows that limiting the number of unresolved branches in some applications reduces the number of squashing cycles and the number of dynamic instructions that the core decodes and re-executes. For example, the number of squashing cycles due to branch misprediction reduces by 1.9× for a UBT size of 2 compared to a UBT size of 16 in `458.sjeng`. A potential future work can deploy a performance-aware and dynamic control of the speculation level to limit the squashing cycles for problematic branches. To have a balanced trade-off for the performance, power, and area, we use a UBT size of 16 entries (as a direct-mapped memory). Levioso consumes only 1.72% more power over the Unprotected Baseline core with an area overhead of 1.42%.

8 RELATED WORK

Channel-specific defenses. Early defenses for speculative execution attacks aimed to secure individual channels, like data caches by either invisible speculation or undo-based speculation [5, 17, 31, 32, 36]. However, these strategies are shown to be ineffective [2, 21]. InvarSpec [40] is a performance optimization for invisible speculation techniques using program analysis. In addition to inheriting security flaws of the underlying defenses, InvarSpec introduces new vulnerabilities [1].

Secure speculation for sandboxing. STT [39] restricts the *execution* and NDA [35] prevents the *propagation* of data from speculative memory accesses (*i.e.*, tainted). STT does not restrict the execution of tainted stores which has been shown to be vulnerable [24]. Following mitigations adopted same insights while improving the performance [38]. Dolma [24] attempts to protect non-speculative secrets, but its performance benefits come from allowing execution for some speculative instructions (under certain conditions) that can lead to exploits through resource contention [2].

Secure speculation for constant-time. Some works [10, 12, 33] adopt dynamic secrecy tracking by manually labeling the secret regions of the memory; speculative execution of the instructions tainted by secret data will be restricted. However, these defenses cannot provide security for legacy software without labeling the secret regions. SPT [8] is a hardware-only defense supporting secure speculation for constant-time that assumes all memory regions are secret and declassifies a region only if it leaks non-speculatively. All these defenses implement taint tracking mechanisms in the hardware and require extensive changes to track the taints in all the components that can potentially process or store sensitive information (*e.g.*, all registers and L1 caches). However, Levioso exploits compiler dependency information and only looks up a small table (UBT) for restrictions.

9 CONCLUSION

We present Levioso, a hardware/software co-design to efficiently and comprehensively protect against speculative execution attacks. Levioso detects and marks true branch dependencies via static compiler analysis and the hardware applies restrictions only when necessary. Levioso significantly reduces the performance overhead compared to prior defenses (with the same threat model) from 51% and 43% to just 23%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Arash Pashrashid, Shweta Shinde, and Prateek Saxena for their valuable feedback.

REFERENCES

[1] Pavlos Aimoniotis, et al. 2021. It's a Trap! - How speculation invariance can be abused with forward speculative interference. arXiv:cs.CR/2109.10774
[2] Mohammad Behnia, et al. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS 2021*.
[3] Atri Bhattacharyya, et al. 2019. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS 2019*.
[4] Nathan Binkert, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* (2011).
[5] Thomas Bourgeat, et al. 2019. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO 2019*.
[6] Guoxing Chen, et al. 2019. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *EuroS&P 2019*.
[7] Yun Chen, et al. 2024. GadgetSpinner: A new transient execution primitive using the Loop Stream Detector. In *HPCA 2024*.
[8] Rutvik Choudhary, et al. 2021. Speculative Privacy Tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO 2021*.
[9] Ron Cytron, et al. 1989. An efficient method of computing static single assignment form. In *POPL 1998*.
[10] Lesly-Ann Daniel, et al. 2023. ProSpeCT: Provably secure speculation for the constant-time policy. In *USENIX Security 2023*.
[11] Deep Dive: CPUID enumeration and architectural MSRs 2023. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/overview.html#MDS-CPUID.
[12] Jacob Fustos, et al. 2019. SpectreGuard: An efficient data-centric defense mechanism against spectre attacks. In *DAC 2019*.
[13] Ali Hajiabadi, et al. 2021. NOREBA: A compiler-informed non-speculative out-of-order commit processor. In *ASPLOS 2021*.
[14] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* (2006).
[15] Jann Horn. 2018. Speculative execution, variant 4: Speculative store bypass.
[16] Intel, Indirect Branch Restricted Speculation 2018. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.
[17] Khaled N Khasawneh, et al. 2019. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *DAC 2019*.
[18] Paul Kocher, et al. 2019. Spectre Attacks: Exploiting speculative execution. In *S&P 2019*.
[19] Esmaeil Mohammadian Koruyeh, et al. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT 2018*.
[20] Chris Lattner et al. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*.
[21] Mengming Li, et al. 2022. unXpec: Breaking undo-based safe speculation. In *HPCA 2022*.
[22] Sheng Li, et al. 2013. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *TACO* (2013).
[23] Moritz Lipp, et al. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security 2018*.
[24] Kevin Loughlin, et al. 2021. Dolma: Securing speculation with the principle of transient non-observability. In *USENIX Security 2021*.
[25] Giorgi Maisuradze et al. 2018. ret2spec: Speculative execution using return stack buffers. In *CCS 2018*.
[26] Daniel S McFarlin, et al. 2013. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?. In *ASPLOS 2013*.
[27] Hamza Omar et al. 2020. IRONHIDE: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *HPCA 2020*.
[28] Arash Pashrashid, et al. 2023. HidFix: Efficient mitigation of cache-based Spectre attacks through hidden rollbacks. In *ICCAD 2023*.
[29] Harish Patil, et al. 2021. ELFies: Executable region checkpoints for performance analysis and simulation. In *CGO 2021*.
[30] Kim Phillip. 2022. LKML: [PATCH 0/3] x86/speculation: Support Automatic IBRS. (2022).
[31] Gururaj Saileshwar et al. 2019. Cleanupspec: An "undo" approach to safe speculation. In *MICRO 2019*.
[32] Christos Sakalis, et al. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA 2019*.
[33] Michael Schwarz, et al. 2020. ConTExT: A generic approach for mitigating Spectre. In *NDSS 2020*.
[34] Michael Schwarz, et al. 2018. NetSpectre: Read arbitrary memory over network. *CoRR* abs/1807.10535 (2018).
[35] Ofir Weisse, et al. 2019. NDA: Preventing speculative execution attacks at their source. In *MICRO 2019*.
[36] Mengjia Yan, et al. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO 2018*.
[37] Yuval Yarom et al. 2014. Flush+Reload: A high resolution, low noise, L3 cache Side-Channel attack. In *USENIX Security 2014*.
[38] Jiyong Yu, et al. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *ISCA 2020*.
[39] Jiyong Yu, et al. 2019. Speculative Taint Tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO 2019*.
[40] Zirui Neil Zhao, et al. 2020. Speculation invariance (InvarSpec): Faster safe execution through program analysis. In *MICRO 2020*.