Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy

Rutvik Choudhary
UIUC, USA

Jiyong Yu
UIUC, USA

Christopher W. Fletcher
UIUC, USA

Adam Morrison
Tel Aviv University, Israel

ABSTRACT

Speculative execution attacks put a dangerous new twist on information leakage through microarchitectural side channels. Ordinarily, programmers can reason about leakage based on the program's semantics, and prevent said leakage by carefully writing the program to not pass secrets to covert channel-creating "transmitter" instructions, such as branches and loads. Speculative execution breaks this defense, because a transmitter might mis-speculatively execute with a secret operand even if it can never execute with said operand in valid executions.

This paper proposes a new security definition that enables hardware to provide comprehensive, low-overhead and transparent-to-software protection against these attacks. The key idea is that *it is safe to speculatively execute a transmitter without any protection if its operands were already leaked by the non-speculative execution.* Based on this definition we design Speculative Privacy Tracking (SPT), a hardware protection that delays execution of every transmitter until it can prove that the transmitter's operands leak during the program's non-speculative execution. Using a novel dynamic information flow analysis microarchitecture, SPT efficiently proves when such an operand *declassification* implies that other data becomes declassified, which enables other delayed transmitters to be executed safely.

We evaluate SPT on SPEC2017 and constant-time code benchmarks, and find that it adds only 45%/11% overhead on average (depending on the attack model) relative to an insecure processor. Compared to a secure baseline with the same protection scope, SPT reduces overhead by an average 3.6×/3×.

1 INTRODUCTION

Speculative execution attacks [16, 22, 37, 41–43, 47, 62] have shaken the foundations of processor security. These attacks coerce *transient*, i.e., doomed-to-squash, instructions to leak secret program data over microarchitectural covert channels (e.g., cache state [78]). In the worst case, these attacks can leak all of program memory. For example, the now-famous Spectre V1 bounds-bypass exploit—`if (i<N) transmit(A[i]);`—leaks data at an attacker-controlled address `&A[i]` by mis-training the directional branch predictor and passing out-of-bounds data to a *transmit instruction* (or *transmitter*). The transmitter's execution creates operand-dependent hardware resource changes that can be observed by an attacker through a microarchitectural covert channel, e.g., cache contention [45, 78].

Ideally, a hardware defense for speculative execution attacks should prevent all *secret* data (whether in registers and data memory) from leaking through speculative execution *with low overhead*, and do so *while being transparent to software*. Here, all prior work runs into a difficult problem. Since hardware alone does not understand program semantics, it does not know what program data is secret and is forced to have incomplete protection or to be conservative. For example, prior work STT [83] reduces protection scope, treating only "speculatively-accessed data" as secret; prior work NDA [74] delays all transmitters until they are non-speculative—regardless of what program data actually is secret. This leads to high overhead, as protection has to be applied where it is not needed [2, 11, 39, 58, 59, 69, 74, 82, 83].

To address these problems, software can explicitly specify what is secret and information flow tracking mechanisms can enable protection only for that secret data [61, 80]. But such an approach creates other issues, such as requiring software/ISA changes which break backwards compatibility.

This paper addresses the above challenge by proposing a new, more precise, definition for what registers and data memory are secret, that hardware can enforce in a software-transparent fashion. The key idea is that *any data that can leak through the program's non-speculative execution should not be treated as secret during the program's speculative execution.* In other words, there is no need to protect what can inherently leak anyway. Among other use cases, this idea enables seamless protection for existing constant-time/data-oblivious code. Such code works around existing ISAs' lack of support for specifying secrets by avoiding passing secrets to transmit instructions [13, 14, 20, 23, 81]. Currently, however, constant-time code only prevents non-speculative leakage and is vulnerable to speculative leakage, because a transmitter might mis-speculatively execute with a secret operand even if it can never execute with said operand in valid executions [16, 19, 81].

We show a motivating example of our idea in Figure 1. Recall, `transmit(...)` leaks its operands. Hence, `val` leaks non-speculatively because the code on Line 2 eventually retires. Thus, our definition allows for disabling protection when the program's speculative execution reaches Line 4. For concreteness, suppose we apply our definition on top of NDA. Normally, NDA would be forced to delay the execution of Line 4 until the branch resolves. Our

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480068

definition would allow Line 4 to execute without delay, since val leaked earlier in the program's non-speculative execution. Importantly, the hardware can determine what data can leak non-speculatively, without software assistance, by recording which transmit instructions retire and conceptually "untainting" those transmit instructions' operands.

```
1 val = ...
2 transmit(val);
3 if (p) {
4   transmit(val);
5 }
```

Figure 1: `transmit(...)` refers to any transmit instruction, one whose execution creates a covert channel as a function of its operands. The data `val` is leaked non-speculatively.

We extend and generalize this idea into a complete framework that aggressively disables protection for data that can be inferred based on non-speculative leakage and the attacker's other prior knowledge. For example, given a simple AND instruction out = AND(in1, in2), if the attacker learns out = 1 because out is leaked non-speculatively *and* the attacker can deduce from its prior (public) knowledge that the AND operator was executed, it can deduce that in1 = in2 = 1 because the only way the output of AND can be 1 is if the inputs are both 1. This implies it is safe to disable protection for both in1 and in2 since the attacker can now infer them anyway. This is akin to applying GLIFT-like [70] ideas in the *backwards* direction. It similarly generalizes to other types of operations (e.g., OR, XOR, ADD) and can be applied compositionally to more complex dataflow graphs.

Putting it all together, we propose *Speculative Privacy Tracking (SPT)*, a microarchitectural implementation of our framework that is optimized to prevent data from being leaked due to speculative execution. At a high level, SPT maintains taint for registers and applies a protection policy for transmitters with tainted operands, e.g., delaying their execution [11, 74, 83] or performing their execution in a fashion that does not create operand-dependent covert channels [82]. We add logic in hardware that tracks when data is transmitted over non-speculative covert channels and apply our novel "untaint" algebra to aggressively disable protection for other data in both registers and memory—whenever we can prove that such data can now be inferred by the attacker, given its prior knowledge and previous non-speculatively transmitted data. Last but not least, SPT provides a strong security property: data that does not leak in a non-speculative execution does not leak in a speculative execution. This enables software to more easily reason about what data it leaks, i.e., without having to worry about speculative execution.

This paper makes the following contributions:

(1) We propose a novel framework for tracking information flows when data becomes declassified (in our setting: leaked non-speculatively).
(2) We propose SPT, a novel hardware defense that applies the above framework to efficiently and comprehensively mitigate speculative execution attacks.
(3) We evaluate SPT on SPEC2017 and constant-time code benchmarks, and find that it adds only 45%/11% overhead on average (depending on the attack model) relative to an insecure processor. Compared to a secure baseline with the same protection scope, SPT reduces overhead by an average 3.6×/3×.

2 BACKGROUND

2.1 Non-speculative Side/Covert Channels

There is significant literature on how non-speculative execution creates covert/side channels in microarchitecture. For example, attacks have been demonstrated that create program data-dependent contention on structures ranging from caches [53, 77–79, 79], to TLBs [29], to page tables [73, 75], to the DRAM [54], to branch predictors [5, 27], and others [8, 10, 23, 26, 30, 31, 50, 66]. From a side channel perspective, these can be grouped into three categories: those that leak a function of the program's memory address pattern [23, 29, 50, 53, 54, 66, 73, 75, 77–79, 79], control-flow decisions [5, 8, 27], and operands of so-called "variable time" instructions [10, 23, 30]. In our terminology, this is equivalent to saying that memory, control-flow and variable time arithmetic instructions are all considered transmitter instructions.[1]

2.2 Speculative Taint Tracking

We build SPT on top of Speculative Taint Tracking (STT) [83], and hence describe STT here. We note that it is possible to build SPT on top of other schemes. However, whichever scheme is used must satisfy an important security property with respect to what data is contained in the ROB in any given clock cycle (which STT does satisfy). We elaborate in Section 2.2.2.

Speculative Taint Tracking (STT) is a framework that protects *speculatively-accessed data*—defined to be data read by transient, i.e., doomed-to-squash, loads (known as *access* instructions)—from being leaked over *any possible microarchitectural covert channel*. By contrast, STT does not protect *non-speculatively-accessed data*, which is defined to be data that is non-speculatively written to the register file. Section 3 expands on this limitation.

2.2.1 STT Details. STT monitors how speculatively-accessed data flows through the pipeline and applies protections when that data is at risk of getting transmitted to an attacker. At a high level, STT implements two policies. First, that *transmit instructions* (Section 1) should not execute until their operands are only a function of non-speculative-accessed data. Second, that the program counter should only be a function of non-speculatively-accessed data. We now discuss these in more detail.

Covert Channels. STT classifies covert channels into two classes: explicit and implicit channels.[2] In an *explicit channel*, data is *directly* passed to a transmitter instruction. For example, loads are transmitters, as their execution makes address-dependent changes to the cache state; hence, passing an address to, and executing, a load forms an explicit channel. In an *implicit channel*, data *indirectly* influences how (or that) an instruction or several instructions execute, and these changes in resource usage reveal the data. For example, a branch instruction, whose outcome determines subsequent instructions and thus whether some functional unit is used. Such branch-based implicit channels are used by NetSpectre [62]

[1] To be consistent with terminology used in prior work—see Section 2.2—we will not refer to branches as transmitters, but they conceptually can be viewed that way.
[2] Fundamentally, a microarchitectural covert channel can communicate a value when explicit information flow at the gate level changes as a function of that value [70, 81]. The abstraction proposed by STT provides a way to reason about when this will occur due to different instructions and microarchitectural optimizations.

and SmotherSpectre [16] to trigger SIMD unit usage and port contention, respectively.

