Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution

Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison[†], Christopher W. Fletcher
University of Illinois at Urbana-Champaign, [†]Tel Aviv University
{jiyongy2, nmantri2, torrella, cwfletch}@illinois.edu, mad@cs.tau.ac.il

***Abstract*—Speculative execution attacks are an enormous security threat. In these attacks, malicious speculative execution reads and exfiltrates potentially arbitrary program data through microarchitectural covert channels. Correspondingly, prior work has shown how to comprehensively block such attacks by delaying the execution of covert channel-creating instructions until their operands are a function of non-speculative data.**

This paper's premise is that it is safe to execute these potentially dangerous instructions early, improving performance, as long as their execution does not require operand-dependent hardware resource usage, i.e., is *data oblivious*. While secure, this idea can easily reduce, not improve, performance. Intuitively, data obliviousness implies doing the worst case work all the time. Our key idea to get net speedup is that it is *safe to predict what will be, and to subsequently perform, the work needed to satisfy the common case*, as long as the prediction itself does not leak privacy.

We call the complete scheme—predicting the form of data-oblivious execution—*Speculative Data-Oblivious Execution (SDO)*. We build SDO on top of a recent comprehensive and state-of-the-art protection called STT. Extending security arguments from STT, we show how the predictions do not reveal private information, enabling safe and efficient speculative execution. We evaluate the combined scheme, STT+SDO, on a set of SPEC17 workloads and find that it improves the performance of stand-alone STT by an average 36.3% to 55.1%, depending on the microarchitecture and attack model—and without changing STT's security guarantees.

***Index Terms*—Security, Speculative execution attacks, Hardware, Information flow**

I. Introduction

Spectre [26] and follow-up attacks [10, 14, 22, 25, 27, 29, 39, 47] represent an enormous threat to processor security. In these attacks, adversary-controlled transient instructions—i.e., speculative instructions bound to squash—access and then transmit sensitive program data over *microarchitectural covert channels* (e.g., the cache [48], port contention [10]). For example in Figure 1, speculative execution bypasses a bounds check due to a branch misprediction and transmits secret data behind that bounds check over a covert channel. Here, the attacker controls the value of `addr`, thus `val` can be any value in program memory and the covert channel can reveal arbitrary program data.

Prior work has pointed out that speculative execution attacks are broken into two components [24, 39]. First, a secret value is speculatively *accessed* and read into architectural state (e.g., a register) due to adversary-controlled speculative execution. For example, the load into array `A` in Figure 1 reads `val` even if `addr` $\geq$ 10 due to a branch misprediction. Second, that secret value is *transmitted/leaked* to the attacker through a *transmitter*, denoted `transmit(...)` in the figure. In this paper, we define a transmitter as any instruction whose execution creates operand-dependent hardware resource usage—i.e., creates a covert channel as a function of the transmitter's operand [51]. For example, the transmitter in Spectre Variant 1 is a load instruction, leaking the secret (load address) over a cache-based covert channel.

```
uint8 A[10];
void victim (size_t addr) {
    if (addr < 10) { // mispredicted branch
        uint8 val = A[addr]; // secret is accessed
        transmit(val); // secret is transmitted
    }
}
```

Fig. 1: Example speculative execution attack. `transmit(arg)`, called the transmitter, denotes any instruction which can create a microarchitectural covert channel, leaking `arg` to the attacker. Variables carrying potentially secret data are colored `green`. If the `if` condition is predicted as true, then the transmitter reveals `val` even though execution eventually squashes.

Using this distinction, prior work has endeavored to block leakage through transmitters. On one hand, *invisible speculation* [2, 23, 36, 37] attempts to execute transmitters "invisibly," e.g., by not modifying cache state for the load transmitter. This is efficient, however has security holes, e.g., due to data-dependent timing effects. On the other hand, *delayed execution* [6, 46, 51] schemes endeavour to completely block all covert channels by monitoring how secrets flow through instructions and delaying the execution of transmitters until their operands become a function of non-speculative data. While this idea can enable strong security, it can incur high overhead as delaying execution can stall the pipeline.

A. This Paper

The goal of this paper is to get the best of both of the above worlds: high performance and high security. The key idea is that it is safe to execute transmitters early, i.e., while their *operands* are still a function of speculative (sensitive) data, as long as their *execution* is made to not require operand-dependent hardware resource usage. We call this *speculative data-oblivious execution (SDO)*.

We start with a state-of-the-art defense framework against speculative execution attacks, called STT [51]. STT provides comprehensive protection for speculatively accessed data, but incurs overhead from delaying the execution of transmitters until their operands are a function of non-speculative data as discussed earlier. We will apply SDO to transmitters in STT, retaining that work's strong security properties while dramatically improving its performance. (See Section III for detailed background on STT.)

To illustrate SDO, consider a simple example where the transmitter in Figure 1 is implemented as a floating point instruction. Floating point instructions typically have operand-dependent behavior: if the operand is subnormal, the instruction may execute on a slow path (e.g., in microcode); otherwise it executes on a fast hardware floating point unit [5]. To simplify the example, we assume this creates two execution equivalence classes: slow and fast, respectively. This forms a covert channel. An attacker can infer which equivalence class a floating point operation belongs to by, e.g., monitoring program runtime [5, 39] and hardware resource usage, and this reveals information about the operand.

To be speculative data-oblivious, the starting point is to execute multiple copies—one for each equivalence class—of any transmitter that computes on speculative data. In our example, we execute two copies of speculative floating point instructions: one for the fast and one for the slow execution. Once both complete, it is safe to forward the result of whichever one was correct to younger, dependent instructions.

While the above idea is sufficient for security, it is low performance. Not only must we execute two versions of the floating point instruction, we must wait to forward the result until the slowest (subnormal) mode completes, to hide which version was actually needed.

The key idea to address this issue is to *predict* that one equivalence class (or a subset of classes) will be correct, and execute only those classes. An obvious pitfall with this idea is security: can the prediction itself reveal private information? Since we build on STT, however, the answer is no. STT automatically ensures that predictors' predictions, and when those predictions are resolved, are completely independent of speculative data.

With an "equivalence class predictor" in hand, we can safely speedup our floating point example. Assuming subnormal inputs are rare, we might statically predict the input will be normal. If the prediction was correct, we execute the floating point operation without delay and incur no overhead. If the prediction was incorrect (hopefully rare), we squash when the inputs become non-speculative and suffer a performance hit.

A remaining issue is: if we predict an equivalence class, as opposed to execute all equivalence classes (the naïve strategy), we will have computed an incorrect result if the prediction is incorrect. For security, we must ensure that any instructions receiving the result do not reveal whether it is correct or not. Fortunately, STT also addresses this issue through its tainting mechanism: by marking the output as tainted, no attacker will be able to learn any bit of the return value.

The above ideas are general, and apply to other types of transmit instructions and more sophisticated predictors. Beyond our simplified floating point example, the bulk of this paper is to design a novel speculative data-oblivious load operation. This is important for performance as prior work shows the lion share of overhead in blocking speculative execution attacks is due to loads [23, 37, 47, 51]. It is also non-trivial, as there are many ways a load can execute that can create covert channels. For example, a load may hit or miss at any cache or TLB level, contend for different resources such as uncore buses, cache banks, lookup DRAM, etc. To address these challenges, we develop a new speculative data-oblivious load operation (an "Obl-Ld" operation, for "oblivious load"), a novel "location predictor" to improve performance for Obl-Ld operations, and additional optimizations that allow us to safely return and forward load data to the pipeline earlier.

To summarize, this paper makes the following contributions.

1) We propose a novel framework enabling Speculative Data-Oblivious Execution (SDO), enabling safe execution of unsafe transmitters.
2) We propose a novel implementation of an SDO operation for speculative loads.
3) We evaluate SDO as an optimization to prior work STT, treating both loads and loads plus floating point operations as transmitters. We find that STT+SDO improves STT's performance on SPEC17 workloads by an average 36.3% to 55.1%, depending on the microarchitecture and attack model—and without changing STT's security guarantees.

