DOI:10.1145/3491201

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

By Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher

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. This paper proposes speculative taint tracking (STT), a high security and high performance hardware mechanism to block these attacks. The main idea is that it is safe to execute and selectively forward the results of speculative instructions that read secrets, as long as we can prove that the forwarded results do not reach potential covert channels. The technical core of the paper is a new abstraction to help identify all microarchitectural covert channels, and an architecture to quickly identify when a covert channel is no longer a threat. We further conduct a detailed formal analysis on the scheme in a companion document. When evaluated on SPEC06 workloads, STT incurs 8.5% or 14.5% performance overhead relative to an insecure machine.

1. INTRODUCTION

Speculative execution attacks such as Spectre[15] have opened a new chapter in hardware security. In these attacks, malicious speculative execution causes doomed-to-squash instructions to *access* and later *transmit* secrets over *microarchitectural covert channels* such as the processor cache.[26]

Consider "Spectre V1" (Figure 1) as an example. On modern processors, branch directions are predicted early in the processor pipeline to enable subsequent instructions to be fetched before the branch's predicate resolves. In a speculative execution attack, the attacker "mistrains" the branch predictor to predict "taken" even if the branch predicate eventually resolves to "not taken." This means that in between branch prediction and resolution, the program speculatively executes down the taken (incorrect) path: accessing a value `secret` potentially outside the bounds of `array1` and passing that value as the address to a second load reading `array2`. For the remainder of the paper, we will consider such speculatively accessed data to be secret.

In the context of Figure 1, the second load forms a microarchitectural covert channel. Specifically, on modern processors, loads result in address-dependent (and by extension `secret`-dependent) hardware resource usage due to the presence of hardware structures such as cache. Thus, an attacker that can monitor the load's hardware resource usage, or the program's execution time, can use that information to infer `secret`.

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 array2 indexed by `secret` is loaded into the cache (Line 3) even though both loads are eventually squashed.

```
1  if (off < array1_size) {          // mispredicts
2    secret = array1[off];            // secret accessed
3    y = array2[64 * secret ]; }      // secret transmitted
```

Making matters worse, an attacker that can freely control `off` can repeat the attack with different `off` to leak different secret values in the victim's memory. Further, although the above example covered Spectre V1, there are many other ways to leak secret data using similar principles. For example, by accessing secret information through other types of processor misspeculation, or by exfiltrating those secrets through other microarchitectural covert channels.

1.1. This paper's defense approach

A secure, but conservative, way to block *all* speculative execution attacks—regardless of source of misspeculation or choice of microarchitectural covert channel—is to delay executing all instructions that can access a secret until such instructions become nonspeculative. In nearly all attacks today, this would imply blocking all loads until they are nonspeculative, which would be tantamount to disabling speculative execution.

This paper proposes a principled, high-performance mechanism that achieves the same security guarantee as the above conservative scheme. The key idea is that *speculative execution is safe unless speculatively accessed data (secrets) reaches a covert channel*. In many cases, speculative instructions either do not have access to secrets or do not form covert channels, and so can execute freely under speculation. For example, the first load in Spectre V1 (Figure 1) forms a covert channel, but that channel only leaks the attacker-selected address `&array1[off]`—not the secret data stored at that address. Thus, this load's execution need not be protected. Likewise, many instructions (e.g., simple arithmetic) do not form covert channels even if their operands are secret

The original version of this paper was published in the *Proceedings of the 52nd International Symposium on Microarchitecture* (October 2019).

values. It is only when the secret is passed to a covert channel (e.g., the second load in Figure 1) that protection must be applied.

To implement this idea, we present speculative taint tracking (STT), a framework that tracks the flow of speculatively accessed data through in-flight instructions (similar to dynamic information flow tracking/DIFT[21]) until it is about to reach an instruction that may form a covert channel. STT then delays the forwarding of the data until it becomes a function of nonspeculative state or the execution squashes due to misspeculation. To be secure and efficient, we address two key challenges.

- **Identifying what is a covert channel.** First, we develop an abstraction that indicates how and when instructions can form covert channels, so as to stall data forwarding only when it becomes unsafe.
- **Identifying what is a secret.** Second, we develop a microarchitecture that determines the earliest time when data should no longer be considered secret, so as to re-enable data forwarding as soon as it becomes safe.

We now describe these two components in more detail.

1.2. New abstractions for describing microarchitectural covert channels

Covert channels come in different shapes and sizes. For example, attackers can monitor how loads interact with the cache,[15] the timing of SIMD units,[20] execution pipeline port contention,[4] branch predictor state,[1] and more. To comprehensively block information leakage through these different channels, it is necessary to understand their common characteristics.

