Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis

Zirui Neil Zhao*, Houxiang Ji*, Mengjia Yan†, Jiyong Yu*, Christopher W. Fletcher*,
Adam Morrison‡, Darko Marinov*, Josep Torrellas*

*University of Illinois at Urbana-Champaign †Massachusetts Institute of Technology ‡Tel Aviv University
{ziruiz6, hj14}@illinois.edu, mengjiay@mit.edu, {jiyongy2, cwfletch}@illinois.edu,
mad@cs.tau.ac.il, {marinov, torrella}@illinois.edu

***Abstract*—Many hardware-based defense schemes against speculative execution attacks use special mechanisms to protect instructions while speculative, and lift the mechanisms when the instructions turn non-speculative. In this paper, we observe that speculative instructions can sometimes become *Speculation Invariant* before turning non-speculative. Speculation invariance means that (i) whether the instruction will execute and (ii) the instruction's operands are not a function of speculative state. Hence, we propose to lift the protection mechanisms on these instructions early, when they become speculation invariant, and issue them without protection. As a result, we improve the performance of the defense schemes without changing their security properties.**

To exploit speculation invariance, we present the *InvarSpec* framework. InvarSpec includes a program analysis pass that identifies, for each relevant instruction *i*, the set of older instructions that are *Safe* for *i*—i.e., those that do not prevent *i* from becoming speculation invariant. At runtime, the InvarSpec micro-architecture loads this information and uses it to determine when speculative instructions can be issued without protection. InvarSpec is one of the first defense schemes for speculative execution that combines cooperative compiler and hardware mechanisms. Our evaluation shows that InvarSpec effectively reduces the execution overhead of hardware defense schemes. For example, on SPEC17, it reduces the average execution overhead of fence protections from 195.3% to 108.2%, of Delay-On-Miss from 39.5% to 24.4%, and of InvisiSpec from 15.4% to 10.9%.

***Index Terms*—Speculative execution defense, Program analysis, Speculation**

I. Introduction

Speculative execution attacks [5], [9], [16], [23], [24], [25], [28], [31], [40], [44], [49] exploit a fundamental vulnerability of modern computer architectures. In these attacks, attackers craft sequences of transient instructions—those that are fetched and executed but do not commit—that leak information by changing the state of structures such as caches.

Since the initial vulnerability disclosures, there has been a flurry of work to block these attacks with a range of schemes that vary in complexity and performance overhead (e.g., [1], [2], [4], [8], [17], [20], [22], [26], [37], [38], [42], [43], [51], [53]). On the one hand, there are software schemes such as load hardening [8]. These schemes are simple to implement but have high overhead and, typically, limited coverage. On the other hand, there are many hardware proposals (e.g., [1], [4], [20], [22], [26], [37], [38], [42], [51], [53]), which have lower performance overhead. Within these schemes, there is a range of solutions with different emphases on complexity versus performance overhead. At this point, it is unclear which schemes will be adopted commercially. However, it is likely that whatever solutions are adopted, they will have to carefully balance coverage, complexity, and performance overhead.

Several of these hardware schemes rely on special mechanisms that protect instructions while they are speculative; when the instructions turn non-speculative, the protections are lifted. For example, speculative loads in InvisiSpec [51] are issued invisibly and are followed-up later with a second memory access. As another example, speculative loads in Delay-On-Miss (DOM) [26], [38] are allowed to access only the L1 cache; when the loads become non-speculative, they can access other cache levels. Finally, in a basic defense scheme that places fences to prevent the execution of speculative instructions, such fences can be removed as the instructions become non-speculative.

In this class of schemes, lifting the hardware protection earlier, while the instruction is still speculative, would increase the scheme's performance. For example, in InvisiSpec, the second memory access would be issued sooner or sometimes not at all; in DOM, loads would be stalled on L1 misses for shorter periods or sometimes not at all; in fence-based schemes, instructions would be stalled on fences for shorter time or sometimes not at all.

In this paper, we make the observation that a speculative instruction can become *Speculation Invariant* at some point before turning non-speculative. By this, we mean that speculative instruction *i* reaches a point when (i) whether *i* will execute is not a function of speculative state, and (ii) the operands of *i* are not a function of speculative state. When a speculative instruction is speculation invariant and its operands are ready, we say it reaches its *Execution-Safe Point* (ESP).

Figure 1 shows two simple examples of speculation invariant loads. Consider the Futuristic threat model [51], where all instructions remain speculative—and therefore squashable — until they reach the Reorder Buffer (ROB) head. Figure 1(a) shows a speculative load following an unresolved branch where the load address *x* is not dependent on any of the two branch paths. We say that *ld x* is speculation invariant and, as soon as *x* is ready, speculative *ld x* reaches its ESP. No

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00094

matter which direction the branch finally takes, *ld x* will always execute and access the same address. Figure 1(b) shows the same speculative load following an earlier load whose return data *y* does not directly or indirectly affect the register that *ld x* uses to generate the *x* address. Once again, *ld x* is speculation invariant and, as soon as *x* is ready, *ld x* reaches its ESP.

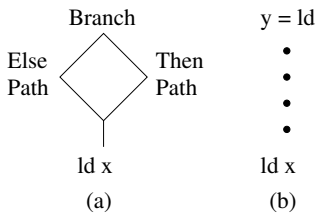


Fig. 1: Examples of speculation invariance.

When a speculative instruction reaches its ESP, we propose to lift any protection and execute the instruction. In the previous examples, we propose to send the load request to memory without protection. With this strategy, the performance of any of the previous protection schemes will improve. At the same time, the protection schemes' security properties will not change: executing a speculation invariant instruction without protection will not reveal any more secrets than the underlying hardware protection scheme would reveal with the non-speculative execution of the instruction.

Unfortunately, hardware structures alone cannot exploit these insights because the hardware is only aware of the current speculative path being executed and does not reason about all possible paths. Instead, we need a program analysis infrastructure to analyze the program and inform the hardware of the speculation invariance of instructions.

In this paper, we introduce *InvarSpec*, a framework to exploit speculation invariance for higher performance without hurting security. InvarSpec includes a program analysis pass that identifies, for each instruction *i* under protection, the set of older instructions (e.g., the branch and the load $y = ld$ in Figure 1) that are *Safe* for *i*—i.e., those instructions that do not prevent *i* from becoming speculation invariant. At runtime, the InvarSpec micro-architecture loads this information and uses it to identify when speculative instructions can execute early, without protection.

InvarSpec is one of the first defense schemes against speculative execution attacks that combines cooperative compiler and hardware mechanisms. It consists of an analysis pass for binaries, currently implemented for x86 binaries, and pipeline micro-architecture that uses this information at runtime.

To evaluate InvarSpec, we apply its analysis pass on the SPEC17 and SPEC06 programs and model its micro-architecture in a cycle-level simulator. Our results show that InvarSpec is effective. On average, using InvarSpec reduces the execution overhead of fence-based protection from 195.3% to 108.2%, the execution overhead of DOM from 39.5% to 24.4%, and the execution overhead of InvisiSpec from 15.4% to 10.9%.

In summary, the paper makes the following contributions:

- Presents *Speculation Invariance* to improve the performance of hardware security schemes against speculative execution attacks without hurting their security properties.
- Develops and evaluates the InvarSpec analysis pass.
- Develops the InvarSpec micro-architecture and uses it, together with the analysis pass, to improve the performance of three existing hardware security schemes.

II. Background & Key Concepts

A. Speculative Execution Attacks

Transient instructions. On an out-of-order processor, some instructions may execute but not subsequently commit; they get squashed and the processor state is rolled back to before their execution. These bound-to-squash instructions are called transient. For example, an instruction may be transient due to being in the shadow of a branch misprediction.

Attack structure. In a speculative execution attack, an attacker exploits the side-effects of transient instructions to learn information it would not be able to learn from a non-transient correct execution. A typical attack consists of a transient load accessing some secret value, which is then forwarded to *transmitter* instruction(s) (or *transmitters*) that leak the secret over a covert channel [22], [53]. These steps are collectively referred to as a disclosure gadget [19].

In general, a transmitter is any instruction whose execution creates operand-dependent micro-architectural resource usage that reveals the operand (even if only partially) [19], [22], [53]. The prototypical example is a load instruction, which causes address-dependent changes to the state of the cache hierarchy by filling and evicting cache lines. As a result, the cache line accessed by the load can be inferred using techniques such as FLUSH+RELOAD [52] or PRIME+PROBE [34].