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

II. Attack/Threat Model & Scope

We assume an adversary that can monitor any microarchitectural covert channel and induce arbitrarily speculative execution from anywhere in the system. For instance, the adversary may operate from within the victim program (*SameThread* [39]), an SMT sibling (*SMT*), or another processor core (*CrossCore*), and can monitor covert channels through the cache/memory system [26], data-dependent arithmetic [20], port contention [10], branch predictors [3], etc.

We adopt STT's scope of protecting speculatively-accessed data, i.e., blocking attacks involving doomed-to-squash access instructions. (Explained in more detail in Section III.) These are the most dangerous attacks, since such access instructions can often be maneuvered to form *universal read gadgets* [30] that read arbitrary memory locations (e.g., Figure 1). Protecting data produced by retired (or bound-to-retire) access instructions is out of scope. This is data that is legitimately accessed according to program semantics, and can therefore be reasoned about by programmers or compilers and blocked using complementary techniques (e.g., [50]).

III. Background: Speculative Taint Tracking

Speculative Taint Tracking (STT) [51] is a framework that protects *secrets*—which are defined to be data read by doomed-to-squash instructions (called *access* instructions)—from being leaked over *any possible microarchitectural covert channel*.

A. STT Details

STT monitors how secrets flow through the pipeline by tracking explicit information flows (instruction data dependencies) and applies a protection policy which is to delay an instruction's execution if that instruction can form a covert channel *and* receives a secret as an operand. Instructions whose execution creates operand-dependent hardware resource usage are called *transmit instructions*. Transmit instructions with secret operands are delayed until either the access instruction(s) that produced the secret becomes non-speculative or the execution squashes due to mis-speculation. Finally, STT applies a policy that eliminates implicit information flows through a combination of mechanisms that ensure that the program counter is never a function of a secret.

Covert Channels. STT classifies covert channels into two classes: explicit and implicit channels.[1] In an *explicit channel*, the secret is *directly* passed to a *transmitter*, which is any instruction whose execution can create operand-dependent hardware resource usage that reveals the secret. For example, loads are transmitters, as their execution makes address-dependent changes to the cache state. In an *implicit channel*, the secret *indirectly* influences how (or that) an instruction or several instructions execute, and these changes in resource usage reveal the data. For example, a branch instruction, whose outcome determines subsequent instructions and thus whether some functional unit is used. (Such branch-based implicit channels are used by NetSpectre [39] and SmotherSpectre [10] to trigger SIMD unit usage and port contention, respectively.)

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

Taint/Untaint Tracking. At a high level, STT features a taint propagation mechanism similar to prior work (e.g., [42]), and proposes a novel "untaint" mechanism to disable protection as soon as doing so is safe. Specifically: STT *taints* the output register of a speculative access instruction. The microarchitecture defines when to *untaint* the output of a speculative access instruction. This point depends on the attack model: In the *Spectre* model (which covers control-flow speculation), it is when all older control-flow instructions have resolved, and in the *Futuristic* model (which covers all forms of speculation), it is when the access instruction cannot be squashed. STT propagates taint/untaint information: the output register of a non-access instruction is tainted if and only if it has a tainted input register. Taint propagation is piggybacked on the existing register renaming logic in an out-of-order core, and is therefore fast. Untainting all dependencies of an access instruction that becomes non-speculative is more difficult, but STT has a fast untaint mechanism that performs untainting in a single cycle. STT does not maintain taint/untaint information in the cache/memory system, only in the physical (non-architected) register file.

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

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

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

STT's implicit channel rules eliminate all implicit channels by making the program counter register (PC) independent of tainted (speculatively accessed) data. STT enforces this invariant efficiently, without needing to delay execution of instructions following a tainted branch.

B. Major Takeaway: STT Makes Prediction Safe

An important corollary of the above is that predictor structures can remain enabled without leaking privacy. The reason is two-fold. Consider branch prediction as an example (an analogous argument follows for other predictors). First, STT ensures that the branch predictor state is never a function of secret data. Further, mis-predictions are squashed only when the prediction's outcome becomes safe to reveal. This policy means the predicted branch directions do not leak privacy and that "what instructions are fetched due to the predictions" is public information. In other words, the only possible source of privacy leakage is through transmitters. Second, STT ensures that if any transmitter fetched on the predicted path can leak

[1] Fundamentally, a microarchitectural covert channel can communicate a value when explicit information flow at the gate level changes as a function of that value [43, 50]. The abstraction proposed by STT provides a way to reason about when this will occur due to different instructions and microarchitectural optimizations.

privacy, then that transmitter is delayed until the threat passes. Note, this argument does not depend on whether the predictor mispredicts by mistake, is intentionally mistrained, etc.

C. SDO Motivation: Source of STT Overhead

The lion share of STT's overhead is due to blocking explicit channels, specifically, delaying execution of tainted loads. The STT paper reports that ignoring implicit channels reduces STT's average overhead on SPEC [21] and PARSEC [11] applications by just 1%–3% over the original 8.5%/14.5% overhead (depending on the attack model). The effect on outlier applications with higher than average overhead is similar. This result shows that STT's implicit channel protection mechanism is cheap and that obtaining better performance requires avoiding the delayed execution of tainted transmitters, particularly loads.

The goal of this paper is therefore to enable transmitters to safely execute early. That is, we will build on STT's mechanisms to block implicit channels and propagate untaint, and implement a new scheme for handling select high-overhead tainted transmitters (loads and floating point operations).

IV. Speculative Data-Oblivious (SDO) Execution

We now present a general methodology for designing an *SDO operation*, which is an SDO implementation for an arbitrary transmit instruction (transmitter). At a high level, an SDO operation is a new implementation of the transmitter that can be executed safely, even if its operands are tainted (Section III).

The microarchitect starts with a transmitter f which takes operands args and returns result, denoted result ← f(args). We will construct an SDO operation for f, denoted Obl-f. This is a two-step process (next two sub-sections). High-level intuition is given in Section I-A.

A. Design Data-Oblivious (DO) Variants

First, the microarchitect designs N data-oblivious variants (called *DO variants*) of f, denoted Obl-f1, . . . Obl-fN. We will see that how to choose N and how to design each variant creates a new design space with performance/design complexity trade-offs. Each variant i for $1 \leq i \leq N$ has the following signature:

$$\mathsf{success?}, \mathsf{presult} \leftarrow \mathsf{Obl\text{-}fi}(\mathsf{args}) \tag{1}$$

where success? is a boolean and presult is whatever datatype is returned by this instruction, i.e., the same return type as the original f. We will abbreviate success? ≡ true as success and success? ≡ false as "fail," where ≡ denotes an equality check that returns true/false.

Informally, each variant must be data oblivious, i.e., not reveal its argument over microarchitectural side channels. A given variant may not be able to return the correct result. For example, if the floating point operand is subnormal the variant for evaluating normal operands will return the wrong result (Section I-A). Each variant indicates whether it has returned the correct data with the success? flag.

We now formalize these requirements in two definitions:

Definition 1. ***(Functional correctness).*** *Consider Equation 1 for* $1 \leq i \leq N$*. For all possible* args*: If* Obl-fi(args) *returns* success*, then* presult ≡ f(args) *must hold. If this function returns* fail*, then* presult *is undefined (we assume* presult *is set to* ⊥*).*

Definition 2. ***(Security).*** *Consider Equation 1 for* $1 \leq i \leq N$*. For any two operand assignments* args *and* args′*: the execution of* Obl-fi(args) *and* Obl-fi(args′) *must create the same hardware resource interference (i.e., which is measurable as a microarchitectural side channel).*

Definition 1 is also important for security, as we will see in Section VII.

Note, these definitions do not require that for some argument args, there must exist an Obl-fi that will return success. For example, in our floating point example we may have $N = 1$, e.g., a DO variant for the fast mode and no DO variant for the slow mode. This is allowed because our scope only considers speculative execution attacks. Continuing the above example, having a single DO variant for floating point operations means we will necessarily see fail on a subnormal input. We will, however, be able to correct that situation by squashing the incorrect speculation when the input becomes non-speculative (see Section IV-C for details).

