Mitigating Speculation-based Attacks through Configurable Hardware/Software Co-design

Ali Hajiabadi  
National University of Singapore

Andreas Diavastos  
National University of Singapore

Archit Agarwal  
National University of Singapore

Trevor E. Carlson  
National University of Singapore

ABSTRACT

New speculation-based attacks that affect large numbers of modern systems are disclosed regularly. Currently, CPU vendors regularly fall back to heavy-handed mitigations like using barriers or enforcing strict programming guidelines resulting in significant performance overhead. What is missing is a solution that allows for efficient mitigation and is flexible enough to address both current and future speculation vulnerabilities, without additional hardware changes.

In this work, we present SpecControl, a novel hardware/software co-design, that enables new levels of security while reducing the performance overhead that has been demonstrated by state-of-the-art methodologies. SpecControl introduces a communication interface that allows compilers and application developers to inform the hardware about true branch dependencies, confidential control-flow instructions, and fine-grained instruction constraints in order to apply restrictions only when necessary. We evaluate SpecControl against known speculative execution attacks and in addition, present a new speculative fetch attack variant on the Pattern History Table (PHT) in branch predictors that shows how similar previously reported vulnerabilities are more dangerous by enabling unprivileged attacks, especially with the state-of-the-art branch predictors. SpecControl provides stronger security guarantees compared to the existing defenses while reducing the performance overhead of two state-of-the-art defenses from 51% and 43% to just 23%.

1 INTRODUCTION

Speculative execution attacks, like Spectre [37], are a major concern in current and future modern processor designs as they exploit the main enabler of their performance, speculative execution [52]. Speculative execution attacks trick the processor into executing unintended paths of the program in a speculative manner and force the victim to access sensitive or secret information. The attacker can then reconstruct the secret data by probing the components containing the footprints of mispredicted and secret dependent paths which have not been rolled back after misspeculation. To mitigate speculative execution attacks, the most comprehensive solutions try to prevent transmitting secret information through any side-channel by restricting the execution of speculative instructions. For example, STT [77] deploys a dynamic taint tracking technique that restricts the execution of instructions that are tainted by speculative loads. Following solutions [20, 47, 63] use the same insight, and propose hardware-only mechanisms to detect and restrict the speculative execution of problematic instructions. These solutions provide secure speculation for sandboxed programs that guarantee their memory accesses are within the authorized address range during the correct execution. However, they fail to provide secure speculation for constant-time programs, since these programs might load the secret data non-speculatively. To protect non-speculative secrets, speculative execution of all instructions tainted by secret values must be restricted. Current solutions for constant-time programs either manually specify secret regions of memory [21, 24, 63] or assume all memory regions are secret and declassify them only if they leak during non-speculative execution [20].

Unfortunately, most of these solutions tend to significantly reduce the benefits of speculative execution, as they follow a conservative approach of restricting the execution of the majority of speculative instructions after an unresolved branch. Our study shows that STT and DOLMA incur 43% and 51% performance overhead over an unprotected processor, respectively. However, our studies show that many of the restrictions introduced by these mitigations are unnecessary because not all instructions after a branch are truly dependent on the branch outcome. Such independent instructions can safely execute without compromising security. Hardware-only mitigations do not have sufficient information on true branch dependencies and all possible control-flow paths of the program, as the processor only views a small window of instructions at any given moment. Our key insight in this paper is that a hardware/software co-design provides a more efficient solution, where the software informs the hardware about true branch dependencies and the hardware applies restrictions only to truly speculative instructions.

In addition, prior solutions offer a limited scope of protection, as they only protect against current attacks exploiting speculative execution. However, modern processors have several components that hold speculative and confidential information (e.g., the Branch Prediction Unit) that can potentially leak, leading to the frequent discovery of new vulnerabilities. Previous research has shown that the branch predictor can leak secret information through the Pattern History Table (PHT) [22]. However, it was not considered a serious threat due to the limited leakage and attack assumptions. We demonstrate a new variant of the attack that an unprivileged attacker can extract a 19-bit private key of RSA without priming the PHT and we show that this attack becomes even more serious with better branch predictors. We discuss that such attacks need to be categorized as speculation-based vulnerabilities because the secret extraction phase of the attack exploits the speculation operations at the processor’s front-end. We call this class of attack

\footnote{We use modified versions of STT (to protect non-speculative secrets) and DOLMA (disabling Delay-on-Miss optimization to protect against speculative interference attacks [16]).}
Speculative Fetch Attacks. Speculatively fetching instructions and redirecting fetch in case of a misprediction creating the timing difference required to recover the control-flow information of the victim.

Defenses for speculative fetch attacks tend to be very challenging and inefficient, and often even impractical. Software-only defenses require the program to remove all the confidential control-flow instructions (i.e., secret dependent branches or branches representing private activities of the users). Such transformation (e.g., if-converting the branches [48]) is challenging for real-world applications, with respect to performance and practicality of removing complex control-flow patterns. On the other hand, hardware-only solutions have no other choice other than disabling the branch predictor since they have no information about the confidentiality of newly fetched branches. We believe that a hardware/software co-design and communication of confidential branches allows high-performance processors to enforce fine-grained front-end restrictions only for sensitive branches.

In this work, we propose SpecControl, a mechanism to provide both comprehensive security and high performance through a configurable hardware/software co-design. To accomplish this, we introduce a software interface for compilers and application developers to communicate true branch dependencies and confidential control-flow instructions to the hardware. Our hardware uses software information to apply restrictions only if necessary, either at the front-end or the back-end of the processor (mitigating both speculative fetch and execution attacks). SpecControl provides secure speculation for both sandboxing and constant-time policies, in addition to a new policy that guarantees the confidentiality of control-flow instructions (called control-flow confidentiality). The SpecControl interface is flexible enough to define new secure speculation policies for future attacks and programs that might require additional or fewer restrictions, without additional changes to the hardware in the future.

