Doppelganger Loads: A Safe, Complexity-Effective Optimization for Secure Speculation Schemes

Amund Bergland Kvalsvik, Pavlos Aimoniotis†, Stefanos Kaxiras†, and Magnus Själander
Amund.Kvalsvik@ntnu.no|Pavlos.Aimoniotis@it.uu.se|Stefanos.Kaxiras@it.uu.se|Magnus.Sjalander@ntnu.no
Norwegian University of Science and Technology, Trondheim, Norway
†Uppsala University, Uppsala, Sweden

ABSTRACT

Speculative side-channel attacks have forced computer architects to rethink speculative execution. Effectively preventing microarchitectural state from leaking sensitive information will be a key requirement in future processor design.

An important limitation of many secure speculation schemes is a reduction in the available memory parallelism, as unsafe loads (depending on the particular scheme) are blocked, as they might potentially leak information. Our contribution is to show that it is possible to recover some of this lost memory parallelism, by safely predicting the addresses of these loads in a threat-model transparent way, i.e., without worsening the security guarantees of the underlying secure scheme. To demonstrate the generality of the approach, we apply it to three different secure speculation schemes: Non-speculative Data Access (NDA), Speculative Taint Tracking (STT), and Delay-on-Miss (DoM).

An address predictor is trained on non-speculative data, and can afterwards predict the addresses of unsafe slow-to-issue loads, preloading the target registers with speculative values, that can be released faster on correct predictions than starting the entire load process. This new perspective on speculative execution encompasses all loads, and gives speedups, separately from prefetching.

We call the address-predicted counterparts of loads *Doppelganger Loads*. They give notable performance improvements for the three secure speculation schemes we evaluate, NDA, STT, and DoM. The Doppelganger Loads reduce the geometric mean slowdown by 42%, 48%, and 30% respectively, as compared to an unsafe baseline, for a wide variety of SPEC2006 and SPEC2017 benchmarks. Furthermore, Doppelganger Loads can be efficiently implemented with only minor core modifications, reusing existing resources such as a stride prefetcher, and most importantly, requiring no changes to the memory hierarchy outside the core.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**; • **Security and privacy → Hardware-based security protocols**.

ACM Reference Format:
Amund Bergland Kvalsvik, Pavlos Aimoniotis†, Stefanos Kaxiras†, and Magnus Själander. 2023. Doppelganger Loads: A Safe, Complexity-Effective Optimization for Secure Speculation Schemes. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589088

1 INTRODUCTION

With the disclosure of Spectre [26] many architectures, previously considered secure, were shown to be unsafe because speculative execution in large part disregarded security vulnerabilities. Since their original introduction, many variations of the attacks have been developed [2, 8, 10, 12, 35, 42, 51], and early mitigation strategies [3, 4, 6, 7, 19, 22, 24, 25, 27, 29, 30, 38–41, 47, 49, 52–56] strive to comprehensively eliminate speculative side-channel attacks at acceptable overheads.

Speculative attacks exploit two key properties of modern computer architectures: First, they rely on transient instructions, which are instructions that are being executed due to an incorrect prediction in the core or as part of delayed exception handling, and that are bound to be squashed. Transient loads are used to access secrets, albeit only for a limited period of time. Second, they rely on side-channels, microarchitectural features that inadvertently leak information, either by persistent microarchitectural state after squashing or by transmitting information to an attacker during the speculative phase. As shown by the example in Figure 1 (a), these two properties enable successful attacks: A secret is first accessed by a speculative load and subsequently leaked through a side-channel by a following transmitting instruction. Some stricter threat models also consider secrets as potentially residing in registers, i.e., a speculative load is not required to first access the secret [13, 40].

Early works, such as InvisiSpec [52] and Ghost Loads [39], provide solutions for mitigating the cache as a potential side-channel, but do not eliminate many other side-channels [17, 23, 36]. Later works, such as Speculative Taint Tracking (STT) [54], Non-speculative Data Access (NDA) [49], Delay-on-Miss [40], DAWG [25], and others, improve on previous work by either achieving better performance, securing more side-channels, or both. Some of the best performing works, such as GhostMinion [3] and speculative data oblivious (SDO) execution [53], are able to comprehensively mitigate Spectre at low performance overheads. However, they require significant overhauls of the core microarchitecture and, more importantly, of the memory hierarchy, which complicates hardware design and is often impractical. Modifications to the memory hierarchy quickly become complex, and hard to verify, as such solutions have to consider the interaction between cores and the impact on cache coherence and memory consistency.

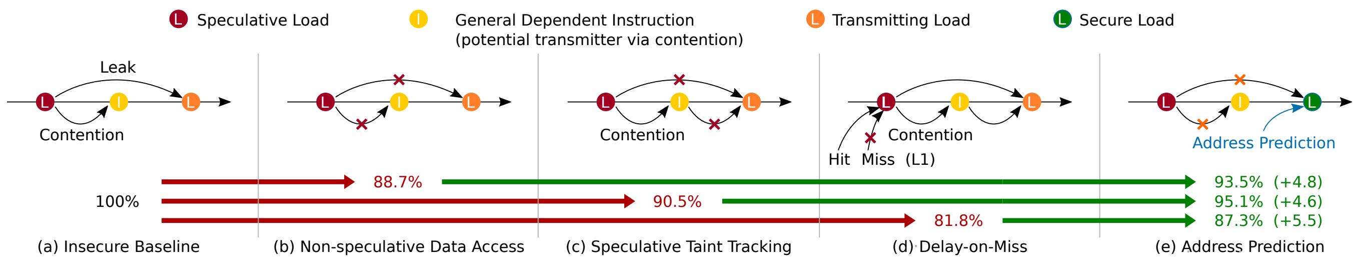

Figure 1: (a) Unsafe Baseline forwards speculatively loaded values, potentially transmitting secrets. (b) Non-speculative Data Access (NDA) performs speculative loads but never forwards potential secrets. (c) Speculative Taint Tracking (STT) performs speculative loads and forwards potential secrets to non-transmitting instructions, enabling ILP. (d) Delay-on-Miss (DoM) allows memory accesses that hit in the L1 data cache to be performed, but delays memory access that miss in the L1 cache until the load becomes non-speculative. (e) Doppelganger Loads address predict dependent loads enabling them to securely access the memory hierarchy, turning transmitting loads into secure loads, while enabling MLP. The performance results (red and green arrows), show that Doppelganger Loads improves the performance by between 4.6 to 5.5 percentage points over the secure schemes, when using a simple strided address predictor.

We make the observation that previous secure speculation schemes that limit the implementation complexity to within the core itself, such as NDA and STT, inadvertently limits the available memory level parallelism (MLP) that can be exploited. Non-speculative Data Access (NDA) [49] offers several strategies for secure speculation. The one we focus on, called permissive propagation (NDA-P), propagates non-speculatively loaded values but delays the propagation of a speculatively loaded value until the load is determined to be non-speculative (Figure 1(b)). By not forwarding speculatively loaded values at all, NDA is able to comprehensively eliminate any-and-all speculative side-channel attacks that rely on speculatively acquired secrets, such as for universal read gadgets. The main drawback of NDA is that it delays all dependent instructions.

Speculative Taint Tracking (STT) [54] partially overcomes this limitation by *tainting* speculatively loaded data and instead preventing instructions that can be used as transmitters to be executed while their operands are tainted. STT enables instruction level parallelism (ILP), as dependent non-transmitting instructions are allowed to execute. However, STT does not issue more dependent loads than NDA, as loads transmit information (Figure 1(c)). Delay-on-Miss takes an alternative approach, allowing unobservable loads to complete as normal, even if they are dependent. If a load's access is a hit in the L1 cache, the load creates no observable timing differences in the cache.[1] DoM uses this insight to increase ILP, while also protecting secrets residing in registers. However, DoM is unable to achieve MLP, especially for the critical long-latency loads, as any access to the L2 cache and onwards would potentially leak the secret.

