

# Detecting speculative data flow vulnerabilities using weakest precondition reasoning

Graeme Smith<sup>[0000–0003–1019–4761]</sup>

Defence Science and Technology Group, Australia  
School of Electrical Engineering and Computer Science,  
The University of Queensland, Australia  
g.smith1@uq.edu.au

**Abstract.** Speculative execution is a hardware optimisation technique where a processor, while waiting on the completion of a computation required for an instruction, continues to execute later instructions based on a predicted value of the pending computation. It came to the forefront of security research in 2018 with the disclosure of two related attacks, Spectre and Meltdown. Since then many similar attacks have been identified. While there has been much research on using formal methods to detect speculative execution vulnerabilities based on predicted control flow, there has been significantly less on vulnerabilities based on predicted data flow. In this paper, we introduce an approach for detecting the data flow vulnerabilities, Spectre-STL and Spectre-PSF, using weakest precondition reasoning. We validate our approach on a suite of litmus tests used to validate related approaches in the literature.

## 1 Introduction

Modern processors liberally employ speculative execution of instructions to optimise performance. Instructions can be executed before earlier instructions in a program based on predictions of the outcomes of the earlier instructions. The intention is to use latent processing cycles rather than waiting for the completion of computations required for the earlier instructions. When a prediction is found to be correct, the speculatively executed instructions are committed to memory. When a prediction is found to be incorrect, the speculatively executed instructions are rolled back, and execution restarted according to the actual outcome.

While the rollback of incorrect speculation maintains a program’s functionality, traces of speculative execution are left in the processor’s micro-architecture and can be exploited by an attacker to gain access to otherwise inaccessible (and hence potentially sensitive) data. The best known such attack, Spectre variant 1 (also known as Spectre-PHT) [22], takes advantage of the pattern history table (PHT), a micro-architectural component used to predict the outcome of a branch instruction. After finding a suitable *gadget* (i.e., code pattern) in a victim program, an attacker can train the PHT to expect a particular outcome and then use this to exploit the gadget. For example, an attacker could train the

PHT to expect the following gadget to execute the body of the if statement. Then, by providing a value of  $x$  greater than  $array1\_size$ , a value beyond the end of  $array1$  is accessed in the if statement’s body. This value is subsequently used to access a particular index of  $array2$ , reading the value at that index into the cache. After rollback, this index can be deduced by a timing attack on the cache [23]. Note that 512 corresponds to the cache line size in bits allowing the attacker to determine, from the affected cache line, the value  $array1[x]$ .

```

r0 := x;
r1 := array1_size;
if (r0 < r1){
    r1 := array1[r0];
    r2 := array2[r1 * 512];
}

```

Since the disclosure of such attacks in 2018, a number of formal methods-based approaches for detecting vulnerable gadgets have been developed [8]. These have mostly focused on attacks exploiting speculation of control flow in a program, such as Spectre-PHT. Significantly less work exists on attacks exploiting speculation of data flow such as Spectre variant 4 (Spectre-STL)[6]<sup>1</sup>. This attack exploits the incorrect prediction that a load is not dependent on an earlier store and hence can be executed first, missing the Store-To-Load (STL) dependency. A similar attack, Spectre-PSF[7, 26], relies on the processor incorrectly predicting that a load *will* depend on an earlier store and speculatively executing the load using the store’s value, referred to as Predictive Store Forwarding (PSF).

In this paper, we provide a weakest precondition-based approach to detecting the data flow Spectre variants, Spectre-STL and Spectre-PSF, building on a recent approach for Spectre-PHT [10]. The existing approach is detailed in Section 2. In Section 3, we describe the data flow variants of Spectre with simple examples before presenting our formal approach to their detection in Sections 4 and 5. In Section 6 we discuss related formal approaches before concluding in Section 7.

## 2 Background

Winter et al. [30] present an information flow logic based on the weakest precondition (wp) reasoning of Dijkstra [15, 16]. The logic introduces additional proof obligations to standard wp rules to ensure a form of *non-interference* [19]: the proof obligations fail when sensitive information can be leaked to publicly accessible variables or through observation of control flow.

This logic forms the basis of the approach to detecting Spectre-PHT vulnerable code by Coughlin et al. [10]. Following [9], that approach employs the notion of a speculative context to track the effects of speculative execution. This

<sup>1</sup> Originally described by Jann Horn at <https://project-zero.issues.chromium.org/issues/42450580>.

is incorporated in a weakest precondition transformer  $wp_s$  which operates over *pairs* of predicates  $\langle Q_s, Q \rangle$ . The predicate  $Q_s$  represents the weakest precondition at that point in the program, assuming the processor is speculating, and  $Q$  the weakest precondition when it is not speculating. The security of a program ultimately depends on the non-speculative predicate  $Q$  holding in the program's initial state: proof obligations from the speculative state  $Q_s$  are taken into account by being transferred to the non-speculative state at points in the program where speculation can begin.