To address this challenge, the paper proposes a new abstraction through which the covert channels on speculative microarchitectures can be viewed, discovers new points where instructions can create covert channels, and discovers a new class of covert channels. We find that all covert channels are one of two flavors, which we call explicit and implicit channels (related to explicit and implicit information flow,[19,22] respectively). 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,[15] as in Line 3 of Figure 1. In an *implicit channel*, data indirectly influences how (or whether) an instruction(s) execute, and these changes in resource usage reveal the data. For example, the instructions executed after a branch reveal the branch predicate.[4, 20] The paper further defines subclasses of implicit channel, based on when the leakage occurs and based on the nature of the secret-dependent condition that forms the channel.

Key advance: safe prediction. Through its investigation of implicit channels, the paper makes a key advance by showing *how to use hardware predictors safely*. Spectre attacks were born from attackers mistraining predictors to leak secrets. Through its abstraction for implicit channels, *STT enforces a policy that prevents arbitrary predictor mistraining from leaking any secret data over any microarchitectural covert channel*. The paper shows how this enables existing predictors to stay enabled without leaking privacy, dramatically improving performance. In the future, we expect the idea of safe prediction to enable further innovation, that is, by enabling the design of new predictors without fear of opening new security holes. Indeed, our follow-on work uses this idea to safely improve the performance of instructions that create explicit channels.[28]

1.3. Mechanisms to quickly and safely disable protection

Once we have mechanisms to block secret data from reaching covert channels, the next question is when and how to disable that protection, if speculation turns out to be correct. This is crucial for performance, as delaying data forwarding longer than necessary increases the chance that later instructions are, themselves, delayed.

STT tackles this problem with a safe but aggressive approach, *by re-enabling data forwarding as soon as data becomes a function of nonspeculative state*. For example, in Figure 1, this corresponds to the moment when the branch predicate resolves. This represents the earliest safe point but is nontrivial to determine in hardware, in general. For example, a delayed instruction's operand(s) may be the result of a complex dependency chain across many control flow and speculative operations. Intuitively, determining that data is a function of nonspeculative state would require retracing a backward slice of the program's execution, which is costly to do quickly.

Despite the above challenges, STT proposes a simple hardware mechanism that can disable protection/re-enable forwarding for an arbitrary instruction in a single cycle, using hardware similar to traditional instruction wake-up logic. The key idea is that to determine whether data is a function of nonspeculative state, it is sufficient to determine whether the *youngest* load, whose return value influences the data, has become nonspeculative. Checking this condition is akin to tracking a single extra dependency for each instruction, as opposed to performing complex backward slice tracking.

1.4. Security guarantees and formal analysis

Alongside the main paper, we formally prove that STT enforces a novel form of noninterference[9] with respect to speculatively accessed data. In a nutshell, we show that, with STT, hardware resource usage patterns over time are independent of data that eventually squashes. We released a companion technical report[29] with detailed formal analysis and a security proof for this property on a processor model implementing STT.

1.5. PUTTING IT ALL TOGETHER

Putting everything together, STT provides both high security and high performance. It does not require partitioning or flushing microarchitectural resources, and does not require changes to the cache/memory subsystem or the software stack. When evaluated on SPEC06 workloads, STT incurs 8.5% or 14.5% performance overhead (depending on the threat model) relative to an insecure machine.

2. BACKGROUND

We now provide additional details about processor microarchitecture. Also see Section 1 for basics on Spectre attacks.

Out-of-order execution. Dynamically scheduled processors execute instructions in parallel and out of program order to improve performance.[11, 23] Instructions are fetched and decoded 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 queuing instructions in a hardware structure called the reorder buffer (ROB)[13] 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, that is, if I_1 precedes I_2 in fetch order, then I_1 is *older* than I_2.

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 instructions younger than the point of misspeculation also get squashed.)

There are multiple types of speculation in modern processors, associated with different instructions and events. For example, to enable immediate fetching of instructions after a branch, that is, before the branch's predicate resolves, modern processors employ branch prediction. Branch predictors are (typically) stateful structures in the processor frontend that predict the direction of the branch based on information such as the branch's program counter and whether the branch historically has been taken/not taken. If the processor backend later resolves the branch predicate and determines the prediction to be incorrect, all subsequently fetched instructions are squashed and control flow is diverted to the correct path.

3. ATTACKER MODEL AND PROTECTION SCOPE

Attacker model. STT assumes a powerful attacker that can monitor any microarchitectural covert channel from anywhere in the system and induce arbitrary speculative execution to access secrets and create covert channels. For example, the attacker can monitor covert channels through the cache/memory system,[15] data-dependent arithmetic,[10] port contention,[4] branch predictors,[1] etc.

We note that the above attacker is very strong, perhaps even unrealistic. The goal is that through defending against such an attacker, we will by extension defend against weaker, more realistic attackers.