The main contributions of this work are:

- A novel hardware/software co-design methodology, called SpecControl, that enables configurable and targeted mitigation for speculation-based attacks;
- Our targeted and automated mechanism reduces performance overhead compared to state-of-the-art solutions, DOLMA [47] and STT [77], from 51% and 43% to just 23%, with negligible power and area overheads;
- We demonstrate a new variant of speculative fetch attacks that allows an unprivileged attacker to extract a 19-bit RSA private key from modern branch predictors. Additionally, we show the possibility of the attack on Intel, AMD, and Apple CPUs2;
- Enhanced security against speculative fetch attacks.

2 BACKGROUND AND MOTIVATION

In prior work [20, 47, 63, 70, 77], an attack is considered speculative only if the secret is leaked at the processor back-end during speculative execution. In this work, we expand this definition to consider an attack to be speculative if any step of the attack exploits speculative operations of the core to recover secret information, even if the secret is transmitted to a channel non-speculatively. In this section, we describe two categories of speculation-based attacks: (1) speculative execution, and (2) speculative fetch attacks.

2.1 Speculative Execution Attacks

Speculative execution attacks aim to bypass the address protection mechanisms of sandboxes or speculatively transmit the secret data of constant-time programs (see Figure 1). In these attacks, the attacker (a) mistrains the branch predictor to always take the victim branch even if the condition is not true (line 2 in Figure 1a and line 4 in Figure 1b), and (b) speculatively (loading the secret into the core in a sandboxed program and) transmitting the secret to a primed side-channel (e.g., the cache). The processor then detects the branch misprediction, rolls back the mispredicted state, and resumes the execution from the correct path. However, the transmitted secret leaves behind persistent state that the attacker can use to extract the secret. Spectre-V1 [37] was the first attack exploiting this vulnerability, with several follow-up variants [11, 18, 29, 34, 39, 49, 65].

2.1.1 Existing Defenses. Many initial defenses for speculative execution attacks [14, 33, 54, 58, 61, 62, 74] focused on solutions to mitigate specific channels, such as caches [35–38, 50]. However, new Spectre variants have demonstrated that other components in the core can act as a side-channel as well, such as the Branch Target Buffer (BTB) [70]. As an attempt to build a more comprehensive set of countermeasures, recent works have focused on mitigating the attack at the source [21, 47, 63, 70, 75–77], by completely restricting speculative execution of instructions that can potentially reveal sensitive information. The focus of these works is to block leaks through any potential channel, but only for speculative execution attacks. However, there are still challenges to be addressed as hardware-only solutions fail to completely solve.

2.2 Speculative Fetch Attacks

Speculative fetch attacks exploit the fact that the branch predictor remembers the decisions originally made by the victim. Figure 2 shows a victim gadget that consists of a confidential branch that is vulnerable to speculative fetch attacks (Branch B1). BranchScope [22] is an example of such attack, where the attacker performs three steps to extract a secret key: (1) the attacker primes one of the PHT entries into a known state (e.g., strongly taken), (2) the victim executes its secret dependent branch (colliding with the primed PHT entry), and (3) the attacker can extract the secret bit by probing the same PHT entry. Unfortunately, BranchScope requires OS control to interrupt the victim exactly at each iteration of the secret dependent branch to extract the secret bit by bit. We consider this attack to be a speculation-based vulnerability as the secret

---

2We have responsibly disclosed our attack to the affected vendors and received the approval to distribute our findings.
We present a new variant of speculative fetch attacks that, unlike previous work [22], allows an unprivileged attacker to successfully launch an attack, without requiring the OS to precisely control the victim execution and without requiring the attacker to prime the branch predictor. We exploit the fact that state-of-the-art branch predictors can remember branch patterns and therefore can extract the confidential information that has already been learned. Our attack only performs two steps without interfering with the victim’s execution:

**Step 1 (victim): Train the PHT with the secret.** The victim runs normally and trains the PHT with its secret information and branch decisions;

**Step 2 (attacker): Measure the execution time of the attack gadget.** After victim execution, the attacker needs to (a) create a PHT collision with the victim branch [16, 36] and (b) make a guess about the secret. The attacker evaluates the correctness of the guess to extract the secret.

Figure 2 shows an example of victim and attack gadgets, along with a timeline of the events during the second step of the attack. If the attacker guesses correctly, the instructions are fetched from the correct path and are not squashed later. However, in the case of an incorrect guess, the incorrectly fetched instructions are squashed and the processor starts fetching and executing instructions from the correct path (speculative execution is disabled in Figure 2). As shown in the figure, the attacker can evaluate the correctness of the guess by measuring the execution time of the attack gadget.

This attack exploits the victim itself to prime and train the PHT state with the secret. Our investigations using gem5 simulation [12] with different branch predictors (Figure 4) show that TAGE-SC-L-64KB [4], the current state-of-the-art branch predictor, is able to learn all 19 bits of the private exponent in a Montgomery Ladder RSA [32] (Figure 3) after 1,000 rounds of decryption. Older branch predictors fail to remember all the bits. For example, MultiperspectivePerceptron64KB [30, 31] provides incorrect predictions for 4 of the secret bits. In other words, deploying increasingly better branch predictors make speculative fetch attacks more serious than was previously realized. In Section 6.4, we demonstrate the feasibility of our attack on real existing CPUs.

