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

Jiyong Yu, Mengjia Yan*, Artem Khyzha**, Adam Morrison**, Josep Torrellas, Christopher W. Fletcher
University of Illinois at Urbana-Champaign *Massachusetts Institute of Technology **Tel Aviv University

1 Introduction

Speculative execution attacks such as Spectre [5] 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 covert channels such as the cache [9]. For example, in Spectre V1 (Figure 1) a branch misprediction enables the attacker to access and leak/transmit arbitrary program data by controlling the out-of-bounds address `&array1[off]`. We refer to such data, which is brought into the pipeline by a speculative instruction, as *secret*.

A secure, but conservative, way to block *all* speculative execution attacks—regardless of covert channel—is to delay executing all instructions that can access a secret until such instructions become non-speculative. In nearly all attacks today, this would imply blocking all loads until they are non-speculative, 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 leak secrets or do not form covert channels, and so can execute freely under speculation. For example, the first load in Spectre V1 forms a covert channel, but it only leaks the attacker-selected address `&array1[off]`—not the secret data in that address. Likewise, many instructions (e.g., simple arithmetic) do not form covert channels even if their operands are secret values.

The paper presents 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 [7]) until it is about to reach an instruction that may form a covert channel. STT then delays the forwarding of the data until the instruction becomes non-speculative or the execution squashes due to miss-speculation. To be secure and efficient, we address two key challenges.

- First, we develop an abstraction that indicates how and when instructions can form covert channels, so as to delay data forwarding to the latest safe time.
- Second, we identify and develop a microarchitecture to indicate exactly when data should be considered secret, so as to re-enable data forwarding at the earliest safe time.

1.1 Challenge #1: New abstractions for describing *all* microarchitectural covert channels

Covert channels come in different shapes and sizes. For example, attackers can monitor how loads interact with the cache [5], the timing of SIMD units [6], execution pipeline port contention [2], branch predictor state [1] and more. To comprehensively block leakage through these different channels, it is necessary to understand their common characteristics.

```
if (off < array1_size) {   // mispredicts
  x = array1[off];    // secret accessed
  y = array2[x]; } // secret transmitted
```

Figure 1: Spectre Variant 1.

To address this challenge, the paper proposes a new abstraction through which to view covert channels on speculative microarchitectures, discovers new points that 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 (which are related to explicit and implicit information flow [8]). 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 [5]. In an *implicit channel*, data indirectly influences how (or that) 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 [2, 6]. The paper further defines sub-classes of the 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 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, i.e., 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 [11].

1.2 Challenge #2: 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 delayed instructions reach the head of the reorder buffer (ROB) and block retirement.

STT tackles this problem with a safe but aggressive approach, *by re-enabling data forwarding as soon as data becomes a function of retired register file state*. This represents the earliest safe point, but is non-trivial to implement in hardware. 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 non-speculative information would require retracing a backwards 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 retired state, it is sufficient to determine whether the *youngest* load, whose return value influences the data, has become non-speculative. Checking this condition is akin to tracking a single extra dependency for each instruction, as opposed to performing complex backwards slice tracking.

1.3 Security guarantees and formal analysis

Alongside the main paper, we formally prove that STT enforces a novel form of non-interference [3] with respect to speculatively accessed data. In a nutshell, we show that hardware resource usage patterns over time are independent of data that eventually squashes (covering microarchitectural interference- and timing-based attacks). We released a companion technical report [12] with detailed formal analysis and a security proof for this property.

1.4 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 Attacker Model & Protection Scope

Attacker Model. STT assumes 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 example, the attacker can monitor covert channels through the cache/memory system [5], data-dependent arithmetic [4], port contention [2], branch predictors [1], etc.

Scope: Protecting Speculatively Accessed Data. We distinguish attacks based on whether the access instruction is doomed-to-squash (*transient*) or bound to retire (*non-transient*). STT's goal is to block attacks involving doomed-to-squash access instructions, shown in Figure 2. These attacks can access data that a correct (not miss-speculated) 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., [10]).