B. Design Predictor To Select Which DO Variant

Second, the microarchitect designs a *DO predictor*, which selects some DO variant Obl-fi ($1 \leq i \leq N$) to execute in place of f. Executing the DO variant in place of f will ensure args does not leak. That is, a DO predictor is a pair of functions:

$$i \leftarrow \mathsf{predict}(\mathsf{inp}) \tag{2}$$

$$\mathsf{update}((\mathsf{inp},\ actual\ i)) \tag{3}$$

where $1 \leq i \leq N$. predict, given an input (e.g., the PC), predicts which of the N DO variants is most likely and update updates the predictor state (if any) to influence future predictions. A prediction is considered correct iff Obl-fi returns success.

Importantly, the predictions made by the predictor, and the updates to the predictor must be a function of non-sensitive information. We will see how this is achieved in the context of malicious speculative execution next, in Section IV-C. The predictor's internal implementation can be arbitrarily simple or complex, and is free to select a different i for each dynamic instance of a given transmitter f.

C. Putting It All Together: Protecting Speculative Data

To protect speculative data, we combine DO variants and the DO predictor with STT (Section III). Pseudo-code for the complete construction, called an *SDO operation*, is given in Figure 2.

First, instead of delaying execution of a transmitter with tainted operands as in STT, the tainted transmitter f is unconditionally predicted on by the DO predictor and issued as a

DO variant (Lines 3-7). The case statement in the pseudo-code therefore corresponds to an implicit branch (Section III) with predicate success?. Likewise, each DO variant corresponds to a non-transmitter by Definition 2. We follow STT's principles (Section III-B) to make the DO predictor/implicit branch not leak privacy. That is, we require that predictor updates be a function of untainted data, e.g., the program counter/PC, and that predictor resolution/squashes be delayed until the implicit branch predicate becomes untainted (Lines 11-16).

Second, results are unconditionally forwarded to dependent instructions (regardless of success?) and tainted/protected by STT (Line 8). Once args becomes untainted, it is safe to reveal success? because it is a function of the prediction and args. By extension, it is safe to update the predictor and/or squash/re-issue the transmitter (since the implicit branch predicate is now untainted).

```
1  // Part 1: On issue of f with PC pc, with tainted operands args
2  Obl-f(args):
3    switch( predict(pc) ):
4      case 1: success?, presult ← Obl-f1(args) break;
5      case 2: success?, presult ← Obl-f2(args) break;
6      ...
7      case $N$: success?, presult ← Obl-fN(args) break;
8    return presult;
9
10 // Part 2: When args becomes untainted
11 if (success?): // Note: success? no longer tainted
12   update((pc, $prediction$))
13 else
14   // Required: squash instructions starting at pc
15   // Optional: call update, if correct prediction is known
16   return f(args)
```

Fig. 2: Pseudo-code for translating a transmitter f(args) into an SDO operation Obl-f(args). Sensitive/tainted values are colored green. The predictor input is assumed to be the transmitter's PC, since this is what we evaluate in the paper. Part 1 occurs when the SDO operation issues. Part 2 occurs when args becomes untainted.

At a high level, this construction combined with STT's mechanisms ensures that the predictor always makes predictions based on public information and that args never leaks before becoming untainted. Section VII formally argues how this does not change STT's security guarantee.

V. Speculative Data-Oblivious Loads

We now use the framework from Section IV to build a high-performance SDO operation for loads. We call it an Obl-Ld operation. Loads are notorious transmitters, leaking privacy over the cache/memory side channel. They are also notoriously high-overhead to protect [23, 37, 47, 51]. For example, STT reports nearly all overhead comes from delaying the execution of loads until their arguments are untainted [51].

While the focus of this section is on loads, the framework in Section IV applies to any transmitter. Thus, to demonstrate generality, the evaluation shows how SDO improves overhead when the framework is applied to both loads and loads plus floating point operations (Section I-A).

A. High-level Design Overview

To design the Obl-Ld, we must decide what DO variants to design (Section IV-A). This is not trivial, as there are many distinct ways a load can execute from a covert/side channel perspective. For example, a load may hit or miss at any cache or TLB level, contend for different resources such as uncore buses, cache banks, lookup DRAM, etc.—and each of these imply different hardware resource usages, timings, etc.

For this paper, we decided to design DO variants which are capable of looking up specified level(s) of cache in an address-independent fashion.

First, this leads to a relatively simple design. We need to design one DO variant per cache level. Importantly, by the SDO operation semantics in Section IV-C, which DO variant we predict is public information. That is, it is public knowledge *which level* cache tag/data arrays we are accessing and we only need to hide the load address while looking up that specific cache tag/data array. Thus, to implement the DO variants for monolithic private caches (e.g., the L1 or L2), a straightforward design might serialize access to the cache and only check but not update state (to hide whether a hit/miss occurred, and to eliminate timing variations from, e.g., cache bank conflicts [49]). For shared caches (such as the sliced L3) and DRAM, this is more challenging and discussed in Section VI-B.

Second, this design allows us to potentially eliminate most of the overhead from protecting loads. For example, if a given load has data in the L2 and the DO predictor correctly predicts "L2", then the Obl-Ld latency will be comparable to that of an insecure load. That is, a majority of the lookup time is simply to send a request to the L2, not to hide minor timing behaviors such as bank conflicts.

Note, our goal in designing Obl-Ld variants for every level in the cache hierarchy plus the DRAM is to explore the design space. Our evaluation (Section I-A) finds that, in terms of the associated performance/complexity trade-offs, systems may be best served by implementing only Obl-Ld variants for the caches (not DRAM).

B. *Basic* Obl-Ld *Design*

To implement the Obl-Ld, we design a DO variant for each cache level (i.e., Obl-Ld1 for predicting data is in the L1, Obl-Ld2 for L2, etc.). When a load with a tainted address issues, we lookup a DO predictor (called the *location predictor*, see Section V-D) to predict a level/variant. Predictions are made based on the load's PC. The chosen DO variant proceeds to lookup *all* cache levels from the L1 to the level predicted. For example, predicting data is in the L3 creates lookups in the L1, L2 and L3. Each lookup only checks if there is a tag match in that cache level and returns either data/⊥ accordingly. Importantly, a lookup makes no address-dependent state changes to the cache.

We will assume each DO variant is securely implemented (i.e., satisfies Definition 2) for now. In other words, when accessing a specific cache level, Obl-Ld does not create observable address-dependent hardware resource interference. We will describe how this is implemented in Section VI-B.

Designing the Obl-Ld to lookup all levels from the L1 to the predicted level is important for functional correctness and security. Consider an alternate design that only looks up the predicted level (e.g., looks up only the L2 if we predict L2). Such a design violates Definition 1: if there is dirty data in the L1, such a design will return stale data. As we will see in Section VII, this violates STT's semantics for access instructions and creates a security problem.

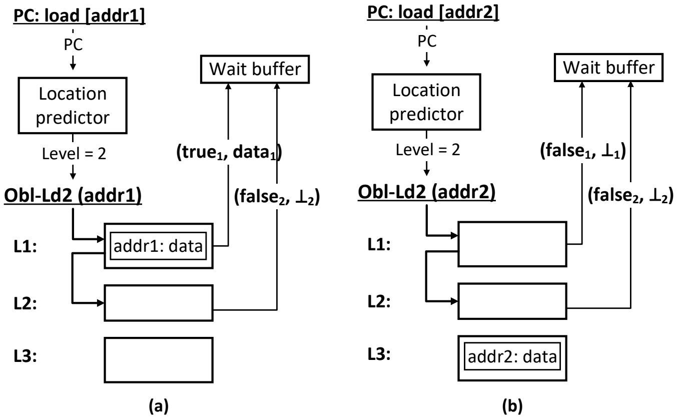

Fig. 3: Example Obl-Ld operation given two different addresses addr1 (present in L1) and addr2 (present in L3). In both cases, resource contention and operation timing is a function of the prediction, not the address.

A structure at the core called the *wait buffer* receives responses from each cache level. Each cache that is looked up returns either (success, presult) if data was present in that level, akin to a cache hit, or (fail, $\perp$) otherwise. In our current design, each DO lookup returns a word, not the whole cache line. We denote a response from cache level i with subscripts, e.g., ($\mathsf{success}_2$, $\mathsf{presult}_2$) for a response from the L2. Once all responses are received, the wait buffer forwards to the pipeline:

- $\mathsf{presult}_i$: for the smallest i such that we have $\mathsf{success}_i$
- $\perp$: if we have fail_i for all i

That is, we forward the correct data from the cache level closest to the core, as in a regular cache.

