Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data

Jiyong Yu
University of Illinois at Urbana-Champaign
jiyongy2@illinois.edu

Mengjia Yan
University of Illinois at Urbana-Champaign
myan8@illinois.edu

Artem Khyzha
Tel Aviv University
artkhyzha@mail.tau.ac.il

Adam Morrison
Tel Aviv University
mad@cs.tau.ac.il

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletch@illinois.edu

ABSTRACT

Speculative execution attacks present an enormous security threat, capable of reading arbitrary program data under malicious speculation, and later exfiltrating that data over microarchitectural covert channels. Since these attacks first rely on being able to *read* arbitrary data (potential *secrets*), a conservative approach to defeat all attacks is to delay the execution of instructions that read those secrets, until those instructions become non-speculative.

This paper's premise is that it is safe to *execute and selectively forward* the results of speculative instructions that read secrets, which improves performance, as long as we can prove that the forwarded results do not reach potential covert channels. We propose a comprehensive hardware protection based on this idea, called Speculative Taint Tracking (STT), capable of protecting all speculatively accessed data.

Our work addresses two key challenges. First, to safely selectively forward secrets, we must understand what instruction(s) can form covert channels. We provide a comprehensive study of covert channels on speculative microarchitectures, and use this study to develop hardware mechanisms that block each class of channel. Along the way, we find new classes of covert channels related to implicit flow on speculative machines. Second, for performance, it is essential to disable protection on previously protected data, as soon as doing so is safe. We identify that the earliest time is when the instruction(s) producing the protected data become non-speculative, and design a novel microarchitecture for disabling protection at this moment.

We provide an extensive formal analysis showing that STT enforces a novel form of non-interference, with respect to all speculatively accessed data. We further evaluate STT on 21 SPEC and 9 PARSEC workloads, and find it adds only 8.5%/14.5% overhead (depending on attack model) relative to an insecure machine, while reducing overhead by 4.7×/18.8× relative to a baseline secure scheme.

MICRO '52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358274

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

Security, Speculative execution attacks, Hardware, Information flow

ACM Reference Format:
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 '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358274

1 INTRODUCTION

Spectre [31], Meltdown [35] and follow-up attacks [8, 13, 24, 30, 32, 36, 46, 54, 57, 58] based on speculative execution have opened a new chapter in hardware security. In these attacks, adversary-crafted sequences of transient instructions—i.e., speculative instructions bound to squash—access and then transmit sensitive program data over *microarchitectural covert channels* (e.g., the cache [59]). For example, Spectre Variant 1, shown in Figure 1, bypasses a bounds check due to a branch misprediction and transmits secret data behind that bounds check over a cache-based covert channel [31]. Since the address `addr` can take an arbitrary value, `val` can be any value in program memory, meaning the covert channel can reveal arbitrary program data. (In this paper, we denote a potentially secret value in `green`.)

```
uint8 A[10];
uint8 B[256*64];
void victim (size_t addr) {
    if (addr < 10) { // mispredicted branch
M1:     uint8 val = A[addr]; // secret is accessed
M2:     ... = B[64 * val]; // secret is transmitted
    }
}
```

Figure 1: Spectre Variant 1 assuming a 64 byte cache line size. Variables carrying potentially secret data are colored `green`. If the `if` condition is predicted as true, then the cache line of B indexed by `val` is loaded to the cache (load M2) even though both loads are eventually squashed.

Prior work has pointed out that speculative execution attacks are broken into two components [29, 46]. First, a secret value is speculatively *accessed* and read into architectural state (e.g., a register) due to adversary-controlled speculative execution. For example, load M1 in Figure 1 reads val even if addr ≥ 10 due to a branch misprediction. Second, that secret value is *transmitted* over a covert channel (formed using one or more younger instructions). For example, load M2 in Figure 1 transmits the secret over a cache-based covert channel (displacing the attacker's line in the cache).

Using this distinction, a conservative scheme to protect *all* speculatively accessed data is therefore to delay the execution of any instruction deemed capable of accessing a secret (*access instruction* for short) until it becomes non-speculative. For example, if we define access instructions to be "all loads," it isn't possible for val in Figure 1 to leak an out of bounds value through the covert channel formed by load M2, since we delay executing load M1 until the branch resolves (and squashes). On the other hand, this scheme has high overhead, as delayed execution blocks execution for all dependent instructions.

1.1 This Paper

The key observation underpinning this paper is that one can improve the above conservative scheme's performance, without hurting security, by *executing and selectively forwarding* the results of speculative access instructions to younger instructions, as long as those younger instructions cannot form a covert channel. For example, suppose the microarchitect designs simple arithmetic (e.g., adds, xors) to have data-independent timing (e.g., implemented with a single-cycle ALU). Then, it is safe to execute and forward the result of load M1 to these dependent instructions, because their execution cannot reveal the result's value. By issuing load M1 and the arithmetic early, we improve performance if the branch resolves with a correct prediction.

This paper designs a framework to selectively forward data in this fashion, providing an efficient mechanism to comprehensively protect all speculatively accessed data. At a high level, our scheme tracks the flow of results from access instructions, through their def-use chains in a manner similar to dynamic information flow tracking [17, 49], until those results reach an instruction, or sequence of instructions, that may form a covert channel. *Only at that later point* do we stop forwarding the access instruction-dependent value. To be secure and efficient, this approach needs to solve two key technical challenges:

Challenge 1: Blocking leakage through all covert channels. First, paramount to deciding when to forward the results of speculative access instructions (called *secrets* for short) is having a complete understanding of how instructions can form covert channels in speculative execution attacks. This is not trivial, as prior attacks have shown there to be many ways to leak a secret (e.g., through loads that interact with the cache [31], SIMD units [46], and port contention [8]).

A key contribution of this paper is a comprehensive study of how instructions can be used to create covert channels and communicate data. In particular, we find that all covert channels are one of two flavors, which we call explicit and implicit channels. First, in an *explicit channel* data is directly passed to an instruction whose execution creates operand-dependent hardware resource usage, and that resource usage reveals the data. For example, how a load impacts the cache depends on the load address [31]. Second, 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, the instructions executed after a branch reveal the branch predicate [8, 46].

Implicit channels are related to implicit flow from the information flow literature [43], which is notoriously difficult to deal with in side channel research [52]. To our knowledge, we are the first to study and provide comprehensive protection for implicit channels in the speculative execution attack setting. Along the way, we also discover new ways that these channels can leak, and also find entirely new forms of implicit channels, unique to speculative microarchitectures.

Challenge 2: Disabling protection as soon as access instructions become non-speculative. Second, to be efficient, it is important to disable protection on data produced by access instructions, as soon as doing so is safe. Consider the example in Figure 1. Here, a simple, secure scheme is to execute and forward data from load M1, yet wait to issue load M2 until load M2 reaches the head of the ROB. This is overly conservative. In fact, it is safe to issue load M2 as soon as the branch resolves in a correct prediction, as this is the soonest point when the data returned by load M1 is no longer considered secret (i.e., load M1 is no longer speculative). To reiterate: *at that earlier point*, we can issue load M2. This is important for performance. The later we delay issuing load M2, the greater the chance it delays instruction retirement in the ROB.

The general principle is that it is safe to disable protection on data, as soon as the data's producer access instruction(s) have all become non-speculative. This is technically challenging for a variety of reasons, as data can be the result of complicated def-use chains through potentially many access instructions, and other instructions such as arithmetic. Yet, our solution requires simple hardware and can disable protection on any protected data in a data-independent number of cycles (e.g., 1 cycle), regardless of the complexity of def-use dependencies through older instructions.

Putting everything together, we call our combined protection scheme Speculative Taint Tracking, or STT for short.

Security guarantees and formal analysis. In addition to proposing STT itself, we provide an extensive formal analysis and prove that STT enforces a novel form of non-interference [38] with respect to speculatively accessed data, given a powerful adversary that can monitor potentially any covert channel at cycle granularity. We show how this implies that with STT enabled, *arbitrary speculative execution is only able to leak retired register file state as opposed to arbitrary program memory*. This means STT comprehensively defeats the worst Spectre attacks, e.g., those which form a universal read gadget [37] such as Spectre Variant 1. We provide an overview of our analysis in this paper, and provide proof details in a companion technical report [61].

Contributions. To summarize, we make the following contributions:

(1) We provide a comprehensive study of covert channels on speculative microarchitectures, including the first in-depth look at implicit channels (related to implicit flow), new ways

implicit channels can leak, and new forms of implicit channels not yet exploited by speculative attacks.

(2) Based on our study of covert channels, we propose a general framework for preventing speculatively accessed data from leaking over any covert channel.
(3) We propose a novel scheme to quickly disable protection on flows of data, once the data's producer access instruction(s) becomes non-speculative.
(4) We formalize our protection mechanisms and show they are able to achieve a strong security definition, akin to non-interference [38], with respect to data returned by speculative access instructions.
(5) We extensively evaluate STT on 21 SPEC and 9 PARSEC workloads, and find it adds only 8.5%/14.5% overhead (depending on threat model) relative to an insecure machine, while reducing overhead by 4.7×/18.8× relative to the baseline secure scheme from Section 1.