**2.2.2 Attack Surface.** Any program with control-flow decisions that carry confidential information is vulnerable to speculative fetch attacks. Many existing cryptographic implementations have secret dependent branches, like the RSA modular exponentiation in GnuPG 1.4 [2], OpenSSL v1.0.1e [55], MbedTLS 3.0.0 [51], LibreSSL 3.3.6 [60], and wNAF algorithm in secp256k1 curve (found in OpenSSL v1.1.0h [23]). In addition, many real-world applications have branches that can leak confidential information (e.g., users’ Bluetooth connections [6] and battery properties [5] in the Linux kernel).

**2.2.3 Existing Defenses.** To defend against speculative fetch attacks, software-only defenses adopt strict policies like data-oblivious programming [46, 53, 59, 66] (e.g., removing secret dependent branches). Adapting to such guidelines is challenging for real-world and control-flow intensive programs, and even many cryptographic modules still choose to use secret dependent branches. These applications secure non-speculative leaks of the program with lightweight spot defenses based on the adversaries’ capabilities. On the other hand, hardware-only defenses have no prior information about newly fetched branches at the front-end and their only option is to disable the branch predictor, an extreme measure that significantly reduces performance, especially in highly speculative modern processors. We show that, for example, a secure baseline processor with no branch predictor can introduce up to a 3.72X performance overhead. Note, that clearing the branch predictor after context switches does not solve the problem for hyperthreading-based scenarios and also

![Figure 4: Investigation of different branch predictors in gem5 and their prediction for RSA private key after 1,000 rounds of decryption. The results show that TAGE-SC-L-64KB remembers the entire key. Note, that a Taken decision means that the secret bit is 0 based on the RSA code in Figure 3. The secret is (1011001111101111111).](image-url)
requires the trust of the OS to not exploit the branch predictor’s state before clearing it.

This speculative fetch vulnerability is a microarchitectural leak that software verification tools cannot reason about (as not all of the microarchitectural features of processors are disclosed); they can only analyze the correct execution of the program. In this work, we introduce a new secure speculation policy, called control-flow confidentiality, where the hardware guarantees secure speculation for confidential branches.

2.3 Challenges of the State-of-the-Art

In this work, we identify four challenges of state-of-the-art defenses for speculation-based vulnerabilities:

1. **Hardware-only solutions for speculative execution attacks apply unnecessary restrictions for the instructions that will not leak data due to speculation.** Our analysis shows that not all instructions after unresolved branches are truly dependent on the branch [27].

2. **Software-only and hardware-only solutions cannot efficiently protect confidential control-flow decisions.** Existing mechanisms are challenging for real-world applications and incur prohibitive performance overheads.

3. **Future security challenges:** The rapid development of new speculative attacks [9, 10, 42, 72] invalidates many of the existing state-of-the-art (performance-wise and security-wise) defenses, and they are unable to recover from newly discovered attacks. Hence, application developers and CPU vendors currently need to fall back to heavy-handed protections (e.g., fences, or strict programming guidelines). A fine-grained solution providing sufficient protections that limits performance penalties would help to alleviate these issues.

4. **Future performance challenges:** While new speculation vulnerabilities emerge rapidly, our understanding of the programs and vulnerable workloads can continue to improve. We believe that even a hardware/software co-design, as we propose in this work, needs to be flexible enough to benefit from future compiler and programming enhancements. For example, future compilers might be able to extract more accurate and less conservative instruction dependencies or new efficient programming disciplines that require fewer restrictions from the hardware.

2.4 Our Approach: HW/SW Co-design

Our key insight is that we can enable fine-grained and targeted restriction of speculative fetch and speculative execution only for problematic instructions using targeted hardware/software communication. In this work, we propose a software interface that allows compilers and application developers to communicate confidential branches (to restrict them at the front-end and defend against speculative fetch attacks) and true branch dependencies (to only restrict the execution of the instructions that are truly dependent on unresolved branches and defend against speculative execution attacks). Our proposed microarchitecture only modifies the front-end logic to detect and restrict the fetch of confidential branches and adds a new table at the backend, the Unresolved Branches Table (UBT), that keeps track of the live unresolved branches residing in the ROB and applies execution restrictions to an instruction only if it depends (as informed by the compiler) on any of the branches in the UBT.

Our hardware/software co-design is flexible enough to allow compilers and developers to apply or relax front-end and back-end restrictions without additional hardware updates to provide (1) a configurable methodology to defend against future vulnerable instructions and paths of the programs, and (2) the flexibility to configure the hardware restrictions based on future advancements of programs and compilers, hence, providing a potential for future performance improvements.

3 THREAT MODEL

In this paper, we focus on securing speculation for software following (1) sandboxing policies, (2) constant-time policies [21, 26], as well as (3) control-flow confidentiality, covering all known speculative fetch, and speculative execution attacks (see Section 5 for the definitions and analysis of our security guarantees).

Speculative execution attacks include all Spectre-like attacks [11, 18, 29, 34, 37, 39, 49, 65]. Speculative fetch attacks refer to speculative vulnerabilities introduced at the front-end, like BranchScope [22] and our new variant of the attack. The assumptions we make for speculative fetch attacks are: (1) the Pattern History Table (PHT) must be shared between the attacker and the victim. This means that the attacker and the victim must run on the same physical core and share PHT entries, (2) the attacker can trigger or is able to watch for and validate the victim code execution, and (3) the attacker can launch the attack with user-only privileges in the system, without requiring any OS capabilities.