Scope: protecting speculatively accessed data. A speculative execution attack consists of two components.[14, 20] First, an instruction that reads a potential secret into a register, making it accessible to younger instructions. We call this instruction the *access instruction*.[14] Second, a younger instruction or instructions that exfiltrate the secret over a microarchitectural covert channel. The access instruction is almost always a load,[15, 24] but some attacks use a privileged register read.[5]

We distinguish attacks based on whether the access instruction is doomed-to-squash (*transient*) or bound to retire (*nontransient*). STT's goal is to block attacks involving doomed-to-squash access instructions, as shown in Figure 2. These attacks can access data that a correct (not misspeculated) execution would never access, which often results in being able to read from any location in memory. Attacks involving bound-to-retire access instructions are out of scope. They can only leak *retired (or bound-to-retire) register file state*, not arbitrary memory, and their leakage can be reasoned about by programmers or compilers and blocked using complementary techniques (e.g., Data-oblivious ISAs[27]).

4. ABSTRACTION FOR COVERT CHANNELS

STT proposes a novel abstraction for covert channels (Figure 3). In our abstraction, covert channels are broken into two classes: *explicit* and *implicit* channels. An *explicit channel*, related to explicit flow in information flow,[19, 22] 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 example is a load instruction's changes to the cache state. An *implicit channel*, related to implicit flow,[19, 22] is one where data *indirectly* influences how (or whether) an instruction or several instructions execute, and these changes in resource usage reveal the data. An example is a branch instruction, whose outcome determines subsequent instructions and thus whether some functional unit is used.

We further find new ways that implicit channels can leak, and find entirely new classes of implicit channels. Figure 4 gives examples of "traditional" (Figure 4(a)) and new (Figure 4(b) and (c)) channels. We denote the value being revealed through the channel as secret. The examples assume the attacker can monitor the cache-based covert channel, that is, the program's memory access pattern. We note that in many cases (e.g., Figure 4(a) and (b)), the load can be replaced by any instruction; in particular, not necessarily

Figure 2. STT's scope is to protect speculatively accessed data from leaking over any microarchitectural covert channel. Protecting values that are accessed nonspeculatively is outside of scope.

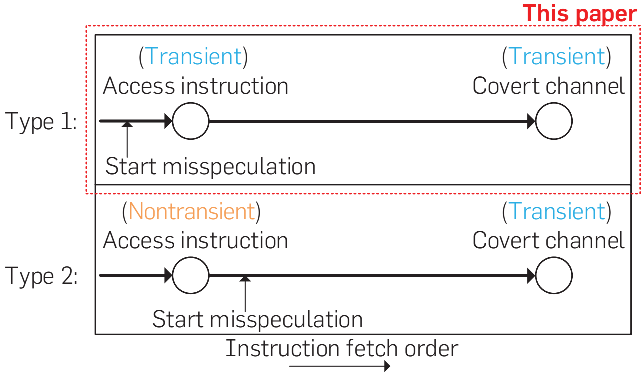


Figure 3. STT's new classification schema for microarchitectural covert channels.

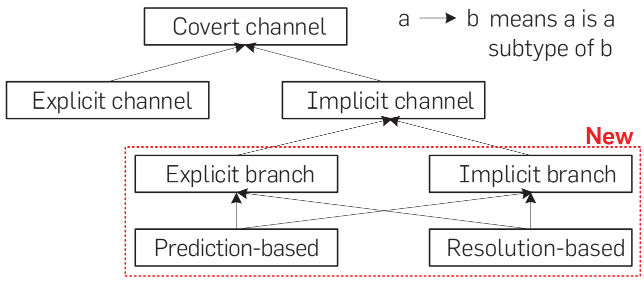


Figure 4. Examples of implicit covert channels revealing `secret`. Assume an older speculative access instruction has already read `secret` into a register, for example, Line 2 in Figure 1. The attacker can see the sequence of load addresses sent to the memory system. For stores, we assume address translation and other address-dependent actions occur when the store retires. `rX`, `rY`, and `rZ` are registers. Each of these covert channels can be "plugged into" existing attacks as the "Covert channel" in Figure 1. For example, we can replace Line 3 with one of (a)–(c) above.

(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)`

one that forms an explicit channel. Case in point, secret is not passed directly as the load address in any of the examples, yet still leaks.

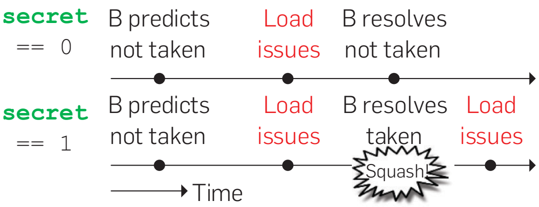
4.1. Prediction- versus resolution-based leakage

First, we find that implicit channels can leak 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 attacker'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 were 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 pipeline squashes. See the code snippet in Figure 4(b), whose timing is shown as a function of the secret in Figure 5. If the attacker knows the branch will predict not taken (e.g., by priming it beforehand[15]), a squash means the branch was actually taken. The attacker can observe the squash through different effects, for example, program timing or the fact that the load issues twice.