We have open-sourced our simulation infrastructure used for performance studies here: https://github.com/cwfletcher/stt.

2 BACKGROUND

Out-of-order Execution. Dynamically scheduled processors execute instructions in parallel and out of program order to improve performance [21, 53]. Instructions are fetched in the processor *frontend*, *dispatched* to *reservation stations* for scheduling, *issued* to execution (functional) units in the processor *backend*, and finally *retired* (at which point they update architected system state). Instructions proceed through the frontend, backend and retirement stages in order, possibly out of order, and in order, respectively. In-order retirement is implemented by queueing instructions in a reorder buffer (ROB) [26] in instruction fetch order and retiring a completed instruction when it reaches the ROB head. Instructions are referred to by their age in the ROB, i.e., if I_1 precedes I_2 in fetch order, then I_2 is *younger* than I_1.

Speculative Execution. Speculative execution improves performance by executing instructions whose validity is uncertain instead of waiting to determine their validity. If such a speculative instruction turns out to be valid, it is eventually retired; otherwise, it is *squashed* and the processor's state is rolled back to a valid state. (As a byproduct, all following instructions also get squashed.)

3 ATTACKER MODEL

We assume a powerful adversary that can monitor any microarchitectural covert channel from anywhere in the system, and induce arbitrarily speculative execution to access secrets and create covert channels. (For a more formal definition, see the BitCycle adversary from [60].) For example, the attacker can monitor covert channels through the cache/memory system [31], data-dependent arithmetic [20], port contention [8], branch predictors [3], etc. As in [58], the adversary may try to induce and monitor malicious speculative execution by priming predictors, caches, etc.—from within the victim thread itself (*SameThread* [46]), or from an external context such as an SMT sibling (*SMT*) or nearby processor core (*CrossCore*).

4 SCOPE: PROTECTING SPECULATIVELY ACCESSED DATA

A speculative execution attack consists of two components [29, 46]. First, an instruction that reads a potential secret into a register, making it accessible to younger instructions. We call this instruction the *access instruction* [29]. Second, a younger instruction or instructions that exfiltrate the secret over a covert channel. The access instruction is almost always a load [8, 13, 24, 30, 32, 36, 46, 54, 57], but some attacks use a privileged register read [11].

We further distinguish attacks based on whether the access instruction is *transient* or *non-transient*, i.e., doomed to squash or bound to retire, respectively [11]. Figure 2 shows the general schema. Note that the covert channel must be transient. Otherwise, all older instructions—including the access instruction—are also non-transient, which means that the attack is a traditional side channel attack (e.g., [39]) and out of scope.

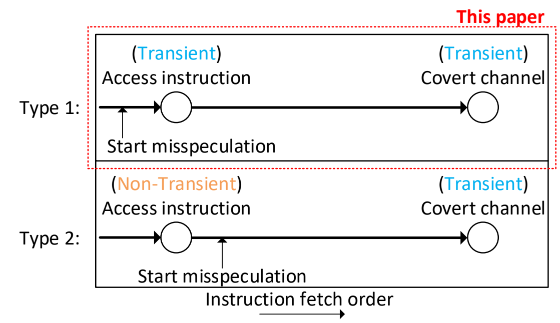

Figure 2: Schema for speculative execution attacks. Attacks can be classified into two types, depending on whether the instruction accessing the secret is transient (top) or non-transient (bottom). This paper protects data returned by transient access instructions.

This paper's goal is to block attacks involving transient access instructions, which are arguably the most dangerous speculative execution attacks. The reason is that a transient access instruction can often be maneuvered to access data that its correct (not misspeculated) execution would never access. The worst such attacks can read from any location in memory, which is referred to as a *universal read gadget* [37]. For example, in Spectre Variant 1 (Figure 1), misspeculating that a bounds check passes allows the transient access instruction—load `M1`—to read from an arbitrary out-of-bounds address. There are additional universal read gadgets that exploit different program constructs and covert channels [37].

In contrast, attacks involving non-transient access instructions cannot create a universal read gadget, because they can only leak *retired (or bound to retire) register file state*. Figure 3 depicts such an attack. Here, a `secret` is legitimately accessed by the program, i.e.,

```
secret = *addr; // retired (non-transient) access instruction
...
if (...) { // mispredicted branch
  b = B[64 * secret]; // secret is transmitted
}
```

Figure 3: Example speculative execution attack involving a non-transient access instruction. Blocking this class of attack is out of scope.

the access instruction retires. Later, a transient covert channel created through a branch misprediction exfiltrates `secret`. Although such leakage is important to address, it is clearly less dangerous than leaking all of program memory. Moreover, potential leakage of retired state can be reasoned about by programmers and compilers and blocked using complementary techniques (e.g., [60]). Therefore, we consider such leakage out of scope.

5 COVERT CHANNELS IN SPECULATIVE EXECUTION ATTACKS

As discussed in Section 1, STT executes and selectively forwards the results of speculative access instructions (which are deemed *secrets*) to younger instructions. For security, it is essential to understand how instructions, computing on secrets, can be used to create covert channels. For this, we propose a novel abstraction for covert channels in the speculative execution attack setting, shown in Figure 4. We note that, although our scope is protecting data read by speculative access instructions (Section 4), our analysis here applies to covert channels following non-speculative access instructions as well.

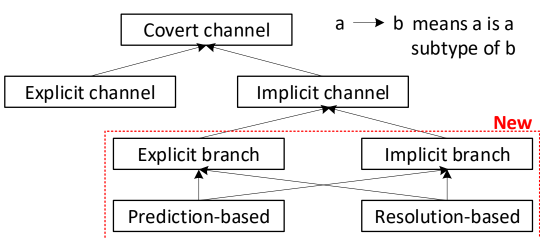

Figure 4: Covert channel classification on speculative microarchitectures.

5.1 Explicit vs. Implicit Channels

To start, we classify all covert channels as one of two types: explicit channels and implicit channels. An *explicit channel*, related to explicit flow in information flow [43, 52], is one where data (e.g., a secret) is *directly* passed to an instruction whose execution creates operand-dependent hardware resource usage, and that resource usage reveals the data. An *implicit channel*, related to implicit flow [43, 52], is one where data *indirectly* influences how (or that) an instruction or several instructions execute, and these changes in resource usage reveal the data. Examples of explicit channels are memory instructions (e.g., load `M2` from Figure 1), variable latency arithmetic instructions [20] and prefetch instructions. Importantly, that load `M2` executes is not secret; it is *how* the load executes (i.e., brings a line into cache at a secret-dependent set) that leaks. Recent speculative execution attacks have also started exploiting implicit channels. Examples are branches with secret-dependent predicates, which influence the instruction cache footprint, program timing [46], execution unit port usage [8], etc.

A key contribution in this paper is finding new ways that implicit channels can leak (Section 5.2), and finding entirely new classes of implicit channels related to what we call "implicit branches" (Section 5.3). Figure 5 gives examples of "traditional" (Figure 5(a)) and new (Figure 5(b)-(c)) channels. We denote the value being revealed through the channel as `secret`. The examples assume

(a) Control dependency:	(b) Squash dep. (new):	(c) Alias dep. (new):

`if (secret)`
`load rX <- (rY)`
	`if (secret)`
	`rX += 64`
	`load rY <- (rZ)`
		`store rX -> (secret)`
		`load rY <- (rZ)`

Figure 5: Examples of implicit covert channels revealing `secret`. Assume an older speculative access instruction has already read `secret` into a register, e.g., `M1` in Figure 1. The attacker can see the sequence of load addresses sent to the memory system. rX, rY and rZ are registers. Each of these covert channels can be "plugged into" existing attacks as the "Covert channel" in Figure 2. For example, in Spectre V1 (Figure 1) we can replace load `M2` with one of (a)-(c) above.

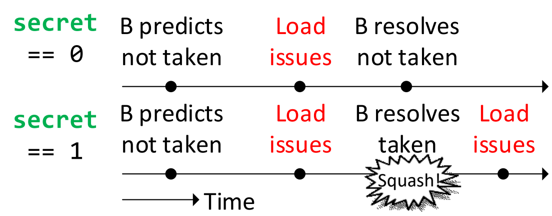

Figure 6: Resolution-based implicit channel for example Figure 5 (b). When the branch (B) resolves, it leaks the secret based on whether a squash occurs. There is an analogous case when the (public) predictor state takes the branch.

the attacker can monitor the cache-based covert channel, but in many cases (e.g., Figure 5(a) and (b)) the load can be replaced by an instruction *whose execution timing/etc. does not depend on its input*. Importantly, `secret` is not passed directly as the load address in any of the examples, yet still leaks.