The rules of  $wp_s$  are defined over a high-level programming language representing assembly programs. The syntax of an instruction,  $\alpha$ , and a program,  $p$ , is defined as follows.

$$\begin{aligned} \alpha &::= \text{skip} \mid r := e \mid r := x \mid x := e \mid \text{fence} \mid \text{leak } e \\ p &::= \alpha \mid p ; p \mid \text{if } (b) \{p\} \text{ else } \{p\} \mid \text{while } (b) \{p\} \end{aligned}$$

where  $r$  is a register,  $x$  is a local or global variable (i.e., a memory location which in this paper can be an array access of the form  $a[e]$ ),  $b$  a Boolean condition and  $e$  an expression. Both  $b$  and  $e$  are in terms of registers and literals only, as in assembly code. The language includes a `fence` instruction which prevents reordering of instructions (in the context of a processor's memory model) and also terminates current speculative execution. A special *ghost* instruction<sup>2</sup> `leak`  $e$  is inserted into a program to indicate that the following instruction(s) are part of a gadget that leaks the value  $e$  through a micro-architectural side channel when executed (speculatively, or otherwise).

Before analysing a program with the logic, `leak` instructions are inserted for each gadget of interest during a pre-pass over the code. Since typical gadgets can be detected syntactically, this is a straightforward task to mechanise. The expression  $e$  of the inserted `leak` instruction is based on what information leaks when the gadget is used in an attack. For the example of Section 1, `leak r1` would be inserted immediately above the access to `array2`. After this pre-pass, the code is analysed using the logic to determine whether the information leaked is possibly sensitive and hence the gadget causes a security vulnerability. Since the pre-pass can be customised for different gadgets, the overall approach can be adapted to a variety of attacks, including new attacks as they are discovered.

## 2.1 Rules of $wp_s$

**Skip** A `skip` instruction does not change the  $\langle Q_s, Q \rangle$  tuple.

$$wp_s(\text{skip}, \langle Q_s, Q \rangle) = \langle Q_s, Q \rangle$$

**Register update** For each register or variable  $v$  in a program, the logic includes an expression  $\Gamma_v$  which evaluates to the *security level* of the information held by the variable. The possible values of security levels form a lattice  $(L, \sqsubseteq)$  where

<sup>2</sup> A ghost instruction is not part of the actual code and is used for analysis purposes only.

each pair of elements  $a, b \in L$  has a *join*, i.e., least upper bound, denoted by  $a \sqcup b$ , and a *meet*, i.e., greatest lower bound, denoted by  $a \sqcap b$ . The rule for updating a register  $r$  to the value of an expression  $e$  updates both  $r$  and  $\Gamma_r$  as follows, where  $\Gamma_E(e) = \bigsqcup_{r \in \text{regs}(e)} \Gamma_r$  is the join of the security levels of the registers,  $\text{regs}(e)$ , to which  $e$  refers.

$$wp_s(r := e, \langle Q_s, Q \rangle) = \langle Q_s[r, \Gamma_r \setminus e, \Gamma_E(e)], Q[r, \Gamma_r \setminus e, \Gamma_E(e)] \rangle$$

where  $Q[x_1, \dots, x_n \setminus y_1, \dots, y_n]$  replaces each free occurrence of  $x_i$  (for  $1 \leq i \leq n$ ) in  $Q$  with  $y_i$ .

**Load** Each variable  $x$  has a programmer-defined *security policy*  $\mathcal{L}(x)$  denoting the highest security level that  $x$  may hold. This level may vary as the program executes [25, 24] and hence  $\mathcal{L}(x)$  is an expression in terms of other variables. For example,  $\mathcal{L}(x) = (\text{if } y = 0 \text{ then } \textit{secret} \text{ else } \textit{public})$ , where  $\textit{secret}, \textit{public} \in L$ , captures that variable  $x$  may hold *secret* information when  $y = 0$  and *public* information otherwise.

When loading the value of a variable  $x$  into a register  $r$ , it is possible that the security level of that value is undefined, e.g., when it has been set to an input value. Hence,  $\Gamma_r$  in the non-speculative state  $Q$  is updated to the meet of  $\Gamma_x$  and its maximum possible value,  $\mathcal{L}(x)$ .

In the speculative state  $Q_s$ ,  $r$  and  $\Gamma_r$  are updated with values from memory (referred to as the *base state* and denoted with a  $\flat$  superscript) when  $x$  is not defined in the speculative context, i.e., an earlier store to  $x$  has not occurred, and  $x$  and  $\Gamma_x$  otherwise. This is required to support concurrency since, during speculative execution, another thread may change a value in memory (the base state) but cannot change the corresponding value in the speculative state. Hence, values in the base state and speculative state can differ. The superscripts avoid the base state variables being affected by speculatively executed assignments.