These techniques all provide different security guarantees, but limit the available MLP to some extent. Another approach exemplified by speculative data-oblivious execution (SDO) [53] is able to reclaim lost MLP by performing secret-independent prediction, specifically as an optimization on STT. By predicting where in the memory hierarchy a requested cache line resides, SDO makes the *timing behavior* of the load independent of speculatively-loaded values. However, more importantly, it does not do the same for the load's address, which can still depend on speculatively loaded values. Because of this, considerable effort must be expended to change the memory hierarchy to ensure that the address never leaks. In many situations, such changes may not be practical. In contrast, our main contribution is to make the address of unsafe loads independent of speculatively loaded values, which means that we work with an unmodified cache hierarchy.

Instead of making loads timing-independent, we focus on assuring that the loads are secret-independent, by predicting their addresses, and preloading registers with the values of the predicted-address loads. We call these address-predicted loads, *doppelgangers* of the actual loads. The security of prediction based on non-speculative data has previously been demonstrated for data prefetching [39], value prediction [40], branch prediction [54], and execution latency prediction [53].

Predicting the address is safe as long as the predictor is trained on committed addresses, but the data fetched by an address predicted load might still contain a secret. Address predicted loads preload values into their target registers, and delay the propagation of ready data until the underlying scheme declares it safe, and the address has been verified. This ensures that potentially incorrect data is never released into the core, which removes the need to introduce squashing and rollback on mispredictions, such as for DoM with value prediction [40]. On an address misprediction, the actual load with the correct address is issued. Doppelganger Loads can be applied on top of many different secure speculation schemes, without affecting the threat model of the underlying scheme, i.e., Doppelganger Loads are threat-model transparent as we demonstrate in Section 4. In Section 5 we show how Doppelganger Loads can be efficiently implemented on the various secure speculation schemes in a complexity-effective, low-cost manner. Our results (methodology in Section 6 and evaluation in Section 7), summarized in Figure 1, show the performance degradation of a model implementation of NDA (NDA-P), DoM, and STT, as well as the performance gains from improving these implementations with address implementations.

[1]replacement state in the L1 is updated retroactively.

2 BACKGROUND AND MOTIVATION

Speculative side-channel mitigations have only existed for a short period of time, originating shortly after the initial publication of the Spectre attacks [26]. The computer architecture field has rapidly come up with novel strategies, such as invisible execution [52], taint-tracking [54], and shadow tracking [39], that use greatly different approaches to handle speculative side-channel attacks. When discussing prior work, however, it is important to keep in mind that different schemes have addressed different threat models, with different understandings of *speculation* and *leaking secrets*. We focus our discussion on NDA (permissive propagation), STT, and DoM. For the sake of brevity, we assume familiarity with speculative execution attacks.

2.1 Non-Speculative Data Access

Weisse et al. propose Non-speculative Data Access (NDA) [49] that aims to block speculative execution attacks at their source, by preventing them at the earliest feasible stage. They identify that in order to block attacks that acquire secrets from memory and transmit a secret speculatively, it is enough to delay speculative loads from propagating their results until they are non-speculative, as they are then bound to become architecturally visible. NDA refers to this strategy as *permissive propagation*, henceforth referred to as *NDA-P*. In terms of universal read gadgets, or other attacks that rely on speculatively acquiring a value from memory, this strategy blocks the leakage of secrets in a complexity-effective manner: By preventing the origin of secrets through a delayed propagation mechanism, all potential leakage of that secret is prevented.

This strategy has the benefit that all forms of transmitters are blocked without the need for evaluating what microarchitectural effects that might be observable, which is a very difficult task: Observability is partially microarchitecture dependent, and there might exist novel methods of transmitting information through previously thought unobservable units (see related work for more details on other attacks). However, the limitation of parallelism incurs heavy performance penalties, as all forms of dependent loads are blocked, and instruction-level parallelism is also prevented.

2.2 Speculative Taint Tracking

Speculative Taint Tracking (STT) [54] is proposed with the intent of mitigating with low overhead Spectre attacks as universal read gadgets. Based on its threat model, STT's mitigation strategy is based on preventing the transmission of speculatively accessed values. Notably, STT does not protect against secrets in registers. STT blocks *explicit channels* and *implicit channels* [54] that otherwise could transmit speculatively loaded values. To achieve this, STT taints the output of all speculative load instructions, untainting them only when they are no longer speculative, i.e., bound-to-commit, which STT calls "reaching the visibility point." Taints propagate via the registers from input operands to the result.

Explicit channels are formed by instructions whose execution is observable, i.e., can transmit a secret in a side channel. STT prevents the transmission of secrets via explicit channels by selectively delaying instructions. More specifically, instructions which do not have tainted operands execute as normal, including transmitting instructions. Instructions that depend on tainted values can execute as normal if they are not transmitting, i.e., their execution is unobservable, or their observable execution is independent of the tainted value. Transmitting instructions that receive tainted inputs are delayed until they reach their visibility point.

Implicit channels are formed when speculatively loaded values indirectly change the execution of one or more instructions, and these changes are visible through a side channel. Implicit channels can be branches whose predicates depend on speculatively loaded values, or microarchitectural decisions that can be re-cast as *implicit branches*, such as store-to-load forwarding when the address of a store depends on a speculatively loaded value.

Because implicit channels may be combined with some form of prediction, STT further discerns implicit channels into prediction-based and resolution-based [54]. To prevent the transmission of secrets via prediction-based implicit channels, STT prevents tainted data from affecting the predictors. Respectively, for resolution-based implicit channels, STT delays (explicit and implicit) branch resolution until the branch's predicate becomes untainted [54].

2.3 Delay-on-Miss

Delay-on-Miss (DoM) [40] instead aims to hide speculative execution, rather than blocking potentially dangerous speculative execution, separating it from NDA and STT. Under DoM, loads that are speculative, which is tracked through the use of shadows [39], are allowed to issue to the cache, but fail if they miss in the L1 cache, and are reissued once they are non-speculative. Coupled with a delayed replacement policy update, this makes loads that hit in the L1 cache unobservable, while enabling both independent and dependent memory accesses. Notably, as DoM does not discriminate between data acquired speculatively and non-speculatively when executing speculative instructions, it also protects secrets residing in registers. As DoM only protects the memory hierarchy, it can still leak secrets acquired both non-speculatively and speculatively through other side-channels, such as timing differences.

DoM was the first to propose using a non-speculatively updated prediction to speed up secure speculation. In particular, DoM used value prediction (VP) but it was later shown that it was not so successful in terms of accuracy and coverage, even with state-of-the-art VTAGE value predictors [34], and because it had to be validated in-order it did not yield significant improvement in MLP [41].

2.4 Motivation: Unlocking MLP

All the previously discussed schemes limit MLP in some way. For NDA-P and STT, independent loads can safely be issued, which is better than delaying all loads under speculation, but loads that base their address on the value of other loads are delayed until the older load is non-speculative. For DoM, any number of memory requests can be issued, but are only serviced if they hit in the L1 cache. Longer latency loads that reside in the lower levels of the cache hierarchy cannot be serviced or parallelized unless they are non-speculative, reducing the amount of available MLP. These schemes all introduce a limitation on what is already the bottleneck of modern processor performance: the memory wall [50]. Therefore, a logical solution to improve the performance of these schemes, especially in the cases where the schemes have a significant overhead, is to attempt to improve MLP in a safe manner. Doppelganger Loads aim to increase MLP, by allowing dependent loads, that would normally be delayed in secure-speculation schemes, to issue using a predicted address that is not a function of speculatively loaded values.

3 THREAT MODELS

In this section, we describe the threat models that the underlying secure speculative execution schemes utilize. As there are some key differences between them, we aim to illuminate what they consider as part of the threat model, and the basic motivation for their individual threat models. Crucially, our Doppelganger Loads optimization can be applied to defenses that have different threat models.

What is common for the threat models we target, is that secrets are never transmitted as part of non-speculative execution, or more specifically, non-speculative leakage is out of scope for these schemes. There are many non-speculative attacks that are able to observe parts of program execution using contention in the memory hierarchy, but they are not part of these threat models.