5.2 Prediction- vs. Resolution-based Leakage

We make a key observation that on speculative machines, implicit channels can leak secrets at two points: when a control-flow prediction is made (if any) and when that prediction is resolved. Recall, branch prediction and resolution occur in the processor frontend and backend, respectively (Section 2). This creates new types of leakage depending on the adversary's capability. In the following, consider a branch whose predicate depends on a secret.

At *prediction time*, the sequence of instructions fetched after this branch is fetched (after branch prediction but before resolution) leaks secrets if the predictor structures have been updated based on secret information at some time in the past. For example, if an attacker runs repeated experiments and the branch predictor is updated speculatively based on how the branch resolves, the branch predictor "learns" the secret and will make future predictions based on the secret.

At *resolution time*, the branch can also leak the secret *even if the predictor state has not been updated based on secret data*, because incorrect predictions will cause a pipeline squash. See the code snippet in Figure 5(b), whose timing is shown as a function of the secret in Figure 6. If the attacker knows the branch will predict not taken (e.g., by priming it beforehand [31]), a squash means the branch was actually taken. The adversary can observe the squash through different effects, e.g., program timing or the fact that the load issues twice. Importantly, Figure 5(b) would not be considered a leak in traditional implicit flow, because the load is control- and data-independent of the branch.

5.3 Explicit vs. Implicit Branches

We make a key observation that on speculative machines, non-control flow instructions that speculate can similarly influence control flow based on conditions in the pipeline. For example, in Figure 5(c) there is no control flow instruction and the load address seemingly does not depend on secret data. Note, stores in isolation don't form covert channels because they are not performed until they retire.

Yet, there may still be an implicit channel. For example, on a machine that performs memory dependence prediction [42], if the store address resolves after the load is issued, the load will squash based on whether `secret==rZ`, causing a similar pipeline disturbance as discussed above.[1] Likewise, if store-load forwarding is enabled, the load conditionally accesses the L1 cache depending on whether `secret==rZ`. Many additional hardware mechanisms, e.g., memory consistency speculation [19], value prediction [34], etc., create similar issues.

An important observation is that hardware optimizations like those above can be modeled as *implicit branches*, whereas explicit control-flow instructions like branches can be viewed as *explicit branches*. That is, the store bypass in Figure 5 (c) can be rewritten as "if (`secret` == rZ) { rY = rX; } else { load rY <- (rZ); }" where the "implicit branch" direction is predicted if `secret` has not yet resolved. In this sense, implicit branches may also leak at prediction and/or resolution time (Section 5.2), e.g., if the architecture uses a store set predictor [15].

Summary. To summarize, covert channels can be explicit or implicit, and implicit channels can be further broken down based on when they leak and their branch type. The next section uses these observations to block leakage through all channel types with a unified mechanism. For reference, Table 1 specifies channel types for existing attacks and a variety of hardware optimizations.

Table 1: Classifying existing attacks and covert channel-creating hardware structures. A channel's *Type* can be either Explicit (Exp) or Implicit (Imp), c.f. Section 5.1. An implicit channel's *Branch Type* is likewise Exp or Imp, c.f. Section 5.3. Attacks utilizing implicit channels may be either prediction- or resolution-time (Section 5.2), thus we leave that field out.

Channel	Spectre PoC?	Type	Branch Type
Cache timing [40, 59]	Spectre V1 [31]	Exp	-
Execution unit timing [6, 20]	-	Exp	-
SIMD utilization	NetSpectre [46]	Imp	Exp
Port contention [5]	SmotherSpectre [8]	Imp	Exp
Store-load forwarding	-	Imp	Imp
Mem. dep. prediction [42]	-	Imp	Imp
Mem. consist. speculation [19]	-	Imp	Imp
Value prediction [34]	-	Imp	Imp

6 SPECULATIVE TAINT TRACKING

Speculative Taint Tracking (STT) is a low-overhead framework that protects data accessed under misspeculation, such as data obtained by an out-of-bounds array access. We refer to such data, that a non-speculative execution would never read, as *secret*.

In a manner similar to dynamic information flow tracking (DIFT) [17, 49], STT "taints" secret data. The STT framework (Section 6.1) defines which data should be tainted, which instructions might leak it and thus should be protected, and when protection can be disabled. Section 6.2 describes how STT tracks the flow of tainted data between instructions and how—in contrast to conventional DIFT schemes—it automatically "untaints" data once the instruction that produces it becomes non-speculative. Based on taint information, STT applies novel protection mechanisms to block explicit covert channels (Section 6.3) and implicit covert channels (Sections 6.4–6.5).

6.1 Framework & Concepts

STT has three characteristics, which are set at design time.

Which data should be tainted? The microarchitecture classifies instructions capable of reading secrets under speculative execution as *access instructions*. We focus on the case where access instructions are loads, as this will be sufficient to block universal read gadget attacks (Section 4). STT taints the output of any speculative access instruction.

When can data be untainted? The microarchitecture specifies when a speculative access instruction is no longer considered a security threat, referred to as the instruction's *visibility point* [58]. The visibility point depends on the attack model. In the *Spectre* model, an instruction has reached the visibility point if all older control-flow instructions have resolved. In the *Futuristic* model, an instruction has only reached this point if it cannot be squashed. (The futuristic model protects data read by any possible hardware speculation, blocking additional attacks such as Meltdown.) Instructions reach the visibility point in fetch order. We call access instructions before and after the visibility point *unsafe* and *safe*, respectively, as instructions which have passed the visibility point are not speculative from a security perspective. STT untaints the output of an access instruction once it becomes safe.

Who can leak secrets? The microarchitecture classifies certain instructions as *transmit instructions*. (Note that an instruction can be neither, either, or both an access and a transmit instruction.) STT considers the execution of a transmit instruction as an explicit covert channel that leaks its argument (Section 5). Classifying only loads as transmitters will block memory system-related explicit covert channels. Classifying all instructions that have operand-dependent hardware resource usage as transmitters will block all explicit channels. We assume that stores trigger a cache coherence invalidation only on retirement, or else are defined as transmitters.

To block implicit channels, STT requires the microarchitect to classify *explicit branch* instructions, which affect control-flow, and to identify the *implicit branches* that represent additional sources of data-dependent resource usage, e.g., store-to-load forwarding, memory consistency speculation [19], etc. Section 6.4.2 discusses implicit branches in detail.

6.1.1 Identifying Access Instructions, Transmit Instructions and Implicit Branches. Here, we describe how microarchitects can identify access and transmit instructions, and implicit branch conditions.

[1] Note, this is not the already known Spectre Variant 4 (SSB) attack [25, 58]. In that attack, an *access instruction* reads stale data through a store bypass. Our attack is concerned with store bypass used as a covert channel.

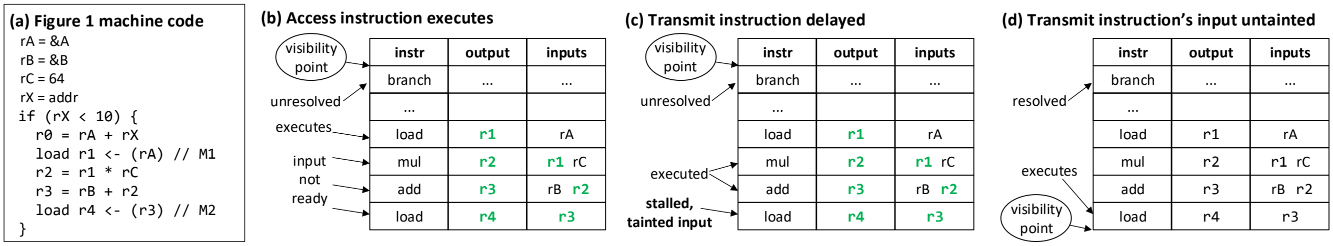

Figure 7: Snapshots of ROB state during the STT execution of the Spectre V1 code, in the Spectre threat model. (Tainted registers are green.)

An instruction should be an access instruction if it has the potential to read a secret. Except for loads, there are only a handful of such instructions (e.g., privileged/configuration register reads or I/O instructions), which can be identified manually.

An instruction should be a transmit instruction if its execution creates operand-dependent resource usage that can reveal the operand (partially or fully). Identifying implicit branches is similar: the architect must analyze whether the resource usage of some in-flight instruction changes as a function of *some other* instruction's operand. Examples of both transmitters and implicit branch-based channels are given in Table 1. This informal definition can be formalized by analyzing (offline) how information flows in each functional unit at the SRAM-bit and flip-flop levels to determine whether resource usage depend on the input value, in the style of the OISA [60] or GLIFT [52] formal frameworks. We leave such analysis to future work.

6.2 Taint and Untaint Propagation

STT tracks information flow from access instructions to younger in-flight instructions. STT *taints* the output register of an unsafe access instruction and propagates taint using standard taint tracking rules, namely that an instruction's output register is tainted if any of its input registers are tainted. Unlike conventional DIFT, STT *automatically* untaints data. When an access instruction becomes safe, its output register is *untainted.* Untaint information is also propagated, so that when all the data dependencies of an instruction become untainted, the instruction's output is untainted.