STT further characterizes implicit channels by *when* they leak data and what *type* of "branch" operation they feature. An implicit channel can leak either when a prediction is made (e.g., a branch prediction) or when a resolution occurs (e.g., when a branch resolves). An implicit channel can feature either an *explicit* branch, which is a control-flow instruction, or an *implicit* branch, which is a conceptual branch that occurs due to hardware mechanisms that change how instructions execute. For example, store-to-load forwarding can be viewed as an implicit branch that checks for an address alias to determine if a load will access the cache.

Tracking S-Taint. At a high level, STT features a taint propagation mechanism similar to prior work (e.g., [68]), and proposes a novel "untaint" mechanism to disable protection as soon as doing so is safe. For the rest of the paper, we refer to STT's taint as *s-taint* (for *speculative taint*) to differentiate it from the distinct tainting mechanism proposed in this work. Specifically: STT *s-taints* the output register of a speculative access instruction. The microarchitecture defines when to *s-untaint* the output of a speculative access instruction. This point in time depends on the attack model: in the *Spectre* model (which covers control-flow speculation), it is when all older control-flow instructions have resolved, and in the *Futuristic* model (which covers all forms of speculation), it is when the access instruction cannot be squashed. STT propagates s-taint/s-untaint information: the output register of a non-access instruction is s-tainted if and only if it has an s-tainted input register. S-Taint propagation piggybacks on the existing register renaming logic in an out-of-order core, and is therefore fast. S-Untainting all dependencies of an access instruction that becomes non-speculative is more difficult, but STT has a fast mechanism to s-untaint in a single cycle. STT does not maintain s-taint/s-untaint information in the cache/memory system, only in the physical (non-architected) register file.

Implications of S-Untainting. S-Untainting the output of an access instruction (i.e., a load) occurs only if the execution of that instruction corresponds to a correct speculation for the given attack model. Consequently, once all inputs of a transmitter are s-untainted, the transmitter becomes *safe* and its inputs can be revealed, as they are guaranteed not to originate from a misspeculated execution.

Protection Policies. Based on s-taint information, STT blocks *all* covert channels by applying a uniform rule across each type, illustrated in the following table:

Explicit Channels are blocked by delaying the execution of transmit instructions until their operands are s-untainted.
Prediction-based Implicit Channels are eliminated by preventing s-tainted data from affecting the state of any predictor structure.
Resolution-based Implicit Channels are eliminated by delaying the effects of branch resolution until the (explicit or implicit) branch's predicate becomes s-untainted.

2.2.2 Key requirement: ROB contents are independent of tainted data. STT's implicit channel rules imply a powerful property, namely that the program counter register (PC), and by extension the ROB contents, does not depend on speculatively-accessed data. More generally, STT ensures that the PC and ROB contents are independent of s-tainted data. STT enforces this invariant efficiently, without needing to delay execution of instructions following an s-tainted branch.

3 MOTIVATION: PROTECTING NON-SPECULATIVE SECRETS

Efficient comprehensive defenses from speculative execution attacks [11, 74, 82, 83] block all leakage of *speculatively-accessed* data, i.e., data produced (written to a register) by a speculative instruction. This approach blocks universal read gadgets [48]—i.e., leakage of arbitrary program memory accessed by a transient instruction such as an out-of-bounds array access. This approach does not, however, protect data once the instruction that produced it becomes non-speculative—that is, *non-speculatively-accessed* data. For instance, STT allows speculative transmitters whose operands are non-speculative to execute without protection. This precludes protecting important classes of privacy-critical programs, as discussed next.

Motivating example: constant-time code. Constant-time (or data-oblivious) code [9, 23, 81] is a pervasive coding discipline for computing over secret data without leaks. Constant-time code reads secrets (plaintext, cipher keys, personal information, etc.) from memory into non-speculative state and performs its computation without passing said secrets as arguments to control-flow or transmit instructions [6, 10, 13, 14, 25, 28, 35, 49, 51, 52, 55, 60, 63, 71, 85].

Unfortunately, the constant-time technique assumes correct execution semantics and does not prevent speculative leakage [16, 19, 81]. For instance, consider a static transmit(rX) instruction in constant-time code. Although the transmitter's operand would never contain secret data in a correct execution, mis-speculation from a state in which rX architecturally (non-speculatively) holds a secret can cause execution to jump to the transmitter's address, resulting in the transmitter executing and leaking the secret. This can happen due to misprediction of loop branches [81], function returns [19], indirect calls [16], etc.

Schemes like STT do not block such attacks on constant-time code, because the leaked data was accessed non-speculatively. A sufficient approach to block these attacks is to expand protection to all non-speculatively-accessed data. But this leads to high overhead, e.g., due to delaying every transmitter until it becomes non-speculative [74]—even if that transmitter leaks what is semantically public data. Software could alternatively specify what data requires protection, but this would require software and ISA changes, which breaks backwards compatibility.

SPT's core observation is that program semantics implicitly provide information about what data should be protected speculatively. These are *non-speculative secrets*, defined as non-speculatively-accessed data that is never transmitted over a non-speculative covert channel. SPT's goal is to transparently—without modifying the software or ISA—infer what is non-speculatively secret and to only apply protection to non-speculative secrets as opposed to all non-speculatively-accessed data.

4 ATTACK/THREAT MODEL

We assume an adversary that can monitor any speculative or non-speculative microarchitectural side/covert channel and arbitrarily induce speculative execution from anywhere in the system. For instance, the adversary may operate from within the victim program (*SameThread* [62]), an SMT sibling (*SMT*), or another processor core (*CrossCore*), and can monitor the channels described in Section 2.1, plus carry out any Spectre attack.

We protect both speculatively-accessed data and non-speculative secrets (Section 3). That is, our protection scope is broader than STT's. We explicitly do not protect any data transmitted over a non-speculative covert/side channel, and use information regarding what data leaks in this way to dramatically improve performance.

5 IMPLICATIONS OF DECLASSIFICATION

As discussed in Section 1, the key idea in this paper is that when data is leaked over a non-speculative covert channel, it should also be allowed to leak over speculative covert channels (i.e., its protection should be disabled). This is related to a more general concept in information flow called *declassification* [57], which provides a framework for explicitly downgrading (leaking) certain values and analyzing the resulting security implications. Using this framework, the first transmitter in Figure 1 (Line 2) can be conceptualized as declassify(`val`), indicating that—due to non-speculative leakage being out of scope—we have explicitly downgraded the security level of `val` from private to public.[3]

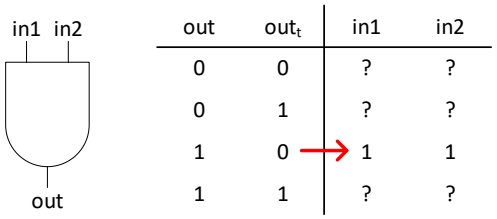
We make a key observation that when a value becomes declassified, it often results in a ripple effect where the attacker can deduce additional values while not actually learning any "new" information beyond the declassified value. As a very simple example, suppose we run the public program `r2 = r1; declassify(r2);` and `r1` is private. Because `r2` was declassified and the program was public, i.e., the attacker knows `r2` is just a copy of `r1`, the attacker can now deduce `r1`. In short, when an attacker learns something new, it can use that new information plus its prior knowledge to try and deduce additional information.

The above has important implications for efficient defenses. For example, if the defense is aware that the declassification occurred, it can stop trying to protect other information that it knows can now be inferred by the attacker. In short, there is no reason to protect `r1` once `r2` has been declassified.

The rest of this section develops a novel "untaint" algebra that expands on this idea. Note, this taint is distinct from STT's s-taint (Section 2.2). We will implement this algebra in hardware, in the context of mitigating speculative execution attacks, in the later sections of the paper.

To explain ideas, we consider examples in terms of how information flows through boolean operations. (Later in the paper, we will replace boolean operations with dynamic instructions.) Similar to other dynamic information flow schemes [70, 81], we assume data is either public (untainted) or private (tainted), representing "tainted" as logic 1. We use the terms *value* and *taint* to represent a data value and its taint, respectively.

[3]Note that declassify(...) is shown in the style of a software function to make declassifications clear in the exposition. All declassifications will be performed by the hardware, transparent to the software.



out	out_t	in1	in2
0	0	?	?
0	1	?	?
1	0	1	1
1	1	?	?

Figure 2: (Left) AND gate example. (Right) Truth table for information flow in the backwards direction, using the AND gate as an example. out_t **denotes the taint on** out**. The key takeaway is that when the output becomes untainted, e.g., due to a declassification event, we can deduce** in1 = in2 = 1 **if** out = 1**.**

5.1 Forward Information Flow

To start, consider a simple 2-input AND gate, with input bits in1 and in2, output bit out as shown in Figure 2. The question is: under what circumstances is the output tainted? The goal is to mark the output untainted whenever possible, to "improve performance" (akin to disabling protection) but without revealing any information about tainted data beyond what can be inferred using prior (public) information.

The generic answer is: the output is untainted iff the OR of the input taint bits is 0, i.e., neither input is tainted. Prior work on GLIFT [70] recognized that when the semantics of the AND operation are taken into account (what we refer to as the attacker's "prior knowledge", in the above discussion), the output can be untainted more aggressively.

For example, suppose input in2 is tainted. If the other input in1 is 0 and untainted, it is "public knowledge" that the output will be 0 (because 0 & in2 = 0). Hence, it is safe to untaint the output. However, if in1 is 1 and untainted, the output becomes a function of in2 (because 1 & in2 = in2). Hence, we must keep the output tainted. Finally, in the case where both inputs are tainted, we clearly have to mark the output tainted.

Now, suppose we add support for declassification. This enables us to re-apply the GLIFT rules dynamically. For example, if in1 and in2 are initially tainted and in1 is later declassified and its value is 0, we can dynamically re-apply the above rules and untaint the output.