3.1 Non-speculative Data Access and Speculative Taint Tracking

NDA [49] provides a detailed exploration of various threat models, and which actions are allowed without leaking potential secrets. For the purposes of this work, we focus on the NDA-P model, which aims to protect secrets in memory. This threat model is shared by STT [54], which was motivated by preventing the universal read gadget attacks that are feasible using Spectre.

The threat model assumes an adversary that can monitor all microarchitectural covert channels and can induce speculation from any part of the system, including simultaneous multithreading (SMT) and cross-cores. The list of covert channels includes the memory hierarchy, timing differences in execution, port contention, and control flow. The adversary is attempting to acquire data in memory it would not have access to non-speculatively, and then exfiltrate this information through a covert channel before the data access is squashed by the system. The authors of STT consider this to be the most dangerous form of attack, since it is capable of being used to form universal read gadgets, powerful attack tools that can read arbitrary data, essentially exposing all mapped-in memory.

NDA-P and STT both do not block the transmission of secrets that are already loaded in registers prior to speculation.

3.2 Delay-on-Miss

DoM [40], instead of focusing on all covert channels, assumes an adversary is aiming to exfiltrate a secret through observing the memory hierarchy. DoM's threat model considers leakage through observable, secret-dependent execution of instructions that alter the memory hierarchy, but does not consider leakage through timing differences in execution, port contention, or control flow. Unlike NDA-P and STT, DoM's threat model states that secrets might also reside in general purpose registers, and might have been acquired non-speculatively. DoM, STT, and NDA-P all consider speculatively acquired data being transmitted through the memory hierarchy as leakage, but DoM does not consider it as leakage if it is observable through other channels.

4 DOPPELGANGER LOADS

Having explained the underlying schemes, we now illustrate how they can be enhanced through the use of Doppelganger Loads, the address-predicted counterparts of unsafe loads.

4.1 Overview

Let us, for the moment, consider only load transmitters and cache side channels. Depending on which secure speculation scheme we consider, we discuss how other explicit and implicit side channels are treated, later on.

The key insight from NDA-P and STT is that a load is a potential transmitter when it receives as input an address that is dependent on speculatively loaded value(s). This is why, in both STT and NDA-P, a dependent load is not allowed to issue: In STT, a dependent load receives a tainted input for its address; in NDA-P, speculatively loaded values are not even propagated to other instructions to form addresses for dependent loads.

The key insight of our work is that if we can safely predict the addresses for such dependent loads, we can then issue them without leaking speculatively loaded values. By safely predicting addresses we mean that both the predictions and when those predictions are resolved, are completely independent of speculative data [54].

A Doppelganger Load stands in for a delayed dependent load by:

i) predicting the address of the *dependent load*,
ii) preloading the value into the load's destination register, and
iii) propagating the preloaded value when the load becomes safe (according to the underlying secure speculation model), if the predicted address matches the load's resolved address; otherwise, issuing the load with its resolved address when it is safe according to the underlying secure speculation model.

4.2 Doppelganger Loads for NDA-P and STT

Let us consider the security for Doppelganger Loads when used in NDA-P and in STT. A doppelganger is supposed to stand in for a delayed dependent load that is a potential transmitter. But how do we know which loads are potential transmitters? The answer is that we don't. We try to *opportunistically* produce a doppelganger for any load we can (details can be found in Section 5). If we fail to produce a doppelganger for a load, that load falls under the normal operation of the secure speculation scheme (NDA-P or STT).

Security is compromised if a dependent load is squashed as part of mis-speculated execution: If the dependent load has made any visible changes in the microarchitectural state, it may have leaked information that encodes a speculatively loaded value (a secret). Secure speculation models such as NDA-P and STT ensure that the execution of a dependent load will be delayed, so it will not make any microarchitectural changes before it gets squashed. However, its doppelganger may have made changes in microarchitectural state, i.e., its doppelganger might have missed in the cache.

To reason about the security of a doppelganger load we can view it as a new *implicit channel*, Figure 2, one that is both prediction-based and resolution-based, as defined by Yu et al. [54]. As a *prediction-based implicit channel*, it is eliminated by preventing speculative data from affecting the state of the predictor. As a *resolution-based implicit channel*, it is eliminated by delaying the effects of the implicit `imp_if` until the predicate `(ap!=r1)` becomes non-speculative

```
// Conventional load
PC1:  load  r2, [r1]

// Load with its Doppelganger
PC1: {
    ap = predict(PC1)    // ap stored in LQ[PC1]
    load  r2, [ap]       // Doppelganger issues
    impl_if (ap != r1)
        load  r2, [r1]   // Load re-issues
}
```

Figure 2: A doppelganger load as an implicit channel. "ap" means address prediction and is stored in the LQ entry. Note that the doppelganger load and the re-issued load are one and the same instruction and use the same physical destination register.

— in other words, the address of the original load, r1, becomes non-speculative (untainted in STT or propagated in NDA-P).

The implicit channel created by the doppelganger is safe since neither the address prediction nor its resolution are dependent on speculative values. Thus, the doppelganger itself does not leak any information.

Now let us consider if seeing only the doppelganger, i.e., the conventional load gets squashed, or if seeing both a mispredicted doppelganger and the load, i.e., the load commits and is on the correct path of execution, can leak any information.

First, we consider the case where a dependent load is squashed, but its doppelganger missed in the cache, i.e., made changes in the microarchitectural state. In this case, we are safe because the only information that has leaked is the predicted address, and that cannot depend on speculative values. Note that it does not matter if it is a "correct" or a "wrong" prediction (in the sense of matching the address of the load) because there is no way to tell.

Second, if a dependent load eventually commits, we might observe in the microarchitectural state two misses instead of one, but that does not constitute speculative information leakage. In order to observe two misses, one for the dependent load and one for its doppelganger, instead of just one, it is necessary for the dependent load to commit (become non-speculative) in which case all older instructions must have also committed (become non-speculative). Thus, it is impossible to infer an illegally-accessed speculative value just by seeing a dependent load and its doppelganger as two distinct misses, because that can happen only in correct execution.

4.3 Doppelgangers in other explicit or implicit channels

Changes affected by their doppelgangers (e.g., a cache miss with a wrong prediction) are inconsequential, as the address prediction cannot be dependent on speculatively loaded values. But can such doppelganger changes be employed to leak information in other explicit or implicit side-channels?

In STT, any prediction-based or resolution-based implicit channel (in which doppelgangers could be used as tracers) is secret-independent. In NDA-P, the same is ensured because of its no-propagation policy. In DoM, we discuss how we ensured it in Section 4.6. For example, in all three schemes, STT, NDA-P, and DoM, we ensure that branches are taken and resolved in a secret-independent way. Thus, using doppelgangers to reveal the execution path, actually reveals the branch prediction, not any speculatively-loaded secret.

```
if (r1 < size) {
      load  r2, [r1]
      store r3, [r2]
PC3:  load  r5, [r4]
}
```

(a) Store-to-load forwarding

```
if (r1 < size) {
    load  r2, [r1]
    store r3, [r2]
PC3:{ // Load behaviour
    impl_if {r2 == r4)
        r5 = r3
    impl_else
        load  r5, [r4]
    }
}
```

(b) Implicit channel

```
if (r1 < size) {
    load  r2, [r1]
    store r3, [r2]
PC3:{ // Behaviour of a single load instruction
    ap = predict(PC3)   // ap stored in LQ[PC3]
    load  r5, [ap]      // Doppelganger issues
    impl_if (r2 == ap) // Store-to-load forward
        r5 = r3         // 1) Update preloaded value
    impl_if (ap != r4) // Address misprediction
        impl_if (r2 == r4)
            r5 = r3     // 2) Update preloaded value
        impl_else
            load  r5, [r4] // Load re-issues
    }
}
```

(c) Doppelganger with store-to-load forwarding

Figure 3: Store-to-load forwarding with Doppelganger Loads.