Figure 2 shows Spectre V1 [24], an example of a transient execution disclosure gadget. It exploits the misprediction of a bounds-checking branch to perform an out-of-bounds array load (Line 2), which can read a secret from any memory location. A transmit load then leaks the secret (Line 3).

```
1  if (x < array1_size) { // mispredicted branch
2    uint8 s = array1[x]; // access load
3    uint8 y = array2[s * 64]; // transmit load
4  }
```

Fig. 2: Spectre V1.

Security violations. Attacks are categorized by the relationship between the hardware protection domains of the disclosure gadget and the victim [19]. In a *domain-bypass* attack, the gadget and victim are in different domains. An example is Meltdown [28], where a userspace process reads OS kernel memory. In a *cross-domain* attack, the gadget resides in the victim's domain (which differs from the attacker's domain, which is from where the attacker monitors the covert channel). An example is a network server whose code inadvertently contains a Spectre gadget that can be passed a malicious

input [24]. Finally, in an *in-domain* attack, the attacker circumvents software sandboxing. For example, an array access in JavaScript (compiled by a browser) is subject to a bounds check, producing code such as in Figure 2. Mispredicting the bounds check allows the attacker to circumvent the bounds check.

B. Hardware Defenses

To defend against speculative execution attacks, researchers have proposed hardware-based schemes. Several of these schemes (at least [1], [20], [26], [38], [51]) share a common general approach. First, they deploy a hardware mechanism that protects the relevant transmitter instructions. This protection prevents a transmitter from leaking its operands, blocking the side channel. However, it imposes a performance cost. Later, the protection is lifted when the transmitter's operands become safe to reveal. This execution point is called the instruction's *Visibility Point* (VP) [51]. When an instruction reaches the VP depends on the scheme's threat model, i.e., which types of transient instructions it considers.

Threat models. A popular but weak threat model is the *Spectre* model. It only considers transient instructions caused by incorrect control flow. An instruction reaches its VP when all of its older control-flow instructions have resolved. Another model is the Futuristic model [51], which we rename to the more descriptive name *Comprehensive* model. This model considers transient instructions caused by all types of squashes. An instruction reaches its VP only when it cannot be squashed anymore, which is most often when it reaches the ROB head. In this paper, we use the Comprehensive model.

Protection mechanisms. Most defense schemes target cache and TLB-based side channels. They typically apply a variety of protection mechanisms to loads. For example, InvisiSpec [51] and SafeSpec [20] issue speculative loads invisibly. CleanupSpec [37] records the state generated by speculative loads, to be able to undo it on a squash. DOM [26], [38] delays speculative loads that miss in the L1 cache, but allows L1-hitting speculative loads to execute. CSF [42] prevents speculative loads from changing visible cache state by inserting stalling fences. All of these mechanisms introduce performance overhead.

III. Speculation Invariance

A. Main Idea

As pointed out above, several defense schemes (at least [1], [20], [26], [38], [51]) use hardware mechanisms to block leakage while a transmitter is potentially transient and thus *unsafe*. If one could disable such mechanisms before the transmitter reaches its VP and, therefore, execute the transmitter speculatively without protection, one would reduce the overhead of these defense schemes.

In this paper, we propose a combined compiler and hardware scheme called *InvarSpec* that allows the lifting of these protection mechanisms for speculative instructions. The key idea is to identify *Speculation Invariant* instructions and allow them to execute while speculative without protection.

> A speculative instruction *i* becomes *Speculation Invariant* when (i) whether *i* will execute is not a function of speculative state, and (ii) the operands of *i* are not a function of speculative state. When an instruction is speculation invariant and its operands are ready, we say that the instruction reaches its *Execution-Safe Point* (ESP).

Intuitively, ESP is the earliest point when speculative instruction *i* can execute and is guaranteed to eventually commit using the exact same operands—no matter how many times it is squashed by older instructions due to incorrect speculation. At an instruction's ESP, InvarSpec permits its speculative execution without protection.

Since the definition of speculative instruction depends on the threat model (e.g., Spectre or Comprehensive as defined in Section II-B), speculation invariance and ESP for an instruction depend on the threat model. For example, assume that the branch in Figure 1(a) is unresolved and that there is no unresolved branch between the two loads in Figure 1(b). In Figure 1(a), *ld x* is speculation invariant under both Spectre and Comprehensive; in Figure 1(b), *ld x* is only speculative (and speculation invariant) under Comprehensive.

Figure 3(a) shows four points in the lifetime of a load instruction—which we use as a representative transmitter. Time increases to the right. The Ready point is when the load operands become available and the load is ready to be sent to memory speculatively. Current defense schemes place restrictions on what the load can do at this point. Sometime later, the load becomes speculation invariant and reaches its ESP. At this point, with InvarSpec, the load can be sent to memory without protection. Later, the load reaches its VP, where it becomes non-speculative and can be safely sent to memory without protection. Finally, the load retires. Effectively, InvarSpec moves the safe point of sending the load to memory from VP to ESP, reducing the overhead of the defense mechanism.

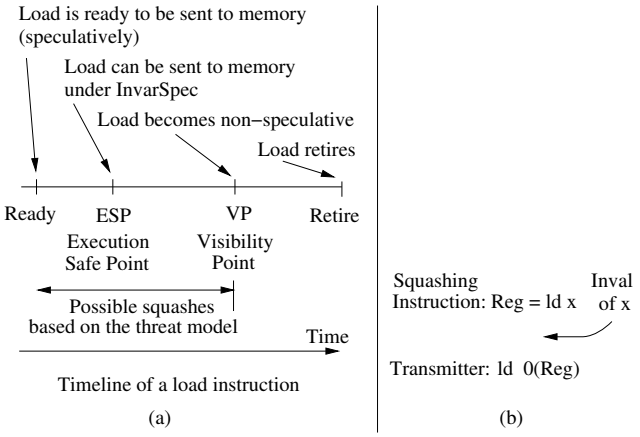


Fig. 3: Supporting speculation invariance.

According to the threat model that we use (Section IV), it is safe to expose the side effects of a speculation invariant instruction. Its execution does not reveal any more secrets than

the underlying hardware defense scheme would reveal with the non-speculative execution of the instruction.

B. Definitions

The InvarSpec framework has two important types of instructions: *Transmitters* and *Squashing* ones. Transmitters (e.g., loads) are inherited from the defense scheme that InvarSpec augments. Squashing instructions are those that can cause squashes that may lead to security violations. Squashing instructions are defined by the threat model. For the Spectre model, they are branches; for the Comprehensive model, they are branches, loads, and any instructions capable of causing exceptions. For example, a load may be squashed on reception of an invalidation for the address that it loaded, due to the processor's memory consistency mechanisms.

In this paper, we use loads as the transmitters and apply the Comprehensive model. In addition, we focus our analysis on the most challenging squashing instructions: branches (which can be mispredicted) and loads (which may be involved in exceptions or consistency violations and, on re-execution, can read a new value). Instructions other than loads may also be involved in squashes due to exceptions, but they are much easier to handle. We discuss exceptions in Section III-E.

Any instruction i that follows a squashing one and that has executed speculatively, may have to be squashed. Only if i had reached its ESP when it executed, it is guaranteed that, even after the squash, i will be re-executed and will use the same operands. For this reason, in InvarSpec, it is key to identify when a transmitter reaches its ESP. Only then can the transmitter execute without protection.

If InvarSpec only uses hardware support to identify when a transmitter reaches its ESP, it produces conservative results. The hardware uses the following algorithm.

> A transmit instruction i reaches its ESP when its operands are ready and each of its older squashing instructions in the ROB has: (i) executed and (ii) produced its final result. As a shorthand for conditions (i) and (ii), we will say that the older squashing instruction has reached its *Outcome Safe Point* (OSP)—i.e., the point where its result will not change irrespective of any future squashes.

When has an executed squashing instruction "produced its final result"? If we do not consider loads, we can say that non-load squashing instructions have produced their final result when all older branches have resolved. However, loads work differently. A load may reach its ESP (because it is also a transmitter) and execute, then get squashed, and then, as it re-executes with the same operand (i.e., the memory address), it may read a different value from memory—if another thread has written to the same location in between. An example is shown in Figure 3(b), where *ld x* reads a value into *Reg*, then gets squashed by an invalidation of x and then, as it re-executes, reads a different value from location x. Consequently, for a load to reach its OSP, it has to reach its ESP, execute, and then reach a point where it cannot be squashed anymore—typically, the ROB head.

Therefore, for the TSO-based x86 architecture and squashing instructions that we consider, the condition for an executed squashing instruction i to reach its OSP is as follows. First, if i is not a load, i reaches its OSP when (i) all the older branches in the ROB are resolved and (ii) there is at most one older load in the ROB, which is at a point where it cannot be squashed anymore. Second, if i is a load, i reaches its OSP when is at a point in the ROB where it cannot be squashed anymore. As indicated before, under the Comprehensive threat model, loads cannot be squashed anymore only when they are at the ROB head.