5.2 Backward Information Flow

Even more interestingly, the above observation about declassification also works in the backwards direction, enabling a novel untaint operation that flows backwards. Consider the AND gate as before. Suppose the output of the AND gate is 1 and tainted. If the output becomes declassified/untainted, we can use that information plus our knowledge of AND gate semantics to infer that in1 = in2 = 1. Cases where the attacker can deduce inputs from outputs in this way are shown in Figure 2 (note, when the output is tainted, we cannot use/know its value to deduce information about the inputs).

This idea can also simultaneously take into account information about both inputs and outputs. For example, suppose in1 and in2 are tainted and in1 = 0, in2 = 1. If the output becomes untainted, we cannot untaint the inputs because it could have been the case that either (or both) of in1 or in2 were 0. On the other hand, suppose that both the output and in2 become untainted. In that case, we can

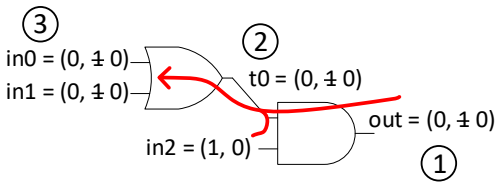


Figure 3: Backwards information flow through a composition of operators. Each wire is a tuple (value, taint)**. Strikethrough on taint indicates a declassification or that the value can now be inferred due to operator semantics and declassifications. The flow of untaint is given by the red backwards arrow.**

now untaint in1 because $\text{out} = 0 \wedge \text{in2} = 1 \rightarrow \text{in1} = 0$ due to the semantics of AND.

The above concepts generalize to the other boolean operations such as OR, XOR, NOT, etc., enabling a new untaint algebra in the backwards direction.

5.3 Composition to complex dataflow graphs

Beyond the simple AND operator, the above concepts extend to compositions of operators. For example, Figure 3 shows our AND gate example, where one of the inputs to the AND gate is the output of an OR gate. The same ideas in propagating untaint forwards and backwards apply. For example, suppose t0 is initially tainted, in2 is initially untainted and out is initially tainted. If out becomes untainted due to a declassification ①, we can now infer that $\text{t0} = 0$ because $\text{in2} = 1 \wedge \text{out} = 0 \wedge \text{out}_t = \text{in2}_t = 0$ and thus untaint t0 ②. Now that t0 is untainted, we can further back-propagate untaint through the OR gate because if the output of OR is 0, it must be the case that the inputs to the OR are 0 ③.

6 SPECULATIVE PRIVACY TRACKING

Building off the novel taint algebra introduced in the previous section, Speculative Privacy Tracking (SPT) is a hardware protection scheme that comprehensively, transparently, and efficiently eliminates *novel* microarchitectural leakage from speculative execution—i.e., leakage of data which the non-speculative program execution does not leak. SPT's security guarantee is detailed in Section 6.2.

SPT "taints" data considered secret and tracks information flow from tainted data to transmit and control flow instructions (Section 6.3). Speculative transmit instructions with tainted operands are considered *unsafe* and SPT applies a protection policy to them, e.g., delaying their execution. Speculative control-flow instructions are protected using STT's principles for blocking implicit channels (Section 6.4). SPT treats a transmit/branch instruction becoming non-speculative—and thus non-speculatively leaking its operands—as a *declassification* of the operands, which allows SPT to apply the ideas of Section 5 to untaint the operands and data that can be inferred from them (Sections 6.4–6.7). Speculative transmit instructions whose operands become untainted as a result of this process can execute without protection, thereby improving performance.

6.1 Baseline Microarchitecture Requirements

As with prior schemes that block all microarhictectural covert channels [11, 74, 83], SPT requires the microarchitecture to identify transmit and control-flow instructions, and for each such instruction type, which of its operands get "leaked" as a result of its execution. That is, which operands cause operand-dependent resource usage that can reveal the operand (partially or fully) when the instruction executes. Like STT, SPT requires the microarchitecture to identify implicit branches and their predicates.

6.2 Security Property

SPT's goal is to prevent any program data—whether in memory or registers—that is not leaked by the non-speculative execution from leaking as a result of mis-speculated execution. We consider data to have leaked if some function of it is transmitted over a microarchitectural covert channel—i.e., gets passed as an operand to a transmit or control-flow instruction (Section 2.1). In the following, we formally define SPT's security guarantee.

Given an execution E of a microarchitecture, we define data as *secret* if there is no dataflow from it to an operand of a transmit or control-flow instruction, whose execution leaks that operand, either partially or fully. (By "dataflow" we refer to flow through registers, memory (across loads and stores), or a combination of both.) This definition of "secret" can be applied to a microarchitecture's *non-speculative* and *speculative* executions, i.e., the executions induced by retired instructions or by all (including transient) instructions, respectively. Based on these definitions, we define the security property provided by SPT:

Definition 1 (**Security**). *If data D is secret with respect to the non-speculative execution, then D is secret with respect to the speculative execution.*

Because SPT's security property guarantees that any data which is secret with respect to the non-speculative execution does not get leaked as a result of speculative execution, it encompasses comprehensive protection [11, 74, 83] (against any microarchitectural channel) of both speculatively-accessed data and non-speculative secrets (Section 3) from all leakage introduced by speculative execution. Speculatively-accessed data (STT's focus) is "secret with respect to the non-speculative execution" because it is not even accessed, let alone leaked, by the non-speculative execution. Non-speculative secrets, which are additionally not leaked by the non-speculative execution, are by definition "secret with respect to the non-speculative execution." Both types of data are thus covered by SPT's security guarantee.

Software does not specify what is secret. SPT and its security definition do not require software to explicitly identify secrets to the hardware. All that SPT requires to block data from being leaked as a result of mis-speculation is for that data to not be leaked by the non-speculative execution of the program. This property has several important implications.

First, it means that SPT can seamlessly protect existing constant-time programs that were written to not leak specific data over non-speculative covert channels (Section 3). SPT provides the guarantee that standard reasoning, without considering invalid mis-speculated executions, can be used to protect secrets even from speculative leakage.

Second, SPT's security definition implies that future mechanisms for annotating data as secret by software are orthogonal and complementary to SPT. On one hand, if the software can define certain

data as public to the hardware, then SPT can use that information to disable protection of such data even if it does not leak through the program's non-speculative execution. On the other hand, if the software defines certain data to be secret and the hardware implements mechanisms to prevent it from leaking over non-speculative covert channels, SPT will ensure that it, additionally, does not leak over speculative covert channels.

Section 6.9 discusses additional aspects of our security property.

6.3 Taint Tracking, Propagation, & Protection

SPT tracks whether data is used as an argument for a transmit/branch instruction in the program's non-speculative execution, so that it can determine whether that data should be protected. Following Definition 1, data that has not been leaked by a non-speculative transmitter/branch is *tainted*. Thus, all program data (memory and architectural registers) starts off as tainted. As data gets leaked over *non-speculative* microarchitectural covert channels, it is declassified (untainted). SPT's notion of taint is thus conceptually different than STT's s-taint (Section 2.2), which only taints data brought into the pipeline by a speculative instruction and automatically untaints data once the instruction that accessed it becomes non-speculative. In contrast, SPT may *never* untaint data; in particular, data satisying the definition of non-speculative secret (Section 6.2) is never untainted.

Taint Tracking. Conceptually, SPT tracks the taint of data in registers and the memory system. In practice, however, taint tracking throughout the memory system is expensive [24, 68]. To trade off hardware complexity and precision, an implementation of SPT can opt to not track taint below a certain cache level. In this case, all data read from lower levels is conservatively considered tainted. Here, for concreteness, we describe an implementation that tracks taint in registers and in the L1 D-cache (L1D) (see Section 7 for the microarchitectural implementation). For simplicity, we first describe SPT assuming it only tracks whether registers are tainted. Later sections extend taint tracking to the L1D (Section 6.8) and to byte (instead of register-level) granularity (Section 7.2).

Taint/Untaint Propagation. SPT propagates taint across in-flight instructions using standard taint tracking rules, similar to STT. When an instruction enters the frontend and its registers are renamed, the instruction's output (physical) register is tainted if any of its operand (physical) registers are tainted. Output registers of loads are always (conservatively) tainted at rename, since the loaded data's taint status is not yet known.

Crucially, registers may get untainted dynamically. First, due to declassification events: the operands of transmitters/branches are untainted when the instruction becomes non-speculative. Second, in the case of loads, a load's output may become untainted if the load later reads data which is untainted (Sections 6.7–6.8).

To further exploit untaint event information, SPT implements a novel *untaint propagation* mechanism, which uses the rules from Section 5 to untaint additional registers whenever some register gets untainted. We detail this process in Sections 6.4–6.7.

Protection Policy. SPT prevents speculative transmit instructions whose operands are tainted from leaking these operands by applying a *protection policy*. In this paper, we use a "delayed execution" policy [11, 74, 83] that delays a transmitter's execution until its operands become untainted or it becomes non-speculative (according to the attack model, see Section 2.2.1). However, SPT can use other comprehensive policies such as executing a transmitter in a data-oblivious fashion that does not leak its operands [82].

6.4 Implicit Channel Protection via Secret-Independent PC

SPT reuses STT's mechanisms for blocking implicit channels. We observe that given a notion of "taint" for registers, STT's mechanisms make the sequence of instructions fetched/executed/squashed independent of tainted data (Section 2.2.1). These mechanisms consist of preventing tainted data from affecting the state of any predictor and delaying the effects of (explicit/implicit) branch resolution until its predicate becomes untainted. This "PC is independent of tainted data" property blocks all implicit channels from leaking tainted data.