**Out of Scope:** We do not consider Meltdown-type attacks [15, 45, 64, 67, 68, 71] that exploit the delay between exceptions and when they are handled. New processors patch this class of attacks [1]. In addition, attacks exploiting non-speculative leaks of the programs [19, 25, 41, 44, 57, 69] are out of scope of this work.

4 SPECCONTROL DESIGN

SPECCONTROL design deploys a configurable solution that efficiently applies minimal restrictions to guarantee strong protections. It provides a lightweight software interface (Section 4.1) that allows the compiler and the applications developers to mark true branch dependencies for targeted front-end or back-end restrictions. This information is communicated to the hardware and the processor applies the proper restrictions for marked instructions (Section 4.2).

Figure 5 shows the SPECCONTROL design, highlighting the modifications over existing processor designs and compilers. SPECCONTROL has always-on security for sandboxed and constant-time programs, even for legacy binaries not compiled by our compiler. But, if needed, SPECCONTROL protections can be disabled by simply instrumenting the binary and manipulating the restriction bits of the x86 instruction prefix bits. We built and verified the end-to-end and automated design of SPECCONTROL in LLVM compiler [40] and gem5 simulation [12].

4.1 SPECCONTROL Interface

SPECCONTROL interface has two main features: (1) marking true branch dependencies of the instructions (Section 4.1.1); this will help
the processor to only restrict speculative execution of the instructions that are truly dependent on unresolved branches, (2) Marking the confidential control-flow instructions (Section 4.1.2); this informs the processor to restrict the fetch unit upon marked instructions. While these two features mitigate both speculation execution and speculative fetch attacks, SpecControl interface is highly configurable to apply or relax other fine-grained restrictions.

4.1.1 Communicating Branch Dependencies. While the hardware assumes all instructions after an unresolved branch are control dependent, a static compiler analysis shows that not all instructions are truly dependent on the branch’s outcome. Therefore, restricting the execution of all instructions after unresolved branches is not required to provide secure speculation; since independent instructions do not leave unintended persistent changes if a branch is mispredicted. Current hardware designs, however, cannot reason about true branch dependencies, as they do not have information about the entire program and its control flow [27].

Algorithm 1 shows how we detect all the dependent instructions of a conditional branch that performs a traversal search over the data-flow graph (DFG). First, we initialize the working_set with the branch control dependent instructions (i.e., all the instructions reachable between the branch and its reconvergence point; line 3). Next, we visit all the direct dependent instructions of the working_set (line 9). Direct dependencies are determined through the compiler def-use and alias analysis. If two instructions have a may-alias dependency we consider it as a dependency in order to provide a fully non-speculative and conservative list of dependences. At the end of this search, all the dependent instructions will be in the working_set of the input branch (line 10). This compiler pass statically considers all possible paths of the program after a branch and conservatively considers all may-alias dependencies as true dependencies, hence, it is sound by design and does not miss any potential dependency.

Marking dependencies. First, we assign a static identifier (BranchID) to each branch (see BranchID in Figure 5). Second, we mark each instruction with the BranchID of the most recent dependent branch detected in the previous step (see Dependent BranchID in Figure 5)\(^1\). We embed this information in the prefix bytes of the x86 ISA instructions. Note, that other ISAs like RISC-V can define new instructions to communicate the compiler information but at the cost of extra instruction decoding. Our experiments show that 4 bits are enough to represent the BranchID. In addition, we set the BD Informed 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 (see Table 1 for detailed options).

4.1.2 Communicating Speculation Restrictions. The SpecControl interface also allows the developers and the compiler to directly mark instructions that need to be restricted either at the front-end or the back-end of the processor. In our baseline x86 ISA, we use two prefix bits of the instructions to inform the hardware about these marked restrictions: one bit that indicates if the instruction needs to be restricted at the front-end (Front-end restricted in Figure 5) and one bit to restrict the instruction at the back-end (Back-end restricted). Table 1 shows all restriction markings in SpecControl. ResFE refers to front-end restriction and ResBG means restricted at the back-end. Resnone also indicates no restriction.

To address speculative fetch attacks, we need to mark secret dependent and confidential branches as ResFE, which restricts the branch at the front-end from accessing/updating the branch predictor. For the evaluation of this work, we deploy a state-of-the-art taint instruction depends on all prior unresolved branches (similar to state-of-the-art hardware-only solutions [47]), by marking the branch as BRinformed.

---

\(^1\)In the rare case of an instruction depending on multiple independent branches, we either need to communicate multiple branches or conservatively assume that the

---

**Figure 5:** SpecControl design. The highlighted components are added by SpecControl on top of the baseline compiler and processor and the hashed components are modified by SpecControl. UBT is the Unresolved Branches Table that keeps the live unresolved branches in the ROB. Note, that the Speculation Restrictions and Branch Dependencies are set per instruction through prefix bytes of x86 ISA (11 bits required for each instruction). The Load/Store Queue is hashed, similarly to prior work [21, 47, 70, 77], we updated it to prevent store-to-load bypass attacks [29].

**Algorithm 1: Branch Dependents Traversal**

```
Input: Branch BR
Output: Determining all the dependents of BR
1  working_set.clear()
2  processed_insts.clear()
3  working_set ← BR.control_dependents
4  while ¬working_set.empty() do
5     inst ← working_set.pop()
6     if inst ∈ processed_insts then
7         continue
8  end
9  foreach dep ∈ inst.direct_dependents do
10     BR.depenents.insert(dep)
11     working_set.push(dep)
12  end
13  processed_insts.insert(inst)
```

---
### 4.2 SpecControl Microarchitecture

The requirements for hardware implementation of SpecControl are: (1) to collect the software information communicated by SpecControl interface, (2) to prevent the speculative fetch of the instructions that are marked as confidential (i.e., ResFE), and (3) to prevent speculative execution of the instructions that are dependent on at least one unresolved branch in the ROB or directly marked as back-end restricted (i.e., ResBE). Table 2 shows a detailed event-action description of the SpecControl microarchitecture.