Putting it all together, an example Obl-Ld is shown in Figure 3. In (a) and (b), we have a load with different addresses but the same location prediction. The takeaway is that the resource usage (cache lookups, timing, etc.) is a function of the prediction, not the address.

When the load address becomes untainted, we follow the framework described in Section IV-C. If the prediction was correct, we proceed with execution and update the DO predictor (if applicable). Else, we squash execution starting at the load and re-issue as a regular load.

Virtual memory. Every load first needs to consult the TLB/page tables for address translation, and hits/misses in the TLB can also leak privacy [19]. As observed by prior work, L1 TLB miss rates are low [37]. Thus, we adopt a simplified strategy based on the same ideas as the main load operation. Conceptually, we design a single DO variant that looks up the L1 TLB. On an L1 TLB hit (success), the rest of the access proceeds as discussed above. On an L1 TLB miss (fail), we likewise continue with the prediction and Obl-Ld access, but with $\perp$ as the translation. That is, we do not consult the L2 TLB until the address becomes untainted and execution squashes.

C. *Advanced* Obl-Ld *Design*

1) Memory Consistency Issues: In a standard multiprocessor system, memory consistency is maintained using speculation [18]. Loads execute out of order—speculating that their output satisfies the memory consistency model rules—and bring the accessed cache lines into the core's L1. Memory consistency violations are detected based on invalidations of these cache lines from the L1. Under consistency model-specific conditions, a load is squashed and re-issued if the cache line it read gets invalidated from the core's L1. This mechanism has correctness and security implications for the Obl-Ld implementation.

Correctness: Handling Missed Invalidations. Unlike a standard load, an Obl-Ld may read from a cache line that is not in the core's private L1. In this case, the core will not get notified if the line is later invalidated (due to a coherence transaction or cache eviction), which can lead to an undetected consistency violation. To address this problem, we adopt InvisiSpec's validation/exposure mechanism [13, 47]. Specifically, when an Obl-Ld that did not obtain its value from the L1 becomes *safe* (i.e., its address is untainted), its output is *validated*. A validation performs a standard cache access, bringing the line read by the Obl-Ld into the L1, and compares the word read by the Obl-Ld to its up-to-date value. If they match, the Obl-Ld's output is valid; otherwise, the Obl-Ld is squashed and re-issued (this time as a standard load, since it is safe). Following an Obl-Ld validation, the core receives future invalidations of the line and is able to maintain memory consistency as usual. Because validation delays a load's retirement, we adopt InvisiSpec's *exposure* optimization, which avoids validating loads that the memory consistency model would not have required squashing had an invalidation been received. For such loads, validation is replaced by an expose operation that brings the accessed line to the L1 asynchronously, without delaying the load's retirement. (InvisiSpec [47] details the conditions under which validation can be replaced by exposure.)

Security: Handling Consistency Squashes. Our attacker model assumes that squashes are observed by the adversary, e.g., through the re-execution of the load. Squashing a load due to an invalidation could therefore leak the load's address. To prevent such a leak, we delay consistency squashes until the affected load's address becomes untainted. This delay is simply an application of STT's implicit channel protection rule to the implicit branch created by the memory consistency check, whose predicate compares the addresses of the load being squashed and of the memory access that triggers the invalidation [51]. The fact that an invalidation occurs implies that the access triggering it has an untainted address [51], so we only need to wait for the Obl-Ld to become untainted.

2) Event Interleavings: Modern out-of-order processors maintain multiple in-flight operations concurrently, meaning that the different steps in an Obl-Ld can occur in a variety of orders and interleavings. We now discuss the different possible cases.

Starting when the Obl-Ld executes, there are four events which control the behavior of the load:

A: The load is ready to issue but is unsafe/tainted, hence issues as an Obl-Ld.

B: The Obl-Ld operation completes when the responses from all predicted cache levels reach the wait buffer. At this point, data is safe to forward to dependent instructions.

C: The Obl-Ld becomes safe, i.e., its address is untainted.

D: The validation for the load completes.

As discussed in Section V-C1, validations are needed to enforce memory consistency if the Obl-Ld returns success from a lower-level cache, and the load must be re-issued if the Obl-Ld returns fail. We will refer to both of these loads as validations (event D) for simplicity.

The above events are partially ordered. In the following, we say $X \prec Y$ if event X happens before Y in time. For example, we must have $A \prec B$ because an Obl-Ld must issue before it completes. $C \prec D$ also holds since the validation is sent after the Obl-Ld becomes safe. On the other hand, $B \prec C$ may not hold, because the load may become safe before its Obl-Ld completes.

In this way, there are only three possible event orderings that can occur: $A \prec B \prec C \prec D$, $A \prec C \prec B \prec D$ and $A \prec C \prec D \prec B$, discussed in detail below and shown in Figure 4.

1) Event ordering is $A \prec B \prec C \prec D$:
[A] When an unsafe load issues an Obl-Ld request, it allocates a wait buffer entry. [B] When the wait buffer entry receives all responses from the accessed cache levels, the Obl-Ld request completes and the result is written to the register file/forwarded to dependent instructions. [C] Later, when the load eventually becomes safe, we issue a validation. If the Obl-Ld returned fail, we also immediately (i.e., before untainting) squash all subsequent instructions since their execution is based on an incorrect value. [D] When the validation completes: If the Obl-Ld returns success: SDO compares the result from the corresponding Obl-Ld and the current validation. If the values differ, a consistency violation might occur, therefore all subsequent instructions are squashed and the result of the validation is forwarded to re-issued dependent instructions. If the Obl-Ld returned fail (and we squashed): the validation result is forwarded to dependent instructions.
2) Event ordering is $A \prec C \prec B \prec D$:
[A] Same as Case 1. [C] Before the wait buffer receives all Obl-Ld responses, the load becomes safe. Instead of waiting for Obl-Ld completion, SDO issues a validation at this moment. [B] When the Obl-Ld completes: If the Obl-Ld returns success, SDO writes back the Obl-Ld result, and wakes up dependent instructions. If the Obl-Ld returned fail, forwarding incorrect data is unnecessary since it is now safe to reveal that the fail occurred. Thus, SDO drops the Obl-Ld result and waits for the result from the validation. [D] Same as Case 1.
3) Event ordering is $A \prec C \prec D \prec B$:
[A] and [C]: Same as Case 2. [D] Since the validation completes earlier than the Obl-Ld, and the validation result is always correct (i.e., a 'guaranteed success'), SDO writes back the validation result directly to complete the execution of the load. [B] Since the load was completed by its validation, the Obl-Ld result is ignored.

Importantly, an Obl-Ld returning fail will only result in a squash (which hurts performance) in Case 1: when the Obl-Ld completes before the load becomes safe.

Additional performance optimization: Early forwarding from wait buffer In the scheme described so far, an Obl-Ld operation is considered to have completed when all expected responses have reached the wait buffer. Only then can the wait buffer forward its result to dependent instructions.

We can improve this scheme's performance by leveraging the observation from before: when a load becomes safe, its address and success/fail status is safe to reveal to the attacker. Thus, if a load becomes safe, it is safe to forward data from the wait buffer early, i.e., as soon as the wait buffer detects there has been a success.[2]

3) Other Considerations:

Store-to-Load Forwarding. An Obl-Ld may get its data from the store queue due to store-to-load forwarding. Here, we adopt STT's policy: the Obl-Ld issues unconditionally, but the correct data is forwarded from the store queue instead of from the wait buffer once all Obl-Ld responses have returned.

Updating the location predictor. We update the location predictor when the load becomes safe, if its Obl-Ld returned success (per Section IV). If the Obl-Ld returns fail, we cannot perform the location predictor update as the actual location of the load data is still unknown. In this case, we wait for the validation and update the predictor with the level that the validation finds data in.