3 Abstraction for Covert Channels

STT proposes a novel abstraction for covert channels (Figure 3). In our abstraction, covert channels are broken into two classes:

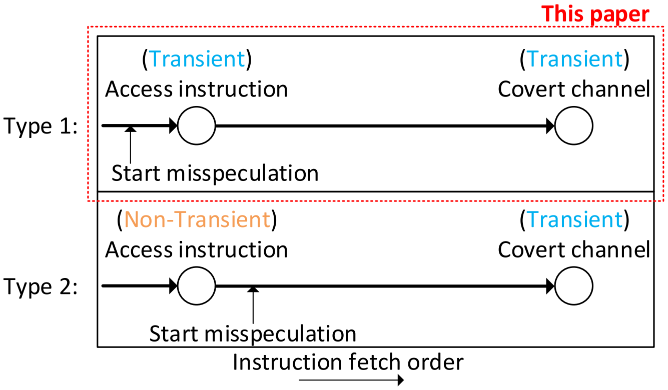


Figure 2: STT's scope is to protect speculative accessed data from leaking over any microarchitectural covert channel. Protecting values which have retired is outside of scope.

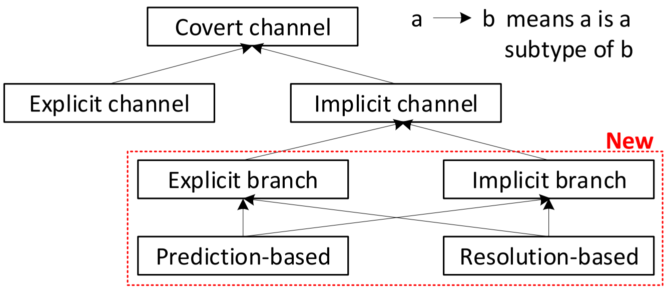


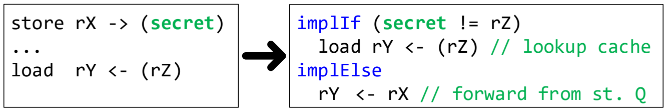
Figure 3: Speculative microarchitectures covert channel abstraction.

explicit and *implicit* channels. An *explicit channel*, related to explicit flow in information flow [8], 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 [8], 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 distinguish between implicit channels, depending on when they leak secrets and what type of branch they feature. First, we find that implicit channels can leak at two points: when a prediction is made (e.g., a branch prediction) and when a resolution occurs (e.g., a branch resolves). Second, we find that implicit channels can feature either an *explicit* or an *implicit* branch. An explicit branch is a control-flow instruction, while an implicit branch is a conceptual branch formed in the hardware due to an optimization. For example, store-to-load forwarding between a store and a younger load can be viewed as an implicit branch that checks for an address alias as shown in Figure 4. Written this way, it is clear that store-to-load forwarding can create a covert channel: depending on whether there is an alias, the processor either looks up the cache or forwards from the local store queue.

3.1 Insights from analysis of implicit channels

Since it was proposed in the paper, the classification for implicit channels has proven to be a robust and useful way to represent and pinpoint the root cause of hardware security vulnerabilities. For example, in the NetSpectre attack [6], it might be said



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

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

Figure 4: Rewriting a store-load pair as an implicit branch. implIf reveals potential covert channels as a function of memory aliasing to the older store. This occurs if the microarchitecture supports store-to-load forwarding or memory-dependence speculation.

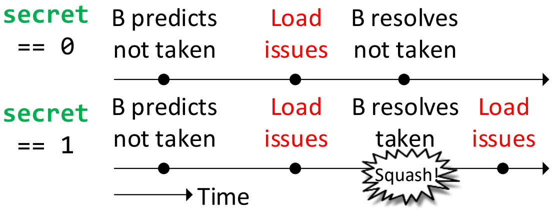


Figure 5: Resolution-based implicit channel due to secret-dependent pipeline squashes. When the branch (B) resolves, it leaks the secret based on whether a squash occurs, as this causes the younger load to execute once or twice. There is an analogous case when the (public) predictor state takes the branch.