#### 4.2.1 Front-end Restrictions

The fetch unit is modified to check and validate the speculation restriction bits of a fetched instruction. If the restriction bits of a branch are marked as ResFE, the fetch unit does not use the branch predictor to speculatively fetch subsequent instructions and instead stalls. When the branch resolves, the processor resumes fetching instructions from the correct, non-speculative path (step ❶ in Table 2). This way, unsafe branches will not leave traces in the speculative components of the front-end (e.g., the branch predictor); hence, protecting against speculative fetch attacks.

#### 4.2.2 Back-end Restrictions

For back-end restrictions, we add an extra bit to the ROB, called Backend Restricted (see Figure 6). If an instruction marked by the compiler as ResBE enters the ROB then the execution of the instruction will be restricted (Backend Restricted bit set to 1 in the ROB, step ❺). When the instruction becomes safe (e.g., all input operands are non-speculative) and guaranteed to commit (e.g., for loads and stores it means they resolve in Load/Store Unit and their page-table access succeeds) then the Backend Restricted bit is set to 0 and the instruction is allowed to execute and forward its results to the dependent instructions (step ❹).

In addition, the Backend Restricted bit is used to restrict the instructions that are not analyzed by our compiler (e.g., instructions marked as BDinvalid and branches marked as BRinvalid). All instructions after a branch marked as BRinvalid will be restricted at the back-end until the branch resolves (step ❺). Hence, we provide always-on security for speculative execution attacks. The developers and the compiler can disable the protections by marking the instructions as BDno and BRno, as explained in Table 1.

#### 4.2.3 Branch Dependency Restrictions

To correctly apply branch dependency restrictions communicated by the compiler, we introduce a new hardware structure, called the Unresolved Branches Table (UBT). The UBT stores all live unresolved branches (see Figure 6). The ROB is also modified to: (1) indicate if the execution of an instruction is directly restricted by branch dependency (Restricted bit in Figure 6) and (2) store the Branch ID of the most recent dependent branch (Dependent Branch in Figure 6).

When a branch instruction 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) (step ❼). Note, that the branch should have BRvalid marking which means it has been analyzed by the compiler. In case of a full UBT, we stall inserting new instructions to ROB (i.e., ROB full) until at least one of the prior branches resolves and then resume decoding and inserting instructions to the ROB (see Section 6.2.2 for the impacts of UBT size). For every decoded instruction entering the ROB, the processor checks if the instruction’s most recent

---

### Table 1: SpecControl options to mark restrictions for an instruction.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Front-end</th>
<th>Back-end</th>
<th>Action</th>
<th>Tag</th>
<th>Front-end</th>
<th>Back-end</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resno</td>
<td>0 0</td>
<td>No additional restrictions</td>
<td>BDinvalid</td>
<td>X</td>
<td>INVALID</td>
<td>The instruction will be dependent on the most recent branches</td>
<td></td>
</tr>
<tr>
<td>ResFE</td>
<td>1 0</td>
<td>Restricted at the front-end</td>
<td>BDno</td>
<td>0</td>
<td>VALID</td>
<td>No branch dependency restriction</td>
<td></td>
</tr>
<tr>
<td>ResBE</td>
<td>0 1</td>
<td>Restricted at the back-end</td>
<td>BDvalid</td>
<td>1</td>
<td>VALID</td>
<td>The instruction will be dependent the branch specified by BranchID</td>
<td></td>
</tr>
</tbody>
</table>

Note, that setting both front-end and back-end bits means restricting at both stages, however, it has the same functionality as ResFE; because if an instruction is restricted at the front-end then the next instructions will be non-speculative at the back-end and no extra restriction will be applied. This policy ensures secure speculation for legacy binaries (except for control-flow confidentiality).

---

Legacy binaries can also perform this by instrumentation.
Table 2: Events and actions in SpecControl. For instruction \(\text{Inst, Inst.DependentBranchID}\) is specified by the compiler in the prefix bits and for branch \(\text{BR, BR.BranchID}\) is an ID assigned by the compiler and specified in the prefix bits as well (see Figure 5).