Figure 5. Resolution-based implicit channel for Figure 4(b) due to secret-dependent pipeline squashes. When the branch (B) resolves, it leaks the secret based on whether a squash occurs, as this causes the younger load to execute once or twice. There is an analogous case when the branch is predicted taken.



4.2. Explicit versus implicit branches

Second, we find that implicit channels can feature either an *explicit* or an *implicit* branch. For example, in Figure 4(c), there is no explicit control-flow instruction and the load address seemingly does not depend on secret data.

Yet, there may still be an implicit channel. For example, consider a machine that performs store-to-load forwarding. With this optimization, the processor can forward data (**`rX`**) directly from the older in-flight store to the younger load's output register (**`rY`**), as opposed to waiting for the store to retire and accessing the cache, if the store/load addresses alias, that is, if secret==**rZ**. Store-to-load forwarding thus creates an implicit channel, as whether a cache access is performed depends on the secret.

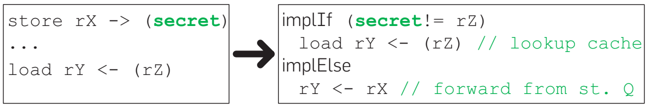
Another common technique with similar implications is memory-dependence speculation.[18] This optimization allows a load to (speculatively) read from cache even if older in-flight stores have unresolved addresses, that is, it speculates that store-to-load forwarding will not be needed. In our example, if the older store address later resolves and we have that secret==**rZ**, the load and younger instructions will squash, causing a similar pipeline disturbance as discussed in Section 4.1. (Note, this is not the already known Spectre Variant 4 (SSB) attack.[12, 25] 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.)

An important observation is that hardware optimizations such as those above can be modeled as *implicit branches*, whereas explicit control-flow instructions such as branches can be viewed as *explicit branches*. That is, the store-load pair in Figure 4(c) can be rewritten as shown in Figure 6, 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 4.1). For example, memory-dependence speculation is sometimes implemented with a stateful predictor called a store set predictor,[6] which tries to guess when store-load pairs will address alias, which can similarly "learn" functions of secret data.

4.3. Insights from analysis of implicit channels

Since it was proposed in the paper, the classification for

Figure 6. Rewriting a store-load pair as an implicit branch. implIf reveals a potential covert channel as a function of memory aliasing to the older store. This occurs if the microarchitecture supports store-to-load forwarding or memory-dependence speculation.



```
store rX -> (secret)
...
load rY <- (rZ)
```

```
implIf (secret!= rZ)
  load rY <- (rZ) // lookup cache
implElse
  rY <- rX // forward from st. Q
```

implicit channels has proven to be a robust and useful way to represent and pinpoint the root cause of microarchitectural attack vulnerabilities. For example, in the NetSpectre attack,[20] a secret branch predicate conditionally causes a SIMD instruction to be issued, which triggers a SIMD unit power-on event. A common misconception is that the attack root cause is SIMD unit power-on time. STT's abstraction shows, however, that the root cause is an explicit branch and that "fixing" the SIMD unit does not prevent the attack.

Even more subtly, the abstraction demonstrates and provides cases where implicit flow and privacy leakage *do occur* despite not occurring according to program semantics. For example, at the software level, neither Figure 4(b) nor (c) would be flagged as creating covert channels. Figure 4(b) would not be considered a channel because the load is control-and data-independent of the branch. Likewise, Figure 4(c) would not be considered a channel because, although there is possible information flow from **`rX`** to **`rY`** due to address aliasing, this information flow does not (seemingly) impact the memory access pattern. Generally speaking, the analysis shows that in advanced processors, subtle microarchitectural decisions that are orthogonal to program semantics must be taken into account to reason about possible microarchitectural covert channels.

Finally, the abstraction applies to a large set of microarchitectural optimizations. For example, the representation of store-to-load forwarding and memory-dependence speculation (Figure 6) also captures the behavior of memory consistency speculation,[8] value prediction,[16] and other optimizations. For reference, Table 1 specifies the channel types for existing attacks and a variety of hardware optimizations. As we will see in the next sections, being able to represent different optimizations as *predictions on implicit branches* will enable STT to apply a uniform mechanism to block leakage through a variety of structures (e.g., branch, store set, etc., predictors).

Table 1. Classifying existing attacks and covert channel-creating hardware structures.