We defer the details of STT's taint/untaint tracking implementation to Section 7. At a high level, taint propagation is piggybacked on the existing register renaming logic in a modern out-of-order core. As an instruction enters the frontend and its registers are renamed, the instruction's output register is tainted if (1) it is an access instruction or (2) any of its input (physical) registers are tainted. Tainting is therefore fast. Propagating untaint is non-trivial, because dependency chains can be long and each instruction can have many data dependencies whose taint status needs to be tracked. STT addresses these challenges with a novel fast untaint algorithm in Section 7. In this section, we simply assume that taint/untaint information is available.

Unlike prior DIFT schemes [17, 49, 50, 60], STT does not require tracking taint in any part of the memory system (TLB, caches, or memory) or across store-to-load forwarding. The reason is that the taint of the output of a load—which is an access instruction—is determined *only* based on whether it has reached the visibility point: If a load is unsafe, its output is always tainted. If a load is safe, every instruction on which it depends has also reached its visibility point (since this happens in-order) and so the load's output is not tainted.

6.3 Blocking Explicit Channels

STT blocks explicit channels by delaying the execution of any transmit instruction whose operands are tainted until they become untainted.[2] This scheme imposes relatively low overhead because it only delays the execution of transmit instructions if they have tainted operands. For example, a load that only *reads* a (potential) secret but does not transmit one—such as load `M1` in Figure 1—executes without delay. Load `M2`, however, will be delayed and eventually squashed, thereby defeating the attack.

Figure 7 depicts this scenario in detail. Figure 7(a) shows a sequence of instructions executing the Spectre V1 code; load `M1` is an access instruction. In Figure 7(b), the access instruction has executed, and its output and all dependencies are tainted. Non-transmit dependent instructions can freely execute, but any transmit dependent instruction like `M2` is stalled (Figure 7(c)). If the speculation succeeds (i.e., `rX` < `10`), the branch resolves as correct and the access instruction becomes safe (assuming the Spectre threat model defined in Section 6.1). In this case, its output becomes untainted and the transmit instruction is allowed to execute (Figure 7(d)). Although in this example the transmit instruction becomes safe together with the access instruction, this is not true in general (e.g., if there is an unresolved branch between them). Thanks to STT's untaint mechanism, however, even an unsafe transmitter (i.e., that has not reached the visibility point) whose input becomes untainted can execute without having to delay until it reaches the visibility point or head of ROB.

In contrast, if the branch is mispredicted (i.e., `rX` ≥ `10`) the transmitter remains stalled until it is eventually squashed along with the access instructions it depends on.

Protection strategies. STT can apply different protection strategies to transmit instructions with tainted arguments. We chose to delay execution for simplicity, and also because this allows us to prove non-interference (Section 8). Yet, other protections are possible which create security-performance trade-offs. For example, one can combine STT with a scheme such as InvisiSpec [58], which would allow loads with tainted arguments to be executed earlier.

6.4 Eliminating Implicit Channels

STT blocks implicit channels by enforcing an invariant that the sequence of instructions fetched/executed/squashed never depends

[2] Notice that for loads, delaying execution implies delaying the TLB lookup.

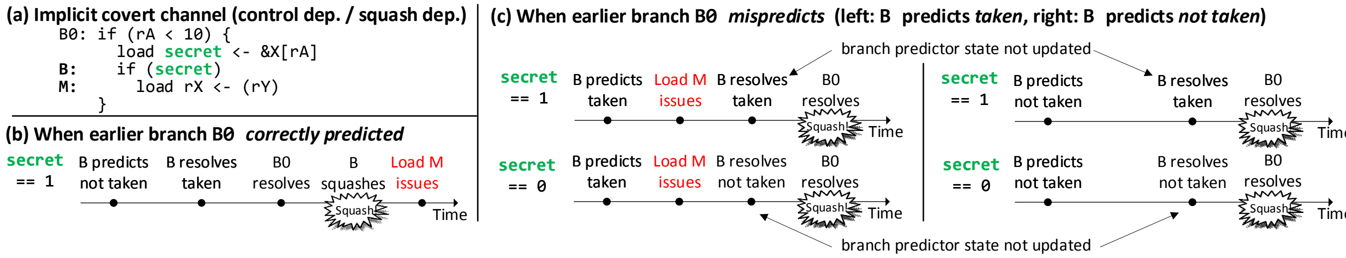

Figure 8: STT executing the code in (a), which includes an untainted branch `B0`, an access instruction reading `secret`, and an implicit channel.

on tainted data. That is, *STT makes the program counter independent of tainted data.*

A key challenge in enforcing this invariant is how to maintain efficiency. For example, a strawman DIFT approach to block implicit channels would be to consider the execution of *any* instruction following a branch with a tainted predicate (or *tainted branch*) as an implicit channel, and delay the execution of all such instructions. This approach would impose high overhead, as it requires delaying execution of all instructions following a tainted branch until the branch predicate becomes untainted. Even then, such an approach would not block implicit channels caused by implicit branches (Section 5.3), which are unique to the speculative execution setting.

To efficiently maintain the STT program counter invariant, we introduce two general principles to neutralize the sources of implicit channels identified in Section 5.2:

> **Prediction-based channels** are eliminated by preventing tainted data from affecting the state of any predictor structure.
>
> **Resolution-based channels** are eliminated by delaying the effects of branch resolution until the branch's predicate becomes untainted.

In the following, we discuss how STT applies these principles to eliminate implicit channels over explicit and implicit branches. Implementation details are presented in Section 7.

6.4.1 Explicit Branches. To prevent implicit channels through explicit branches (i.e., control flow instructions), STT modifies the baseline microarchitecture as follows.

Prediction-based channels. STT requires that every frontend predictor structure be updated based only on untainted data. This makes the execution path fetched by the frontend unaffected by the output of unsafe access instructions. STT passes a branch's resolution results to the direct/indirect branch predictors only after the branch's predicate and target address become untainted; if the branch gets squashed before this, the predictor will not be updated.

Figure 8(c) demonstrates the effect of STT on a speculative execution of the code snippet in Figure 8(a), in which the branch `B0` is mispredicted as taken. No matter how many experiments the attacker runs, the predicted direction of the branch `B` will not be a function of `secret`, because the branch predictor is not updated when `B` resolves. As a result, the execution path does not depend on `secret` (top vs. bottom)—it only depends on the predicted branch direction (left vs. right).

STT does not need to change how the *return address stack* (RAS) [27] (which predicts `RET` results) is updated; we discuss this issue shortly.

Resolution-based channels. STT delays squashing a branch that resolves as mispredicted until the branch's predicate becomes untainted. As a result, a transient branch with a tainted predicate (such as the branch `B` in Figure 8(c)) will never be squashed and re-executed, preventing the implicit channel leak shown in Figure 6. As Figure 8(c) shows, the transient branch `B` is eventually squashed once an older (mispredicted) branch with an untainted predicate squashes. Thus, the squash does not leak any information about the transient branch's resolution. Importantly, note that it is safe to resolve a branch *as soon as* its predicate becomes untainted, even if an *older branch with a tainted predicate* has not yet resolved.

To summarize, where the strawman DIFT approach would stall the execution of any instruction following a tainted branch, STT lets the instructions execute, and only increases the latency of *recovering* from a *tainted branch* misprediction. For example, in Figure 8(b), the load `M` does not execute immediately after the tainted branch `B` resolves, because `B`'s predicate is tainted at this point. This is in contrast to a modern processor, which squashes a mispredicted branch and starts executing the correct path immediately upon resolving the branch [1, 4, 12]. Fortunately, tainted branch mispredictions are only a small fraction of overall branch mispredictions (Section 9), which are infrequent in the first place because successful speculation requires accurate branch prediction.

Handling the RAS. With STT, the RAS is updated by the frontend as usual: as `CALL`s and `RET`s are fetched, they push and pop return addresses from the RAS. The reason is that STT makes the predicted execution path up to a `CALL`—and therefore the return address it pushes—independent of tainted data. Thus, `RET`s can safely pop from the RAS as usual, as the values they pop do not depend on tainted data.

STT does need to delay squashes due to RAS misprediction until the mispredicted `RET` reaches its visibility point, because the RAS misprediction resolution depends on the return address the `RET` reads from the stack. We assume that the baseline microarchitecture can repair the RAS after a squash to undo the effects of squashed `CALL`s [47]. We do not require a perfect RAS repair algorithm. Our only requirement is that the repair does not depend on tainted data. In particular, any repair algorithm whose input is only the RAS state [47] is fine.

6.4.2 Implicit Branches. Implicit branches frame microarchitectural mechanisms that observably change how the processor executes instructions as being caused by a branch "injected" into the execution (Section 5.3). Speculations—such as memory dependence speculation [42], value prediction [34], memory consistency speculation [19], etc.—are *predictions* of an implicit branch.

Formulating microarchitectural mechanisms as implicit branches allows STT to block leakage through them using a similar mechanism as was used to block prediction- and resolution-based channels for explicit branches. This section walks through this process for several common optimizations. While we cannot exhaustively discuss all known processor optimizations, the successful systematic application of STT's principles is evidence that they should generalize to other optimizations.