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Branch BR (marked as (\text{Resp}_0)) fetched while BR is unresolved: block fetch</td>
</tr>
<tr>
<td>2.</td>
<td>Instruction (\text{Inst}) (marked as (\text{Resp}_0)) enters ROB (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 1)</td>
</tr>
<tr>
<td>3.</td>
<td>Instruction (\text{Inst}) (marked as (\text{Resp}_0)) resolves (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 0)</td>
</tr>
<tr>
<td>4.</td>
<td>Branch BR (marked as (\text{BR}_{\text{valid}})) enters ROB while UBT is full: stall inserting new instructions to the ROB (\text{UBT}[\text{BR.BranchID}] = \text{SeqNum})</td>
</tr>
<tr>
<td>5.</td>
<td>Instruction (\text{Inst}) (marked as (\text{BR}<em>{\text{valid}})) enters ROB if UBT ([\text{Inst.DependentBranchID}]) exists: (\text{ROB}[\text{Inst}].\text{Restricted} = 1); (\text{ROB}[\text{Inst}].\text{DependentBranch} = \langle \text{Inst.DependentBranchID}, \text{UBT}[\text{Inst.DependentBranchID}] \rangle) if an unresolved branch marked as (\text{BR}</em>{\text{invalid}}) or indirect jump not marked as (\text{BR}_{\text{no}}) exists in ROB: (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 1)</td>
</tr>
<tr>
<td>6.</td>
<td>Branch BR (marked as (\text{BR}_{\text{valid}})) resolves for all entries (E) in ROB: if (\text{ROB}[E].\text{DependentBranch} = \langle \text{BR.BranchID}, \text{BR.SeqNum} \rangle): (\text{ROB}[E].\text{Restricted} = 0) Remove BR from UBT</td>
</tr>
<tr>
<td>7.</td>
<td>Branch Inst (marked as (\text{BR}<em>{\text{invalid}})) or indirect jump Inst (marked as (\text{BR}</em>{\text{valid}}) or (\text{BR}<em>{\text{invalid}})) resolves for all entries (E) in ROB after Inst: if (E) is marked as (\text{BR}</em>{\text{invalid}}) or (E) is indirect jump and marked as (\text{BR}_{\text{valid}}): break else if (E) not marked as (\text{Resp}_0): (\text{ROB}[E].\text{BackendRestricted} = 0)</td>
</tr>
</tbody>
</table>

5 SECURITY ANALYSIS

A processor with secure speculation guarantees the absence of additional vulnerabilities on top of the non-speculative leaks of a given program (i.e., the leaks during the correct execution). Two main programming policies (also known as software contracts [26]) are (1) sandboxing, and (2) constant-time execution that a processor requires to protect them with secure speculation. These policies are defined as followed:

Definition 5.1 (Sandboxing policy). The sandboxing policy requires the program to ensure that all memory accesses are not outside the authorized address range.

Definition 5.2 (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.

SpecControl secure speculation for sandboxing and constant time policies: To provide secure speculation for the sandboxing policy, the hardware should restrict the execution of speculative memory accesses and their dependent instructions. However, secure speculation for constant-time programs (commonly used for cryptographic algorithms) requires the hardware to avoid speculative execution of the instructions that are tainted by secret values, even if the secret itself has been loaded non-speculatively. SpecControl supports secure speculation for both sandboxed and constant-time programs since it restricts the execution of all speculative instructions. The performance benefits of SpecControl come from allowing the execution of the instructions that are guaranteed to be independent of the speculation sources and speculative instructions. In other words, SpecControl assumes that all instructions after speculation sources (e.g., conditional branches and indirect jumps) are secret dependent, unless the compiler declassifies the instructions that are independent of the speculation sources (i.e., they will leak during the non-speculative execution as well). While

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Branch BR (marked as (\text{Resp}_0)) fetched while BR is unresolved: block fetch</td>
</tr>
<tr>
<td>2.</td>
<td>Instruction (\text{Inst}) (marked as (\text{Resp}_0)) enters ROB (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 1)</td>
</tr>
<tr>
<td>3.</td>
<td>Instruction (\text{Inst}) (marked as (\text{Resp}_0)) resolves (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 0)</td>
</tr>
<tr>
<td>4.</td>
<td>Branch BR (marked as (\text{BR}_{\text{valid}})) enters ROB while UBT is full: stall inserting new instructions to the ROB (\text{UBT}[\text{BR.BranchID}] = \text{SeqNum})</td>
</tr>
<tr>
<td>5.</td>
<td>Instruction (\text{Inst}) (marked as (\text{BR}<em>{\text{valid}})) enters ROB if UBT ([\text{Inst.DependentBranchID}]) exists: (\text{ROB}[\text{Inst}].\text{Restricted} = 1); (\text{ROB}[\text{Inst}].\text{DependentBranch} = \langle \text{Inst.DependentBranchID}, \text{UBT}[\text{Inst.DependentBranchID}] \rangle) if an unresolved branch marked as (\text{BR}</em>{\text{invalid}}) or indirect jump not marked as (\text{BR}_{\text{no}}) exists in ROB: (\text{ROB}[\text{Inst}].\text{BackendRestricted} = 1)</td>
</tr>
<tr>
<td>6.</td>
<td>Branch BR (marked as (\text{BR}_{\text{valid}})) resolves for all entries (E) in ROB: if (\text{ROB}[E].\text{DependentBranch} = \langle \text{BR.BranchID}, \text{BR.SeqNum} \rangle): (\text{ROB}[E].\text{Restricted} = 0) Remove BR from UBT</td>
</tr>
<tr>
<td>7.</td>
<td>Branch Inst (marked as (\text{BR}<em>{\text{invalid}})) or indirect jump Inst (marked as (\text{BR}</em>{\text{valid}}) or (\text{BR}<em>{\text{invalid}})) resolves for all entries (E) in ROB after Inst: if (E) is marked as (\text{BR}</em>{\text{invalid}}) or (E) is indirect jump and marked as (\text{BR}_{\text{valid}}): break else if (E) not marked as (\text{Resp}_0): (\text{ROB}[E].\text{BackendRestricted} = 0)</td>
</tr>
</tbody>
</table>
our static compiler analysis might over-approximate the dependencies, it is sound by design and will not declassify any true branch dependency.

In addition to previous software contracts, we introduce a new policy, called control-flow confidentiality:

Definition 5.3 (control-flow confidentiality policy). Control-flow confidentiality policy requires the program to mark confidential control-flow instructions and guarantee that the correct execution of the program will not leak confidential information based on the adversaries’ capabilities.