4.4 Doppelganger Loads and Store-to-load forwarding

Store-to-load forwarding (assuming memory dependence prediction [14, 31]) is another implicit channel (both prediction-based and resolution-based) as shown by Yu et al. [54], see Figure 3a and 3b. The interaction of doppelganger loads with store-to-load forwarding must not leak any information. Information leakage can happen, for example, if a store can make a doppelganger invisible by passing it the store value instead of letting it go to memory. In such a case, the attacker knows that the store address matches the predicted address. Because doppelganger loads are not delayed by the secure speculation mechanisms, preventing them from appearing can expose information that normally should be protected. Thus, doppelganger loads issue independently of store-to-load forwarding. Store-to-load forwarding takes place transparently by replacing the doppelganger preloaded value with the store value. This is similar to the store-to-load forwarding optimization proposed by Yu et al. [54], but for the doppelganger load. The combined store-to-load forwarding implicit channel with the doppelganger implicit channel is shown in Figure 3c. For correct store-to-load forwarding, we discern two cases:

(1) A store with a resolved address matches the predicted doppelganger address: In this case the doppelganger is issued but the value that is preloaded into the register is that of the older resolved store, not the one that comes from memory. This happens transparently and cannot be observed, since the doppelganger does not propagate its results.
(2) A doppelganger bypasses older store(s) with unresolved addresses: In conventional store-to-load forwarding, a resolved address of an older store squashes all issued younger loads with the same address. This is not necessary for a doppelganger where the predicted address matches the store address, since the doppelganger does not propagate its results. Instead, it is enough for the store forwarding to override the doppelganger register preload.

In either case, the resulting behavior is that the doppelganger appears in memory but the resulting preloaded value is that of a matching older store (if there is one), not the value that comes from memory. The behavior of the actual load remains unaffected with respect to both the doppelganger and the store-to-load forwarding implicit channels.

Because a doppelganger does not propagate its preloaded (or store-to-load forwarded) value before its load would propagate its value under NDA-P or STT, the rest of the protection mechanisms of the underlying secure speculation scheme remain unaffected. Specifically, for STT and NDA-P, both implicit channel protections and explicit channel protections remain in full force and with full scope.

4.5 Doppelganger Loads and Memory Consistency

In general, memory consistency is maintained by allowing speculative reordering of instructions and squashing in case such reordering is observed by another core by invalidations that reach the L1 or core [20]. Invalidations snoop a core's load queue (LQ) and squash matching loads that have executed out-of-order. This consists of an implicit channel [54]. The predicted address of a doppelganger load can be matched in the LQ by an invalidation, but the doppelganger itself is not squashed. The invalidation is noted and takes effect when the preloaded data is propagated, if the doppelganger has executed out-of-order with respect to older loads. If the address of the doppelganger was mispredicted, then the invalidation is ignored. The load's behavior, after the doppelganger validation, remains the same with respect to invalidations, as in the underlying secure speculation models.

4.6 Doppelganger Loads for DoM

In contrast to STT and NDA-P, DoM has no notion of dependent loads, but instead follows an entirely different philosophy. DoM prevents speculative load transmitters from leaking in the memory hierarchy side-channel, by simply delaying all microarchitectural change in the memory hierarchy (i.e., delaying L1 misses[2]) until it is safe to do so. Furthermore, DoM does not distinguish between speculatively loaded secrets and non-speculatively loaded secrets, thereby providing *register protection* — STT and NDA-P do not.

[2]DoM also delays replacement update for hits until it is safe to do so.

```
if (mispredict) {
    // hit -- DoM allows
    load  r1, [secret]
    if (r1) {
        // AP and Miss
        load  r2, [X]
    } else {
        // AP and Miss
        load  r3, [Y]
    }
}
```

(a) Secret loaded speculatively hitting in the L1D cache.

```
// Secret in register
load  r1, [secret]
if (mispredict) {
    if (r1) {
        // AP and Miss
        load  r2, [X]
    } else {
        // AP and Miss
        load  r3, [Y]
    }
}
```

(b) Secret loaded non-speculatively into a register.

Figure 4: Examples of Doppelganger Loads in implicit channels.

Because of these differences, allowing a doppelganger (address-predicted) load to miss in DoM is predicated upon conditions that ensure that no leakage occurs, as per the DoM threat model.

Figure 4 demonstrates two examples of Doppelganger Loads in implicit channels. In the first example (Figure 4a), a secret is speculatively loaded into `r1` (allowed by DoM), passed as a predicate to the `if` (similarly to [1, 2]) and leaked by allowing one of the two independent Doppelganger Loads to cause a miss on a *distinct prediction address*. DoM normally does not allow leakage through the memory hierarchy, so the same example would be protected. The second example (Figure 4b) is similar, but now shows how a non-speculative secret loaded in a register is leaked. Again, this breaks DoM's register protection.

It is clear from these examples that one can use the visibility of doppelganger misses to leak DoM secrets via implicit channels. Such implicit channels can be formed by explicit branches (e.g., the `if(r1)` branch in Figure 4), by implicit branches (e.g., store-to-load forwarding as in Figure 3b), or the implicit channel of the doppelganger itself (see Figure 2). DoM does not protect against implicit channels by default, because it expects that all speculative change in the memory hierarchy will be delayed. Our insight is that DoM with Doppelganger Loads is vulnerable to implicit channels, as now there can be speculative change from the Doppelganger Loads.

In DoM, we must eliminate implicit channels while allowing address-predicted doppelgangers to effect changes in the memory hierarchy. We accomplish this as follows:

- The explicit branch channel is handled in the same way as proposed by Yu et al. [54] by resolving branches in order, i.e., only once a branch becomes non-speculative.
- The implicit doppelganger side-channel is handled by only resolving the `impl_if(ap != r1)` in Figure 2 and by only propagating preloaded values that missed in the L1 cache once the load becomes non-speculative. In other words: Doppelgangers that miss in the L1 cache behave as DoM misses by propagating only when the load is non-speculative; doppelgangers that hit in the L1 cache behave as DoM hits, propagating when the address is validated.
- Finally, the store-to-load forwarding side-channel is handled by never propagating store-to-load forwarded values until they would be visible by the underlying DoM scheme,

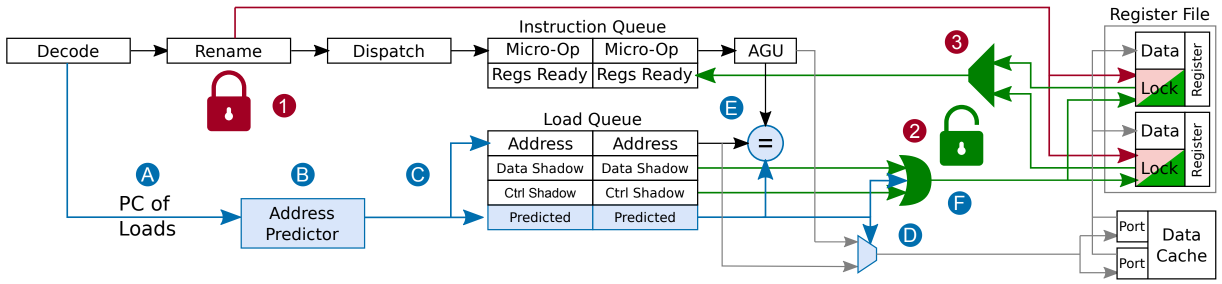

Figure 5: Doppelganger Loads for NDA-P: New structures required by the mechanism are highlighted in blue. Mechanisms required by NDA-P are highlighted in red and green for lock/unlock respectively.

which is the natural behavior of doppelgangers in the first place. In other words: Doppelgangers that hit in the L1 cache propagate the store value once the address prediction is verified; and doppelgangers that miss in the cache propagate the store value once the load becomes non-speculative, i.e., at the same point in time as the original DoM accesses would occur if they were not address predicted.

5 IMPLEMENTATION DETAILS

First, we detail how Doppelganger Loads works when used with NDA-P. Implementations for STT and DoM differ slightly, and are discussed later. The goal of this section is to demonstrate that Doppelganger Loads can be implemented on top of secure speculation schemes with complexity-effective modifications of the existing apparatuses and negligible additional hardware cost.