Implicit branch without prediction. Consider a store-to-load forwarding design without memory dependence speculation (e.g., [23]). In such a design, a load stalls until the addresses[3] of all older stores have resolved. As shown in Figure 9(a), store-to-load forwarding creates an implicit channel because the load `M2` accesses the cache only if it does not alias with the store S, i.e., if the secret `r2` is not 17. Figure 9(b) shows store-to-load forwarding framed as an implicit branch, denoted by impIf in the code.

This implicit branch only creates a resolution-based implicit channel, because the branch is not predicted. We eliminate this channel by delaying the resolution of the implicit branch until its predicate is untainted. In our example, this means that `M2` (and so all instructions data-dependent on it) are delayed until both `r2` and `r0` become untainted, which means they are delayed until the mispredicted explicit branch `B` squashes. (Section 6.5 describes an STT optimization that handles store-to-load forwarding more efficiently.)

In general, the store-to-load forwarding implicit branch predicate checks that *every* older in-flight store does not alias with the load, i.e., it is a conjunction of the "no alias" predicate for each older store. For example, Figure 9(b) should have:

$$\text{impIf}\ \ (\text{AND}_{\{S\,|\,\text{in-flight store S older than M2}\}}\ \text{S.addr}\ \neq \&\text{Y[r0]})\,.$$

Crucially, the implicit branch predicate never depends on prior explicit or implicit branches. This ensures that implicit branch predicates do not grow more complicated as more speculative instructions enter the ROB. Because of STT's invariant that instructions fetched/executed/squashed thus far are independent of tainted data, we need only guarantee that subsequent instructions do not create an implicit channel by (in this case) delaying the branch's resolution until its predicate is untainted. Implementation-wise, our STT microarchitecture (Section 7) efficiently tracks the taint of addresses in the load-store queue (LSQ), which allows resolving the implicit branch as part of the store-to-load forwarding logic.

Implicit branch with prediction. Consider now *memory dependence speculation*, where the processor might execute a load (read from memory) speculatively and squash it if an older store ends up aliasing with it. This is simply a *prediction* of the store-to-load forwarding implicit branch. We eliminate its predictor-based channel by requiring that the relevant predictor (e.g., a store set predictor [15]) be updated only by untainted data, i.e., only after the

[3] We refer to physical addresses simply as "addresses."

(a) Store-to-load forwarding:
```
   r0 = 17
B: if (r1 < size) {  // mispredict
M1:  load r2 <- &X[r1]    // access
S:   store r0 -> &Y[r2]
M2:  load r3 <- &Y[r0] // transmit
   }
```

(b) Store-to-load forwarding implicit branch:
```
   r0 = 17
B: if (r1 < size) {    // mispredict
M1:  load r2 <- &X[r1]     // access
S:   store r1 -> &Y[r2]
     implIf (r2 != r0)
M2:    load r3 <- &Y[r0] // transmit
     else
       r3 = r1
   }
```

Figure 9: Store-to-load forwarding implicit channel due to implicit branch.

implicit branch predicate becomes untainted. The prediction typically also depends on the LSQ state—for example, a prediction is made only if there are older stores with unresolved addresses. In this case, eliminating the predictor-based channel also requires delaying the prediction until the relevant LSQ state becomes untainted, e.g, until the addresses of older stores become untainted. We eliminate the resolution-based channel by delaying the squashing of the load on a misspeculation (misprediction of the implicit branch) until the branch's predicate becomes untainted.

Squashing implicit branches early-on. An advantage of implicit branches is that the microarchitecture knows the structure of their predicates. In some cases, this knowledge allows STT to untaint an implicit branch predicate early-on, based on the observation from GLIFT [52] that "the output of a logical function should only be untrusted if some untrusted input actually had an opportunity to affect the output."

For example, suppose that the memory dependence predictor predicts that the implicit branch in Figure 9(b) is taken, and that this turns out to be a misprediction. Naively, STT would need to delay squashing the implicit branch until the load's address and the addresses of *all* older stores become untainted, as the predicate is a function of them. However, the GLIFT observation implies that we need to delay the squash only until *one* term that evaluates to *false* becomes untainted—i.e., until the address of some older aliasing store and the load are untainted. At this point, the result of the AND becomes a function of only untainted data—the attacker only learns that an alias between untainted addresses exists.

Statically-predicted implicit branches. Several common forms of speculation can be formulated as implicit branches that are predicted statically, and therefore have no predictor-based channels. We only need to eliminate resolution-based channels by identifying the branch's predicate and, if the implicit branch is mispredicted, delay the resulting squash until the predicate becomes untainted. Below, we consider the example of memory consistency speculation in a multicore processor. Load-load ordering is another example with similar characteristics.

Memory consistency speculation. A memory consistency model (or memory model) specifies the order in which a processor's memory operations are performed and observed by other processors in the system [48]. Memory consistency speculation [19] allows the processor to maintain any required ordering between loads while still issuing them out of order. The idea is that if two loads, `M1` and `M2`, must appear to execute in program order (`M1` before `M2`), then `M2` can still execute before `M1` but will be squashed if the data loaded by M2 is invalidated by another processor or gets evicted from the cache before M2 retires.

Memory consistency speculation is thus a static prediction of *true* for an implicit branch predicate that the cache line a load accesses remains valid until the load retires. It turns out that with STT's delayed execution protection strategy, the resolution of this implicit branch predicate can only occur when it becomes untainted. Therefore, its resolution does not need to be delayed—i.e., consistency squashes can be performed when signalled, as usual.

The reason is that a consistency squash of load L can be signalled only after L accesses the cache, which implies that L's address is untainted. In addition, the memory access that triggers the line's invalidation/replacement is independent of tainted data. This holds because such a memory access occurs either because of a load/store instruction or a hardware prefetch. With STT, loads and stores access memory only if their address argument is untainted, which also implies that hardware prefetching state at the caches is a function of untainted data. Therefore, if the implicit memory consistency branch predicate evaluates to *false*, it is due to an untainted term.

6.5 Optimizing Store-to-Load Forwarding

We now describe an optimization that allows resolving the store-to-load forwarding implicit branch without waiting for its predicate to become untainted. The insight is, because store-load forwarding can only result in two observable outcomes (issue the load, or forward from a prior store) it is feasible to hide which occurs. Specifically: when the load address becomes untainted, we issue the load unconditionally. (That is, we do not wait for the prior store addresses to become untainted.) If forwarding should occur, we ignore the value read from memory and use the forwarded store value as the load's output. The load's output register is written only after the memory access completes, to guarantee that the timing of younger instructions is unchanged from the "no forwarding" case.

The optimization maintains STT's property that tainted data does not influence the execution path. The reason is that the resolution of this implicit branch only determines whether the load will access memory (and its output). It does *not* influence the execution path as long as the load is unsafe. This principle is general to other implicit branches whose resolution does not determine whether instructions retire/squash.

7 MICROARCHITECTURE

We now present a microarchitecture for STT. The key challenge in the implementation is how to implement the automatic untaint operation (Section 6.2). We present a unified mechanism that implements taint and untaint, and show how it can be used to block-/eliminate both explicit and implicit channels. In the following, instructions are labeled with monotonically increasing numbers, which we loosely refer to as their position in the ROB (younger instructions are assigned larger numbers).

7.1 Main Ideas

We make a key observation that helps implement untainting. Since instructions reach their visibility point in program order (Section 6.1), to untaint the arguments for an instruction i, it suffices to wait for the youngest access instruction that is causing the taint for i to reach the visibility point. We call this instruction the *Youngest Root of Taint* (YRoT) of i.

With this approach, we do not need to track exact def-use chains between instructions. Conceptually, we track the position of the YRoT for each instruction in the ROB. Then, we broadcast the ROB position of each access instruction as it reaches the visibility point. Each younger instruction whose YRoT is smaller or equal to the broadcasted value becomes untainted. If multiple access instructions reach the visibility point in the same cycle, we can broadcast the maximum index of all of them (instead of all of their indices). We note that logic indicating which instructions reach the visibility point is provided by prior work on InvisiSpec [58].

Untainting can trigger different operations. When a transmit instruction's arguments become untainted, it can execute (Section 6.3). When an explicit branch predicate is untainted, subsequent instructions are squashed if the branch was mispredicted (Section 6.4.1). Finally, when an implicit branch predicate is untainted, any observable effect related to the implicit branch, e.g., a squash due to memory dependence prediction (Section 6.4.2), can occur.

7.2 Hardware Changes to Frontend

We now describe an architecture that implements STT (Figure 10). Our architecture is meant to model a modern speculative out-of-order multicore with optimizations such as those described in Section 6.