The appendix describes how to handle store-load aliasing.

C. Using Program Analysis Information

A program analysis pass can help the hardware algorithm just described to be more aggressive. It can identify, for each transmitter, the set of older squashing instructions that are *Safe* for the transmitter.

> *Safe* instructions for an instruction i are older squashing instructions that, even if they have not executed and generated their final result (i.e., they have not reached their OSP), they cannot prevent i from becoming speculation invariant.

Intuitively, the transmitter can become speculation invariant despite the fact that these older squashing instructions have not yet completed. Consequently, the hardware *does not need to consider them* when determining whether the transmitter is speculation invariant.

What are safe branches and safe loads for the x86 architecture? For a given load i, safe branches are those whose outcome cannot affect whether i will execute and what operands i will use. An example was shown in Figure 1(a). For a given load i, safe loads are those whose return data cannot affect directly or indirectly the address that i loads from. An example was shown in Figure 1(b). If, instead, load i is control dependent on a branch or data dependent on a load, then the branch or load is not safe for i.

The InvarSpec framework includes an analysis pass that takes a source or executable program and determines, for each transmitter, the set of safe squashing instructions. It then places these instructions' program counters (PCs) in a *Safe Set* (SS) for the transmitter.

At runtime, when a transmitter is about to execute and the InvarSpec hardware wants to determine whether the transmitter has reached its ESP, the hardware computes the ESP condition described in the box of Section III-B. However, the hardware also reads the transmitter's SS and prunes from the computation all of the squashing instructions in the ROB that are in the SS of the transmitter. Specifically, older branches and loads that are in the SS do not need to have reached their OSP for the hardware to conclude that the transmitter has reached its ESP. As a result, the transmitter reaches its ESP sooner and can execute sooner.

Finally, from this discussion, it is clear that we want the squashing instructions j that are not in the SS of the transmitter to reach their OSP as soon as possible. Sadly, each of them

needs to fulfill the conditions listed on Section III-B, which require that even older squashing instructions execute and reach their OSP. Fortunately, we can speed-up this process if we also generate the SS for each squashing instruction j. Any instruction in j's SS can be disregarded as we compute the conditions for j to reach its OSP. With this insight, we help j reach its OSP sooner. Hence, InvarSpec also builds the SS for squashing instructions.

D. The Complete InvarSpec Framework

The InvarSpec framework has two parts: (i) an analysis pass that generates the SS for transmit and squashing instructions, and (ii) hardware that, at runtime, loads the SSs and computes the ESP conditions. The analysis has two levels of support. The first one, called *Baseline*, populates the SS of instruction i with only those squashing instructions that are safe for i no matter what execution path the program takes.

The second level, called *Enhanced*, is more aggressive. It additionally places in the SS of i some squashing instructions that are *not* safe for some execution paths—as long as the hardware can detect when these paths are executed and prevent i from being executed until i is indeed speculation invariant. With this support, when the other paths are followed, i can be executed earlier. *Enhanced* improves the analysis by exploiting dynamic path execution behavior.

E. Handling Exceptions

Branches and loads are challenging instructions because, when they cause or are involved in a squash, they may change (i) what subsequent program instructions execute, and (ii) what operand values such subsequent instructions take.

Consider now exceptions. We assume an environment with no self-modifying code and no attacker-tampered executable. Here, there are two cases to consider. One is when the OS is able to service the exception and resume the program execution. The second case is when the exception causes program termination.

For the first case, InvarSpec's analysis only considers exceptions that involve the re-execution of loads, since loads may read a new value on re-execution. When only non-load instructions are involved, the re-execution after the exception is the same as the execution before. Hence, non-loads involved in exceptions do not need to be considered by InvarSpec.

The second case is when the exception causes program termination. In this case, we argue that no harm occurs from executing any speculation invariant transmitters that appear after the excepting instruction in program order. The reason is that such instructions are, by definition, control- and data-flow independent of the excepting instruction. As a result, unless the programmer or compiler explicitly places a fence in the code, the programmer can have no expectation about their execution order with respect to the excepting instruction—e.g., a different compiler could have hoisted these speculation invariant transmitters above the excepting instruction. Overall, program-termination exceptions do not affect InvarSpec's analysis either.

In summary, InvarSpec's analysis only needs to be concerned with exceptions that involve the re-execution of loads and are non terminating. Hence, the analysis of squashing instructions is limited to branches (which mispredict) and loads (which are involved in non-terminating exceptions and consistency violations).

IV. Threat Model

InvarSpec inherits the transmitters and the threat model from the hardware defense scheme that it augments. In this paper, we augment defense schemes that use loads as the transmitters and Comprehensive as the threat model (Section II-B). As indicated above, in this threat model, the analysis only needs to focus on two types of squashing instructions: branches (which can mispredict) and loads (which can re-load a new value after a non-terminating exception or a memory consistency violation—i.e., when certain speculatively loaded data receives an invalidation or suffers a cache eviction). The other type of instructions involved in exceptions are handled by existing hardware and the OS. In our model, victim and attacker can run on different cores or on different SMT contexts of a core.

InvarSpec allows speculative transmitters that are speculation invariant to execute speculatively without protection. InvarSpec is secure because it does not change the security properties of the defense scheme that it augments. Indeed, the execution of these speculative instructions does not reveal any more information than the underlying defense scheme would reveal with the non-speculative execution of the instructions.

We are expressly not considering attacks where the exact timing of when these speculative instructions execute would create a side channel. The defense schemes discussed [1], [20], [26], [38], [51] use the same assumption.

We assume that the SS information generated by the analysis pass for a program and attached to its executable is correct (e.g., signed and checked for trusted binaries). This is the case in the cross- and in-domain settings (Section II). In these settings, the victim is compiled by a benign compiler that generates a correct SS. In contrast, in the domain-bypass setting, the program itself is malicious. However, domain-bypass attacks [28], [33], [39], [44], [45] exploit an implementation issue—deferred handling of exceptions—which is fixed in upcoming processors [11], and so are not the focus of forward-looking defenses. We consider them out of scope.

We further assume that a program's SS information is not tampered with. In the cross-domain setting, the victim has no motivation to tamper with its own SS. In the in-domain setting, the sandbox prevents any attacker-controlled code from tampering with the SS (which would be computed or verified by the sandbox's trusted runtime system). Moreover, the integrity of the SS in distributed software packages can be verified together with the integrity of the entire package, using well-established integrity verification techniques such as digital signatures [10], [32]. In all of these cases, if an attacker is able to tamper with the victim's SS, then she is able to modify the binary, which means that she can mount much more harmful attacks than speculative execution attacks.

V. The InvarSpec Analysis Pass

InvarSpec includes an intra-procedural program analysis pass that accepts as input a program in source code or binary. Source code is preferred, since it allows a better analysis because it contains more information. InvarSpec is also told what kind of instructions are transmitters and squashing ones, and the threat model. InvarSpec can support multiple threat models and augment multiple hardware defense schemes.

The analysis pass generates, for each transmit and squashing instruction i, the set of squashing instructions that are safe for i. The program counters (PCs) of these safe squashing instructions form the Safe Set (SS) for i. The InvarSpec pass has two levels: the *Baseline* analysis and the more aggressive *Enhanced* analysis. We consider each in turn.

A. Baseline Analysis

1) Basic Algorithm: InvarSpec starts by generating the Program Dependence Graph (PDG) [12] of each procedure in the program. The PDG represents the dependence relationships among the instructions in the procedure. Each instruction is a node, and a directed edge from node i to node j means that i is directly control or data dependent on j. The edge is labeled "CD" if it is a control dependence, or "DD" if it is a data dependence.

The algorithm to generate the PDG of a procedure takes as inputs the procedure's control-flow graph (CFG) and data-dependence graph (DDG). The DDG includes dependencies through both registers and memory. For each instruction i, the algorithm adds an outgoing edge to all the instructions d that i directly depends on.

InvarSpec then computes the SS for each transmit and squashing instruction in the procedure. Algorithm 1 shows the pseudo-code for *getSS*, which computes the SS for instruction i. *getSS* takes as inputs i and the CFG, DDG, and PDG of the procedure. It first computes *ancSI*, which is the set of all the squashing instructions that are ancestors of i in the CFG. These are potential candidates for the SS. Then, Line 3 calls *getIDG*, which computes the Instruction Dependence Graph (IDG) of i.