D. Predictor Design

We now describe a design for several location predictors. Suppose a load required data from cache level i and the predictor predicts j. We say the predictor is accurate and precise if $i == j$, accurate but imprecise if $i < j$ and not accurate if $i > j$. The goal is to be accurate and precise. When accurate but imprecise, the load incurs a larger delay because we must wait for the level j request to return. When

[2] Here, we assume levels of the cache respond in order, i.e., L1 first, L2 second, etc. If responses can arrive out of order, success_i cannot be forwarded to the pipeline until all responses j for $1 \leq j < i$ arrive at the wait buffer.

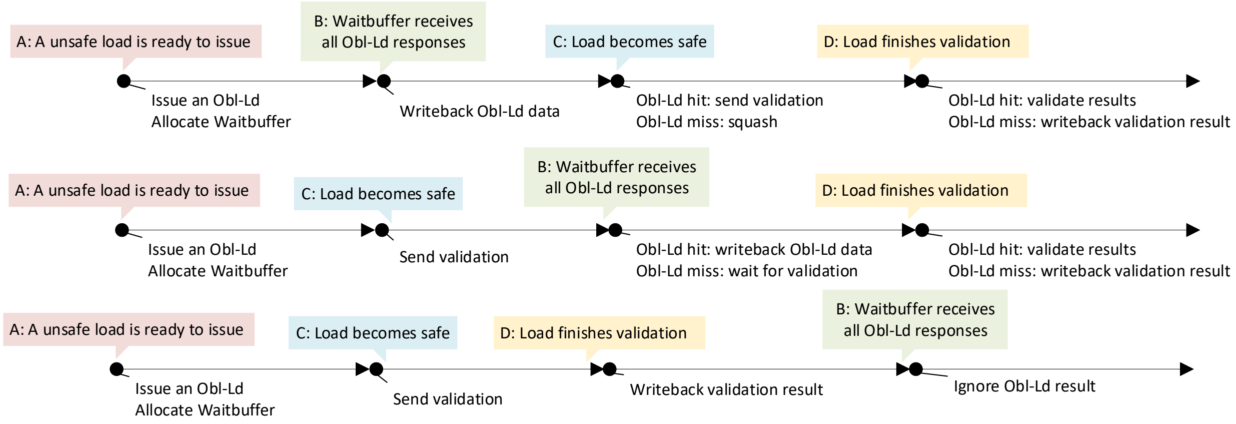
A: A unsafe load is ready to issue
B: Waitbuffer receives all Obl-Ld responses
C: Load becomes safe
D: Load finishes validation
Issue an Obl-Ld
Allocate Waitbuffer
Writeback Obl-Ld data
Obl-Ld hit: send validation
Obl-Ld miss: squash
Obl-Ld hit: validate results
Obl-Ld miss: writeback validation result
A: A unsafe load is ready to issue
C: Load becomes safe
B: Waitbuffer receives all Obl-Ld responses
D: Load finishes validation
Issue an Obl-Ld
Allocate Waitbuffer
Send validation
Obl-Ld hit: writeback Obl-Ld data
Obl-Ld miss: wait for validation
Obl-Ld hit: validate results
Obl-Ld miss: writeback validation result
A: A unsafe load is ready to issue
C: Load becomes safe
D: Load finishes validation
B: Waitbuffer receives all Obl-Ld responses
Issue an Obl-Ld
Allocate Waitbuffer
Send validation
Writeback validation result
Ignore Obl-Ld result

Fig. 4: Obl-Ld operation flow with the three possible event sequences (top: $A \prec B \prec C \prec D$; middle: $A \prec C \prec B \prec D$; bottom: $A \prec C \prec D \prec B$) and corresponding actions.

not accurate, we may incur a squash, depending on when the load becomes safe (Section V-C2).

The goal of this paper is to show the SDO framework is viable, not to invent a state-of-the-art predictor. We therefore first evaluate several simple static predictors that always predict to a specific cache level. Intuitively, static predictors to larger j (lower cache levels) will become more accurate but less precise. We also designed a simple dynamic predictor based on analyzing benchmark traces of high-overhead loads in vanilla STT, in particular to what levels loads hit and in what order. We observed that a given static load's cache level access pattern falls into one of two categories:

1) The cache level access pattern changes at a coarse granularity. That is, there are regions of continuous hits to a single cache level, i.e., low spatial locality.
2) The cache level access pattern consists of mostly L1 hits with predictable, singular lower level hits in between, i.e., high spatial locality. One common pattern in this category is accessing memory sequentially with a constant stride, i.e., one L1 miss per N memory accesses.

To capture both patterns, we design a *hybrid location predictor* which internally chooses between 2 predictors—a greedy predictor (greedy) and loop predictor (loop)—on each lookup, based on a per-load saturating confidence counter. Our predictor follows the template in Equation 2. The load's static PC (which is public in STT; Section III) is used as the predictor's input. greedy and loop are designed to capture access pattern 1 and 2, respectively. greedy predicts the lowest cache level (highest j) that has been seen in the last m dynamic instances of a given load. That is, it favors imprecision over inaccuracy to avoid potential mis-predictions. loop's behavior resembles a loop branch predictor: it predicts the frequency of lower-level accesses and tries to predict whether the next access will be an L1 hit or a hit in that lower level.

VI. Microarchitecture

Here, we describe microarchitecture changes to support SDO. We assume a baseline STT microarchitecture. We make modest changes to the processor pipeline to support Obl-Ld operations and enforce their consistency (Section VI-A). We then propose implementations for data-oblivious accesses to the various levels of the cache/memory hierarchy (Section VI-B). Figure 5 illustrates the baseline architecture and the modifications made for SDO.

A. Changes to Processor Pipeline

Our starting point is the STT microarchitecture, which extends a modern speculative out-of-order pipeline with (1) taint propagation, which is piggybacked on the existing register renaming logic; (2) a fast (single cycle) untaint mechanism; and (3) STT's protection rules for blocking implicit and explicit channels (Section III).

On top of STT, we add the location predictor (Section V-D) as well as control logic to perform the Obl-Ld execution events (Section V-C). To implement Obl-Ld execution logic, we extend each load queue entry with the following fields.

1) Obl-Ld State (4 bits): Stores whether the load is an Obl-Ld and if so, the current state of the Obl-Ld execution state machine described in Section V-C.
2) Actual Level (2 bits): The highest cache level for which a DO variant succeeded. This is used to update the location predictor once the load becomes safe.
3) Validation/Exposure (1 bit): Indicates if, when the Obl-Ld becomes safe, an expose should be performed instead of a validation. This field indicates Exposure in one of two cases: if the Exposure condition defined in InvisiSpec [47] is satisfied when the Obl-Ld is issued,[3] or if the Obl-Ld's lookup in the L1 succeeds.
4) Pending Squash (1 bit): Indicates if this load should be squashed once it becomes safe. (Also needed by STT, but written here because SDO adds a new case where it is required.)

[3] This is a condition over the state of the load queue that identifies when the load cannot possibly be reordered with older memory operations. (Importantly, we do not require InvisiSpec hardware to detect when the condition is satisfied.) Refer to [47, Appendix A] for details.

The wait buffer (which stores data returned by DO variants) is implemented using the output register of the Obl-Ld. That is, the output register is repeatedly overwritten (with the ready bit not yet set) when each cache level access returns.

B. Changes to Memory Subsystem

The crux of the Obl-Ld implementation are the DO load variants, which perform a data-oblivious lookup of some level of the cache/memory hierarchy. To satisfy SDO's security definition (Definition 2), an Obl-Ld must not have any address-dependent hardware resource usage; its resource usage can only be a function of untainted (public) state. This means that an Obl-Ld cannot make any address-dependent change to cache state, but also requires avoiding every other type of address-dependent resource contention. We generally achieve this property by partitioning an Obl-Ld from other loads (standard or DO), either spatially or temporally (by serializing resource access). As we shall see, while an Obl-Ld may contend for resources with other loads, the contention results only from the fact that the Obl-Ld is executing—which is public, due to STT's implicit channel protections—and not from the Obl-Ld's address.

Below, we discuss what a DO lookup entails for each level in the memory hierarchy, for completeness. We find, however, that limiting Obl-Lds to the on-chip caches suffices to reap most of SDO's performance gains (Section VIII). We thus posit that a microarchitecture which implements only on-chip cache DO variants is a sweet spot from a complexity/performance trade-off standpoint.