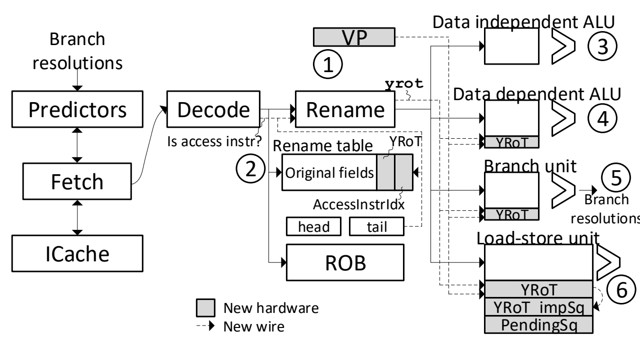

Figure 10: Microarchitecture with support for STT. Shaded block represent new hardware added to support our scheme; dashed lines are new wires. The per-instruction Youngest Root of Taint is denoted yrot, whereas fields added to hardware tables are denoted YRoT.

In the processor frontend (up to dispatch to execution units) the main hardware changes are to add logic to generate the Youngest Root of Taint (Section 7.1) for fetched instructions. We reuse logic from the InvisiSpec paper [58] paper to generate the visibility point (VP, shown as ①). For example, in the Spectre model the VP equals the ROB index of the oldest unresolved branch.

Tracking Youngest Root of Taint. We calculate the Youngest Root of Taint (YRoT) in the processor rename stage (Figure 10 ②). We add two new fields to the entries in the rename table (which maps logical registers to physical registers): YRoT and the *access instruction ROB index* (AccessInstrIdx), both of which require $\log_2$(ROBSize) bits. YRoT tracks the Youngest Root of Taint of the instruction that last produced each logical register in program order.

AccessInstrIdx records the ROB index of the last producer for each logical register, if that producer was an access instruction, or −1 otherwise.

Suppose a new instruction is being renamed. Then we calculate the Youngest Root of Taint for that instruction (denoted `yrot`) as the YRoT of the argument, in the rename table, which reaches the visibility point the latest. If an argument was directly produced by an access instruction, that instruction's index in the ROB directly becomes the YRoT for that argument (which is known through the AccessInstrIdx field). For example, suppose that this instruction has source registers Rs1 and Rs2. Let `RT` be short for rename table. Assume that when the source for a logical register Rx is in retired architectural state, `RT[Rx].YRoT = -1`. Then at rename time, we compute the instruction's `yrot` as:

```
yrot = max(
  ((RT[Rs1].AccessInstrIdx == -1) ?
    RT[Rs1].YRoT : RT[Rs1].AccessInstrIdx),
  ((RT[Rs2].AccessInstrIdx == -1) ?
    RT[Rs2].YRoT : RT[Rs2].AccessInstrIdx));
```

Recall, younger instructions have larger ROB indices, hence the use of max. If the instruction has a different number of arguments, the same max is taken over all of them.

If the instruction has a destination register, call it Rd, then we update the rename table as `RT[Rd].YRoT = yrot`. As previously mentioned, `RT[Rd].AccessInstrIdx` is set to the instruction's index in the ROB, if the instruction is an access instruction (determined in decode), or -1 otherwise. If the instruction does not update a register (e.g., a branch or store), the rename table is not updated because no logical register is being updated.

After rename, the instruction is dispatched to a reservation station (RS) based on its type. Depending on the instruction type, `yrot` may travel with the instruction and be stored in the RS (see below).

Importantly, this design assigns each instruction a `yrot` without adding new ports to the rename table or ROB. The YRoT and AccessInstrIdx fields for each entry are read along with the logical to physical register mappings that are read per-argument in the rename table already. As with the normal logical-to-physical register mappings, the YRoT and AccessInstrIdx in each entry needs to be restored after a squash.

7.3 Hardware Changes to Backend

At dispatch time, each instruction travels with its `yrot`. We now describe logic changes at the reservation stations (RS) for each instruction type, based on whether those instructions can form explicit and/or implicit channels (Section 5).

Our goal is design simplicity, and note that many optimizations are possible. While we cannot cover every proposed microarchitectural optimization, we cover an example for instructions that cause neither, both or one of explicit/implicit channels. The mechanisms can be generalized to other instructions. Recall, the microarchitecture is responsible for denoting each instruction as potentially creating explicit and/or implicit channels (Section 6.1).

Data-Independent Arithmetic. In the simplest case, the instruction performs a simple task that cannot create either an explicit or implicit covert channel (e.g., single-cycle add, xor without side effects), shown in Figure 10 ③. In this case, there are no changes to the RS and the `yrot` is dropped. Such instructions can execute as soon as their arguments are available, even if they are tainted. This is key for performance.

Data-Dependent Arithmetic. Instructions may be capable of creating only explicit channels, such as arithmetic with data-dependent timing (e.g., a multiplier [20]), see Figure 10 ④. If the microarchitecture classifies these as transmitters, the `yrot` of each instruction waiting in the RS is stored alongside the instruction in a new field called YRoT. When the VP changes, we calculate for each instruction i in the RS:

$$\text{instr } i \text{ can execute } = \texttt{yrot}_i < \text{VP}.$$

Our current design performs these checks in parallel, for each RS entry, in a fashion similar to instruction wakeup logic that checks if a dependency is ready.

Recall, `yrot` is based on each instruction's def-use chains, not where each instruction appears in program order. So, instruction wakeup due to `yrot` still allows instructions to execute out of order once their arguments become untainted.

Branches. Instructions may be capable of creating only implicit channels, such as conditional/unconditional branches and jumps, see Figure 10 ⑤. We handle these cases with the same mechanism as in the previous case for Data-Dependent Arithmetic: the `yrot` is stored alongside each branch in the branch RS (branch unit) and wakeup occurs by performing the same comparison between `yrot` and VP. This ensures that branch resolution only occurs when the branch's predicate and target (if any) is untainted. This design also avoids any modification to the branch predictors in the frontend, because only executions based on untainted data will update the predictors.

Loads and Stores. Finally, there are instructions which can create both explicit and implicit channels such as stores and loads, see Figure 10 ⑥. Discussed in Section 5.1, memory instructions are an important type of explicit channel. At the same time, loads and stores can also create implicit channels due to hardware features such as store-to-load forwarding, memory dependence speculation and memory consistency checks across cores (Section 6.4.2).

In isolation, a store creates neither explicit or implicit channels because we assume stores are performed at retirement. Yet, stores may alias with younger loads; we thus must store the `yrot` for each store, calculated in rename, along with the store, to calculate the predicate for implicit branches.

We block channels related to loads as follows. First, loads are stored in the LSQ with their `yrot`, as with previous cases, in a new YRoT field. Loads are also assigned two additional fields: PendingSquash (1 bit), and YRoT_impSquash (same width as the YRoT field). YRoT is used in the same way as before, to notify when the load address is untainted (can no longer form an explicit channel), at which point it is safe to perform the load.

When the load address becomes untainted, it may require store-load forwarding or memory dependence speculation. We handle store-load forwarding as discussed in Section 6.5: we unconditionally perform the load unless all prior stores are resolved and untainted. PendingSquash and YRoT_impSquash are used to handle memory dependence speculation. After the load is performed, if it suffers an alias to an earlier store whose address resolves late, we set YRoT_impSquash to the YRoT of the store causing the alias and set the PendingSquash bit (but do not perform the squash). If PendingSquash is set and YRoT_impSquash < VP, we perform the squash. (This checks requires analogous logic as comparing the normal `yrot` values to the VP.) This is equivalent to performing the squash when the implicit branch predicate becomes a function of untainted data (see Section 6.4.2). As explained in Section 6.4.2, a memory consistency violation simply squashes when it is signalled.

Multiple late resolving stores may alias with the load, resolving one after another. In this case, YRoT_impSquash is set to the min YRoT_impSquash of any store causing an alias. This is important for security. Once any memory violation occurs, it will eventually cause a squash, unless a squash is triggered beforehand by older instructions in the ROB. If the memory violation itself causes the squash, we must reveal that squash only when the implicit branch predicate is untainted. This moment is exactly when the min YRoT_impSquash of any alias reaches the visibility point. The logic for repeatedly updating the YRoT_impSquash in this fashion piggybacks off of the existing LSQ logic for detecting aliases.

8 SECURITY ANALYSIS

We formally prove that STT in the Spectre threat model enforces a novel notion of non-interference [38] appropriate for enforcing privacy of speculatively accessed data. (The formal proof for the futuristic model is ongoing work.) Our proof applies to a strong adversary that observes the instructions fetched, when and which functional units are busy (i.e., resource usage and port contention), and the target address of every cache/memory access. (See Section 3.) This section summarizes the key ideas and results; the details appear in a companion technical report [61].

We formally model processors as state machines. We define an STT machine that is a detailed model of a speculative out-of-order processor with STT. In addition to registers and memory, its *state* includes hardware structures such as branch predictors, the ROB, LSQ, etc. Its state also includes taint bits for registers. A *processor logic* defines how the state changes at every cycle. In each cycle, the processor logic performs *events* that modify the machine's state. These events model microarchitectural events such as instruction fetch, execution by a functional unit, squashes, retirement, tainting/untainting, etc. The STT machine models the protections described in Section 6 (e.g., delaying execution of tainted transmit instructions) and the fast untaint mechanism described in Section 7.