Since SPT ensures that secret data is tainted, the above design guarantees the following property:

Property 1 (**PC is independent of secrets**). *The sequence of instructions fetched/executed/squashed does not depend on secret data.*

Property 1 is the foundation for SPT's analysis of the implications of declassifying data. Property 1 implies that the contents of the ROB at every cycle does not depend on secret data. Consequently, we can assume that the attacker knows the dataflow between in-flight instructions (through registers) and the operations they perform, as this information is obtainable from the ROB contents plus knowledge of the program being executed. Importantly, however, the attacker may not know dataflow through memory, which depends on tainted store/load address operands.

6.5 Untainting via Secret-Independent PC

An implication of Property 1 is that the output of an instruction can be untainted if it is determined only by the content of the ROB. Specifically, SPT untaints the output register of "load immediate" instructions such as `r1 = 17`, where the immediate constant is encoded into the instruction and does not reside in a register operand. Another example (which we do not currently exploit) is to untaint the output of instructions such as `r1 = r1 − r1` (which are often used to zero-initialize registers) regardless of their inputs' taint.

6.6 Untaint Propagation Over Instructions

SPT treats a transmit/branch instruction becoming non-speculative as a declassification of any operand that would have been leaked by the instruction's execution. In response, SPT dynamically *untaints* these operands when the transmit/branch instruction becomes non-speculative. SPT then dynamically and continuously applies a generalized untaint algebra (based on Section 5) to further untaint data that can be inferred from the now-declassified operands. (This is in contrast to prior work (except STT) where, once a physical register's taint is set, that taint does not change until the register is overwritten.)

SPT propagates register untaint events in each cycle by applying untaint algebra rules to each in-flight instruction. Evaluation of the

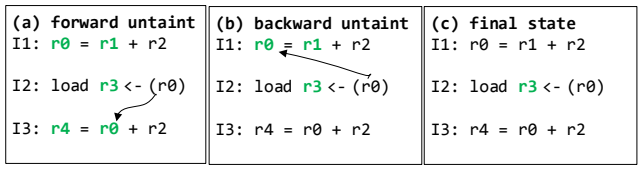


Figure 4: Untaint propagation (green registers are tainted). Suppose I1, I2, and I3 are in the ROB and we consider the clock cycle when I2 becomes non-speculative. This means r0 becomes untainted, which triggers the following forward (a) and backward (b) untaint events, resulting in a final state (c) with r1 and r4 untainted.

rules may untaint more registers, and the process continues in the next cycle. Importantly, the fact that SPT continuously re-evaluates the taint induced by every instruction means that untainting of a register by instruction I can cause instructions both younger and older than I to untaint registers in the next cycle(s). We detail the implementation of the untaint mechanism in Section 7.

Untaint rules consist of *forward* and *backward* untaint rules, which untaint an instruction's output or inputs, respectively, based on untainting of its inputs or output, respectively. To allow a single-cycle implementation, each rule is a function of the instruction's type and the taint of its registers. For example, we do not consider rules that require executing the instruction (e.g., comparing numbers). This paper describes the rules used in our evaluation, which we do not claim to be exhaustive; in fact, they are conservative, e.g., do not apply the GLIFT [70] insights.

The rules described in this section are based on the attacker's knowledge of the contents of the ROB, i.e., the type of each in-flight instruction and the dataflow graph induced by instruction register dependencies. In Section 6.7, we describe untainting mechanisms that leverage what the attacker learns about dataflow between instructions through memory.

Forward (Output) Untainting. We use the conservative taint propagation rules. For each instruction whose output is only a function of its operands (e.g., not loads, whose output also depends on memory): if all of the instruction's operands are untainted, then its output register becomes untainted. Figure 4(a) shows an example. The output of load instructions is untainted when and only when the data they read becomes untainted through other rules (Sections 6.7–6.8).

Backward (Input) Untainting. We use backward untaint rules for several instruction types: ① For register MOV instructions (r2 = r1): if the output is untainted, then the operand is untainted. ② For invertible arithmetic instructions (e.g., ADD): if the output is untainted and all but one of the inputs are untainted, then the remaining input is untainted. Figure 4(b) shows an example, in which r0 and r2 get untainted, so the instruction r0 = r1 + r2 untaints r1, because the attacker can learn r1's value (r1 = r0 − r2).

Convergence. By repeatedly applying the above forward/backward rules after a register becomes untainted/declassified, the untaint status of all other registers will eventually converge. An important property of SPT's taint algebra is that the taint of an instruction's registers can change from tainted to untainted, but never back. This property implies that SPT's untaint process examines each in-flight instruction at most 3 times (assuming instructions have ≤ 2 operands).

6.7 Untaint Propagation Through Memory

Dataflow between instructions extends into memory through stores and loads. SPT propagates untaint across memory dataflow edges created by store-to-load forwarding (this section) and by L1D accesses (Section 6.8). The challenge here is that whether there is a memory dataflow edge may itself be a secret, if one of the store/load address operands is tainted. Untaint propagation must not leak the existence of such secret edges (i.e., address aliases) to the attacker, as that leaks information about tainted (secret) data.

For store-to-load forwarding, our goal is to support two untaint rules for a store/load pair S, L in which L reads the data written by S: ① If S's data operand gets untainted, then L's output register should be untainted (forward untaint). ② If L's output register gets untainted, then S's data operand should be untainted (backward untaint).

SPT builds on STT's store-to-load forwarding security mechanism (described below). However, STT's mechanism is designed only to hide the fact that store-to-load forwarding occurs. It does not support propagating untaint in the above manner, which is what SPT adds.

Our starting point is when a load's address becomes untainted, and so can execute. At this point, the load/store queue (LSQ) checks for possible forwarding. We note that the forwarding decision is made based on the virtual addresses of instructions in the LSQ, and so can consider tainted store addresses, i.e., before the relevant stores execute and perform address-dependent TLB lookups, page walks, or cache state changes.[4] Following STT, secret-dependent store-to-load forwarding can then be hidden as follows: when a load's address becomes untainted, it always accesses the cache. Once the cache access completes, the load's output register is written with the forwarded value (ignoring the memory data) if forwarding should occur, otherwise it is written with the memory data. The only exception, in which a load does not access the cache, is when the fact that store-to-load forwarding should occur is known to the attacker, as described later.

Adding untaint propagation to STT's mechanism requires care, because STT's mechanism makes the "forwarding" and "no forwarding" cases indistinguishable to the attacker *as long as the load's output remains tainted*. Figure 5 shows how adding untaint propagation can reveal the forwarding decision and thereby leak information about tainted data. In the figure, there are several stores with tainted addresses but untainted values. Propagating untaint across store-to-load forwarding allows a younger transmitter to leak the untainted value, which reveals which store forwarded the data and thus its tainted address.

To address this problem, SPT delays propagating untaint between a store-to-load forwarding pair (S, L) until the implicit branch condition of the forwarding becomes untainted. We denote this event by the condition STLPublic(S, L) which is defined over the LSQ. STLPublic(S, L) holds if and only if ① S's data is forwarded to L, ② L's address is untainted, and ③ the addresses of all stores older than L and younger than S (including S) are untainted. Thus, when

[4] If it later turns out that an incorrect decision was made, because a younger store whose virtual address does not alias with the load's address does alias with it physically, the load is squashed. This is handled as usual with STT's principles, and the squash is performed only once the relevant implicit branch—involving the addresses of the store, the load, and all stores in between them in the LSQ—becomes untainted [83].

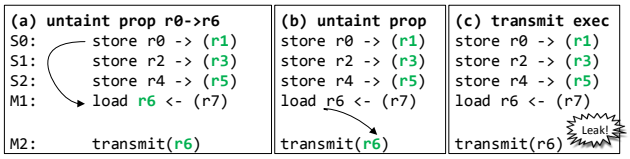


Figure 5: Naive untaint propagation over store-to-load forwarding leaks `tainted` data. Suppose r1= r7: (a) propagating r0's taint untaints $M1$'s output, (b) propagating r6's taint untaints transmitter $M2$'s input, (c) $M2$ leaks r6, revealing r6 = r0 and thus that r1= r7.

STLPublic(S, L) holds, the attacker knows based on non-secret information that L obtains its data via forwarding from store S (and no other store).

Memory dependence speculation. A load L can be speculatively issued to memory before older store addresses are computed [56]. If it later turns out that store-to-load forwarding from some store S was required, L gets squashed. Again, STT's implicit channel protection handles this securely by delaying the squash until STLPublic(S, L) holds, at which point SPT can also propagate untaint (if needed).

6.8 Tracking Taint for L1D Data

We describe an implementation of SPT that tracks L1D taint at byte-granularity for each line. For this purpose, the L1D taint bits are stored in a *shadow L1* structure inside the core, so that SPT does not require out-of-core modifications to the L1D itself. (We detail the implementation in Section 7.) When a cache line is brought into the L1D, all of its bytes are tainted. When a load accesses the L1D, the taint of its output register is set according to the taint of the data read (i.e., we also track taint at byte granularity in the register file).

SPT uses two untaint rules for the L1D: ① **Stores.** When untainted store data is written to the L1D, the L1D taint of the written address range is cleared. The untainting occurs after the L1D allocates a cache line for the stored data, at which point the corresponding taint bits in SPT's shadow L1 are cleared. ② **Loads.** If a load receives data from the L1D and its output register is already untainted, the L1D taint of the read address range is cleared.[5]

6.9 Discussion

SPT allows data to be leaked by speculative execution at a later time than when it was leaked by the non-speculative execution and/or through a different type of transmit instruction. This provides a meaningful, precise security guarantee because (1) SPT never allows speculatively-accessed data to leak, thereby blocking leakage of arbitrary program memory [48], and (2) SPT reduces leakage to that which is implied by program semantics. The latter property means that unmodified constant-time programs can regain their security in the context of speculative execution.