1) Baseline Memory Subsystem Model: We assume the following baseline, which is based on commercial processors. Each core has private L1 instruction and data caches and a private L2 data cache. The L3 cache is shared. The caches are all physically tagged set-associative caches and are write-back, write-allocate. All caches are *banked*, i.e., the data array is arranged in several banks. Concurrent accesses to different cache lines that target the same bank are serialized; otherwise, they are served in parallel. The shared L3 cache is distributed: it is organized in multiple set-associative slices (one slice per core). A hash function (set at design time) determines the slice associated with a cache line. Caches are kept coherent with a MESI-style [41] coherence protocol.

The caches can sustain multiple concurrent misses. Each cache maintains an array of miss status holding registers (MSHRs), each of which stores all information related to an outstanding miss. A cache miss on a line allocates an MSHR if there is no outstanding miss on that line; otherwise, the information in the MSHR is augmented with the new miss and no new request is issued to the next cache level.

A single, shared memory controller (MC) connects the system to DRAM. The MC schedules memory accesses (by L3 misses) to maximize DRAM row buffer hits and bank/rank parallelism. Therefore, DRAM access latency is a function of recent and outstanding requests. The caches and memory controller communicate over a ring interconnect.

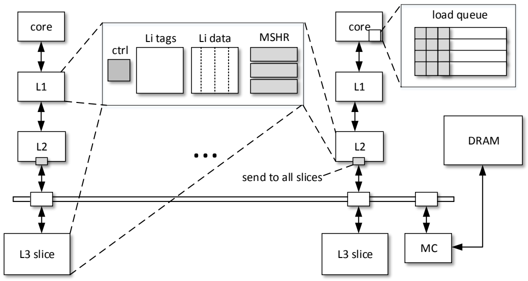

Fig. 5: Microarchitecture with SDO support. Shaded blocks represent modified hardware.

2) Modifications for Obl-Ld*:* We describe how to avoid address-dependent resource contention throughout the path of a cache/memory access.

Issuing Obl-Ld **requests to the cache controller.** In a baseline microarchitecture, address-dependent scheduling can affect when a load gets issued to the cache hierarchy, e.g., as a result of scheduling loads to minimize bank or port contention. To address this, the Obl-Ld scheduling logic must be address-independent, e.g., an Obl-Ld should be issued as soon as it is ready. Similarly, the choice of which cache port to access should also be address-independent. Note that an Obl-Ld request might not successfully issue immediately when ready due to other regular or Obl-Ld loads that are also pending to access the same cache. Importantly, such contention doesn't reveal any information about the Obl-Ld's address, because our design maintains the invariant that resource contention is a function of public (untainted) state.

Cache bank access. Accessing only the bank(s) dictated by a load's address constitutes address-dependent resource usage. Specifically, it risks creating a bank conflict with another request, with the resulting contention leaking the address. Our solution is therefore for an Obl-Ld to access all cache banks. Namely, after the Obl-Ld enters the cache, all succeeding requests are blocked until the Obl-Ld request completes its lookup. This approach guarantees that the resource use—and hence, effect on other requests—is independent of the Obl-Ld's address.

Storage of outstanding Obl-Ld **miss state.** We require the following to avoid address-dependent MSHR usage. First, every Obl-Ld must allocate an MSHR; it cannot share an MSHR with any other request. Second, the MSHR choice must be address-independent (e.g., first available MSHR). In this way, any MSHR contention created by an Obl-Ld follows only from the fact that the Obl-Ld is executing and accessing a specific cache level, both of which are public (untainted) information.

LLC slice access. Similarly to bank access, an Obl-Ld cannot access only the LLC slice dictated by its address. Consequently, the Obl-Ld variant accessing the L3 must send a request to all LLC slices. The correct slice returns success or fail, depending on whether the request hits or misses; all other slices necessarily return fail. The MSHR between the

L2 and L3 is de-allocated when all responses arrive, at which point it forwards a single response back to the core.

DRAM modules access. Implementing data-oblivious DRAM accesses not only requires changes to the on-chip memory controller (similar to [45]), but also to the DRAM modules themselves. For example, an Obl-Ld cannot directly fetch data from the row buffer, which has shorter access latency compared to accessing un-buffered rows [34].

As mentioned above, we find that designing a DO variant for DRAM is unnecessary, because data usually resides in the cache (Section VIII). However, simply limiting predictions to the L3 would result in a failed Obl-Ld (and subsequent squash) for data that does reside only in DRAM. We avoid this problem by allowing the DO predictor to predict that data is in DRAM, and delaying the issue of that load, until it is safe, in that case (i.e., reverting to STT's default protection). In this way, we do not squash unnecessarily.

VII. Security

We build SDO on top of STT, and denote our combined scheme STT+SDO. Our security goal is to preserve STT's security guarantee. Cited from the STT [51] paper: "at each step of its execution, the value of a *doomed* (transient) register, that is, a register written to by a speculative access instruction that is bound to squash, does not influence future visible events in the execution." This implies blocking leakage through all microarchitectural covert channels including pressure in the cache, arithmetic unit ports, total program execution time, etc (Section II).

To prove security, we argue that STT+SDO does not change STT's security semantics for transmitters and access instructions, Claims 1 and 2 below, respectively.

A. Security for Transmit Instructions

We first analyze SDO operations in general, and then analyze our Obl-Ld operation.

Claim 1. *Implementing transmitter* f(args) *as SDO operation* Obl-f(args) *(Figure 2) leaks equivalent privacy as delay-executing* f(args) *until* args *are untainted.*

Proof. In STT's terminology: Obl-f(args) is equivalent to executing a non-transmitter in the shadow of an implicit branch, which STT guarantees cannot violate security. From Figure 2, the case statement on Line 3 is by definition an implicit branch whose (i) predictions and (ii) updates/resolutions are a function of non-speculative data. (i) holds by Equation 2. (ii) holds by Figure 2, Lines 11-16, i.e., namely we adopt STT's policy for delayed predictor update/resolution until args is untainted. Finally, by Definition 2 the DO variant executed in the shadow of the branch—Obl-fi(args) for some i—is a non-transmitter. □

The Obl-Ld operation (Section V) is a valid SDO operation from Claim 1. First, the location predictor (Section V-D) takes the load PC as input, which is a function of non-speculative data due to STT [51, 52]. The predictor's delayed resolution/updates follow the same argument as above. Second, the design for each Obl-Ld variant–for each memory level (Section VI-B)–satisfies Definition 2.

B. Security for Access Instructions

STT untaints the output of an access instruction when the instruction reaches its *visibility point*, which is the point at which the instruction is considered non-speculative with respect to the threat model (Section III). Younger transmitters may reach their visibility point at the same time as their producer access instruction(s). For example, in the Spectre model, if there are no unresolved branches between access and dependent transmit instructions. STT's protection no longer applies to such transmitters. It is therefore important to establish that the output of the access instruction—which may be forwarded to these transmitters—is the result of a correctly-speculated execution and hence not a secret. This property holds for STT, because STT only delays execution of instructions, without changing how they execute. Without careful attention, one might conclude that the property does not hold for STT+SDO, because a DO variant might return fail. Here, we prove this property does in fact hold for STT+SDO.

Claim 2. *In STT+SDO, data returned by an access instruction is untainted only if that data corresponds to correct speculation, for the given attack model.*

Proof. We consider Obl-Ld operations, since these are the only access instruction changed in this paper. Suppose an Obl-Ld access instruction, denoted ainstr, reaches its visibility point. Let X be ainstr's output. We now proceed in cases.

Case 1 (ainstr **has forwarded** X **to younger instructions**). We have two sub-cases:

i ainstr **has returned in** success: By Definition 1, success implies X is correct with respect to the current speculative path. For a given attack model, having reached the visibility point implies that the current speculative path is correct speculation, thus the claim holds.[4]

ii ainstr **has returned in** fail: X may or may not correspond to correct speculation. As described in Sections IV-C and V-B, ainstr (and younger instructions) are squashed at the same moment that the data is untainted. Thus, the claim holds.