Figure 5 shows an integrated address predictor employed on top of a slightly modified NDA-P scheme. The green and red components show the implementation of NDA-P, where if there exists unresolved speculation at the time that a load instruction enters rename, its destination registers are locked, as shown by ①. This prevents the register from being propagated, until the lock is freed. Instead of using NDA's speculative tracking within the ROB, we use *shadow tracking*, first proposed for Ghost Loads [39] and subsequently used in DoM [40]. For our purposes we focus on tracking speculation originating from unresolved control flow, and unresolved store addresses. When there are no such shadows for a given load, its output would normally be unlocked, as shown by ②, and then propagated to dependent instructions, as shown by ③.

The address predictor is integrated with the front-end and load queue (Figure 5). Load instructions are known at the decode stage of the processor. The address predictor is both indexed and tagged with the load PCs, as shown by Ⓐ. Full PC tags are used to prevent aliasing. The address predictor Ⓑ is capable of processing as many PCs as there can be loads in the decode stage. The predicted addresses are sent to the load store unit (LSU), storing each prediction as the address of the associated load instruction in the load queue and setting the *predicted* bit, Ⓒ. The address predictor may fail to predict an address for a given PC, in which case no address is stored for the instruction and the predicted bit is not set.

In a given cycle, if there are fewer issued memory requests than the system is capable of issuing, available slots will be filled by predicted addresses, as shown by Ⓓ. Issued predicted addresses will set their corresponding load-queue entries to *executed*. Non-predicted addresses are always prioritized for execution, as, if we are on the correct path of execution, they are likely to be on the critical path.

When a load address is resolved (generated by an address generation unit) for a load with a doppelganger predicted address, a comparison is performed between the generated and the predicted address stored in the load queue, as shown by Ⓔ. If the addresses match, the prediction is correct and the predicted bit is unset. If the addresses do not match, the load queue entry is updated with the resolved address, and the executed and predicted bits are cleared. The load is replayed. Any response to the memory request that used the incorrect predicted address will be discarded.

If a response to a load using a predicted memory address comes to the core before the prediction is verified, the loaded value is written to the destination register of the load. However, the register is not propagated as ready until both the address is verified (i.e., the predicted bit is cleared), and the load is non-speculative, with speculation usually being the last to resolve Ⓕ. Note that this adds another check to the freeing mechanism of ②. If an older store instruction aliases with a doppelganger load, the store value is written to the preloaded register (instead of the memory value) without being propagated, ensuring correct store-to-load forwarding (see Section 4.4). Key to the security of the whole approach is that the address predictor is trained (updated) strictly by non-speculative loads when they commit (not shown in Figure 5).

5.1 Implementation Cost

The advantage of this scheme is multifold:

- A load and its doppelganger share the same load queue (LQ) entry, and by using the address slot in the LQ entry until the actual address is resolved, we ensure that predicted addresses use no additional space.
- The load and its doppelganger share the same physical destination register, which means rename is not modified and no additional registers are required. If the doppelganger is mispredicted, the preloaded value is always discarded before the load is re-issued, which results in requiring only one physical destination register at any given time.
- When the address prediction is correct, no extra resources are used, except those necessary for the address predictor.

- When the address prediction is incorrect, an extra address translation and memory request is issued, but no extra storage is necessary, and there is no need to squash any dependent instructions. This ensures that the penalty of incorrect address predictions remains small.
- The address predictor can be shared with a conventional strided prefetcher, with the only difference that the current address, instead of a future load address, being predicted.
- No modifications are needed to the memory hierarchy. A doppelganger access behaves exactly as any other memory access in the memory hierarchy.

The address predictor in this paper is deliberately chosen to deliver just the ground performance level that address prediction can provide. We do this to clearly establish the validity of our approach in relation to other approaches. There are many avenues worth exploring to further improve the benefit that such a scheme can provide, least of all by improving the coverage and accuracy of the predictor itself. We use a simple PC-based stride predictor that is unable to capture many address patterns that a more advanced predictor, such as for example a "bouquet" of predictors [33], would be able to find. However, it comes "for free" as it can be implemented simply as modification of an ubiquitously present PC-based stride prefetcher. More specifically, in "address prediction mode" the prefetcher is tasked to predict the address of the current instance of the load based on its history, while in "prefetching mode" it predicts future instances of the load based on the current (resolved) load address. Other changes may also be necessary for security, e.g., using full PC tags to prevent aliasing in prefetcher/predictor entries. Area saving optimizations such as the ones used in [45] can be used to cut down cost.

5.2 Modifications for STT

STT does not delay propagation at the source, but instead taints instruction output(s) and delays execution of transmitting instructions that depend on tainted data. The locking mechanism is still in-place for address-predicted loads. When a predicted load is verified to have the right address, it can be propagated immediately, but its data taints as it would normally under STT. If the prediction is incorrect, a load is issued if its operands are untainted, or whenever they become untainted. We investigate the performance impact of delaying address-predicted loads until they are non-speculative in Section 7.

5.3 Modifications for DoM

The original DoM does not delay the propagation of the output of any instruction, but instead delays speculative loads that miss in the L1 cache. The locking mechanism is in-place for only the address-predicted part of the load queue. The duration for which the lock remains locked depends on if the doppelganger hits in the L1 cache or not. If it hits in the cache, then the lock is released as soon as the predicted address is validated, i.e., corresponding to the same behavior as a conventional DoM load that hits in the L1 cache. If it misses in the cache, the lock is only released once the corresponding load becomes non-speculative, i.e., at the same time as a conventional load that missed in the L1 cache is re-issued according to DoM.

Table 1: System Configuration

Processor	
Decode width	5 instructions
Issue / Commit width	8 instructions
Instruction queue	160 entries
Reorder buffer	352 entries
Load queue	128 entries
Store queue/buffer	72 entries
Address predictor/prefetcher	1024 entries, 8-way, 13.5 KiB storage
Memory	
L1 D cache	48KiB, 12 ways
Access latency	5 cycles roundtrip
Number MSHRs	16
Private L2 cache	2MiB, 8 ways
Access latency	15 cycles roundtrip cache
Shared L3 cache	16MiB, 16 ways
Access latency	40 cycles roundtrip cache
Memory access time	13.5ns

To protect DoM from implicit channels, all branches are resolved in order and the second load of mispredicted doppelgangers are only issued once the load is non-speculative. We achieve this without introducing any taint tracking mechanisms, instead relying solely on the existing DoM shadow tracking mechanism to: i) delay branch resolution until a branch is no longer covered by a shadow; and ii) delay the issuing of the load of a mispredicted doppelganger until the load is no longer under a shadow. This mechanism is similar to how DoM, delays non-predicted loads that miss in the L1 cache. This protects DoM with Doppelganger Loads from implicit channels in the form of: i) explicit branches dependent on a secret; ii) implicit store-to-load forwarding branches; and iii) implicit doppelganger address-prediction branches.

6 METHODOLOGY

To compare our scheme to the current state-of-the-art, we implement NDA-P, STT, and DoM in a common simulation infrastructure using the gem5 [11] simulator, version 22. We use the SPEC CPU2006 [15] and CPU2017 [16] benchmark suites, as a representative for single-threaded programs. We run all inputs for each benchmark. We run detailed simulations using the out-of-order (o3) CPU, in both cases using simulation points. The simulation points were taken using simpoint profiling [44] of the first 100 billion instructions, with up to five simpoints per benchmark run, in which simpoint intervals were 100 million instructions and a warm-up period of one million instructions was present before execution. SPEC CPU2006 was run using the syscall emulation mode, while SPEC CPU2017 was run using the full system mode.

We configure the gem5 o3 CPU to resemble the hardware configuration of the IceLake CPU, including properties such as ROB size, decode width, and load queue size. Additionally, we change some other properties within the default configuration for gem5 to be in line with current high performance processor, such as increasing the amount of MSHRs for the L1 instruction and data caches from four to sixteen (Table 1).