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. Consider a simple example where a load is control- and data-independent of a sensitive branch, e.g., "if (secret == rV) { rX <− rW; } load rZ <− (rY);". How this load executes is important, to understand whether a potential cache-based covert channel exists. Traditional software-level analysis would indicate that the execution of the load is independent of the secret (the branch outcome). Yet, on a speculative microarchitecture, STT's abstraction would classify this code as a resolution-based implicit channel. As shown in Figure 5, if the branch miss-speculates and subsequently squashes, the load may execute either once or twice depending on the value of secret.

Finally, the abstraction applies to a large set of microarchitectural optimizations. For example, the representation of store-to-load forwarding (Figure 4) also captures the behavior of memory-dependence speculation with a store set predictor. Here, the abstraction models the store set predictor as a *prediction* on the implicit branch (implIf in the figure). As we will see, 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).

4 STT: Design

4.1 Framework & concepts

STT requires that the microarchitect define 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 on identifying access and transmit instructions. An instruction should be classified an access instruction if it has the potential to read a secret. Except for loads, there are only a handful of such instructions, which can be identified manually.

An instruction should be classified 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 [10] or GLIFT [8] formal frameworks. Automatically performing such analysis is important future work.

4.2 Taint & 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 non-access 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. In contrast, it is difficult to propagate "untaint" to all depdendencies of an access instruction that becomes safe in a single cycle. We address this with a single-cycle implementation for untaint in Section 5.

Unlike prior DIFT schemes [7], 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.

4.3 Blocking covert channels

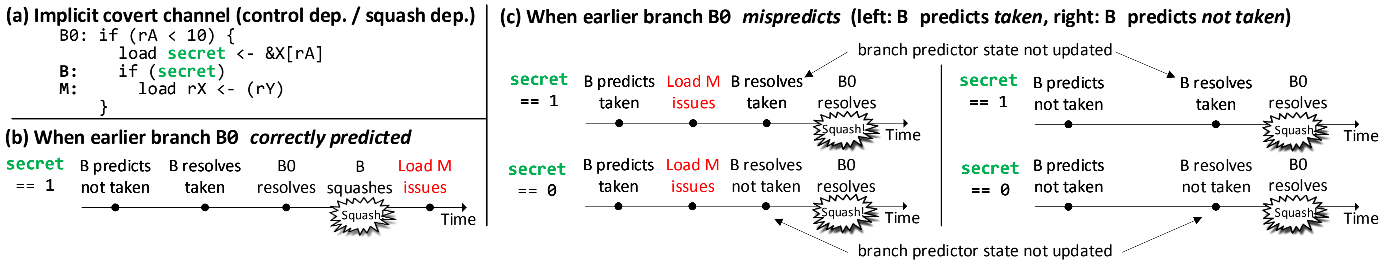
Given STT's rules for tainting/untainting data and its abstraction for covert channels, STT blocks all covert channels by applying a uniform rule across each type.

4.3.1 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 *reads* a (potential) secret but does not transmit one—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.

4.3.2 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,



```
B0: if (rA < 10) {
      load secret <- &X[rA]
B:    if (secret)
M:      load rX <- (rY)
    }
```

Figure 6: STT executing the code in (a), which includes an untainted branch B0, an access instruction reading secret, and an implicit channel.

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 branch's predicate becomes untainted.

STT's principles can be applied to efficiently make *any* hardware predictor impossible to exploit as a covert channel for leaking speculatively-accessed secrets.

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. 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 6(c) demonstrates the effect of STT on a speculative execution of the code snippet in Figure 6(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 6(c)) will never be squashed and re-executed, preventing the implicit channel leak discussed in Section 3.1. As Figure 6(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 6(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 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.