We show that the STT machine provides the following non-interference security guarantee: at each step of its execution, the value of a *doomed* register, that is, a register written to by a speculative access instruction that is bound to squash, does not influence future visible events in the execution. The key challenge is that when an instruction executes, we do not know whether it is going to squash or not. We address this by considering a simple in-order processor model, which we use to verify the STT machine's branch predictions against their true outcome, obtained from the in-order processor. Specifically, at each step of the STT machine's execution of a program, in our formal analysis we maintain an auxiliary bit of state, `mispredicted`, that is only set to `true` if the prediction of one of the preceding branches differs from the outcome of a corresponding branch in the in-order execution of the program.

With the way to identify whether the STT machine mispredicts at each point of the execution, we are able to distinguish doomed registers: a register r is doomed in a state σ with a given mispredicted flag, if it gets tainted in the shadow of what the in-order processor identifies as a misprediction, i.e., while `mispredicted = true`. For each register, we also maintain auxiliary state indicating whether it is doomed. We refer to the STT machine state coupled with its auxiliary state as an *extended* state. Given two extended STT states, κ_1 and κ_2, we say that $\kappa_1 \approx \kappa_2$ holds if κ_1 and κ_2 only differ by values of doomed registers.

We prove the following theorem, which states that values of doomed registers neither influence which events the STT machine executes at each cycle nor get leaked into the rest of the state by those events. We parameterize the theorem by an *observability function* view, which models the adversary's view [60], i.e., it projects event traces onto the parts the adversary can observe. The theorem holds in particular for the strong adversary described above, which observes the entire event trace.

Theorem 1. *At any cycle t, given two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds, if τ_1 and τ_2 are the sequences of events the STT processor logic performs at the cycle t from κ_1 and κ_2 respectively, then the following holds:*

(a) $\mathsf{view}(\tau_1) = \mathsf{view}(\tau_2)$ *holds, and*
(b) for the extended states κ_1' and κ_2' resulting from executing τ_1 and τ_2 respectively, $\kappa_1' \approx \kappa_2'$ holds.

The theorem proves that $\kappa_1 \approx \kappa_2$ is an invariant preserved by each cycle of the machine execution. Since this invariant is inductive, we obtain the following corollary: at any cycle t in any extended state κ, changes to doomed registers do not influence the future of the execution of the machine. In particular, they never influence the program counter's value, which is sufficient to eliminate traditional implicit covert channels.

9 EVALUATION

9.1 Experimental Setup

Simulator setup. We evaluate STT with the Gem5 [10] simulator, which models the performance implications of speculative instructions. We run SPEC CPU2006 [22] and PARSEC 3.0 [9] benchmarks, as representatives of both single-threaded and multi-threaded programs. For SPEC, we use the *reference* input size, and launch detailed simulation for 1 billion instructions after skipping the first 10 billion instructions. PARSEC benchmarks are all run with eight cores with the *simmedium* input size (except x264, whose input size is *simsmall*). A detailed architecture specification used for all schemes is shown in Table 2.

Microarchitectural features modeled in Gem5. In Gem5, instructions that create explicit channels are loads. The simulator also models the implicit channels discussed in Section 6, including direct/indirect branches, jumps, calls/returns, as well as implicit branches formed by store-load forwarding, memory dependence speculation (with a store set predictor), memory consistency checks and load-load ordering.

Configurations. We evaluate the following design variants, as shown in Table 3. We evaluate a baseline scheme DelayExecute

Table 2: Parameters of the simulated architecture.

Parameter	Value
Architecture	1 core (SPEC) or 8 cores (PARSEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

which conservatively delays execution for all access instructions until they reach the visibility point. This models the strawman, secure scheme from Section 1. Our main proposal DelayExecute+STT only applies this protection to tainted transmitters, and additionally eliminates implicit channels.

Table 3: Evaluated configurations.

Configuration	Description
Unsafe	An unmodified insecure Gem5 processor as baseline.
DelayExecute	Delay the execution of every transmit instruction until it reaches the visibility point.
DelayExecute +STT	STT implemented on top of DelayExecute, therefore only transmitters with tainted arguments are delayed.
DelayExecute +STT-ExpOnly	DelayExecute+STT without handling implicit channels. Thus this configuration has weaker security.

For each configuration, we evaluate the two visibility points from [58], namely Spectre and Futuristic, together with both Total Store Ordering (TSO) and Release Consistency (RC) memory consistency models.

Penetration testing. Prior to performance modeling, we evaluated whether our framework blocked Spectre variants, such as Spectre V1 (Figure 1) and confirmed the attack was blocked.

9.2 Main Performance Result

Figures 11 and 12 compare the execution time of configurations Unsafe, DelayExecute and DelayExecute+STT on the single-threaded SPEC and multi-threaded PARSEC applications, respectively. The goal is to show the performance overhead of DelayExecute+STT (our complete proposal) relative to Unsafe and performance improvement relative to the naive but secure DelayExecute scheme. All individual benchmark results use the TSO model, and execution times are normalized to Unsafe for each memory model.

SPEC analysis. With the Spectre threat model, STT improves overhead on average relative to naive DelayExecute from over 40% without STT to 8.5% with STT, in TSO. RC results are similar. The main reason is that only a small portion of all speculative loads (transmitters) are tainted due to older speculative loads (access instructions).

The savings is even more pronounced using the Futuristic model, where overhead drops from around 3× to 14.5%. This makes sense because Futuristic is a more restrictive model that forces longer delays before loads can execute. The fact that Futuristic overhead is close to Spectre overhead (14.5% to 8.5%) is an important result. Futuristic was designed in [58] to be a holistic threat model, taking into account all possible reasons for an instruction to be speculative. The small difference between the two models suggests that supporting comprehensive security definitions is viable with STT without sacrificing much performance.

Table 4: The effect of delaying predictor updates and resolution (for explicit and implicit branches). All numbers assume TSO + Futuristic.

	Benchmark Suite	SPEC2006		PARSEC	
	Protection Mechanism	Unsafe	DelayExecute+STT	Unsafe	DelayExecute+STT
1	# explicit br. misp. / # explicit branches	8.81%	9.04%	3.62%	3.85%
2	# tainted explicit br. misp. / # explicit br. misp.	N/A	8.87%	N/A	28.81%
3	# implicit br. misp. / # implicit branches	0.008%	0.01%	0.022%	0.018%
4	# tainted implicit br. misp. / # implicit br. misp.	N/A	15.5%	N/A	7.74%

PARSEC analysis. The multi-threaded PARSEC workloads in Figure 12 exhibit the same trends seen in the SPEC workloads. For the Spectre model, DelayExecute+STT reduces the overhead of DelayExecute from 78% to 24% in TSO model, and from 103% to 30% in RC. For the Futuristic model, DelayExecute+STT lowers the overhead (over 3x in both TSO and RC) of DelayExecute to 27% and 36% for TSO and RC, which are close to the weaker Spectre model.

Overhead from implicit branch protection. A central component in STT is mechanisms that eliminate implicit channels by delaying predictor updates and explicit/implicit branch resolutions until branch predicates are untainted. Figure 11 and 12 include results for DelayExecute+STT-ExpOnly, which shows performance for a weaker security guarantee that ignores implicit channels. For SPEC workloads, ignoring implicit branches reduces overhead on average relative to DelayExecute+STT by around 1%, for all memory models and visibility points. The number increases to 3% for PARSEC workloads. The takeaway is that protection against implicit branches using our mechanisms is very cheap.

For more insight into implicit channel overhead, Table 4 shows the explicit and implicit branch misprediction rates for both the Unsafe baseline and DelayExecute+STT, under TSO in the Futuristic model. We see that, delaying branch predictor updates only increases explicit branch misprediction rate by 0.2% relative to the unsafe baseline, and the rate of memory violations (implicit branch misprediction) is close for all schemes. Second, the percentage of branch mispredictions where STT would delay resolution—and thus may incur performance overhead—is small. These are the explicit and implicit branch mispredictions that occur when the branch is tainted. For example, with DelayExecute+STT on SPEC, only 0.8% of all dynamic branches are both tainted and mispredicted (8.87% of the 9.04% mispredicted branches). The situation is even more apparent for memory violations, as they are very rare (< 0.1%, regardless of whether the implicit branch predicate is tainted). Results for PARSEC are similar.