The IDG of i is a subgraph of the PDG that includes i plus all the instructions that may affect whether i executes or the values of i's source operands. Intuitively, the instructions in the IDG should not be placed in the SS for i. If i is a load, the IDG does *not* contain stores that may update the memory *location* that i loads. Such stores are in the DDG because the DDG captures all the data dependencies, including those that affect the load's result; such stores are *not* in the IDG because they cannot affect whether i executes or the values of i's operands.

getIDG first creates an empty IDG graph (Line 9). It then adds to the graph all the instructions that i has direct control dependence on or that i's source operands have direct data dependence on. Finally, for each such instruction, *getIDG* calls *addDescGraph*, which adds to the IDG all the descendants of the instruction in the PDG.

Back to *getSS*, Line 4 collects all the squashing instructions from the IDG into *deps*; i itself is not in *deps* unless it depends on itself (due to a program loop). Finally, Line 5 subtracts *deps* from *ancSI*. The result is the SS of i.

Algorithm 1: Computing the SS for an instruction.

```
1  Function getSS(i, CFG, DDG, PDG) is
2      ancSI ← {a ∈ getAnces(CFG, i) | isSquashInsn(a)}
3      IDG ← getIDG(i, CFG, DDG, PDG)
4      deps ← {d ∈ getDesc(IDG, i) | isSquashInsn(d)}
5      SS ← ancSI \ deps
6      return SS
7  end
8  Function getIDG(i, CFG, DDG, PDG) is
9      IDG ← DirectedGraph()
10     for d in getCtrlDeps(CFG, i) do
11         addNode(IDG, d)
12         addEdge(IDG, i, d, "CD")
13         addDescGraph(d, IDG, PDG)
14     end
15     for d in getDataDeps(DDG, i) do
16         if ¬(isLoad(i) ∧ isStore(d)) then
17             addNode(IDG, d)
18             addEdge(IDG, i, d, "DD")
19             addDescGraph(d, IDG, PDG)
20         end
21     end
22 end
```

2) Procedure Calls: The InvarSpec analysis pass is intra-procedural and, therefore, only considers dependencies inside a procedure. Interactions between procedures are handled as follows. First, consider a caller procedure. InvarSpec conservatively assumes that the callee may modify any memory location. Hence, InvarSpec treats a procedure call instruction as a store that may alias with any subsequent loads. For registers, InvarSpec uses calling conventions, which preserve some register values.

Second, consider a callee procedure. The SS of an instruction does not contain PCs of squashing instructions outside of the procedure. This design conservatively assumes that all squashing instructions outside of the procedure are unsafe. While this design is conservative, it is sound.

In a recursive procedure, the caller is the same as the callee. In this case, more dependencies may exist between instructions in the procedure than captured by our intra-procedural analysis. To see why, consider Figure 4. In the example, instruction *ld x* is a transmitter, and *br* is a squashing instruction that we would prefer to be in the SS of *ld x*. However, because the call is recursive, and the branch decides whether the call is executed, the *ld x* in the callee depends on the *br* in the caller. More generally, if a recursive procedure call (Line 3) has a control dependence or a data dependence (e.g., due to call arguments) on a squashing instruction, that squashing instruction should not be placed in the SS of any other instruction in the procedure.

Unfortunately, we cannot simply solve the problem via

```
1  foo() {
2    if (...) { // br
3      foo(); // call
4    }
5    ld x; // ld
6  }
```

Fig. 4: Code snippet with a recursive call.

program analysis: because of procedure pointers and indirect recursive calls, it is typically hard to identify recursive functions. Hence, we use hardware as follows. We still place the above squashing instruction in the SS of *ld x*, but the hardware places a fence at the beginning of each procedure. Such fence only prevents the execution of subsequent transmitters until the call instruction reaches the ROB head. With this support, the callee is not affected by squashing instructions from the caller. In practice, this support causes only a minor slowdown to the code run with InvarSpec, since compilers typically inline short functions in the caller. Our fence support handles not only direct but also indirect recursion.

3) Soundness & Completeness of Analysis: Our analysis labels squashing instructions as safe or unsafe. Soundness considers whether an unsafe squashing instruction may be labeled as safe, and completeness whether a safe squashing instruction may not be labeled as such.

The InvarSpec analysis is sound because it closely follows the definition of speculation invariance within procedures: no execution path from a safe squashing instruction to a transmitter can affect whether the transmitter executes or what source operands it uses. When our analysis cannot determine all execution paths, e.g., due to indirect jumps, it conservatively does not place the squashing instruction in the SS.

The InvarSpec analysis is not complete, due to at least two reasons. The first one is that it is not inter-procedural, which would be expensive and potentially unsound. The second one is the limitations of pointer-aliasing analysis. Incompleteness hurts performance but not correctness.

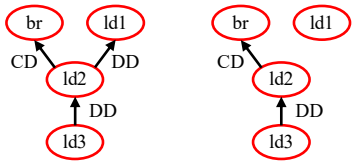
B. Enhanced Analysis

1) Key Insight: The Baseline analysis considers all possible dependencies when generating an SS. However, some of the dependencies may not occur on all execution paths. If we could neglect such dependencies, unless they really do occur, we could make the SS bigger, which could lead to speed up.

To illustrate the problem, consider the code in Figure 5(a), where *ld*3 is a transmitter. Assume that *ld*1 takes a long time to execute (e.g., because *z* misses in the cache), while *br* resolves quickly and, typically, is not taken. Figure 5(b) shows the IDG of *ld*3. We see that *ld*3 has a data dependency on *ld*2; *ld*2 has a control dependency on *br* and a data dependency on *ld*1.

Given this IDG, using InvarSpec's Baseline analysis, *ld*3's SS will not contain *ld*2, *br*, or *ld*1 because they are in *ld*3's IDG. They can affect the execution of *ld*3. Hence, InvarSpec will not send *ld*3 to memory until all three instructions have reached their OSP.

```
y = ld z; // ld1
if (a) { // br
  x = ld y; // ld2
}
ld x; // ld3
```



(a) (b) (c)

Fig. 5: Code pattern that can be sped-up: (a) source code, (b) IDG of transmitter *ld*3, and (c) pruned IDG of *ld*3 computed by the Enhanced analysis.

However, consider the case when *br* quickly resolves as *not taken*, and *ld*1 takes a long time to complete. In this case, *ld*3 is stalled by *ld*1, although *ld*3 has *no* runtime dependency on *ld*1.

The root of this stall is that InvarSpec's Baseline analysis does not consider the true runtime dependencies (i.e., the Baseline analysis is not flow/path-sensitive [14]). If, instead, we consider the path taken, we can show that, since *ld*3 depends on *ld*1 only if *ld*2 appears in ROB (i.e., *br* is taken), putting *ld*1 in *ld*3's SS is actually safe.

Specifically, if *br* resolves and *ld*2 appears in the ROB, *ld*2 effectively *shields* *ld*3 from *ld*1: *ld*3 will not be sent to memory until *ld*2 reaches its OSP. By that time, *ld*1 has reached its OSP. Hence, the scheme does not need to directly check for *ld*1. Placing *ld*1 in *ld*3's SS still results in a correct execution.

If, instead, *br* resolves and *ld*2 does not appear in the ROB, *ld*3 can safely execute without waiting for *ld*1. Hence, putting *ld*1 in *ld*3's SS keeps correctness and makes the execution faster than with the Baseline analysis. Overall, in *ld*3's IDG, we can effectively remove the edge to *ld*1 (Figure 5(c)).

2) Understanding the Enhanced Analysis: Based on the previous discussion, InvarSpec's Enhanced algorithm involves taking the IDG of an instruction *i* and removing some of the squashing instructions. The squashing instructions that are removed can be placed in the SS of *i*; the ones that remain cannot.

To understand when a squashing instruction can be removed, we need to understand when a squashing instruction shields another. Specifically, given an instruction *i* that depends on a squashing instruction *j*, which in turn depends on a squashing instruction *k*, when does *j* shield *i* from *k*?

We have seen in Figure 5(b) that if the edge from *j* to *k* is a data dependence, *j* shields *i*, and we can remove the edge from *j* to *k* (i.e., the edge from *ld*2 to *ld*1). Instruction *i* cannot reach its ESP until *j* reaches its OSP, and in turn *j* cannot reach its ESP (let alone its OSP) until *k* reaches its OSP. By the time *j* reaches its OSP, *k* cannot affect *i* anymore.