SpecControl secure speculation for control-flow confidentiality: To provide such a guarantee, SpecControl avoids any speculative activity (including the front-end) for confidential control-flow instructions. SpecControl is the first solution that supports secure speculation for control-flow confidentiality through its software interface. SpecControl provides the liberty for the software-level defenses to explore lightweight mitigations for non-speculative leaks based on the adversaries’ capabilities without additional vulnerabilities due to the microarchitectural level speculations.

6 EVALUATION

We evaluate SpecControl compared to an unprotected baseline out-of-order core, and a secure baseline with the same protection scope as SpecControl (addressing both speculative fetch and execution attacks). Additionally, we compare the modified version of DOLMA [47] and STT [77] which are still vulnerable to speculative fetch attacks. Table 3 shows more details of our evaluated designs.

6.1 Experimental Setup

Simulation Environment. We implemented SpecControl on top of the gem5 [12] DerivO3CPU and run simulations in syscall emulation (SE) mode. For power analysis, we modify McPAT [43] version 1.3 to support secure speculation for sandboxing and constant-time

We disable the delay-on-miss optimization of DOLMA as it is shown to be vulnerable to speculative interference attacks [10].

Table 3: Evaluated designs in our studies.

<table>
<thead>
<tr>
<th>Design</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected Baseline</td>
<td>Normal out-of-order core with no protection</td>
</tr>
<tr>
<td>SpecControl</td>
<td>Our proposed design</td>
</tr>
<tr>
<td>STT</td>
<td>STT and DOLMA implementations adopted from [47] to support secure speculation for sandboxing and constant-time (as defined)</td>
</tr>
<tr>
<td>DOLMA</td>
<td>STT-Spectre (M+R) and DOLMA-Default (M+R)</td>
</tr>
<tr>
<td>Secure Baseline</td>
<td>Disabling speculation (no branch predictor)</td>
</tr>
</tbody>
</table>

Table 4: System configuration for simulation (Intel Golden-Cove-like processor).

<table>
<thead>
<tr>
<th>L1d Cache</th>
<th>32KB, 8-way</th>
<th>Branch Predictor</th>
<th>TAGE-SC-L-64KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>32KB, 8-way</td>
<td>LQ/SQ size</td>
<td>192/114 entries</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256KB, 8-way</td>
<td>ROB size</td>
<td>512 entries</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>1MB, 16-way</td>
<td>IQ size</td>
<td>97 entries</td>
</tr>
<tr>
<td>F/D/I/C Width</td>
<td>8/8/8/8</td>
<td>BF (INT/FP) size</td>
<td>280/332 entries</td>
</tr>
<tr>
<td>Data Prefetcher</td>
<td>Stride</td>
<td>UBT size</td>
<td>16 entries</td>
</tr>
</tbody>
</table>

Figure 7: Performance results of SPEC CPU2006 [28]. SpecControl protects all known speculative fetch and speculative execution attacks, while DOLMA and STT (hashed bars) only protect against speculative execution attacks.

it is not architecture-specific and can be easily ported to other architectures. In addition, we integrated a taint tracking mechanism in our compiler to mark secret dependent branches for crypto modules [13].

Benchmarks. We use C/C++ applications from SPEC CPU2006 [28] benchmark suite and the ELFie [56] methodology to generate representative (SimPoint) executables with a region size of 1 billion instructions. Also, we investigate a set of realistic crypto modules and standard microbenchmarks for constant-time enforcements [13, 46, 59, 73].

6.2 Performance Results

6.2.1 Performance of SPEC CPU2006 Workloads. Figure 7 shows the performance results for SPEC CPU2006 applications for five different designs. DOLMA and STT show an average performance overhead of 51% and 43% compared to the Unprotected Baseline, respectively. SpecControl’s performance overhead is just 23% on average. This means that the SpecControl 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 SpecControl (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 SpecControl and DOLMA in some cases. However, prior work [47] demonstrates the vulnerability of STT against Spectre attacks transmitting data via stores. Note, that none of the branches in SPEC CPU2006 applications are marked as RESGE since they do not process confidential information.

Using a naive implementation (Secure Baseline) to protect the system against all speculation-based attacks will incur a significant performance degradation of 2.26× on average compared to the
6.2.2 Impacts of UBT Size. Figure 10 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 458.sjeng). However, we observe that in some cases the SpecControl 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. Figure 11 depicts the breakdown of the commit cycles for 458.sjeng with UBT sizes of 2 and 16. 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. A potential future work can deploy a performance-aware and dynamic control of 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). SpecControl consumes only 1.72% more power over the Unprotected Baseline core with an area overhead of 1.42%.

6.2.3 Performance of Cryptographic Workloads. Figure 12 shows the performance results for our cryptographic benchmarks. We mark all sensitive branches of these applications via secret taint tracking and restrict them at the front-end (ResFE). These applications are much smaller than SPEC CPU2006 with less intense control-flow leading to fewer opportunities to improve performance. However, the results show that SpecControl is still able to benefit from the compiler informed branch dependencies and improve the performance of these applications compared to STT and DOLMA, which only mitigate speculative execution attacks. For example, SpecControl shows 10% performance overhead for dijkstra, while STT and DOLMA incur 30% overhead. For the same reason we explained for SPEC CPU2006, STT shows better performance for two applications as it does not restrict the execution of tainted stores.