9.3 InvisiSpec vs. STT

InvisiSpec [58] is a prior hardware mechanism for blocking speculative execution attacks. The two schemes have different security trade-offs: On one hand, InvisiSpec only blocks covert channels through the cache hierarchy, whereas STT can block any covert channel. On the other hand, STT does not prevent leaking secrets which are part of retired state (Section 4) whereas InvisiSpec does

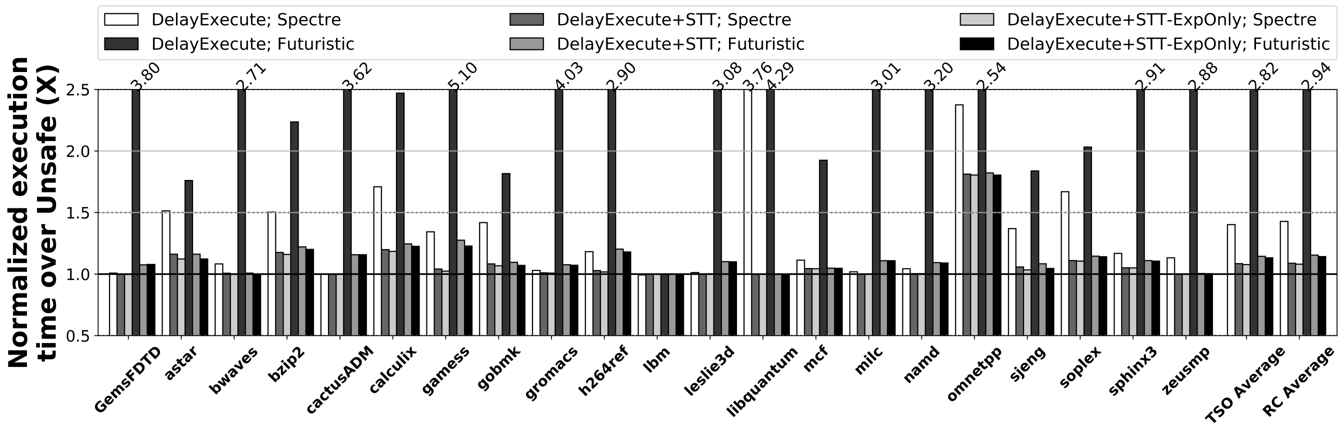

Figure 11: Execution time (normalized to Unsafe) of SPEC benchmarks with the schemes DelayExecute, DelayExecute+STT, DelayExecute+STT-ExpOnly, in two visibility points (Spectre and Futuristic). All per-benchmark results assume TSO; averages for TSO and RC are given on the right.

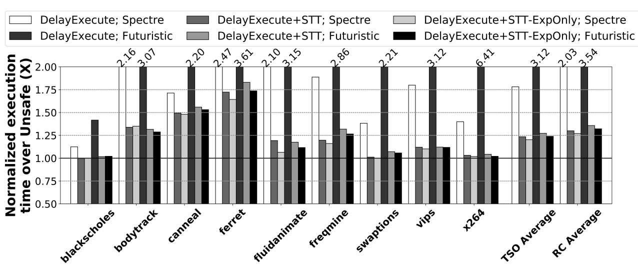

Figure 12: Execution time (normalized to the Unsafe baseline) of PARSEC benchmarks. Results shown and methodology follow Figure 11.

handle this case. As we mentioned, STT is sufficient to prevent the universal read gadget, which is the most dangerous class of attacks.

We compare overhead of InvisiSpec and STT running SPEC2006, using the Spectre and Futuristic threat models.[4] We find InvisiSpec and STT have 7.6% and 8.5% overhead relative to Unsafe for the Spectre model, respectively; and 18.2% and 14.5% overhead in the Futuristic model, respectively.

10 RELATED WORK

STT builds on the rich literature of hardware DIFT [14, 16–18, 41, 49, 51, 52, 55, 60]. Most DIFT works consider threat models that occur outside of speculative execution and do not involve covert channels (we note exceptions below). Our conceptual contribution is in exposing new types of implicit flow in the speculative execution attack setting (Section 5).

Several hardware defenses work for speculative execution attacks: InvisiSpec [58], SafeSpec [28], and DAWG [29] only block covert channels through the cache hierarchy. Conditional Speculation [33] and Selective Delay [44] additionally block covert channels through the memory system (e.g., DRAM contention). In contrast, STT can block all covert channels (Section 6).

Oblivious ISA Extensions (OISA) [60] and GLIFT [52] can achieve non-interference on speculative architectures (similar to Section 8), but impose restrictive, potentially low performance programming models such as data oblivious programming. Context Sensitive Fencing [50], OISA, and ConTExT [45] require taint tracking through memory, while STT does not require taint tracking for retired state or the memory system. Context Sensitive Fencing briefly mentions implicit flow. They do not mention the new, different ways branches can leak (Section 5.2) or implicit branches (Section 5.3). Their scheme—tainting the PC—is relatively high overhead (11.8% for explicit branches), relative to our 1-3% overhead for both explicit and implicit branches.

NDA [56] and SpecShield [7], which are concurrent to our work, share a similar high-level strategy as STT: to restrict propagation of potential secrets to covert channels. However, neither NDA nor SpecShield provide an abstraction to classify all covert channels, identify implicit branch-based implicit channels, distinguish between prediction- and resolution-time leakage for implicit channels, or propose an untaint mechanism which is as aggressive as STT’s. For example, SpecShield delays forwarding tainted data to covert channels until the youngest instruction dependency (as opposed to the youngest access instruction; Section 7.1) producing that data reaches the visibility point. As a result, both NDA and SpecShield propose multiple design variants, each with security-performance trade-offs. By contrast, STT proposes a single design that is both high performance and high security.

11 CONCLUSION

This paper proposes Speculative Taint Tracking (STT), a novel protection framework to comprehensively protect speculatively accessed data from speculative execution attacks. STT has two key novelties: a framework to eliminate both explicit and implicit covert channels, and a new taint/untaint procedure capable of waking instructions up early. Together, these mechanisms enable a high-performance and high-security design, able to enforce strong non-interference properties with respect to speculatively accessed data.

[4] InvisiSpec is evaluated using a recent code update that models the design accurately [2].

ACKNOWLEDGMENTS

This work was funded in part by NSF under grant CNS-1816226, Blavatnik ICRC at TAU, ISF under grant 2005/17, and by an Intel Strategic Research Alliance (ISRA) grant. We thank Joel Emer, Sarita Adve and Shubu Mukherjee for very helpful discussions, and thank the anonymous reviewers for their feedback and insights during the review process.

REFERENCES

[1] 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[2] 2019. InvisiSpec-1.0 simulator bug fix. https://github.com/mjyan0720/InvisiSpec-1.0/commit/f29164ba510b92397a26d8958fd87c0a2b636b0c.

[3] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. 2006. Predicting Secret Keys via Branch Prediction. *IACR* (2006).

[4] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *MICRO'03*.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida Garcia, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. *IACR* (2018).

[6] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *S&P'15*.

[7] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *PACT'19*.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. *arXiv* (2019).

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT'08*.

[10] 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.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security'19*.

[12] Christopher Celio, David A. Patterson, and Krste Asanovic. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report UCB/EECS-2015-167. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. Sgxpectre Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv* (2018).

[14] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *ISCA'08*.

[15] G. Z. Chrysos and J. S. Emer. 1998. Memory dependence prediction using store sets. In *ISCA'98*.

[16] J. R. Crandall and F. T. Chong. 2004. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO'04*.

[17] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *ISCA'07*.

[18] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *MICRO'10*.

[19] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP'91*.

[20] Johann Grossschadl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. *ICISC'09*.

[21] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[22] John L Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 4 (2006), 1–17.

[23] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. 2001. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal* 5 (2001).

[24] Jann Horn. 2018. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[25] Intel. 2018. Q2 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html.

[26] Mike Johnson. 1991. *Superscalar Microprocessor Design.* Prentice Hall Englewood Cliffs, New Jersey.

[27] David R. Kaeli and Philip G. Emma. 1991. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. In *ISCA'91*.

[28] 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*.

[29] 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*.

[30] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv* (2018).

[31] 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 *S&P'19*.

[32] 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*.

[33] 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*.

[34] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *ASPLOS'96*.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security'18*.

[36] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS'18*.

[37] 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* (2019).

[38] John Mclean. 1994. Security Models. In *Encyclopedia of Software Engineering*. Wiley & Sons.

[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA'06*.

[40] Colin Percival. 2005. Cache missing for fun and profit. In *Proc. of BSDCan 2005*.

[41] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO'06*.

[42] Glenn Reinman and Brad Calder. 1998. Predictive Techniques for Aggressive Load Speculation. In *MICRO'98*.

[43] A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[44] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Sjalander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *ISCA'19*.

[45] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. 2019. ConTExT: Leakage-Free Transient Execution. *arXiv* (2019).

[46] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS'19*.

[47] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. 1998. Improving Prediction for Procedure Returns with Return-address-stack Repair Mechanisms. In *MICRO'98*.

[48] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Pub.

[49] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS'04*.

[50] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing : Securing Speculative Execution via Microcode Customization. In *ASPLOS'19*.

[51] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. 2009. Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference. In *MICRO'09*.

[52] 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*.

[53] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2008. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security'18*.

[55] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. 2008. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA'08*.

[56] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO'19*.

[57] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).

[58] 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*.

[59] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security'14*.

[60] 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*.

[61] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher 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.