However, if the edge from *j* to *k* is a control dependence, the behavior is different. An example is the edge from *ld*2 to *br* in Figure 5(b). Branch *br* controls the value of *x* that *ld*3 uses: either the value returned by *ld*2 or not. *ld*3 cannot be sent to memory until *br* has reached its OSP. If we removed the edge from *ld*2 to *br*, *ld*3 would not wait for *br*'s OSP,

which could cause an incorrect execution. Indeed, suppose we remove it. Then, suppose that *br* mispredicts as not taken, and hence *ld*2 is not in the ROB to shield *ld*3. In this case, *ld*3 would be incorrectly sent to memory before *br* reached its OSP. Overall, the edge from *ld*2 to *br* cannot be removed and *br* cannot be in *ld*3's SS.

Consider now when the instruction *i* is control dependent on a squashing instruction *j*, to find out what instructions can *j* shield. Figure 6(a) shows an example code where *ld*2 is the transmitter. *ld*2 is control dependent on *b*2 which, in turn, is control dependent on *b*1 and data dependent on *ld*1. Figure 6(b) shows the corresponding IDG.

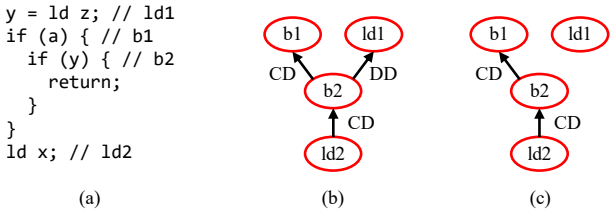


Fig. 6: Code pattern to show when edges can be removed.

In this example, *b*2 shields *ld*2 from *ld*1: *b*2 will not reach its OSP until *ld*1 reaches its OSP, by which time *ld*2 does not need to consider *ld*1. Hence, InvarSpec can remove the edge from *b*2 to *ld*1 and put *ld*1 in *ld*2's SS. The code will now run faster if *ld*1 takes long to execute, *b*1 reaches its OSP quickly and is not taken.

On the other hand, *b*2 does not shield *ld*2 from *b*1. If we removed the edge from *b*2 to *b*1, *ld*2 would not wait for *b*1's OSP, which could cause an incorrect execution. Indeed, suppose we remove the edge. Then, suppose that *b*1 mispredicts as not taken, and hence *b*2 is not in the ROB to shield *ld*2. In this case, *ld*2 would be incorrectly sent to memory before *b*1 reached its OSP. Hence, the *b*2 to *b*1 edge needs to remain in the IDG, and *b*1 cannot be in *ld*2's SS. Figure 6(c) shows the resulting IDG.

Overall, outgoing DD edges from squashing instructions can be removed, while CD edges cannot. The fundamental reason is that runtime data dependencies are path-sensitive—i.e., they are a function of the execution path followed. Control dependencies are path-insensitive, in that they exist irrespective of which of the two paths is taken by the execution.

If a DD edge starts from a non-squashing instruction, the edge cannot be removed. This is because a non-squashing instruction does not prevent a younger instruction from executing and, therefore, cannot shield it.

3) Enhanced Algorithm: Based on the previous discussion, we now outline InvarSpec's Enhanced analysis. Algorithm 2 shows the pseudo-code of function *pruneIDG*, which takes the IDG of an instruction *i* and generates a *pruned* IDG for *i*. The function traverses all the nodes in the IDG except *i* (the IDG root). If an instruction in the IDG is squashing, we check its outgoing edges. All the edges that are DD are removed.

The pruned IDG is then passed to function *getSS* of Algorithm 1 to compute the SS of the instruction. Because some squashing instructions are now unreachable in the pruned IDG, the Enhanced algorithm places more instructions in the SS of the instruction than the Baseline one. The result is a faster execution of the program.

Algorithm 2: Pruning an IDG.

```
1  Function pruneIDG(IDG) is
2      for i in getNodes(IDG) \ {getRoot(IDG)} do
3          if isSquashInsn(i) then
4              for e in getOutEdge(IDG, i) do
5                  if isDataDep(e) then
6                      removeEdge(IDG, e)
7                  end
8              end
9          end
10     end
11     return IDG
12 end
```

C. Truncating the Safe Set

The SS of an instruction can contain the PCs of many instructions. To keep the hardware simpler, we propose to truncate the SS to a fixed size. For performance, we would like to keep only "the most useful" SS PCs. These are the PCs of the safe squashing instructions that are the most likely to be in the ROB when the transmitter enters the ROB. The PCs of safe instructions that are far in dynamic execution and thus already likely out of the ROB are less important to keep.

To find the most useful SS PCs for instruction *i*, the analysis pass statically finds the shortest distance, measured in the number of instructions in the function's CFG, between each safe squashing instruction and *i*. Then, it keeps in the SS the *N* safe squashing instructions with the smallest distances. It further removes those instructions whose distance is larger than the size of the ROB. We call the scheme $Trunc_N$.

In the SS of an instruction *i*, each safe instruction is encoded as the signed difference between the PC of the instruction and the PC of *i*. We call them *Offsets*.

VI. The InvarSpec Hardware

To use the SS information, InvarSpec adds two microarchitecture modules. One compares the SS of an instruction to the older squashing instructions in the ROB; the other holds the SS and brings it to the pipeline on demand. For the second module, we present two possible designs.

A. Comparing the SS in the ROB

InvarSpec adds a hardware buffer in the pipeline that contains an entry for each dynamic instruction *i* in the ROB that is a transmitter (i.e., a load) or a squashing one (i.e., a load or a branch). We call it the *Inflight Buffer* (IFB) (Figure 7). Each IFB entry contains the following information for *i*: (i) its PC, (ii) a bit $\overline{T}$ that tells that *i* is not a transmitter, (iii) a *Ready* bitmask used to periodically check if *i* has become speculation invariant (SI), (iv) a bit set when *i* becomes SI,

and (v) a bit set when i reaches its OSP. The Ready bitmask has as many bits as IFB entries.

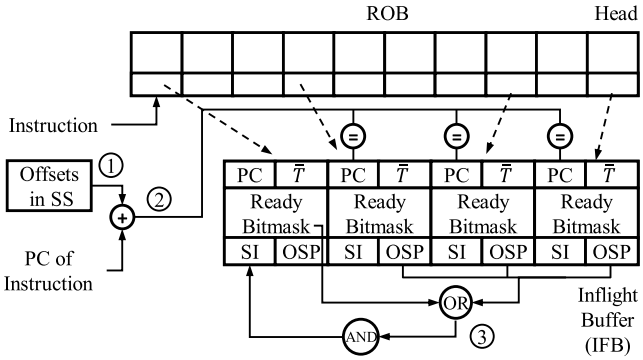


Fig. 7: Hardware to use the SS in the ROB.

ROB entries have pointers to their corresponding IFB entries. IFB entries are allocated and deallocated in program order when the corresponding instruction is inserted in and removed from the ROB, respectively. Both IFB and ROB are circular buffers.

When a transmit or squashing instruction i is inserted in the ROB, its SS is requested (①), as we will see in Section VI-B. The offsets in the SS are summed-up to i's PC, creating a set of safe squashing instruction PCs (②). The resulting PCs are compared to the PCs in the IFB entries that are both older than i and belong to squashing instructions. Note that, in the Comprehensive threat model that we use, transmit instructions are also squashing ones. Hence, the PCs from the SS are compared to the PCs in all the older IFB entries.

Based on these comparisons, the Ready bitmask in the IFB entry for instruction i is set as follows. If IFB entry k has a PC that matches one of the PCs obtained from the SS or has the OSP bit set, we know that the entry cannot prevent i from becoming speculation invariant (SI): either entry k belongs to a safe squashing instruction or to an instruction that has already reached its OSP. In either case, bit k in the Ready bitmask of instruction i is set. Further, the bits for those IFB entries not yet owned by any instructions, and for the entry owned by i, are set. The only Ready bitmask bits of i that remain clear are those for older squashing instructions that are not safe for i and have not reached their OSP.

If i is not a transmitter ($\overline{T}$=1), i can execute as soon as its operands are ready. In our configuration, this is the case for branches. Otherwise, i can only execute when it becomes SI and its operands are ready. In either case, the hardware tries to find when i becomes SI by checking, at every cycle, if the IFB entries that caused Ready bitmask bits to remain clear do set their OSP bit. As seen in Figure 7, this is done by simply taking the OSP bits from all the IFB entries and bit-ORing them with the Ready bitmask (③). When all the resulting bits are set, it means that all the squashing instructions older than i are either safe or have reached their OSP. At this point, i has become SI and sets its SI bit. If i is a transmitter, it can now execute as soon as its operands are ready.

After an instruction has satisfied the condition for its SI bit to be set and has executed, the logic to set its OSP bit depends on what type of instruction it is. Specifically, if it is a branch, the hardware sets its OSP bit right away. If it is a load, setting the OSP bit has to wait until the load reaches the point where it cannot be squashed anymore based on the threat model. For the Comprehensive model that we use, this is when the load reaches the ROB head.