7 MICROARCHITECTURE

We now describe the core microarchitecture changes required for SPT. These consists of taint storage for registers (Section 7.2) and L1D data (Section 7.5), taint/untaint propagation (Section 7.3), and handling of loads and stores (Section 7.4). Figure 6 shows the baseline architecture and the modifications made for SPT.

[5]This case can happen only if the load becomes non-speculative while waiting for the data (Section 8), so the fact that it accessed memory is no longer secret, as all older stores have also become non-speculative.

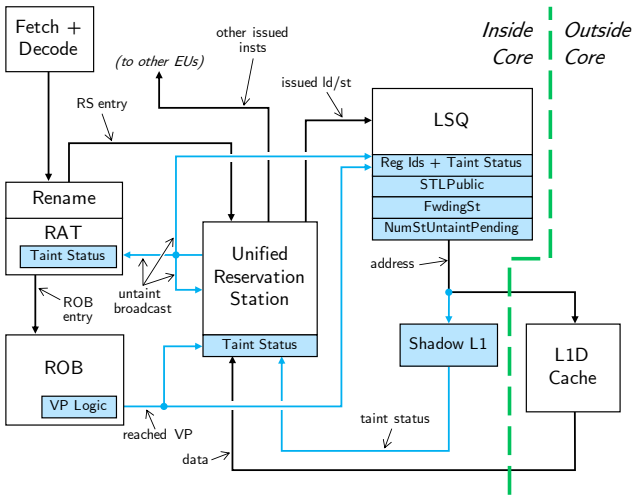


Figure 6: SPT microarchitecture. Blue wires and boxes represent added hardware. For simplicity, most logic changes are not depicted.

7.1 Baseline Processor

We assume a baseline out-of-order (OoO) core design [36, 64]. Instructions are fetched and *renamed* in-order. Renaming maps an instruction's architectural register specifiers to physical registers. Renaming relies on a register alias table (RAT) that maps architectural to physical registers. After renaming, instructions are *dispatched* to a unified reservation station (RS) for scheduling and queued in a reorder buffer (ROB) [38] in program order. The RS *issues* instructions to execution units (EUs), possibly out-of-order. (Loads and stores are issued to a load/store unit, which contains the LSQ.) Executed instructions that reach the head of the ROB are *retired*, committing their results to architectural state in-order.

7.2 Register Taint Storage

We replicate register taint storage across the frontend and backend. (Section 7.3 explains how the replicas are synchronized and Section 7.6 provides the design's rationale.)

In the frontend, we add taint status bits to the entries in the RAT. We use them to determine the taint status of currently renamed-to-physical registers. In the backend, we extend every entry in the RS with taint status bits for each source and destination, which track the taint of the registers of the instruction occupying the slot. We similarly extend each LSQ entry with taint status for the address operand of the entry's instruction (Section 7.4).

By default, the "taint status" of registers is a single bit, indicating if the register is tainted. For registers with partial access modes (e.g., the x86 register EBX corresponds to the lower 32 bits of RAX), the taint status consists of multiple bits that combined can describe the taint of every access mode. For example, for a 64-bit register R, x86 has partial access modes of R[7 : 0], R[15 : 8], R[15 : 0], R[31 : 0], where $R[x : y]$ denotes bits x–y in R. To describe the taint status of such a register, we maintain four taint status bits, corresponding to R[7 : 0], R[15 : 8], R[31 : 16], and R[63 : 32].

7.3 Taint/Untaint Propagation

Tainting. Taint status is computed when an instruction is renamed. For loads, the output register is marked tainted at this point (Section 6.3). For other instructions, if any of the operand registers are tainted then the output register is tainted. Lookup of the taint statuses of the operands and writing the taint status of the renamed outputs happens while accessing their RAT entries during renaming.

Untainting. The ROB is augmented with logic to detect when instructions have reached the *visibility point* (VP), which is the point at which an instruction is considered non-speculative with respect to the threat model. The ROB broadcasts the index of the youngest instruction that has reached the VP. In response, RS slots of older transmit instructions mark their sources as untainted.

Propagation. Untaint propagation consists of two phases which occur every cycle in the RS. The first phase applies the untaint rules from Section 6.6, in parallel, to every entry in the RS. The second phase propagates the untaint event of physical registers untainted in the first phase to the RS slots that share these registers (as an operand or destination). Untaint rules in the first phase are applied locally at each RS slot, so that the taint status computation for one RS slot does not affect the taint status of other slots.

The propagation phase is accomplished by broadcasting the IDs of newly untainted registers on a dedicated bus, akin to the broadcast of ready registers for instruction wakeup. Every RS slot listens to this broadcast, and if the IDs of any of its registers matches a broadcasted ID, it sets its taint status of that register to untainted. (The broadcast only affects the first (untaint) phase of the *next* cycle.) Untaint broadcasts are also sent to the RAT and LSQ to synchronize their taint status.

We limit the number of untaint events broadcast each cycle, referred to as the *untaint broadcast width*. (Otherwise, we would have to support cases of *every* RS slot broadcasting, which requires wiring on the order of the size of the RS, which is too costly.) The RS choice of untaint events to broadcast prioritizes destination registers over source registers within each RS slot, and RS slots corresponding to older instructions over slots corresponding to younger ones. RS slots track which untainted registers have been broadcast by adding a *untaint broadcast flag* to each taint status. When an RS slot untaints a register—including as a result of a VP changes—it sets the corresponding untaint broadcast flag. Only registers with the broadcast flag set are considered for untaint broadcast, and once broadcast, their flag is reset.

7.4 Loads and Stores

For non-load/store transmitters, delayed execution is achieved by the RS delaying issuing of the transmitter to the EU until its operands are untainted. For loads/stores, delaying execution (including TLB accesses, etc.) until the address operands are untainted is handled by the load/store unit, to which loads/stores are issued once their operands become ready. To detect untainting of load-/store addresses, we augment LSQ entries with the IDs of their address operand registers as well as their taint status at the time of issue. LSQ entries listen for VP broadcasts as well as register untainting from the RS, and update their taint status accordingly.

To keep track of STLPublic conditions (Section 6.7), we augment each LSQ entry with a bit that is 1 when the entry corresponds to a load for which some STLPublic condition is true, and 0 otherwise. To update this bit, we further augment each LSQ entry with two extra fields. The first field, FwdingSt, holds the index of the store that is forwarding to the load if forwarding occurs, and -1 otherwise. The second field, NumStUntaintPending, holds the number of stores with tainted addresses involved in the implicit branch if forwarding occurs but STLPublic is initially false; otherwise, it is set to -1. Every cycle, if a store's address becomes untainted, it broadcasts its LSQ index. Every load in the LSQ listens to this broadcast, and if the broadcast index is between (inclusive) FwdingSt and the LSQ index of the load, then it decrement its NumStUntaintPending counter. When this counter reaches 0, the associated STLPublic bit is set to 1.

7.5 Shadow L1 (Taint Tracking of L1D Data)

The *shadow L1* is a core structure that mirrors the set-associative geometry of the L1D. Each entry stores the taint status of the corresponding L1D cache line at byte-level granularity and a valid bit. Shadow L1 entries require no replacement or tag bits because tag checking and eviction mirror L1D decisions, which are made available to the shadow L1 by connecting the L1D tag check logic output and the eviction decision output to the shadow L1. When a cache line is invalidated in the L1D, all of its shadow L1 taint bits are set. Thus, when an L1D line is filled, it is considered tainted.

When a load accesses the L1D, it (in parallel) accesses the shadow L1 to obtain the taint status of the data, or to update it if the load's output register is already untainted. Thus, the logic ANDs the two taint statuses and updates both to the result. To convert the 1-bit-per-byte taint status of the shadow L1 to a 4-bit taint status for the register, we conservatively OR the taint of the bytes. When a store writes to the L1D, the taint status of its data operand overwrites the written bytes' taint status in the shadow L1.

7.6 Performance and Cost Considerations

We eschewed the idea of centralizing the taint status storage since it would require unnecessary exchanges between the hardware that stores the taint status and the hardware that needs it, driving up the clock period. Having the taint status distributed means that the RS does not need to access other structures to compute untaint. Similarly, two-phase untaint propagation simplifies the untaint logic since it does not need to evaluate the entire dataflow graph and propagate untaint through the RS in a single cycle.

Having an untaint broadcast width that is much smaller than the size of the RS may hurt performance since it may take several cycles for the untaint to propagate through the dataflow graph. But keeping the broadcast width small saves a lot on hardware, particularly on wires. Fortunately, our evaluation shows that an untaint broadcast width of 3 is a good trade-off (Section 9.4).

8 SECURITY ANALYSIS

We sketch the proof that SPT satisfies its security guarantee (Definition 1). We prove the contrapositive of Definition 1:

Theorem 1. *If data gets untainted in SPT's speculative execution, then it is not secret in the non-speculative execution.*

We assume that if an instruction X reaches the visibility point, then X's control- and dataflow are as in the non-speculative execution. This assumption holds in the Futuristic model (where the VP is when X becomes non-squashable), in the Spectre model if the processor performs only control-flow speculation, and in a variant of the Spectre model where the VP is augmented to consider data speculation (e.g., memory dependence speculation).

Our proof formally models the processor as a state machine and reasons about the execution as a sequence of events (state transitions), as in [19, 34, 84]. We first prove an auxiliary lemma: that an instruction X's output register can get untainted before it is *ready* (written to by X) only if X reaches the VP.