We evaluate the following schemes:

- **Unsafe Baseline:** A baseline out-of-order processor that is not protected from speculative side-channel attacks. Secrets can leak through explicit and implicit channels.
- **Unsafe Baseline + Address Prediction:** Same as baseline, but with address prediction enabled, to show the performance gain for non-secure speculative execution schemes.
- **NDA-P:** Independent speculative loads are allowed to issue to memory as normal, and complete, but value propagation is delayed until the load is non-speculative.
- **NDA-P + Address Prediction:** Same as NDA-P, but with address prediction enabled. Loads cannot propagate before address is verified and load is non-speculative.
- **STT:** Speculative Taint Tracking mechanism enables speculative loads to complete, but taints the destination registers. Taints propagate through instructions. Tainted registers cannot be used for explicit or implicit channels.
- **STT + Address Prediction:** As STT, but with address prediction enabled. An address predicted loaded value propagates and taints as soon as the prediction is verified and gets untainted once the input operand is untainted.
- **DoM:** Speculative loads are issued to memory as normal, but are delayed if they miss in the L1 cache, and re-issued once they are non-speculative.
- **DoM + Address Prediction:** As DoM, but with address prediction enabled. Correctly predicted loads that hit in the L1 cache propagates as soon as the address is verified. Correctly predicted loads that miss in the cache propagates when the load becomes non-speculative.

In all evaluated schemes, speculation is tracked through the use of control and data shadows [39, 40] and feature a PC-based stride prefetcher with an 8-way associative, 1024-entry structure. Designs with address prediction have both a prefetcher and address predictor that share the same sized prefetcher structure, as described in Section 5.1.

7 EVALUATION

Our main performance results are shown in Figure 6. The first set of benchmarks belong to the SPEC CPU2006 suite, while the second set of benchmarks, belong to the SPEC CPU2017 suite. Some benchmarks belonging to these suites did not run on the unsafe baseline processor due to various issues, and are therefore excluded from the evaluation.

As shown by the geometric mean (GMEAN), address prediction provides a noticeable speedup for all the surveyed secure schemes. NDA-P achieves secure execution at 88.7% of baseline performance, while address prediction improves performance to 93.5% of baseline, reducing the average slowdown by 42.0%. STT, which has the least slowdown of the three evaluated schemes at 90.5%, achieves a similar speedup, improving performance to 95.1% with address prediction, reducing the slowdown of the scheme by 48.2%. Note that the simpler NDA-P with address prediction outpaces the more complex STT. DoM, which has the lowest performance at 81.8% of baseline, improves to 87.3% with address prediction, reducing the total slowdown by 30.3%.

Unsafe Baseline + AP: We also evaluate enabling address prediction for the unsafe baseline, but this achieves a geomean performance improvement of 0.5% (thus, this case is omitted from the graphs). Similar register prefetching, by Shukla et al., demonstrates that address prediction [5] gives modest gains [45]. Address prediction does not always enhance a conventional OoO core but, in our case, it has the potential to recover MLP that is lost due to security.

Coverage and Accuracy: Figure 7 shows the address predictor coverage and accuracy for DoM-AP as a representative for all the schemes. The geomean coverage and accuracy are all within 1% of each other between the evaluated schemes, as are the results for individual benchmarks, which is expected as address prediction is trained on the same non-speculative data (i.e., correct path execution), and encounters the same instructions, with only minor timing differences. For nearly all benchmarks, the accuracy of the predictor remains very high, typically at or above 90%. For a few, such as `xalancbmk_s` and `exchange2_s`, the accuracy is noticeably lower, down to just under 60% and 80%, respectively.

A limitation of our naïve address predictor is its low coverage. At most, benchmarks achieve a 49% coverage (`hmmer`), while the majority of benchmarks are around the geomean of 35% coverage. The low coverage is especially difficult in benchmarks such as `mcf` (9%), in which the low coverage means that despite the relatively high accuracy of the predictor, there is only a limited performance improvement. Generally speaking, the higher coverage correlates positively with the performance gain from address prediction, although this varies. However, looking at the predictor performance and the overall performance of address prediction, there is no clear correlation, e.g., `astar` has more than 35% of all loads correctly predicted, yet receives only a very minor performance gain.

We highlight two benchmarks, `omnetpp_s` and `xalancbmk_s`, which experience a performance penalty with address prediction. For `omnetpp_s`, the slowdown (0.4% for NDA, 1.2% for STT, and 1.5% for DoM) is due to the around 10% increase in L2 accesses, which indicates that the predictor is negatively affecting the amount of useful data in the L1 cache. For `xalancbmk_s`, the slowdown is caused by the noteworthy increase in L1 traffic. For this benchmark, the coverage remains decent, but the accuracy is very low, indicating that the benchmark is not amenable to this form of address prediction. The result is a large increase in the amount of traffic to the L1 cache, which negatively impacts the performance of DoM (-3.2%) while still improving the performance of NDA (+1.9%) and STT (2.7%). Although coverage and accuracy are shared across the three schemes, DoM is uniquely dependent on hitting in the L1 cache to achieve high performance, and the flooding of the cache incurs a performance detriment.

Some benchmarks achieve only a minor speedup with address prediction, such as `sjeng`, `gromacs`, and `wrf`, and a majority of the CPU2017 suite. The lack of improvement to most of the SPEC2017 suite is largely because the default schemes already have a low overhead on these benchmarks. For `bzip2` and `gcc`, there is a considerable speedup for all schemes, and a noticeable increase in the L1 cache accesses. Importantly, there is no increase in the number of accesses to the L2 cache, indicating that address-predicted loads to the L2 cache and further down the memory hierarchy are generally correct. Unlike with `xalancbmk_s` for DoM+AP, this increase is not enough to negatively affect the performance of the L1 cache.

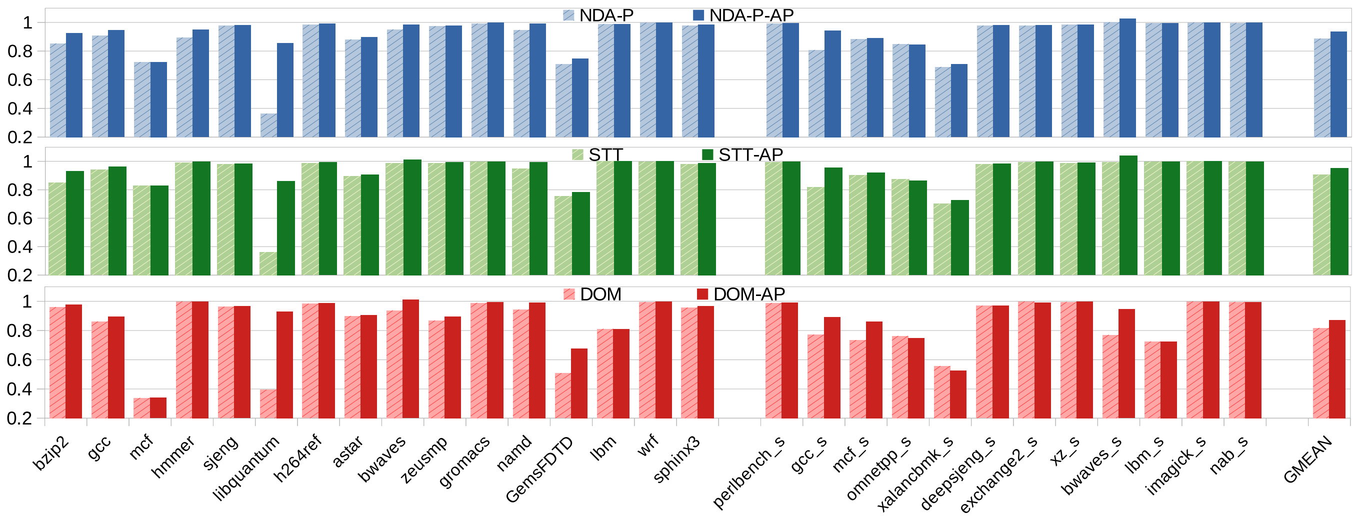

Figure 6: Normalized IPC to baseline of the NDA-P, STT, and DoM schemes enhanced with Doppelganger Loads.