There are two corner cases that are easily solved. First, if the IFB runs out of space, the ROB stops taking in new instructions. Second, if the SS for instruction i is not yet in the pipeline when i in inserted in the ROB, and there are older entries in the IFB that have their OSP bit clear, the hardware assumes that such entries are all unsafe. Hence, the corresponding Ready bitmask bits remain clear.

B. Storing and Bringing the SS to the Pipeline

The InvarSpec pass generates the SSs for the Squashing and Transmit Instructions (STIs) in the program. However, a sizable fraction of the STIs have empty SSs. Hence, we envision the InvarSpec pass to mark in the executable those STIs that have a non-empty SS.

Logically, such a mark can be a set bit in the opcode of the STI. In practice, in the x86 ISA, there is no such bit available. Hence, we can use an approach that has been used by Intel for lock elision: re-purpose a previously-ignored instruction prefix to mark instructions [18]. Specifically, we can reuse the *XRELEASE* prefix—which today is meaningful only for stores—to mark that the prefixed STI (a load or a branch in our case) does have an SS. This means that the encoding of STIs with an SS grows by the 1-byte prefix.

This approach changes the executable, but maintains backward compatibility. Because current processors ignore this prefix for STIs, the new executable runs on any x86 machine.

With this support in place, we now focus on how to store the SS and bring it to the pipeline on demand. We propose two alternatives: a software-based solution that is simple but makes the executable backward incompatible, and a hardware-based solution that is more complex but keeps backward compatibility. We outline each in turn, but we will only evaluate the one that keeps backward compatibility.

Software-Based Solution. In this solution, the InvarSpec analysis pass embeds the SS of an STI in the code of the program, right after the STI. For example, the pass could add an SS with 12 PC offsets of 10-bits each, for a total of 15 bytes. As an STI with prefix is decoded, the decoding hardware extracts the adjacent SS from the code stream. When the STI is inserted in the ROB, its SS is readily available for the operation ① in Figure 7. This solution is simple but not backward compatible.

Hardware-Based Solution. In this solution, the InvarSpec analysis pass stores the SSs in data pages, and the core has a small *SS Cache* that keeps the recently-used SSs close to the pipeline for easy access in the future. Since the most frequently-executed STIs are in loops, a small SS cache typically captures the great majority of dynamic SSs needed.

We propose a simple design where, for each page of code, there is a data page at a fixed Virtual Address (VA) offset that holds the SSs of the STIs in that page of code. Further, the VA offset between *each* STI and its SS is fixed. This design does increase the memory consumed by a program by potentially the size of its instruction page working set (Section VIII-B). However, it enables fast SS access.

Figure 8(a) shows a page of code and its SS page at a fixed VA offset (Δ). When the former is brought into physical memory, the latter is also brought in. The figure shows a prefixed STI and its SS. If the distance between the VAs of two consecutive prefixed STIs is less than the size of an SS, one of the STIs loses the prefix.

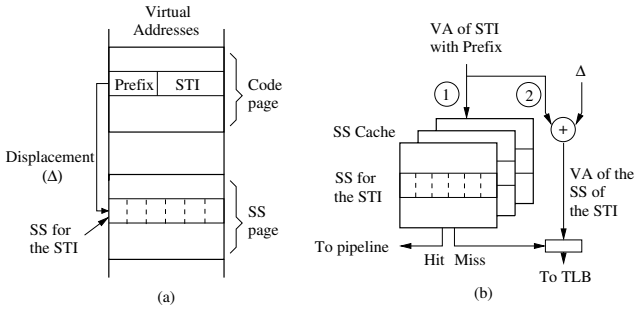


Fig. 8: Hardware solution to store and access the SS. In the figure, STI means Squashing or Transmit Instruction.

Figure 8(b) shows the action taken when a prefixed STI is decoded. The VA of the STI is sent to the SS cache (①). The SS cache is a small, set-associative cache that contains the most recently-used SSs. Due to the good locality of STIs in loops, most of the time, the SS cache hits. In this case, it provides the SS to the pipeline on time to be used when the STI is inserted in the ROB.

On an SS miss, the STI's VA is added to the Δ offset (②) to obtain the VA of the SS. This address is sent to the TLB to obtain the Physical Address (PA). After that, *but only when the STI reaches its Visibility Point (VP)*, a request is sent to the cache hierarchy to obtain the SS, and bring it to the SS cache. As a result, this STI is unable to use its SS; it will be used in a future invocation of the same STI when the SS request hits in the SS cache.

The SS cache does not introduce any side channel because no side effect occurs until the STI reaches its VP. Specifically, on an SS cache miss, we saw that the SS request is not sent to the cache hierarchy until the STI's VP, providing no information to the attacker. On an SS cache hit, the SS cache's LRU bits are not updated until the STI reaches its VP.

VII. Experimental Methodology

Architectures Modeled. We model the architecture shown in Table I using cycle-level simulations with Gem5 [6]. All the side effects of transient instructions are modeled. Our baseline architecture is a conventional processor with no protection against speculative-execution attacks. We call it UNSAFE.

Parameter	Value
Architecture	2.0 GHz out-of-order x86 core
Core	8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, TAGE branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache	32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher
L1-D Cache	64 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher
L2 Cache	2 MB, 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
SS Cache	64 sets, 4-way, 2 cycle RT latency, each entry has 12 10-bit PC offsets ($Trunc_{12}$). For 22nm: area is $0.0088mm^2$, dyn. rd. energy is 2.95pJ, leakage power is 2.31mW
IFB	76 entries. For 22nm: area is $0.0022mm^2$, dyn. rd. energy is 0.99pJ, leakage power is 0.58mW

TABLE I: Parameters of the simulated architecture.

We augment this architecture with several hardware defense schemes that use loads as the transmitters. We use the Comprehensive threat model, with both branches and loads as squashing instructions. The defense schemes are: (i) delaying with fences all speculative loads until they reach their Visibility Point (VP) [51] (FENCE); (ii) Delay-On-Miss, which delays speculative loads that miss in L1 until their VP [26], [38] (DOM); and (iii) InvisiSpec, which executes speculative loads invisibly before their VP [51] (INVISISPEC). We model these defense schemes as they are (D), augmented with the Baseline InvarSpec analysis (D+SS), and augmented with the Enhanced InvarSpec analysis (D+SS++). The resulting configurations are shown in Table II.

Configuration	Description
UNSAFE	Unmodified x86 architecture
FENCE	Delay all speculative loads with fences [51]
FENCE+SS	FENCE augmented with Baseline InvarSpec
FENCE+SS++	FENCE augmented with Enhanced InvarSpec
DOM	Delay speculative loads on L1 miss [26], [38]
DOM+SS	DOM augmented with Baseline InvarSpec
DOM+SS++	DOM augmented with Enhanced InvarSpec
INVISISPEC	Execute speculative loads invisibly [51]
INVISISPEC+SS	INVISISPEC augmented with Baseline InvarSpec
INVISISPEC+SS++	INVISISPEC augmented with Enhanced InvarSpec

TABLE II: Defense configurations modeled.

Applications and Analysis Pass. We run SPEC17 [7] and SPEC06 [15] applications with the reference input size. Because of simulation issues and binary analysis tool malfunction, we do not report on 2 applications out of 23 from SPEC17 and 4 out of 29 from SPEC06. For each application, we use SimPoint [13] to generate up to 10 representative intervals that accurately characterize the end-to-end performance of the application. Each interval contains 50 million instructions. We run Gem5 on each interval with system-call emulation mode with 1 million warm-up instructions.