Channel	Spectre PoC?	Type	Branch type
Cache timing[17, 26]	Spectre V1[15]	Exp	–
Execution unit timing[3, 10]	–	Exp	–
SIMD utilization	NetSpectre[20]	Imp	Exp
Port contention[2]	SmotherSpectre[4]	Imp	Exp
Store-load forwarding	–	Imp	Imp
Mem. dep. prediction[18]	–	Imp	Imp
Mem. consist. speculation[8]	–	Imp	Imp
Value prediction[16]	–	Imp	Imp

A channel's *Type* can be either Explicit (Exp) or Implicit (Imp), c.f. Section 4. An implicit channel's *Branch Type* is likewise Exp or Imp, c.f. Section 4.2. Attacks utilizing implicit channels may be either prediction- or resolution-time (Section 4.1); thus, we leave that field out.

5. STT: DESIGN

STT "taints" secret (speculatively accessed) data as it flows through the pipeline in a manner similar to dynamic information flow tracking (DIFT).[7, 21] The STT framework (Section 5.1) defines which data should be tainted, which instructions might leak it and thus should be protected, and when protection can be disabled. STT tracks the flow of tainted data between instructions in the ROB and automatically "untaints" data once the instruction that produces it becomes nonspeculative (Section 5.2), in contrast to conventional DIFT schemes. Based on taint information, STT applies novel protection mechanisms to block both explicit and implicit covert channels (Section 5.3).

5.1. Framework and concepts

STT requires that the microarchitect defines what instructions write secrets into registers (*access instructions*, mainly loads), what instructions can form explicit channels (*transmitters*), and what instructions form implicit channel branch predicates (for both explicit and implicit branches). Finally, the architect must define the *Visibility Point*, after which speculation is considered safe (e.g., at the point of the oldest unresolved branch, or at the head of the ROB). If the Visibility Point refers to an instruction older than an access instruction, we call the access instruction *unsafe*; otherwise, it is considered *safe*.

We provide guidelines for microarchitects to identify access and transmit instructions. An instruction should be classified as an access instruction if it has the potential to return a secret. Except for loads, there are only a handful of such instructions, which can be identified manually.

An instruction should be classified as 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. This 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 depends on the input value, in the style of the OISA[27] or GLIFT[22] formal frameworks. Automatically performing such analysis is important future work.

5.2. Taint and untaint propagation

Conceptually, in each clock cycle, STT applies the following taint rules to instructions in the ROB:

- **The output register of an access instruction** is tainted if and only if the access instruction is unsafe.
- **The output register of a nonaccess instruction** is tainted if and only if at least one of its input operands is tainted.

In the implementation, taint propagation is piggybacked on the existing register renaming logic in an out-of-order core. Tainting is therefore fast. By contrast, it is difficult to propagate "untaint," to all dependencies of an access instruction that becomes safe, in a single cycle. We address this with a single-cycle implementation for untaint in Section 6.

Unlike prior DIFT schemes,[21] STT does not require tracking taint in any part of the memory system or across store-to-load forwarding. The reason is that because loads are access instructions, the taint of their output is determined only based on whether they have reached the Visibility Point. That is, the output of an unsafe load is always tainted.

5.3. Blocking covert channels

Given STT's rules for tainting/untainting data and its abstraction for covert channels, STT blocks all covert channels by applying a uniform rule across each type.

Blocking explicit channels. STT blocks explicit channels by delaying the execution of any transmit instruction whose operands are tainted until they become untainted. 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 returns a secret but does not have (transmit) a secret operand—such as the load on Line 2 in Figure 1—executes without delay. The load on Line 3, however, will be delayed and eventually squashed, thereby defeating the attack.

Blocking implicit channels. STT blocks implicit channels by enforcing an invariant that the sequence of instructions fetched/executed/squashed never depends on tainted data. That is, *STT makes the program counter independent of tainted data*. To enforce this invariant efficiently, without needing to delay execution of instructions following a tainted branch, we introduce two general principles to neutralize the sources of implicit channels:

- **Prediction-based implicit channels** are eliminated by preventing 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 untainted.

The above principles can be applied to efficiently make *any* hardware predictor impossible to exploit as a covert channel for leaking speculatively accessed data.

Conceptually, the protection mechanism does not need to reason about whether an implicit channel is caused by an explicit or implicit branch: both types have a predicate, and the policy with respect to the predicate is the same in both cases. The implementation, however, must identify the predicate. We illustrate this by showing how the STT microarchitecture handles explicit branches.

Applying Principle #1 (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. Specifically, 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.