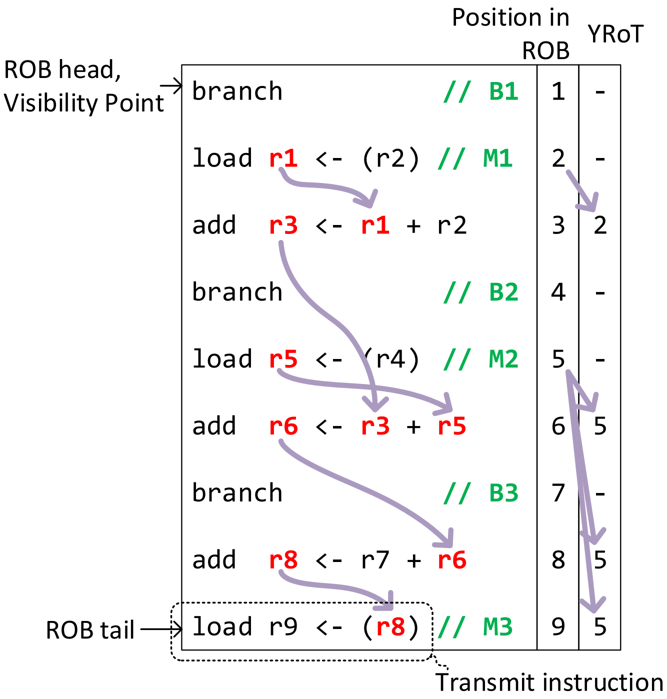
5 STT: Implementation

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 access instructions on which it depends reach the Visibility Point, i.e., becomes safe. Our key observation is that it suffices to track only when the *youngest access instruction* becomes safe, because instructions become non-speculative in program order in the processor reorder buffer (ROB). We call this youngest access instruction the *youngest root of taint (YRoT)*.

Determiming 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 rename, 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 7 shows an example. Assume the Spectre attack



```
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
```

Figure 7: 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 '-', that means the instruction's youngest dependent access instruction is a part of retired state.

model, i.e., that 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 miss-speculations 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 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). Since 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.

Importantly, the above scheme is only secure after applying STT's mechanisms to block *both* explicit and implicit channels (Section 4). That is, the scheme requires that r8 is not a function of speculative data *at the exact moment load M2 becomes non-speculative*. 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).

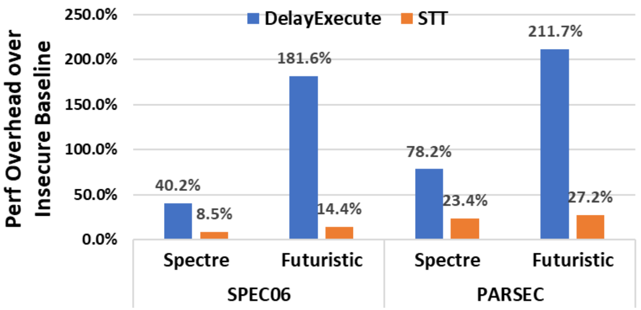


Figure 8: Performance evaluation on SPEC06 and PARSEC benchmark suites. STT outperforms the baseline secure scheme (DelayExecute) with much smaller performance overhead, for both Spectre and Futuristic attacker models.

6 Formal Analysis/Security Proof

We formally prove [12] that STT enforces a novel notion of non-interference: at each step of the execution, the value of a *doomed* register—a register written to by a bound-to-squash access instruction—does 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 including cache, branch predictor, etc.—is 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 non-speculative in-order processor, which allows us to verify the STT machine's branch predictions and determine whether a prediction leads to miss-speculation or not.

7 Evaluation Results

We evaluate STT on 21 SPEC and 9 PARSEC workloads. The results are shown in Figure 8. Relative to an insecure machine, STT adds only 13.0%/18.2% overhead (averaged across all 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.

8 Acknowledgements

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] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*, 2006.
[2] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *CCS'19*, 2019.
[3] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982.

[4] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," 2009.

[5] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P'19*, 2019.

[6] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS'19*, 2019.

[7] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *ASPLOS'04*, 2004.

[8] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS'09*, 2009.

[9] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*, 2014.

[10] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS'19*. `https://eprint.iacr.org/2018/808`.

[11] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative Data-Oblivious Execution (SDO): Mobilizing Safe Prediction For Safe and Efficient Speculative Execution," in *ISCA'20*.

[12] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Formal Analysis," tech. rep., University of Illinois at Urbana-Champaign and Tel Aviv University, 2019. `http://cwfletcher.net/Content/Publications/Academics/TechReport/stt-formal-tr_micro19.pdf`.