6.2.4 Performance of Synthetic Workloads. To better understand the effects of different configurations of SpecControl in a real...
As you can see, by decreasing the size of (tainted) branch restrictions and have the same protection of DOLMA and STT. Related functions, and (3) a more coarse-grained approach and marks all branches in RSA-SpecControl loads. We evaluate three versions of SpecControl: (1) SpecControl-(s:tainted) restricts only the secret dependent branch of RSA at the front-end (line 4 in Figure 3), (2) SpecControl-(s:all) adopts a more coarse-grained approach and marks all branches in RSA-related functions, and (3) SpecControl-(s:none) that adds no front-end restriction and have the same protection of DOLMA and STT. As you can see, by decreasing the size of (Cipher part (as a regular application with less benefit from speculative execution) and increasing the size of (sandboxed part (more benefits from speculation) the performance overhead of SpecControl increases, from 30% in S25/C75 to 57% in S90/C10 (STT and DOLMA show the same trend, from 45% overhead to 64%). In addition, for RSA we cannot observe a significant improvement by precisely marking front-end restrictions and a conservative approach is also sufficient to gain the most benefits of SpecControl. Interestingly, we observe that SpecControl-(s:none) shows a bit more performance loss compared to SpecControl-(s:tainted) (about 4% for S25/C75 workload). This means that not using branch prediction for the tainted branch improves the performance and our simulation results show that the number of squashed instructions drop by 36% compared to the Unprotected Baseline.

6.3 Penetration Tests
Figure 15 shows the penetration tests of our new variant of speculative fetch attacks on gem5. The figure shows the execution time of the attack gadget when the attacker chooses a correct or incorrect guess. A higher timing difference for the incorrect guess means that the attacker can recover the secret. The results confirm that all other designs (the Unprotected Baseline, DOLMA, and STT) leak the secret bit through an observable timing difference, while SpecControl can successfully protect the secret bit by restricting the fetch and front-end updates for the marked branch.

---

Since there is no native implementation of __rdtsc__ on Apple processors, we implemented a timer function by incrementing a shared variable with the __atomicIncrement64__ function.
In addition, we confirm that SpecControl passes the Spectre penetration tests for both speculative secrets (sandboxing case) and non-speculative secrets (constant-time case)\(^9\).

### 6.4 Speculative Fetch Attack on existing CPUs

To demonstrate the feasibility of the attack on real existing CPUs, we run our attack on three different real CPUs, specified in Table 5: (1) Intel, (2) AMD, and (3) Apple. Table 6 shows the average execution time of the attack gadget (running 10K NOPs) for 20 trials on these three CPUs with both correct and incorrect guesses. We observe that the execution time in the case of an incorrect guess is always higher than the case the attacker has a correct guess. For example, for the Intel CPU, we observe at least 1657 cycles of timing difference for incorrect guesses and correct guesses which can be used as a threshold to evaluate the correctness of the guess.

### 7 RELATED WORK

**Channel-specific mitigations.** Early defenses for speculative execution attacks aimed to secure individual channels, like data caches [14, 33, 58, 61, 62, 74]. Most of these solutions either implement invisible speculation of memory accesses or undo the speculation effects on the cache. Both of these two strategies are shown to be ineffective against newer versions of speculative execution attacks [10, 42]. In addition, Weisse et al. [70] show that the attacker can exploit other channels as well to leak secret data, like BTB. InvarSpec [78] is a performance optimization for invisible speculation techniques that uses program analysis to lift the restrictions for speculatively invariant instructions. Besides that, InvarSpec inherits the security flaws of the underlying defenses, it also introduces new vulnerabilities [7].

**Secure speculation for sandboxing.** STT [77] and NDA [70] are the first defenses that introduced taint tracking methodologies to restrict the execution of instructions tainted by speculative memory accesses. While STT restricts the execution of tainted memory accesses, NDA prevents the propagation of the data from tainted instructions. STT does not restrict the execution of tainted stores which has been shown to be vulnerable [47]. Following mitigations adopted the same insights of speculative taint tracking while improving the performance with the same security guarantees [8, 76]. DOLMA [47] attempts to protect non-speculative secrets, but their performance benefits come from allowing the execution of some speculative instructions (under certain conditions) that might lead to exploits through resource contention [10].

**Secure speculation for constant-time.** Some works [21, 24, 63] adopt a dynamic secrecy tracking methodology by manually labeling the secret regions of the memory; speculative execution of the instructions tainted by secret data will be blocked in the processor. However, these defenses cannot provide security for legacy software without labeling the secret regions. SPT [20] is a hardware-only defense supporting secure speculation for constant-time workloads that assumes all memory regions are secret and declassifies a region only if it is leaked non-speculatively as well.

Table 7 shows a summary of existing defenses with respect to the secure specifications they offer. SpecControl offers secure speculation for sandboxing and constant-time policies and is the only one to offer secure speculation for control-flow confidentiality. In addition, SpecControl provides the flexibility to define new software contracts and restrict additional or fewer instructions through its interface based on future speculative attacks and programming policies.

### 8 CONCLUSION

In this work, we present SpecControl, a configurable methodology that implements a hardware/software co-design to efficiently protect against current and future speculation-based attacks. SpecControl proposes a communication interface that allows compilers and application developers to inform the hardware of the true branch dependencies, confidential instructions, and additional fine-grained
constraints in order to apply restrictions only when necessary. SpecControl minimizes performance overheads while allowing for configurable secure speculation policies. Apart from known speculative execution attacks, we also evaluate SpecControl on a new variant of speculative fetch attacks that exploit the Pattern History Table (PHT) in branch predictors to extract the control-flow decisions of the victim. We show that this attack variant is more dangerous than previously reported. With SpecControl we provide stronger security guarantees for speculation-based attacks compared to the state-of-the-art while reducing the performance overhead from 51% and just to 23%.

REFERENCES