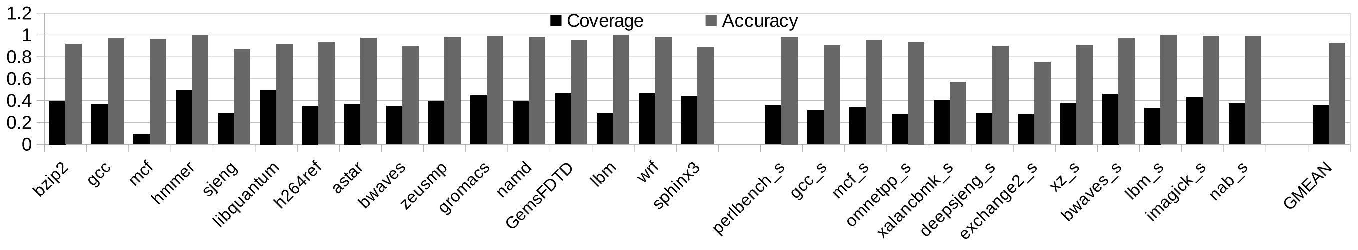

Figure 7: Coverage and Accuracy for address prediction under DoM (similar results observed for NDA-P, STT).

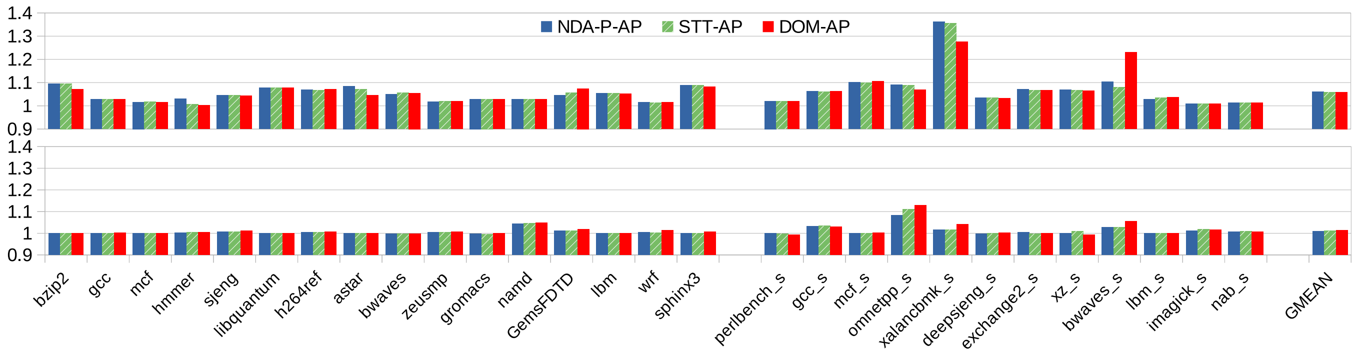

Figure 8: Normalized L1 and L2 accesses. (Upper graph: L1 cache, lower graph: L2 cache.)

For `GemsFDTD` there is a noticeable performance difference between DoM and the other two schemes. As DoM is normally unable to access L2, this indicates that address prediction is capable of providing additional speculative MLP in limited forms for `GemsFDTD`.

The standout benchmark for address prediction is `libquantum`, which recovers 77-88% of total baseline performance with simple address prediction. As evidenced in the lack of increase in the number of accesses to the L2 cache, and the very minor increase in accesses to the L1 cache, this is because the address predictor is able to accurately predict the addresses of critical loads, reflected in its high coverage and quite high accuracy.

Overall, address prediction as employed in this work demonstrates no significant benefits for the baseline but highlights its usefulness for the secure speculation schemes. Normally, a core should make available as much MLP as possible, but the constraints imposed by secure speculation severely constrain it. Address prediction recovers a sizable part of the MLP in a safe, threat-model transparent, way.

8 RELATED WORK

A closely related work is SDO [53], which introduces data-oblivious execution in STT by using prediction (non-speculatively trained). Specifically, SDO allows loads (oblivious loads) to speculatively access a value from a predicted location in the cache hierarchy, continuing execution with null-operators in the case of a mispredict, and not squashing before the squashing is non-speculative.

Our approach differs in many ways:

- The SDO design paradigm requires changes to the memory hierarchy — instead, we investigate how a prediction can unlock more MLP with changes only to the core.
- Because, in SDO, the addresses of the predicted loads can be secret value dependent, care must be taken so that predicted accesses are not visible in the memory hierarchy under any circumstances. This introduces significant complexity in the memory hierarchy implementation details. In contrast, in our approach, predicted addresses are secret value independent, and we can safely release them out in the open, including into the memory hierarchy.
- Oblivious loads require additional buffering — instead Doppelganger Loads preload the target register of the load.
- SDO introduces a new kind of predictor (cache-level prediction) — instead, we rely on existing predictors such as the ubiquitous PC-based stride prefetcher.
- Even with correct prediction, oblivious loads may need validation with one additional access because of memory consistency issues [53], similarly to the validation of invisible loads in InvisiSpec [52], thus, introducing further complexity. In contrast, address prediction for Doppelganger Loads does not raise any consistency issues.

There exist other approaches to recover some lost performance, both compiler-based [48, 56] and hardware-based [55].

InvarSpec [56] identifies instructions that are guaranteed to commit regardless of the outcome of speculation and lifts the defenses for these instructions. However, recent work [2] shows that it can be vulnerable to speculative interference attacks [8]. Tran et al. [48] use compile-time instruction re-ordering for early evaluation of conditions and addresses to enable the hardware to remove the control and exception shadows as early as possible. Compiler-based approaches are beyond the scope of our work.

Other approaches aim to ameliorate mitigation costs associated with speculative memory re-ordering (memory shadows). Pinned Loads [55] resolves memory violations as early as possible so that loads reach their visibility point earlier; Tran et al. [48] propose using non-speculative load-load reordering [37] for the same reason. These works are orthogonal to our approach and can be employed on memory shadows as deemed necessary.

Proposals must also be validated against forms of attacks and vulnerabilities, such as, store-to-load forwarding [54], Speculative Interference [8], Reorder Buffer Contention [2], unXpec [28], DOIN! [1], and DOLMA [30]. For Doppelganger Loads specifically, two types as important: attacks that use a secret-dependent branch for DoM [1, 2], that could inspire an attacker to leak a value by a secret-dependent prediction, yet we answer how this can be mitigated in Section 4.6, and store-to-load forwarding where it can make a doppelganger invisible by passing it the store value instead of letting it go to memory, yet we also answer that in Section 4.4.

Address Prediction: Address prediction was examined as an approach to hide load-access latency in in-order pipelines [9, 18] and to issue loads with unresolved addresses early [21]. More recently, Alves et al. use early address prediction to detect potential load-load reuse before the rename stage, thereby saving energy and time by using the physical register file as an indirect L0 cache [5]. Such research shows the value of knowing addresses early in the pipeline, and expounds on the much more regular nature of addresses compared to values [21]. The closest address prediction work to our work is "Register File Prefetching" [45] which shares many of the same characteristics (but ours yields lower performance for the baseline, considering the methodology—simulator, configuration, simpoints—and predictor configuration differences). To the best of our knowledge, however, ours is the first work that uses address prediction to enable *safe* execution of *delayed* loads in multiple secure speculation schemes, by ensuring that their addresses are independent of any speculatively-loaded secret.

Value Prediction: Unlike address prediction, value prediction suffers from predicted values being propagated before validation, incurring squashes on mispredicted values. Additionally, addresses are easier to predict than values [32, 43].

9 CONCLUSION

In this work, we introduce the Doppelganger Loads architecture, which enables dependent loads to be safely speculatively executed by predicting their addresses, thereby improving memory level parallelism (MLP) of secure speculation schemes that otherwise limit the MLP. We show that Doppelganger Loads can be integrated in existing secure schemes without changing the underlying threat model, i.e., Doppelganger Loads are threat-model transparent. Furthermore, we show how Doppelganger Loads can be implemented at low cost and complexity, without requiring any modifications to the memory hierarchy. Finally, our results show that a simple strided prefetcher can be used for address prediction to improve the performance of three secure speculation schemes, NDA-P, STT, and DoM. The potential to further improve performance by using a more advanced address predictor is left for future work.