Lemma 1. *If the (physical) output register R of X becomes untainted while not ready, then X has reached the VP.*

Proof. By induction over the sequence of events in the execution. The base case (empty sequence) is vacuously true. Inductive step: Since R is an output register, it cannot be an operand or output of older instructions. R can thus only get untainted in one of two cases: ① R is the operand of a younger transmitter that has reached the VP. ② R is the operand of a younger instruction X' performing a backward untaint step. But since R is not ready, the output of X' is not ready, and so by the inductive assumption, X' has reached the VP. In either case, we have an instruction younger than X that has reached the VP, implying that X has reached the VP. □

Consider an attacker that has (a) visibility over non-speculative side channels formed in the victim program's execution and (b) the program's static code. We say that a value X is *inferable by the attacker* at some point in the execution if $X = f(O)$, where f is a function known to the attacker and O are operands of transmitters that have reached the VP by that point. By definition, a value inferable by the attacker is not a secret in the non-speculative execution. The theorem thus follows from the lemma below.

Lemma 2. *The following hold in each clock cycle:*

(1) ROB state is public: *The contents (instructions) of the ROB is inferable by the attacker.*

(2) Untainted data is public: *If data (a register or memory location) D gets untainted, then D is inferable by the attacker.*

Proof. We prove both properties simultaneously by induction over the sequence of events in the execution. The base case is vacuously true. For the inductive step, assume that both properties hold in a prefix α of the execution, and consider the next event e.

Property (1) is a generalization of STT's "ROB contents are independent of speculatively-accessed data" property (Section 2.2.2). Given a notion of "taint" for registers, STT's mechanisms (inherited by SPT) make the sequence of instructions fetched/executed/squashed depend only on untainted data [84]. Property (1) follows because, by the induction for (2), "untainted" in SPT corresponds to non-secret data inferable by the attacker.

For proving Property (2), we need only consider untaint events. Due to space constraints, we sketch the arguments:

• D is a register untainted because it is the operand of a transmitter X that reaches the VP: The claim trivially holds.

• D is a register untainted due to the forward/backward untaint rules (Section 6.6): For example, $D = U + V$, where U and V are untainted. By the induction for (2) on U and V, we have $D = f_U(O) + f_V(O)$, which is known to the attacker (by the induction for (1)).

• D is L1D data at address A that gets untainted because a load L accesses A while having an untainted output register R (Section 6.8): Inductively, $A = f_A(O)$ and $R = f_R(O)$. By Lemma 1, L has reached the VP. Therefore, address A and all the store-to-load forwarding implicit branches involving L are untainted. By the induction for (1), the attacker knows when L executes and accesses the L1D. Since the attacker monitors all covert channels, including the cache, it therefore knows that L accesses A. Thus, the attacker knows that the contents of R is what is stored at memory address A.

• D is L1D data at address A that gets untainted because a store S writes to A while having an untainted value register R (Section 6.8): Because stores write to the cache on retirement, S has reached the VP and thus A is untainted. Inductively, $A = f_A(O)$ and $R = f_R(O)$. As above, by the induction for (1), the attacker knows that the contents of R is what is stored at memory address A.

• D is (a) an output register of a load L from address A, which read an untainted value R from store S via store-to-load forwarding; or (b) the value register of a store S to address A, which was forwarded to load L and L's output register R got untainted: As SPT propagates untaint over store-to-load forwarding only when the relevant implicit branch is untainted (Section 6.7), the induction for (1) implies that the attacker knows L reads its value from S. The claim follows, as inductively, $R = f_R(O)$.

□

9 EVALUATION

We evaluate SPT (on different attack models) relative to an insecure baseline, a secure baseline with the same protection scope, and STT (whose protection scope is narrower than SPT's).

9.1 Experimental Setup

Simulation setup. We use the Gem5 [17] cycle-accurate simulator. For SPT, we implement the equivalent of the microarchitectural changes detailed in Section 7. Table 1 shows the simulated machine. We use total store ordering (TSO) as the memory model. We run the SPEC CPU2017 [1] benchmarks (using the *reference* input size) and three data-oblivious code kernels: AES (bitslice [3]) and ChaCha20 (from BearSSL [4]) ciphers, and `djbsort` [15] sorting. For the SPEC benchmarks, we use SimPoint analysis [65], which breaks the overall execution into representative phases (average of 7 SimPoints per application). We simulate 50 M instructions of each phase and perform a weighted sum to obtain an overall execution time result.

Configurations. Table 2 lists the processor configurations evaluated. Transmit instructions are defined as loads and stores. We evaluate a SecureBaseline design, which delays execution of transmitters until they reach the visibility point, and thus provides the same protection scope as SPT. We also compare to STT, which has a narrower protection scope (i.e., only protects speculatively-accessed data).[6] For SPT, we evaluate a series of configurations that incrementally add the untaint mechanisms described in Section 6—forward register untainting, backward register untainting, and the shadow L1—to arrive at the full SPT design. These configurations

[6] Following [46], and for consistency with SPT, we assume an STT configuration that treats stores as transmitters.

HW Components	Parameters
Pipeline	8 fetch/decode/issue/commit, 32/32 SQ/LQ entries, 192 ROB, 16 MSHRs, LTAGE branch predictor
L1 I-Cache	32 KB, 64 B line, 4-way, 2-cycle latency
L1 D-Cache	32 KB, 64 B line, 8-way, 2-cycle latency
L2 Cache	256 KB, 64 B line, 16-way, 20-cycle latency
L3 Cache	2 MB, 64 B line, 16-way, 40-cycle latency
Network	4×2 mesh, 128 b link width, 1 cycle latency per hop
Coherence Protocol	Two-Level MESI protocol
DRAM	50 ns latency after L2
Untaint broadcast width (SPT only)	3

Table 1: Simulated architecture parameters.

use an untaint broadcast width of 3 (justification in Section 9.4). We further evaluate two *idealized* SPT variants, to evaluate the limits of the individual techniques: SPT {Bwd, ShadowMem} adds taint tracking of every byte in the entire memory to SPT. SPT {Ideal, ShadowMem} further adds single-cycle untaint propagation through the entire dataflow graph that is still represented in the ROB. On each cycle, it performs forward and backward untainting until no further data can be untainted.

Configuration	Description
UnsafeBaseline	An unmodified, insecure processor.
SecureBaseline	Loads and stores delayed until reaching the VP.
SPT {Fwd, NoShadowL1}	Forward untainting only (in RS). No shadow L1.
SPT {Bwd, NoShadowL1}	Forward and backward untainting (in RS). No shadow L1.
SPT {Bwd, ShadowL1} = **full SPT design**	Forward and backward untainting (in RS) plus shadow L1 (L1D taint tracking).
SPT {Bwd, ShadowMem}	Forward and backward untainting (in RS) plus *all memory* taint tracking.
SPT {Ideal, ShadowMem}	*Ideal* forward and backward untainting (in RS) plus *all memory* taint tracking.
STT	Only protects speculatively-accessed data.

Table 2: Evaluated design variants.

Penetration testing. We experimentally confirm that all SPT configurations protect both speculatively-accessed data (a standard Spectre V1 attack) and non-speculative secrets.

9.2 Main Result: Performance of SPT

Figure 7 shows the performance results of all the benchmarks in the Futuristic and Spectre attack models. Execution times shown are normalized to the execution time of UnsafeBaseline. SPT effectively reduces the overhead compared to SecureBaseline, obtaining 3.6× and 3× lower average overhead in the Futuristic and Spectre models, respectively. Compared to UnsafeBaseline, SPT adds 45% and 11% average overhead in the Futuristic and Spectre models, respectively.

Adding each individual SPT untainting technique results in an overhead reduction. SPT {Fwd, NoShadowL1}, which represents our core idea of tracking and propagating untaint, already achieves an average overhead reduction of 3.1× (Futuristic) and 1.9× (Spectre) over SecureBaseline. Backward untainting (SPT {Bwd, NoShadowL1}) further reduces average overhead by 1.8 and 3.5 percentage points in the Futuristic and Spectre models, respectively. Notice that SPT {Ideal, ShadowMem} provides negligible improvement over SPT {Bwd, ShadowMem}, which means that the microarchitectural constraint of limited untaint bandwidth does not, in fact, hurt SPT's performance.

Tracking L1D data taint (SPT {Bwd, ShadowL1}) is also effective. It reduces average overhead over register untainting by 5.2 and 3.2 percentage points in the Futuristic and Spectre models, respectively, and by up to 15.9 percentage points on individual benchmarks (`perlbench` in the Futuristic model). Idealized taint tracking of all memory (SPT {Bwd, ShadowMem}) reduces average overhead by ≈ 1 percentage point over SPT, in both attack models, and has no effect on many benchmarks. Thus, SPT's L1D taint tracking is a good trade-off that reduces hardware cost for almost no performance loss.

Data-oblivious code. As mentioned previously, a key use case for SPT is protecting constant-time (data-oblivious) programs. SPT is particularly effective on our evaluated constant-time code kernels, which suffer a 2.8× average slowdown when run on the SecureBaseline in the Futuristic model (the conservative model appropriate for these security-sensitive programs). SPT reduces the overhead to only 1.10×, an 18× reduction relative to SecureBaseline.

STT. Comparing to STT shows the cost of SPT's comprehensive protection scope (protecting all secrets, not only speculatively-accessed data). SPT adds an extra 3.3 and 26.1 percentage points of overhead over STT in the Spectre and Futuristic models, respectively.

9.3 Breakdown of Untaint Events

Figure 8 shows the breakdown of *untaint events*—i.e. any instance of a register being untainted—into their types. We provide the breakdown for every benchmark for both the Futuristic and Spectre attack models. The results were done by analyzing runs of SPT, i.e. SPT {Bwd, ShadowL1}. Note that while the configurations from Table 2 are inclusive—for example the mechanisms in {Fwd, NoShadowL1} are included in {Bwd, NoShadowL1}—the untaint events from the breakdown graph are exclusive.