Case 2 (ainstr **has not yet returned** X **/ has returned but not yet forwarded**): X may or may not correspond to correct speculation. As described in Section V-C2, we will (a) forward X on success or (b) drop/re-issue ainstr as a normal load on fail. (a) becomes Case 1-i above. In (b), the result will be produced by a normal load, so the claim follows from STT's properties. □

[4] Note, in the case of loads X may eventually cause a consistency violation. Depending on the attack model (e.g., Spectre, Futuristic; Section II), a consistency violation may or may not constitute incorrect speculation. In the Spectre model, there is no issue (consistency violations are out-of-scope). In the Futuristic model, having reached the visibility point implies that a consistency violation can no longer occur (i.e., an access instruction is unsquashable after reaching the visibility point in the Futuristic model).

VIII. Evaluation

We now evaluate the performance of STT+SDO, for a variety of attack models and SDO design variants, relative to vanilla STT and an insecure baseline.

A. Experimental Setup

Simulation setup. We evaluate the performance of SDO using the Gem5 [12] simulator, which models cache port, bank and MSHR contention. We change the simulator to model the additional contention-related overheads caused by SDO. For example, by granting Obl-Ld operations exclusive access to all banks once they access one cache level. Table I details the simulated architecture. We use the x86-like Total Store Ordering (TSO) memory consistency model. We run SPEC CPU2017 [1] benchmarks with the *reference* input size. To obtain representative results for SPEC benchmark performance, we use SimPoint analysis [40] to identify execution fragments representing program phases. For each benchmark, we simulate 10 million instructions of each such fragment, and sum each reported metric (e.g., cycles) over all simulations, weighting each fragment according to its execution phase.

TABLE I: Simulated architecture parameters.

HW Components	Parameters
Pipeline	8 fetch/decode/issue/commit, 32/32 SQ/LQ entries, 192 ROB, 16 MSHRs, Tournament branch predictor
L1 I-Cache	32KB, 64B line, 4-way, 2-cycle latency
L1 D-Cache	32KB, 64B line, 8-way, 2-cycle latency
L2 Cache	256KB, 64B line, 8-way, 12-cycle latency
L3 Cache	2MB, 64B line, 8-way, 40-cycle latency
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	50ns latency after L2

Configurations. We evaluate the following design variants, listed in Table II. We compare STT+SDO (with different predictors, treating both loads and floating point (FP) operations as transmitters with architected DO operations) to STT. Two STT configurations are evaluated: STT{ld} considers only loads to be transmitters; STT{ld+fp} also considers FP operations to be transmitters (Section I-A). For STT+SDO, as discussed in Section V-D, we evaluate both static predictors (always predicting a specific cache level) and the hybrid location predictor (*Hybrid*, which predicts L1, L2 or L3). The size of the Hybrid predictor's internal state is 4 KB. Finally, we evaluate a *Perfect* predictor which always predicts the correct cache level. All SDO configurations protect subnormal FP inputs by statically predicting FP inputs to be normal (as described in Section I-A). All SDO configurations also mitigate leakage through virtual memory translation in the fashion described in Section V-B. For each configuration, we evaluate both the Spectre and Futuristic attack models (Section III).

Penetration testing. Prior to performance modeling, we confirmed that all SDO design variants block the Spectre V1 attack, to which the Unsafe baseline is vulnerable.

TABLE II: Evaluated design variants.

Configuration	Description
Unsafe	An unmodified insecure processor
STT{ld}	STT, delaying the execution of unsafe loads only
STT{ld+fp}	STT, delaying the execution of unsafe loads and fmult/fdiv/fsqrt micro-ops
Static L1	SDO with predictor always predicting L1 D-Cache
Static L2	SDO with predictor always predicting L2
Static L3	SDO with predictor always predicting L3
Hybrid	SDO with proposed hybrid location predictor (Section V-D)
Perfect	SDO with oracle predictor always predicting the correct level

B. Main Result: Performance of STT+SDO vs. STT

Figure 6 compares the execution time, normalized to Unsafe, of STT and the SDO variants on the evaluated SPEC2017 benchmarks. STT+SDO outperforms STT with both Static and Hybrid predictors in all cases. The summary is that (1) our Hybrid predictor outperforms Static predictors in the Spectre model, obtaining an average overhead of 4.19%, which translates to a 44.4%/50.1% improvement relative to STT{ld}/STT{ld+fp}, respectively; and (2) Static L2 is the lowest overhead predictor in the Futuristic model, obtaining an overhead of 10.05%, which is a 36.3%/55.1% improvement relative to STT{ld}/STT{ld+fp}. We now discuss these results in more detail.

Static Predictors. In both the Spectre and Futuristic models, Static L1 has the highest overhead of any SDO variant. The reason is that while predicting higher cache levels (e.g., L1) yields faster Obl-Ld operations, it also incurs more frequent squashes (due to Obl-Ld operations returning fail). As shown in Section VIII-E, performance overhead is strongly correlated to the number of squashes. Relative to Static L1, Static L2 and Static L3 have similar overheads on average, indicating that their relative differences in accuracy (which impacts squash rate) and precision (which impacts memory latency) balance each other out. (See Section V-D for definition of accuracy and precision.)

Hybrid Predictor. While the Hybrid predictor can achieve high accuracy and precision when access patterns fall into the categories discussed in Section V-D, we have found that it causes extra squashes (due to Obl-Ld failures) when accesses unpredictably miss in the L1. In the Spectre model, the overhead of these squashes can be hidden by overlapping computation, which allows the Hybrid predictor to achieve speedup over the Static predictors through its increased precision. See Figure 8 (left): the number of squashes for the Hybrid and Static L2 predictors are similar, but the Hybrid predictor achieves significantly lower overhead. Table III further confirms that the Hybrid predictor has significantly greater precision than the Static L2 predictor. In the Futuristic model, however, performance is more tightly correlated to the number of squashes. Indeed, Figure 8 (right) shows once again that the Hybrid and Static L2 predictors have a similar squash rate, but now with similar overheads.

Perfect Predictor. We show a perfect accuracy location predictor to show what performance potential is possible with SDO, beyond that achievable by our imperfect predictors.

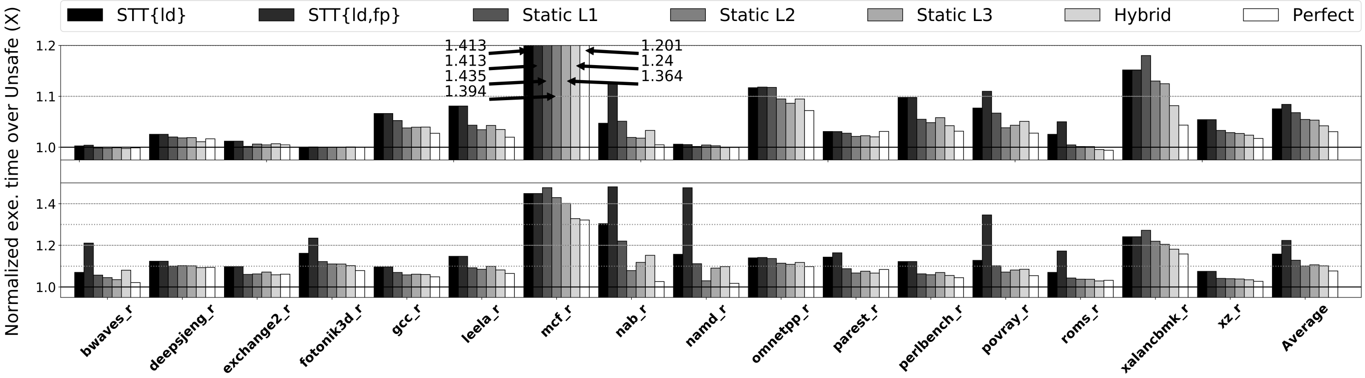

Fig. 6: Execution time (normalized to UNSAFE) of SPEC2017 benchmarks with STT and the proposed SDO design variants (STT+SDO) in Table II. Averages are given on the right. (Upper half: Spectre model; lower half: Futuristic model.)