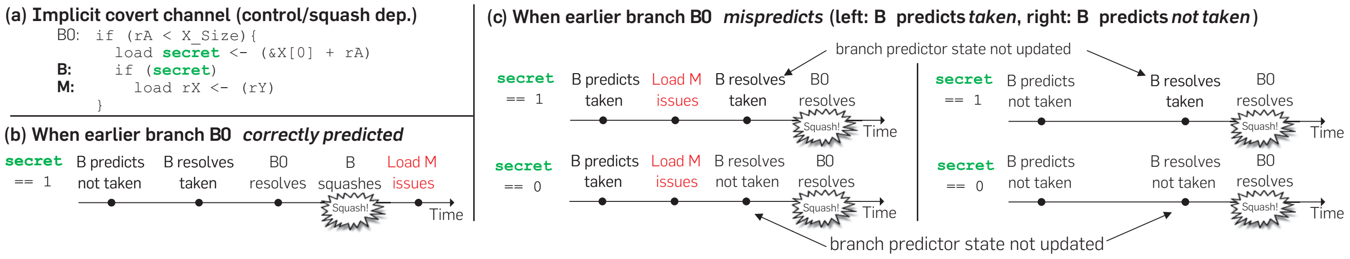
Figure 7(c) demonstrates the effect of STT on a speculative execution of the code snippet in Figure 7(a), in which the branch `B0` is mispredicted as taken. No matter how many experiments the attacker runs, the predicted direction of the branch **B** will not be a function of `secret`, because the branch 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).

Applying Principle #2 (resolution-based channels). STT delays squashing a branch that resolves as mispredicted until the branch's predicate becomes untainted. As a result, a doomed-to-squash branch with a tainted predicate (such as the branch **B** in Figure 7(c)) will never be squashed and re-executed, preventing the implicit channel leak discussed in Section 4.3. As Figure 7(c) shows, the doomed-to-squash 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 branch's resolution. Importantly, 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.

STT only increases the latency of *recovering* from a *tainted branch* misprediction. For example, in Figure 7(b), the load does not execute immediately after B resolves. Fortunately, tainted branch mispredictions are only a small fraction of overall branch mispredictions, which are infrequent in the first place because successful speculation requires accurate branch prediction.

Implicit branches. The paper applies STT's principle to secure several common microarchitectural optimizations

Figure 7. STT executing the code in (a), which includes an untainted branch B0, an access instruction reading `secret`, and an implicit channel (due to branch B).



(a) Implicit covert channel (control/squash dep.)

```
B0:  if (rA < X_Size){
       load secret <- (&X[0] + rA)
B:     if (secret)
M:       load rX <- (rY)
     }
```

(b) When earlier branch B0 *correctly predicted*

(c) When earlier branch B0 *mispredicts* (left: B predicts *taken*, right: B predicts *not taken*) that can be formulated as implicit branches, namely: store-to-load forwarding, memory-dependence speculation, and memory consistency speculation. In the process, the paper details various optimizations and cases which arise when dealing with implicit channels. In particular, whether the explicit/implicit branch has a prediction step can be resolved early or can be optimized in some other way. For example, because store-to-load forwarding can only result in two observable outcomes (issue the load or forward from a prior store), we hide which one occurs by unconditionally accessing the cache.

6. STT: IMPLEMENTATION

We previously assumed untaint information propagated along data dependencies instantly. This is difficult to implement in hardware because a word of tainted data may be a function of complex dependency chains involving many access instructions.

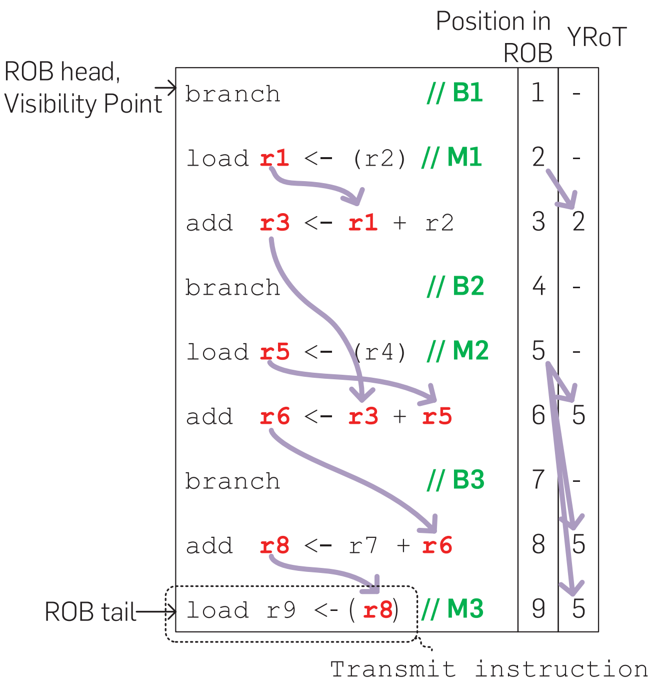
A tainted register needs to be untainted once all the access instructions on which it depends reach the Visibility Point, that is, become safe. Our key observation is that it suffices to track only when the *youngest access instruction* becomes safe, because instructions become nonspeculative in program order in the processor reorder buffer (ROB). We call this youngest access instruction the *youngest root of taint (YRoT)*.