Our InvarSpec analysis pass implementation is based on Radare2 [36], a state-of-the-art open-source binary analysis

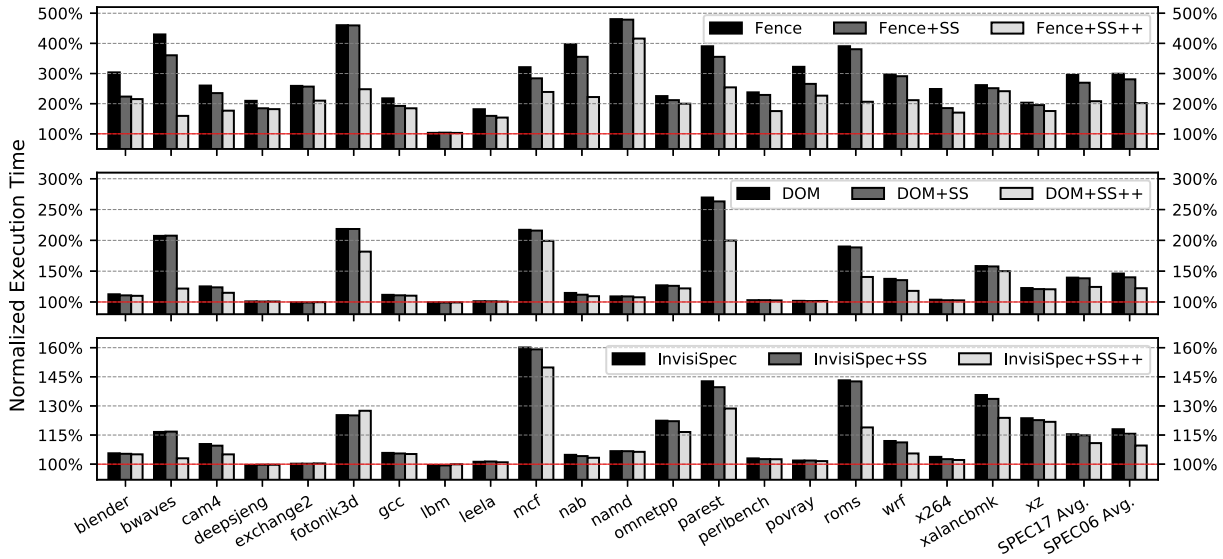


Fig. 9: Execution time of the applications on different architecture configurations, all normalized to UNSAFE. The three plots correspond, from top to bottom, to configurations related to the FENCE, DOM, and INVISISPEC defense schemes. Each plot has a different Y-axis range.

tool. It is performed on x86 binaries. However, it can also be implemented as a compiler pass and be performed on source code during compilation.

VIII. EVALUATION

A. Overall Performance Results

Figure 9 shows the execution time of SPEC17 and SPEC06 applications on all the configurations of Table II. The three plots correspond, from top to bottom, to configurations related to the FENCE, DOM, and INVISISPEC defense schemes. Each plot has a different Y-axis range. All bars are normalized to UNSAFE. Each plot shows each SPEC17 application, the average of SPEC17 applications, and the average of SPEC06 applications.

Going from top to bottom, we see that FENCE is the slowest scheme among all schemes evaluated. On average, it has an overhead of 195.3% on SPEC17 and 199.3% on SPEC06. FENCE+SS++ reduces the average overhead significantly, from 195.3% to 108.2% on SPEC17, and from 199.3% to 101.9% on SPEC06.

DOM exhibits a bimodal behavior on SPEC17 applications. While it has low overhead on about half of the applications, its overhead is very high on the rest. For example, the overhead is 169.6% on `parest` and 107.3% on `bwaves`. On average across all applications, DOM's overhead is 39.5% on SPEC17 and 46.1% on SPEC06. Adding support for Enhanced SS on top of DOM (DOM+SS++) substantially reduces this overhead. Enhanced SS is typically effective in the cases when DOM has high overhead. Specifically, it brings down `parest`'s overhead to 99.7% and `bwaves`'s to 21.8%. It does so by allowing cache-missing loads that are speculation invariant to proceed—rather than stalling them. On average, DOM+SS++ reduces the execution overhead from 39.5% to 24.4% on SPEC17, and from 46.1% to 22.3% on SPEC06.

INVISISPEC's average overhead is 15.4% on SPEC17 and 18.0% on SPEC06. This overhead is lower than the corresponding DOM overhead. INVISISPEC+SS++ speeds-up the execution over INVISISPEC. On average, the overhead of INVISISPEC+SS++ is only 10.9% on SPEC17 and 9.6% on SPEC06. In INVISISPEC+SS++, when a speculative load is ready to issue to memory, if it is speculation invariant, it is issued to memory normally; in INVISISPEC, the load is issued as an invisible load and hence requires two loads.

B. SS Analysis

We evaluate the performance impact of InvarSpec's design choices by conducting sensitivity studies for FENCE+SS++, DOM+SS++, and INVISISPEC+SS++ on SPEC17.

SS coverage. One design decision is how many bits to use to encode an *SS offset*, i.e., the distance between the PCs of a safe instruction and a transmitter. This number affects how many offsets an SS entry can encode.

Figure 10 shows the average normalized execution time of the schemes on SPEC17 when varying the number of bits per SS offset. The size of SS is fixed to 12 offsets. All data are normalized to the corresponding base hardware scheme (FENCE, DOM, and INVISISPEC). We see that, as the number of bits decreases, the execution time increases with different degrees. When the number of bits is smaller than 10, the performance degradation becomes non-negligible. Thus, our design uses 10 bits to encode an SS offset, which provides a performance similar to the unlimited number of bits.

Truncation. Another design decision is the SS size, namely the maximum number of SS offsets to keep in an SS entry.

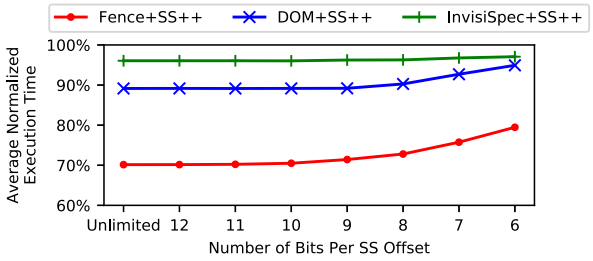


Fig. 10: Normalized execution time when varying the number of bits per SS offset. All execution times are normalized to their corresponding base hardware schemes without InvarSpec.

Figure 11 shows the average normalized execution time of the schemes with various SS sizes. Each SS offset is 10 bits. All data are normalized as in Figure 10. We see that, as the SS size increases, the execution time decreases. Compared to an unlimited SS size, all truncation configurations have a performance degradation. An SS size equal to 12 offsets is a good design point and, therefore, is our default design.

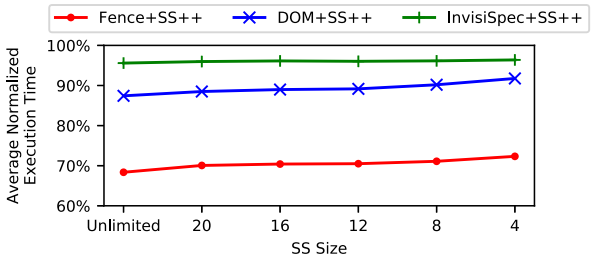


Fig. 11: Normalized execution time when varying the SS size. All execution times are normalized to their corresponding base hardware schemes without InvarSpec.

SS Cache. Figure 12 characterizes the SS cache. It shows the SS cache hit rate (right Y axis) and average normalized execution time of the applications (left Y axis) for different SS cache geometries. We compare our default configuration (4-way set-associative with 64 sets) to geometries with the same associativity but fewer or more sets. We also compare to a fully-associative cache of the same size (256 lines). All execution times are normalized as in Figure 10.

Increasing the SS cache size from our default configuration only slightly decreases the execution time of DOM+SS++ and INVISISPEC+SS++, but FENCE+SS++'s execution time keeps decreasing as the SS cache size grows. Decreasing the SS cache size from our default configuration increases the execution time of every scheme.

The average hit rate shows that the cache size is more important than the associativity. Reducing the cache size for the same associativity decreases the hit rate. However, for the same size, increasing the associativity from 4 to full causes a minimal change. Overall, our default design strikes a good trade-off between performance and hardware complexity.

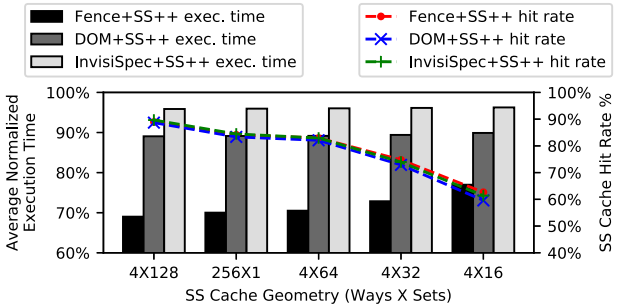


Fig. 12: Normalized execution time and SS cache hit rate when changing the SS cache. Execution times are normalized to the corresponding base hardware schemes without InvarSpec.

Memory Footprint. We measure that, on average, about half of the code pages in an application have at least one non-empty SS. To estimate an upper bound on the amount of memory required to store this SS state at run time, we add-up all the code pages in an application that have at least one non-empty SS. In reality, not all of the SS pages may be in memory at the same time. We call the resulting size the *Conservative SS Footprint*. We also measure the peak memory usage of each application with the reference input at any point in the program execution. We call the resulting size the *Peak Memory during Execution*.