One interesting observation is that, as expected, benchmarks whose untaint events mostly consist of forward untaints (e.g. `fotonik` and `namd` under the Futuristic model) tend to have massive reductions in overhead occur when forward untainting is introduced, but each incremental addition yields almost no additional benefits. The intuition is that there are not enough opportunities to be exploited for the additional hardware to provide any benefit.

In other benchmarks, such as `perlbench` and `povray`, we can see a similar correlation between the prevalence of shadow L1 untaint events and the amount of overhead reduction when introducing the shadow L1. In benchmarks such as `parest` we see little overhead reduction when introducing the shadow L1 due to the low prevalence of like untaint opportunities.

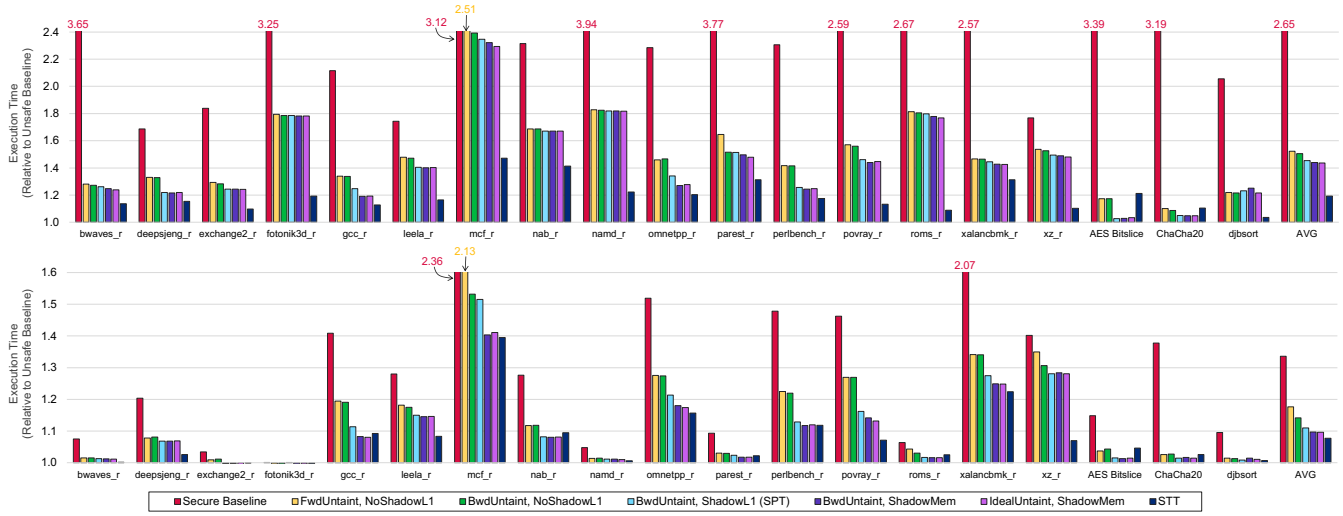


Figure 7: Execution time (relative to UnsafeBaseline) of SPEC2017 and data-oblivious benchmarks of the designs from Table 2. Averages appear on the right. (Top graph: Futuristic attack model; bottom graph: Spectre attack model.)

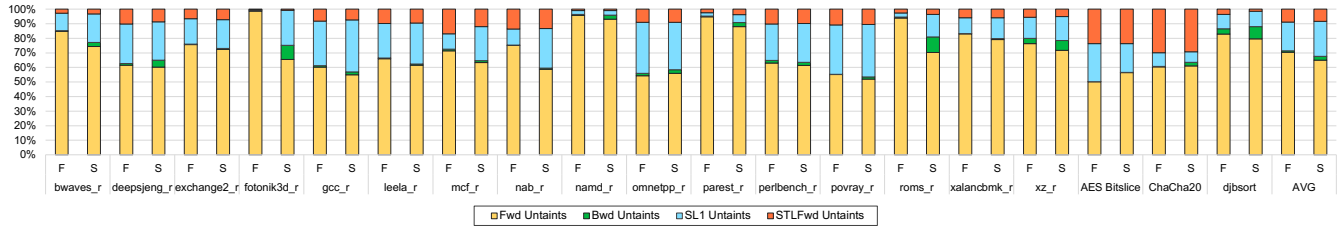


Figure 8: A per-benchmark breakdown of the untaint events that occur when running the SPT {Bwd, ShadowL1} configuration. Runs under the Futuristic model are represented by F, while runs under the Spectre model are represented by S. Note that the untaint events are exclusive even though the configurations from Figure 7 are inclusive.

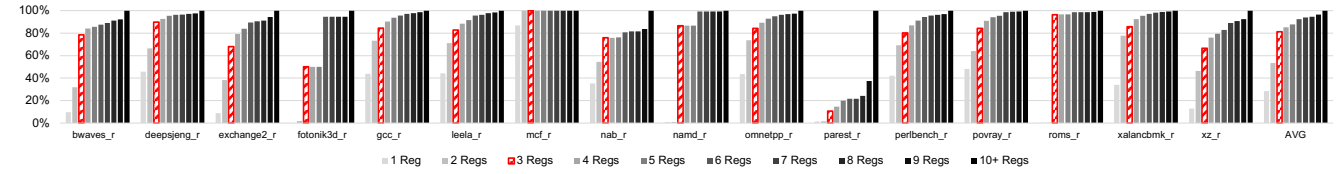


Figure 9: Percentage of cycles with untaint events in which the number of untainted registers is at most $N = 1, \ldots, 10+$ when running SPT {Ideal, ShadowMem} on the SPEC benchmarks.

Finally, we see that `mcf` benefits the most from backward untainting. Yet the prevalence of backward untaint events is low (and in fact is low for most benchmarks). We attribute this surprising reduction in overhead to the fact that while a low volume of data was untainted, the *right* data was untainted.

At the same time, we see more backwards untainting occurring in the Spectre model relative to the Futuristic model. This is because we only backwards untaint to the head of the ROB, and there is more distance between the VP and the head of the ROB in the Spectre model, on average. This indicates that an implementation enabling backwards untainting to instructions older than the ROB head may be able to exploit more opportunity.

9.4 Choosing an Untaint Broadcast Width

To choose the untaint broadcast width in our simulations, we run the SPEC benchmarks on SPT {Ideal, ShadowMem}. For each cycle in which registers are untainted, referred to as an *untainting cycle*, we measure how many registers are untainted in the RS (this number is not bounded with ideal untainting). Figure 9 shows, for each application, the percentage of untainting cycles in which the number of physical registers untainted is $\leq N$, for various values of N. For example, in `bwaves`, roughly 32% of untainting cycles untaint 1 or 2 registers. We see that on average, roughly 81% of untainting cycles untaint at most 3 registers. Thus, a broadcast width of 3 is a good trade-off between coverage of cases and hardware complexity.

Scheme	Data protection scope	Transmitter scope	Receiver scope	Programmer transparent?
InvisiSpec [76]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
SafeSpec [39]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
DAWG [40]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
Delay-on-miss [59]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
Cond. Spec. [44]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
MuonTrap [7]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
CleanupSpec [58]	Spec/Non-spec accessed data	Cache-based	CC, ST	yes
CSF [69]	Spec/Non-spec accessed data	Cache-based	CC, ST	no, user annotates secrets
MI6 [18]	Spec/Non-spec accessed data	All	CC, ST	yes
ConTExT [61]	Spec/Non-spec accessed data	All	CC, ST, SMT	no, user annotates secrets
OISA [81]	Spec/Non-spec accessed data	All	CC, ST, SMT	no, user annotates secrets
STT [83]	Spec accessed data	All	CC, ST, SMT	yes
SDO [82]	Spec accessed data	All	CC, ST, SMT	yes
SpecShield [11]	Spec accessed data	All	CC, ST, SMT	yes
NDA [74]	Spec/Non-spec accessed data	All	CC, ST, SMT	yes
Dolma [46]	Spec/Non-spec accessed data	All	CC, ST	yes
SPT (this work)	**Non-spec secrets**	**All**	**CC, ST, SMT**	**yes**

Table 3: Prior hardware-based mitigations for speculative execution attacks. Transmitter scope refers to what (speculative) covert channels are blocked. Receiver scope refers to where the receiver component of the attacker [40] is allowed to run. CC, ST, SMT mean "CrossCore", "SameThread" and "SMT", respectively, referring to different places the receiver can run [76].

10 RELATED WORK

Prior hardware defenses can be categorized based on whether they block all speculative covert channels and/or are comprehensive (protect both speculatively-accessed and non speculatively-accessed data). Table 3 summarizes the literature along these, and several other, dimensions. Invisible speculation schemes such as InvisiSpec [76], SafeSpec [39], DAWG [40], delay-on-miss [59], conditional speculation [44] and MuonTrap [7] only block speculative covert channels through the cache. MI6 [18] blocks all speculative covert channels given a receiver (attacker) co-located to the same machine, but still leaks termination time to a remote receiver (i.e., does not block NetSpectre [62]). Dolma [46] likewise blocks more speculative covert channels, yet does not block those that can be monitored by a receiver that runs in a sibling SMT context.[7] Other schemes such as STT [83], SDO [82] and SpecShield [11] block all speculative covert channels, but only protect speculatively-accessed data. Finally, we note that NDA [74] is capable of protecting non-speculatively-accessed data by delaying execution of transmitters until they become non-speculative, which incurs a prohibitive performance cost (> 100% [46]).

Relative to the above, this paper's contribution is a novel framework for efficient, comprehensive protection of both speculatively-accessed and non-speculative secrets over all speculative covert channels. The key idea is to selectively disable protection based on inherent declassification events that occur when the program transmits data over non-speculative covert/side channels, i.e., to only protect data that is non-speculatively secret (Section 3). This definition of what is secret is compatible with speculative non-interference [33, 34] and TPOD [21]. To our knowledge, our work is the first to be able to enforce these security definitions in a low-overhead/software-transparent manner.