Determining the YRoT is done through modifications to rename logic in the processor frontend. Specifically, the YRoT for an instruction X being renamed is given by the max of (1) the YRoT(s) of the instruction(s) producing the arguments for X, if those instructions are not access instructions; or (2) the ROB index of the instruction(s) producing the arguments for X, otherwise. (By convention, we assume the ROB index increases from ROB head to tail.) After renaming, the YRoT is stored alongside the instruction in its reservation station and is conceptually an extra dependency for that instruction. When the Visibility Point changes, its new position is broadcast to in-flight instructions, akin to a normal writeback broadcast, and instructions whose YRoT is less than the Visibility Point's new position are allowed to execute (assuming their other dependencies are satisfied). The entire architecture requires modest changes to the frontend rename logic, storage in reservation stations for the YRoT, and logic to compare the YRoT to the Visibility Point which is comparable to normal instruction wakeup logic.

Figure 8 shows an example. Assume the Spectre attack model, that is, the Visibility Point will be set to the ROB index of the oldest unresolved branch. The ROB contains 3 unresolved branches (`B1`–`B3`) and a transmit instruction (`M3`) whose operand/address `r8` is a function of the return value of two access instructions (`M1` and `M2`). `M3` is a transmit instruction (because it is a load) and can potentially leak secrets because misspeculations on branches `B1` and `B2` can influence the data returned by loads `M1` and `M2`, which in turn contribute to the address of `M3` through data dependencies.

On the one hand, the data dependency chain from load `M1` all the way to load `M3` is quite complex. That is, the instruction at ROB index 6 depends on index 5 and index 3, index 8 depends on 6, etc. Re-traversing this dataflow graph to propagate untaint, akin to tracing backwards slices, would be expensive. On the other hand, the YRoT dependency chain is relatively simple. Each instruction just tracks whichever is the youngest load that contributes to its dependency chain (e.g., load `M2` for instructions 6, 8 and 9). When branches `B1` and `B2` resolve, the Visibility Point advances to point to branch `B3` (ROB index 7). As 7 is greater than 5 (the YRoT for the transmit instruction `M3`), `M3` is allowed to execute at this point. Note, the dependency chain could have been more complex, with additional branches and arithmetic dependencies separating load `M2` and load `M3`, but this would not change the moment that it is safe to execute load `M3`.

Figure 8. Example showing YRoT tracking showing a snapshot of ROB state. Addition (add) instructions are used to represent arithmetic (non-loads). If the YRoT is set to '-', it means the instruction's youngest dependent access instruction is a part of retired state.



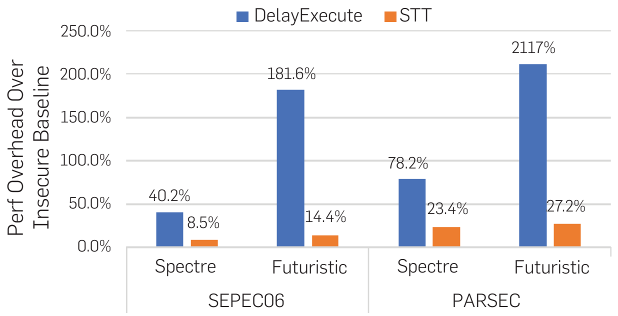
	Position in ROB	YRoT
`branch              // B1`	1	-
`load r1 <- (r2)     // M1`	2	-
`add  r3 <- r1 + r2`	3	2
`branch              // B2`	4	-
`load r5 <- (r4)     // M2`	5	-
`add  r6 <- r3 + r5`	6	5
`branch              // B3`	7	-
`add  r8 <- r7 + r6`	8	5
`load r9 <- (r8)     // M3`	9	5

Importantly, the above scheme is only secure after applying STT's mechanisms to block *both* explicit and implicit channels (Section 5). That is, the scheme requires that `r8` is not a function of speculative data *at the exact moment load M2 becomes nonspeculative*. This requires that branch `B3` not be influenced by speculative data (achieved by protections for implicit channels) and that other intervening instructions that can cause explicit channels not execute until they are likewise safe (achieved by protections for explicit channels).

7. FORMAL ANALYSIS/SECURITY PROOF

We formally prove in a companion document[29] that STT enforces a novel notion of noninterference: at each step of the execution, the value of a *doomed* register—a register written to by a bound-to-squash access instruction—does

Figure 9. Performance evaluation on SPEC06 and PARSEC benchmark suites. STT outperforms the baseline secure scheme (DelayExecute) with much smaller performance overhead, for both Spectre and Futuristic attacker models.



not influence future visible events in the execution. This applies to all microarchitectural timing and interference-based attacks. For instance, the property ensures that the program's completion time and hardware resource usage—for all hardware structures such as cache, branch predictor, etc.—are completely independent of doomed values.