ACKNOWLEDGMENTS

We thank Christos Sakalis for early contributions related to the idea and guidance on gem5 development. Simulations were executed on the NTNU EPIC compute cluster [46]. This work was supported in part by Microsoft Research through its EMEA PhD Scholarship Programme grant no. 2021-020, the Swedish Research Council (VR) grants 2018-05254 and 2022-04959, the VINNOVA grant 2021-02422, the SSF grant FUS21-0067, and Uppsala University funds for Cybersecurity.

REFERENCES

[1] Pavlos Aimoniotis, Amund Bergland Kvalsvik, Magnus Själander, and Stefanos Kaxiras. 2022. Data-Out Instruction-In (DOIN!): Leveraging Inclusive Caches to Attack Speculative Delay Schemes. In *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design*. 49–60. https://doi.org/10.1109/SEED55351.2022.00012

[2] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. 2021. Reorder Buffer Contention: A Forward Speculative Interference Attack for

Speculation Invariant Instructions. *IEEE Computer Architecture Letters* 20 (July 2021), 162–165. Issue 2. https://doi.org/10.1109/LCA.2021.3123408

[3] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 592–606. https://doi.org/10.1145/3466752.3480074

[4] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *Proceedings of the International Symposium on Computer Architecture*. 132–144. https://doi.org/10.1109/ISCA45697.2020.00022

[5] Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. 2021. Early Address Prediction: Efficient Pipeline Prefetch and Reuse. *ACM Trans. Archit. Code Optim.* 18 (June 2021), 39:1–39:22. Issue 3. https://doi.org/10.1145/3458883

[6] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Isolating Speculative Data to Prevent Transient Execution Attacks. *IEEE Computer Architecture Letters* 18 (July 2019), 178–181. Issue 2. https://doi.org/10.1109/LCA.2019.2916328

[7] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*. IEEE Computer Society, 151–164. https://doi.org/10.1109/PACT.2019.00020

[8] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative interference attacks: breaking invisible speculation schemes. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 1046–1060. https://doi.org/10.1145/3445814.3446708

[9] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. 1999. Correlated load-address predictors. *ACM SIGARCH Computer Architecture News* 27 (May 1999), 54–63. Issue 2. https://doi.org/10.1145/307338.300984

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 785–800. https://doi.org/10.1145/3319535.3363194

[11] 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* 39 (May 2011), 1–7. Issue 2. https://doi.org/10.1145/2024716.2024718

[12] 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 *Proceedings of the USENIX Security Symposium*. 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[13] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 607–622. https://doi.org/10.1145/3466752.3480068

[14] G.Z. Chrysos and J.S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*. 142–153. https://doi.org/10.1109/ISCA.1998.694770

[15] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmark Suite. http://www.specbench.org/cpu2006/

[16] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017 Benchmark Suite. http://www.specbench.org/cpu2017/

[17] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky Frontends: Security Vulnerabilities in Processor Frontends. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 53–66. https://doi.org/10.1109/HPCA53966.2022.00013

[18] R. J. Eickemeyer and S. Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development* 37 (July 1993), 547–564. Issue 4. https://doi.org/10.1147/rd.374.0547

[19] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Proceedings of the ACM/IEEE Design Automation Conference*. Association for Computing Machinery, 1–6. https://doi.org/10.1145/3316781.3317914

[20] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. *Two Techniques to Enhance the Performance of Memory Consistency Models*. Computer Systems Laboratory, Stanford University.

[21] José González and Antonio González. 1997. Speculative Execution via Address Prediction and Data Prefetching. In *Proceedings of the ACM International Conference on Supercomputing*. 9. https://doi.org/10.1145/263580.263631

[22] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1–6. https://doi.org/10.1145/3316781.3317903

[23] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. 2021. UC-Check: Characterizing Micro-operation Caches in x86 Processors and Implications in Security and Performance. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 550–564. https://doi.org/10.1145/3466752.3480079

[24] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. 2020. ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage. In *Proceedings of the IEEE Secure Development Conference*. IEEE Computer Society, 96–107. https://doi.org/10.1109/SecDev45635.2020.00029

[25] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 974–987. https://doi.org/10.1109/MICRO.2018.00083

[26] Paul Kocher, Jann Horn, Anders Fogh, 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 *Proceedings of the IEEE Symposium on Security and Privacy*. 1–19. https://doi.org/10.1109/SP.2019.00002

[27] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 39–53. https://doi.org/10.1109/SP40000.2020.00033

[28] Mengming Li, Chenlu Miao, Yilong Yang, and Kai Bu. 2022. unXpec: Breaking Undo-based Safe Speculation. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 98–112. https://doi.org/10.1109/HPCA53966.2022.00016

[29] 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 *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, 264–276. https://doi.org/10.1109/HPCA.2019.00043

[30] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *Proceedings of the USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin

[31] Andreas Ioannis Moshovos. 1998. *Memory Dependence Prediction*. Ph. D. Dissertation. University of Wisconsin.

[32] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. 2018. AVPP: Address-first Value-next Predictor with Value Prefetching for Improving the Efficiency of Load Value Prediction. *ACM Transactions on Architecture and Code Optimization* 15 (Dec. 2018), 49:1–49:30. Issue 4. https://doi.org/10.1145/3239567

[33] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *Proceedings of the International Symposium on Computer Architecture*. 118–131. https://doi.org/10.1109/ISCA45697.2020.00021

[34] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *Proceedings of the International Symposium High-Performance Computer Architecture*. 428–439. https://doi.org/10.1109/HPCA.2014.6835952

[35] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the International Symposium on Computer Architecture*. Association for Computing Machinery, 685–698. https://doi.org/10.1145/3470496.3527429

[36] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *Proceedings of the International Symposium on Computer Architecture*. 14. https://doi.org/10.1109/ISCA52012.2021.00036

[37] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *ACM SIGARCH Computer Architecture News*, Vol. 45. 187–200. https://doi.org/10.1145/3140659.3080220

[38] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 73–86. https://doi.org/10.1145/3352460.3358314

[39] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost loads: What is the cost of invisible speculation?. In *Proceedings of the ACM International Conference on Computing Frontiers*. Association for Computing Machinery, 153–163. https://doi.org/10.1145/3310273.3321558

[40] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*. 723–735. https://doi.org/10.1145/3307650.3322216

[41] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2020. Understanding Selective Delay as a Method for Efficient Secure Speculative Execution. *IEEE Trans. Comput.* 69 (Nov. 2020), 1584–1595. Issue 11. https://doi.org/10.1109/TC.2020.3014456

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 753–768. https://doi.org/10.1145/3319535.3354252

[43] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. 2017. Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 423–435.

[44] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 45–57. https://doi.org/10.1145/605397.605403

[45] Sudhanshu Shukla, Sumeet Bandishte, Jayesh Gaur, and Sreenivas Subramoney. 2022. Register file prefetching. In *Proceedings of the International Symposium on Computer Architecture*. Association for Computing Machinery, 410–423. https://doi.org/10.1145/3470496.3527398

[46] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. 2022. EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure. (Feb. 2022). https://doi.org/10.48550/arXiv.1912.05848 arXiv:1912.05848

[47] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 395–410. https://doi.org/10.1145/3297858.3304060

[48] Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. 2020. Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, 241–254. https://doi.org/10.1145/3410463.3414640

[49] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 572–586. https://doi.org/10.1145/3352460.3358306

[50] Wm A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23 (1995), 20–24. Issue 1. https://doi.org/10.1145/216585.216588

[51] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *Comput. Surveys* 54 (May 2021), 54:1–54:36. Issue 3. https://doi.org/10.1145/3442479

[52] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[53] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *Proceedings of the International Symposium on Computer Architecture*. 707–720. https://doi.org/10.1109/ISCA45697.2020.00064

[54] 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 *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 954–968. https://doi.org/10.1145/3352460.3358274

[55] Zirui Neil Zhao, Houxiang Ji, Adam Morrison, Darko Marinov, and Josep Torrellas. 2022. Pinned loads: taming speculative loads in secure processors. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 314–328. https://doi.org/10.1145/3503222.3507724

[56] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1138–1152. https://doi.org/10.1109/MICRO50266.2020.00094