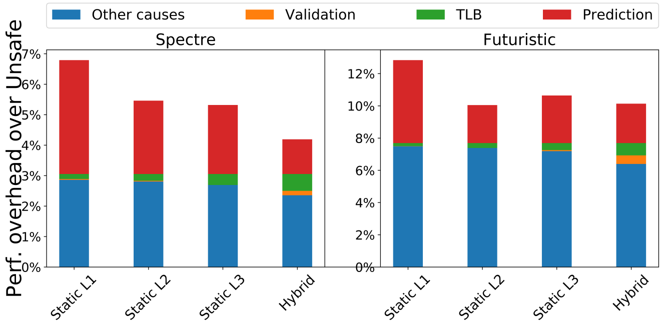

Fig. 7: Performance overhead breakdown (vs. Unsafe) for evaluated SDO variants, averaged over SPEC17 workloads.

Perfect prediction improves performance by 59.5%/63.7% relative to STT{ld}/STT{ld+fp} in the Spectre model, and 51.3%/65.6% relative to STT{ld}/STT{ld+fp} in the Futuristic model. Interestingly, there is still performance overhead, even if the location predictor is perfect. We perform a more detailed breakdown of the sources of remaining overhead in the next sub-section.

C. Sources of Slowdown

To provide more insight, Figure 7 shows a breakdown of what design components contribute what % of the total slowdown. Inaccurate and imprecise cache level prediction is a major source of overhead. Sections VIII-D and VIII-E below perform a deeper analysis of how prediction impacts performance. Validation stall and TLB/virtual memory protection constitute a small portion of the overhead. Remaining slowdown is due to (1) Obl-Ld operations not changing cache state, which leads to more cache misses; (2) implicit channel handling; and (3) the additional memory system contention caused by SDO requests.

D. Predictor Accuracy and Precision

We now measure the accuracy and precision for each SDO predictor. Table III shows that our Hybrid predictor has the highest precision, followed by Static L1, since Static L1 can usually satisfy most memory accesses. Static L2 and L3 have low precision ($< 8\%$), although their prediction tends to be more accurate (leading to fewer squashes) than Static L1/Hybrid.

TABLE III: *Precision* and *Accuracy* of evaluated SDO predictors in the Spectre/Futuristic models, averaged over SPEC17 workloads.

	Spectre		Futuristic	
Configuration	**Precision**	**Accuracy**	**Precision**	**Accuracy**
Static L1	71.87%	71.87%	75.48%	75.48%
Static L2	7.01%	78.74%	6.58%	83.39%
Static L3	4.60%	85.04%	3.71%	89.25%
Hybrid	84.30%	86.49%	84.34%	87.18%

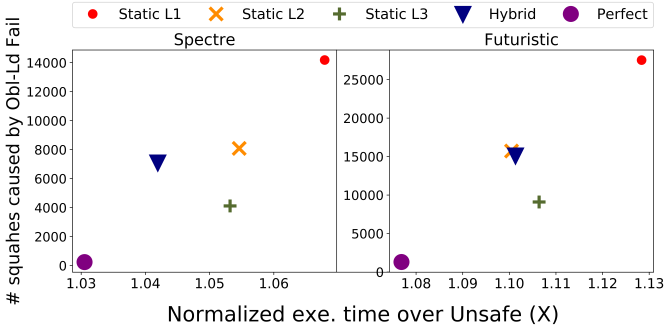

Fig. 8: Relationship between the number of squashes and execution time (normalized to UNSAFE) of all evaluated SDO variants, averaged over SPEC17 workloads.

E. Relationship between Performance and Squashes

We now quantify the performance loss due to inaccurate location prediction, i.e., the pipeline squashes that occur when an Obl-Ld returns fail. Figure 8 shows the correlation between the number of squashes and the performance overhead. The takeaway is that performance overhead is roughly proportional to the number of squashes. An exception to this trend is Static L3, which has the fewest squashes because its predictions are relatively accurate. In this case, fewer squashes are offset by imprecision (longer latency loads).

IX. Related Work

Hardware defenses for speculative execution attacks. InvisiSpec [47], SafeSpec [23], and DAWG [24] only block covert channels through the cache hierarchy. Conditional Speculation [28] and Selective Delay [37] additionally block covert channels through the memory system (e.g., DRAM contention). GLIFT [43] and OISA [50] can block all transient and non-transient covert channels, but impose restrictive, potentially low-performance programming models such as data-oblivious programming. STT [51], NDA [46] and SpecShield [6] strike a balance by blocking only transient covert channels.

We differentiate from these works as follows. Using STT's terminology, speculative data-oblivious execution is a tool to execute transmit instructions early, without leaking any additional information except for the fact that the transmitter executed. This is a stronger security property than that provided by works like InvisiSpec [47] and SafeSpec [23], as those proposals allow data to be forwarded at different times depending on when/where the access hits in memory (e.g., for the load instruction transmitter). Speculative data-oblivious execution is also more general. For example, it applies to any transmit instruction whereas, e.g., InvisiSpec only applies to loads.

At the same time, we view our work as complementary to more comprehensive proposals (such as STT, NDA and SpecShield). As discussed, these works adopt a high-overhead mechanism to block leakage through transmitters (delayed execution) whereas we study how to safely execute transmitters early. We chose to extend STT because it simultaneously achieves high performance and high security, but could have extended NDA or SpecShield as well.

Data-oblivious execution. There is a rich literature which focuses on how to write data-oblivious code on today's processors [4, 5, 7]–[9, 15]–[17, 31]–[33, 35, 38, 44, 53]—a practice also known as "constant-time" programming. These works focus on how to improve security on existing hardware. As a result, they cannot implement high-performance data-oblivious execution as described in this paper (e.g., using the floating point example from Section I, they must wait for the slowest mode to complete).

X. Conclusion

This paper proposes speculative data-oblivious execution (SDO), a new primitive which can be used to mitigate speculative execution attacks in a high-performance and high-security fashion. The key idea is that it is safe to execute an instruction which can form a covert channel, as long as that instruction's execution is independent of sensitive data, i.e., is data oblivious. To reduce the performance overhead of this idea, we extend prior work on STT to design *safe predictors* that specify what data-oblivious behavior is needed to satisfy common case program behavior. Putting it all together, we show that augmenting STT with SDO significantly speeds up vanilla STT without altering security.

Acknowledgment

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 would like to thank the anonymous reviewers for their insightful comments during the review process, which helped to significantly strengthen the paper.

References

[1] "SPEC CPU2017," https://www.spec.org/cpu2017.

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

[3] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*.

[4] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx," in *NDSS'18*.

[5] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P'15*.

[6] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels," in *PACT'19*.

[7] D. B. S. G. Ben A. Fisch, Dhinakaran Vinayagamurthy, "Iron: Functional encryption using intel sgx," in *CCS'17*.

[8] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *PKC'06*.

[9] D. J. Bernstein, "The poly1305-aes message-authentication code," in *FSE'05*.

[10] 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*.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT'08*.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, no. 2, pp. 1–7, 2011.

[13] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *ISCA'04*.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre attacks: Leaking enclave secrets via speculative execution," *CoRR'18*.

[15] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P'09*.

[16] S. Eskandarian and M. Zaharia, "An oblivious general-purpose SQL database for the cloud," *CoRR'17*.

[17] Z. L. L. K. Fahad Shaon, Murat Kantarcioglu, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *CCS'17*.

[18] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *ICPP'91*.

[19] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIEX Security'18*.

[20] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*.

[21] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, no. 4, pp. 1–17, 2006.

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

[23] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. B. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC'19*.

[24] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *MICRO'18*.

[25] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv'18*.

[26] 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*.

[27] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT'18*.

[28] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *HPCA'19*.

[29] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *CCS'18*.

[30] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv'19*.

[31] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P'18*.

[32] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," *IACR'05*.

[33] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *USENIX Security'16*.

[34] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security'16*.

[35] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Security'15*.

[36] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An undo approach to safe speculation," in *MICRO'19*.

[37] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *ISCA'19*.

[38] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel sgx," in *NDSS'18*.

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

[40] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ASPLOS '02*.

[41] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Pub., 2011.

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

[43] 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*.

[44] S. Tople and P. Saxena, "On the trade-offs in oblivious execution techniques," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Springer'17.

[45] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *HPCA'14*.

[46] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *MICRO'19*.

[47] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *MICRO'18*.

[48] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," in *USENIX Security'14*.

[49] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa," *IACR'16*.

[50] 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*.

[51] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO'19*.

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

[53] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *NSDI'17*.