The key challenge in the analysis is how to avoid "looking into the future" to determine if an instruction is doomed to squash. We address this by running the STT machine alongside a nonspeculative in-order processor, which allows us to verify the STT machine's branch predictions and determine whether a prediction leads to misspeculation or not.

8. EVALUATION RESULTS

We evaluate STT on 21 SPEC and 9 PARSEC workloads. The results are shown in Figure 9. Relative to an insecure machine, STT adds only 13.0%/18.2% overhead (averaged across both SPEC and PARSEC benchmarks) depending on whether the attack model considers only control-flow speculation (Spectre) or all types of speculation (Futuristic). Compared to the baseline secure scheme (DelayExecute) described in Section 1, STT reduces overhead by 4.0× in the Spectre model and 10.5× in the Futuristic model, on average. This indicates that defending against stronger attack models is viable with STT without sacrificing much performance.

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. We would especially like to thank our colleagues at Intel who contributed significant feedback throughout the project's development, in particular Fangfei Liu, Matthew Fernandez, Frank McKeen, and Carlos Rozas.

References

1. Aciicmez, O., Seifert, J.-P., Koc, C.K. Predicting secret keys via branch prediction. In *IACR'06* (2006).
2. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C. P., Tuveri, N. Port contention for fun and profit. In *IACR'18* (2018).
3. Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H. On subnormal floating point and abnormal timing. In *S&P'15* (2015).
4. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS'19* (2019).
5. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security'19* (2019).
6. Chrysos, G.Z., Emer, J.S. Memory dependence prediction using store sets. In *ISCA'98* (1998).
7. Dalton, M., Kannan, H., Kozyrakis, C. Raksha: A flexible information flow architecture for software security. In *ISCA'07* (2007).
8. Gharachorloo, K., Gupta, A., Hennessy, J. Two techniques to enhance the performance of memory consistency models. In *ICPP'91* (1991).
9. Goguen, J.A., Meseguer, J. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy* (1982).
10. Großschädl, J., Oswald, E., Page, D., Tunstall, M. Side-channel analysis of cryptographic software via early-terminating multiplications. In (2009).
11. Hennessy, J.L., Patterson, D.A. *Computer Architecture: A Quantitative Approach*, 6th edn. Morgan Kaufmann Publishers Inc., 2017.
12. Intel. Q2 2018 speculative execution side channel update, 2018. https://www.intel.com/content/www/us/en/ security-center/advisory/intel-sa-00115.html.
13. Johnson, M. *Superscalar Microprocessor Design*. Prentice Hall Englewood Cliffs, New Jersey, 1991.
14. Kiriansky, V., Lebedev, I.A., Amarasinghe, S.P., Devadas, S., Emer, J. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO'18* (2018).
15. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y. Spectre attacks: Exploiting speculative execution. In *S&P'19* (2019).
16. Lipasti, M.H., Wilkerson, C.B., Shen, J.P. Value locality and load value prediction. In *ASPLOS'96* (1996).
17. Percival, C. Cache missing for fun and profit. In *Proceedings of BSDCan 2005* (2005).
18. Reinman, G., Calder, B. Predictive techniques for aggressive load speculation. In *MICRO'98* (1998).
19. Sabelfeld, A., Myers, A.C. Language-based information-flow security. *IEEE J. Sel. Areas Commun. 21*, 1 (Jan. 2003), 5–19.
20. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D. Netspectre: Read arbitrary memory over network. In *ESORICS'19* (2019).
21. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S. Secure program execution via dynamic information flow tracking. In *ASPLOS'04* (2004).
22. Tiwari, M., Wassel, H.M., Mazloom, B., Mysore, S., Chong, F.T., Sherwood, T. Complete information flow tracking from the gates up. In *ASPLOS'09* (2009).
23. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev. 11*, 1 (1967), 25–33.
24. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security'18* (2008).
25. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C.W., Torrellas, J. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *MICRO'18* (2018).
26. Yarom, Y., Falkner, K. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security'14* (2014).
27. Yu, J., Hsiung, L., Hajj, M.E., Fletcher, C.W. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS'19*. https://eprint.iacr.org/2018/808.
28. Yu, J., Mantri, N., Torrellas, J., Morrison, A., Fletcher, C.W. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *ISCA'20*.
29. Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., Fletcher, C.W. *Speculative Taint Tracking (STT): A Formal Analysis*. Technical report, University of Illinois at Urbana-Champaign and Tel Aviv University, 2019. http://cwfletcher.net/Content/Publications/Academics/TechReport/stt-formal-tr_micro19.pdf.

Jiyong Yu, Josep Torrellas, and Christopher W. Fletcher, University of Illinois at Urbana-Champaign, IL, USA.

Mengjia Yan, Massachusetts Institute of Technology, Cambridge, MA, USA.

Artem Khyzha and Adam Morrison, Tel Aviv University, Israel.