Table III shows the two metrics for the 5 applications with the largest conservative SS footprint, and the average metrics across all SPEC17 applications. We can see that the memory overhead of storing the SS state is negligible compared to the peak memory that an application uses. The SS only causes a 0.55% memory overhead on average. For `blender`, which has the largest SS footprint, the overhead is only 1.32%.

SPEC17 App.	Conservative SS Footprint (MB)	Peak Memory during Execution (MB)
blender	8.24	626.31
perlbench	8.00	413.09
wrf	7.70	172.15
gcc	5.87	1277.55
cam4	5.27	853.91
SPEC17 Avg.	2.55	462.05

TABLE III: Assessing the memory footprint of the SS state.

C. Hardware Overhead

The main InvarSpec hardware is the SS cache and the IFB. The SS cache is relatively simple because it stores only read-only data. We used CACTI 7.0 [3] to estimate the area and power of the storage component of these structures for 22nm technology. As shown in Table I, the area, dynamic read energy, and leakage power of the storage structures is small.

D. Discussion

Interaction with a JIT Compiler. Our scheme is compatible with a JIT compilation environment. In this case, the dynamic

generation of a binary is augmented with a step that runs the InvarSpec analysis pass and generates the SSs. In practice, this step does not take long because it substantially reuses information that the compiler has just generated. We cannot provide an accurate estimate of this extra execution time because our implementation of the InvarSpec analysis pass is not optimized.

Reducing Execution Overhead Further. There are several approaches that could further reduce the execution overhead with InvarSpec. Three that come to mind are to increase the SS size, increase the SS cache size, and improve the Enhanced compiler analysis. The first approach can only decrease the execution overhead by a few percentage points. Indeed, Figure 11 showed the overhead with unlimited-sized SS entries, which is an upper bound. The second approach also gives modest gains. We have evaluated a configuration with an infinite SS cache with unlimited-sized SS entries. The result is that FENCE+SS++, DOM+SS++, and INVISISPEC+SS++ further reduce the average execution overhead from 108.2% to 90.4%, from 24.4% to 21.8%, and from 10.9% to 10.2%, respectively. The third approach, namely improving the Enhanced compiler analysis, may deliver more significant gains, especially if it involves adding inter-procedural analysis. Such approach likely involves non-trivial effort, and is our future work.

IX. Related Work

As indicated in Sections I and II-B, there are many defense schemes against speculation attacks. Some are software schemes, based on stopping speculation either with fences [2], [17], [43] or by injecting data-dependencies into the code [8], [43]. There are many hardware schemes (e.g., [1], [4], [20], [22], [26], [37], [38], [42], [51], [53]). Of these hardware schemes, many of those that do not consider timing attacks can be extended to support InvarSpec (e.g., [1], [20], [26], [38], [51]). InvarSpec enhances hardware techniques with software information.

The STT [53], SpecShield [4], and NDA [48] hardware schemes have a different threat model than those that InvarSpec extends in this paper. Indeed, the schemes in this paper protect all data from being leaked by speculative execution; STT, SpecShield, and NDA protect only data that is read by mis-speculated execution, and consider data in retired register file state not to be a secret.

To see the difference, consider the example in Figure 13. In the example code, although `secret` would not be leaked in a non-speculative execution, STT, SpecShield, and NDA do not apply protection to the mis-speculated `load(secret)` instruction, because `secret` was read into a register by a bound-to-commit instruction. In contrast, the schemes that InvarSpec extends in this paper do not allow performing the `load(secret)` without protection before the branch resolves.

Despite this difference in protection scope, the main principle of InvarSpec to statically analyze code and dynamically disable defense protection earlier could also be adapted to extend schemes such as STT, SpecShield, and NDA.

```
1  secret = load( secret_ptr );   // soon to commit
2  if (...) { // mispredicted branch
3    load( secret );
4  }
```

Fig. 13: Example that exposes the difference between protecting all data versus only speculatively-read data.

Finally, there are many designs that aim to block cache-based covert channels, using randomization [30], [47], encryption [35], [50], cache partitioning [27], [41], [46], [47], or other mechanisms [21], [29]. They do not address speculative execution attacks.

X. Conclusion

This paper introduced *Speculation Invariance*, and showed that it can be used to reduce the overhead of speculative execution defenses without changing security properties. It also presented the *InvarSpec* framework, which includes a program analysis pass to identify *Safe* instructions, and micro-architecture that uses this information to find and issue speculation invariant instructions earlier. InvarSpec is one of the first defense schemes for speculative execution that combines cooperative compiler and hardware mechanisms. It effectively enhances hardware defense schemes: it reduces the average execution overhead of fence protection from 195.3% to 108.2%, of DOM from 39.5% to 24.4%, and of InvisiSpec from 15.4% to 10.9%.

Acknowledgments

This work was funded in part by Intel under an Intel Strategic Research Alliance (ISRA) grant, NSF under grants CNS 1956007, CNS 1763658, and CCF 1725734, Blavatnik ICRC at TAU, and ISF under grant 2005/17. We thank Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander for kindly sharing their code. We also thank Dimitrios Skarlatos for his help in estimating hardware overhead.

Appendix: Store-to-Load Forwarding

When a load reaches its ESP and there is an older store, we need to ensure that whether the store and the load alias is invisible to an attacker. Otherwise, the attacker could deduce the address of the load. Specifically, if store and load alias and the load gets the data from the store, the attacker can deduce the alias by not observing a load access to the cache hierarchy. To solve this problem, we change the microarchitecture slightly as follows. The load is always issued to the cache hierarchy and if, at this point or later, the store address is found to alias, the load gets the data from the store and ignores the data returned from the cache hierarchy.

Relevant to InvarSpec is to understand when is the point where a load reaches its OSP. Such point requires not only that the load not be squashable anymore. In also requires that all of its older stores have been resolved—and hence that the load has been able to read the correct data, either from memory or from a store. InvarSpec implements this algorithm.

REFERENCES

[1] S. Ainsworth and T. Jones, "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State," in *International Symposium on Computer Architecture (ISCA)*, May 2020.

[2] ARM, "Cache Speculation Side-channels," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper, Oct. 2018.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[4] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2019.

[5] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[6] 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*, 2011.

[7] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[8] C. Carruth, "Speculative Load Hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution," *arXiv e-prints*, p. arXiv:1802.09085, Feb 2018.

[10] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *International Workshop on Information Hiding*. Springer, 2002, pp. 400–414.

[11] I. Cutress, "The Intel Second Generation Xeon Scalable: Cascade Lake, Now with Up To 56-Cores and Optane!" *AnandTech*, Apr. 2019. [Online]. Available: https://www.anandtech.com/show/14146/intel-xeon-scalable-cascade-lake-deep-dive-now-with-optane

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[13] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[14] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[15] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[16] J. Horn, "Speculative Store Bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=15282018.

[17] Intel, "Speculative Execution Side Channel Mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[18] ——, "Intel® 64 and IA-32 Architectures Software Developer's Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, Feb. 2019.

[19] ——, "Refined Speculative Execution Terminology," https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology, 2020.

[20] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[21] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," in *USENIX Security*, 2012.

[22] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987.

[23] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv e-prints*, p. arXiv:1807.03757, Jul 2018.

[24] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[25] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[26] 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 *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 264–276.

[27] J. Liedtke, N. Islam, and T. Jaeger, "Preventing denial-of-service attacks on a /spl mu/-kernel for WebOSes," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 73–79.

[28] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[29] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[30] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, p. 8–16, Sep. 2016. [Online]. Available: https://doi.org/10.1109/MM.2016.85

[31] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[32] L. B. Michael, M. J. Mihaljevic, S. Haruyama, and R. Kohno, "A framework for secure download for software-defined radio," *IEEE Communications Magazine*, vol. 40, no. 7, pp. 88–96, 2002.

[33] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," *arXiv preprint arXiv:1905.12701*, 2019.

[34] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[35] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 360–371.

[36] Radare2, "UNIX-like reverse engineering framework and command-line toolset," https://github.com/radareorg/radare2.

[37] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An Undo Approach to Safe Speculation," in *International Symposium on Microarchitecture (MICRO)*, October 2019.

[38] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 723–735.

[39] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[40] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 279–299.

[41] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2011, pp. 194–199.

[42] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support*

for Programming Languages and Operating Systems, 2019, pp. 395–410.

[43] P. Turner, "Retpoline: a Software Construct for Preventing Branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[44] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[45] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[46] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[47] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on computer architecture*, 2007, pp. 494–505.

[48] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[49] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018.

[50] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security*, 2019.

[51] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[52] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

[53] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.