Finally, software defenses perform program analysis that reasons about mis-speculated execution flows to detect [19, 32, 33, 72] or prevent [72] speculative leakage. Relative to SPT, these tools typically do not scale to large code bases [33], require software changes and, similar to prior hardware defenses, use conservative definitions for what data is secret.

[7] Specifically, Dolma uses delay-on-miss [59] to issue speculative loads early. Thus, if a load with a secret address hits or misses in the L1 cache, Dolma either forwards the result to dependent instructions or delays forwarding. Which of these occurs can easily be monitored by a sibling SMT context through, e.g., port contention effects [12, 16, 67]. By contrast, SPT uses a secure policy to protect such loads by delaying their execution.

11 CONCLUSION

This paper proposed SPT, a novel mechanism to efficiently and comprehensively block speculative execution attacks. The key idea is that it is safe to speculatively execute a transmitter without any protection, if its operands were already leaked by the non-speculative execution. This enables a clean, easy to reason about security property, which extends data-oblivious security guarantees to speculative execution. It also enables a novel framework for selectively disabling protection on data that SPT can prove can be inferred over non-speculative covert/side channels.

ACKNOWLEDGMENTS

This work was funded in part by NSF under grants CNS #1942888, #1954521 and #1816226, ISF under grant #2005/17, and by an Intel Strategic Research Alliance (ISRA) grant. We thank the anonymous reviewers for their insights during the review process.

REFERENCES

[1] 2017. SPEC CPU2017. https://www.spec.org/cpu2017.
[2] 2019. InvisiSpec-1.0 simulator bug fix. https://github.com/mjyan0720/InvisiSpec-1.0/commit/f29164ba510b92397a26d8958fd87c0a2b636b0c.
[3] 2020. Bitslice AES (Bitcoin). https://github.com/bitcoin-core/ctaes.

[4] 2021. ChaCha20 (BearSSL). https://bearssl.org/gitweb/.
[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys via Branch Prediction. In *CT-RSA '07*.
[6] Adil Ahmad, KyungTae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. Obliviate: A Data Oblivious Filesystem for Intel SGX. In *NDSS '18*.
[7] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *ISCA '20*.
[8] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. IEEE S&P '19.
[9] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security '16*.
[10] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *IEEE S&P '15*.
[11] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *PACT '19*.
[12] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *ASPLOS '21*.
[13] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *FSE '05*.
[14] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC '06*.
[15] Daniel J. Bernstein. 2019. djbsort. https://sorting.cr.yp.to/.
[16] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS '19*.
[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 2 (2011), 1–7.
[18] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *MICRO '19*.
[19] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *PLDI '20*.
[20] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *IEEE SecDev '17*.
[21] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *CSF'19*.
[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *IEEE EuroS&P '19*.
[23] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE S&P'09*.
[24] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *ISCA '07*.
[25] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *VLDB Endowment* 13, 2 (2019).
[26] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *CCS '16*.
[27] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS '18*.
[28] Ben A. Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: Functional Encryption using Intel SGX. In *CCS '17*.
[29] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security '18*.
[30] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *ICISC '09*.
[31] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS '16*.
[32] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *CCS '20*.
[33] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *IEEE S&P '20*.
[34] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *IEEE S&P '21*.
[35] Shay Gueron. 2012. Efficient Software Implementations of Modular Exponentiation. *Journal of Cryptographic Engineering* 2 (2012). Issue 1.
[36] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.
[37] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.
[38] Mike Johnson. 1991. *Superscalar Microprocessor Design.* Prentice Hall Englewood Cliffs, New Jersey.
[39] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC '19*.
[40] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO '18*.
[41] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. arXiv:1807.03757 [cs.CR]
[42] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P '19*.
[43] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT '18*.
[44] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *HPCA '19*.
[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P '15*.
[46] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security '21*.
[47] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS '18*.
[48] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv:1902.05178 [cs.PL]
[49] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *IEEE S&P'18*.
[50] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. *International Journal of Parallel Programming* 47, 4 (2019).
[51] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. *ICISC '05*.
[52] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security '16*.
[53] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA '06*.
[54] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security '16*.
[55] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security '15*.
[56] Glenn Reinman and Brad Calder. 1998. Predictive Techniques for Aggressive Load Speculation. In *MICRO '98*.
[57] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and Principles. *J. Comput. Secur.* 17, 5 (Oct. 2009), 517–548.
[58] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *MICRO '19*.
[59] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *ISCA '19*.
[60] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS'18*.
[61] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS '20*.
[62] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS '19*.
[63] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *CCS '17*.

[64] John Paul Shen and Mikko H. Lipasti. 2005. *Modern Processor Design: Fundamentals of Superscalar Processors* (1st ed.). Waveland Press, Inc.
[65] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS '02*.
[66] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS '18*.
[67] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *ISCA '19*.
[68] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS '04*.
[69] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing : Securing Speculative Execution via Microcode Customization. In *ASPLOS '19*.
[70] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *ASPLOS '09*.
[71] Shruti Tople and Prateek Saxena. 2017. On the Trade-Offs in Oblivious Execution Techniques. In *DIMVA '17*.
[72] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 49 (2021).
[73] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS '17*.
[74] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO '19*.
[75] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P '15*.
[76] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO '18*.
[77] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE S&P '19*.
[78] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security '14*.
[79] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *Journal of Cryptographic Engineering* 7 (2017). Issue 2.
[80] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2018. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *IACR '18*.
[81] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS '19*. https://eprint.iacr.org/2018/808.
[82] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *ISCA '20*.
[83] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO'19*.
[84] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. *Speculative Taint Tracking (STT): A Formal Analysis*. Technical Report. University of Illinois at Urbana-Champaign and Tel Aviv University. http://cwfletcher.net/Content/Publications/Academics/TechReport/stt-formal-tr_micro19.pdf.
[85] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI '17*.

A ARTIFACT DESCRIPTION

A.1 Abstract

We briefly describe the process of building and running our version of Gem5. For the full description, see our repository on GitHub at https://github.com/FPSG-UIUC/SPT.

A.2 Requirements

The following are required to build and run our version of Gem5. **Note:** Ubuntu 16.04 is *required*; we attempted to run our repo on Ubuntu 20.04 and were met with irresolvable issues. We advise researchers to create a Docker container based on a base image for Ubuntu 16.04 and install the dependencies there.

Tool	Version
Ubuntu	16.04
Python	2.7
Python	3.5+
SCons	2.5.1
g++	3.5.1

A.3 Cloning and Building

To clone the repository, first `cd` to whichever directory you want your local copy to be. Then run:

```
> git clone https://github.com/FPSG-UIUC/SPT.git
```

We will refer to the folder that has just been created as `$SPT`. To build, run the following commands:

```
> cd $SPT
> /usr/bin/scons ./build/X86_MESI_Two_Level/gem5.fast -j <# CPUs>
```

A.4 Running

A.4.1 Helper Script. Running the Gem5 executable directly is a bit complicated since there are some legacy command-line options from previous projects. Thus it is highly recommended to use the helper script `run_spt.py`. Gem5 will print the actual command when it is run. The script `run_spt.py` must be run with Python version 3.5 or later. The following parameters are available to the script:

Parameter	Values	Description	Requirements
`--executable`	Filesystem Path	The executable you want to run with gem5	Required
`--enable-spt`	n/a	Enables SPT's protection mechanism	Not Required
`--threat-model`	`spectre`, `futuristic`	Which threat model to simulate under	Required if `--enable-spt` is specified
`--untaint-method`	`none`, `fwd`, `bwd`, `ideal`	What type of untaint propagation to allow	Required if `--enable-spt` is specified
`--enable-shadow-l1`	n/a	Enables the shadow L1	Cannot specify both this and `--enable-shadow-mem`
`--enable-shadow-mem`	n/a	Enables shadow memory	Cannot specify both this and `--enable-shadow-l1`
`--track-insts`	n/a	Makes gem5 output detailed taint tracking information	Can only specify if `--enable-spt` is specified
`--output-dir`	Filesystem Path	The directory where the output file `stats.txt` will be generated	Not Required

A.4.2 Running Configurations from the Evaluation. Below are the configurations from Table 2 in Section 9 and the corresponding parameters. Note that `--executable`, `--track-insts`, and `--output-dir` are left to the user to specify and are omitted from the table.

Configuration	Parameters
SecureBaseline	`--enable-spt --untaint-method=none`
SPT{Fwd, NoShadowL1}	`--enable-spt --untaint-method=fwd`
SPT{Bwd, NoShadowL1}	`--enable-spt --untaint-method=bwd`
SPT{Bwd, ShadowL1}	`--enable-spt --untaint-method=bwd --enable-shadow-l1`
SPT{Bwd, ShadowMem}	`--enable-spt --untaint-method=bwd --enable-shadow-mem`
SPT{Ideal, ShadowMem}	`--enable-spt --untaint-method=ideal --enable-shadow-mem`

Note that to run InsecureBaseline, simply provide the `--executable` and nothing else.

A.4.3 Getting the Results. When Gem5 is run, it will create a file called `stats.txt` in the directory you specified with `--output-dir`. If you did not specify `--output-dir` then `stats.txt` will be in `$SPT/m5out`.

In the file are many statistics, though the one of most interest will be `numCycles`, which specifies how many cycles the program took to execute. There are many other statistics provided by Gem5 that have comments next to them briefly describing what they are.

On top of the statistics provided by Gem5, we provide some custom statistics as well. See the `README` in our repository for a brief description of them.