Whether or not a variable  $x$  has been defined in the speculative context is captured by a Boolean ghost variable  $x_{def}$ .

$$wp_s(r := x, \langle Q_s, Q \rangle) = \langle (x_{def} \Rightarrow Q_s[r, \Gamma_r \setminus x, \Gamma_x]) \wedge (\neg x_{def} \Rightarrow Q_s[r, \Gamma_r \setminus x^\flat, \Gamma_{x^\flat} \sqcap \mathcal{L}(x)[\text{var} \setminus \text{var}^\flat]]), Q[r, \Gamma_r \setminus x, \Gamma_x \sqcap \mathcal{L}(x)] \rangle$$

where  $\text{var}$  is the list of program variables, and  $\text{var}^\flat$  the same list with each element decorated with a  $\flat$  superscript. Note that when  $x_{def}$  holds, we can use  $\Gamma_x$  directly, rather than the meet with  $\mathcal{L}(x)$ .

**Store** A store to a variable,  $x := e$ , sets  $x_{def}$  to true and replaces each occurrence of variable  $x$  and  $\Gamma_x$  with expression  $e$  and security level  $\Gamma_E(e)$ , respectively, in both  $Q_s$  and  $Q$ . Additionally, in the non-speculative case non-interference is ensured by checking that

- (i) the security level of  $e$  is not higher than the security classification of  $x$ , and
- (ii) since  $x$ 's value may affect the security classification of other variables, for each such variable  $y$ ,  $y$ 's current security level  $\Gamma_y \sqcap \mathcal{L}(y)$  does not exceed its updated security classification when  $x$  is set to  $e$ .

Such checks are not required in the speculative case since, while speculating, values are not written to shared memory.

$$\begin{aligned} wp_s(x := e, \langle Q_s, Q \rangle) = & \langle Q_s[x, \Gamma_x, x_{def} \setminus e, \Gamma_E(e), true], \\ & Q[x, \Gamma_x \setminus e, \Gamma_E(e)] \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\ & (\forall y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \setminus e]) \rangle \end{aligned}$$

**Fence** The fence instruction terminates any current speculative execution. Hence, any proof obligations in the speculative state beyond the fence do not need to be considered at the point in the program where a fence occurs.  $Q_s$  is therefore replaced by *true* and  $Q$  is unchanged.

$$wp_s(\text{fence}, \langle Q_s, Q \rangle) = \langle true, Q \rangle$$

**Leak** The instruction *leak*  $e$  leaks the value of expression  $e$  via a micro-architectural side channel, introducing a proof obligation into both  $Q_s$  and  $Q$ .

$$wp_s(\text{leak } e, \langle Q_s, Q \rangle) = \langle Q_s \wedge \Gamma_E(e) = \perp, Q \wedge \Gamma_E(e) = \perp \rangle$$

where  $\perp$  denotes the lowest value of the security lattice. Requiring that the leaked information is at this level ensures that the attacker cannot deduce anything new from the information, regardless of the level of information they can observe.

**Sequential composition** As in standard wp reasoning, sequentially composed instructions transform the tuple one at a time.

$$wp_s(p_1 ; p_2, \langle Q_s, Q \rangle) = wp_s(p_1, wp_s(p_2, \langle Q_s, Q \rangle))$$

**If statement** In the case of Spectre-PHT, speculation can begin at an if statement. Hence, it is at this point in the reasoning that the speculative proof obligation manifests itself as a proof obligation in the non-speculative state. For ease of presentation, we assume that the guard  $b$  does not change during speculation, hence the speculative proof obligation can be evaluated in the context of the guard.<sup>3</sup> The speculative proof obligation is from the opposite branch to the one that should be executed, with each variable  $x_{def}$  set to false (leaving just the predicates in terms of base variables) and all  $b$  subscripts removed (to identify these base variables with variables in the non-speculative state).

There is an additional proof obligation  $\Gamma_E(b) = \perp$  on the non-speculative state since, in concurrent programs, the value of  $b$  can readily be deduced using timing attacks (even when the statement's branches do not change publicly accessible variables) [24, 29]. An if statement might occur within a speculative context (when nested in or following an earlier if statement, for example). The branch that is followed speculatively is, in general, independent of that actually executed later. Hence, the speculative proof obligations from both branches are conjoined to form the speculative precondition.

Given  $\langle Q_{s1}, Q_1 \rangle = wp_s(p_1, \langle Q_s, Q \rangle)$  and  $\langle Q_{s2}, Q_2 \rangle = wp_s(p_2, \langle Q_s, Q \rangle)$ , we have

<sup>3</sup> An alternative rule that does not require this assumption is provided in [10].

$$\begin{aligned}
wp_s(\text{if}(b)\{p_1\}\text{ else } \{p_2\}, \langle Q_s, Q \rangle) = \\
\langle Q_{s1} \wedge Q_{s2}, (b \Rightarrow Q_1 \wedge Q_{s2}[var^b, d_1, \dots, d_n \setminus var, false, \dots, false]) \wedge \\
(\neg b \Rightarrow Q_2 \wedge Q_{s1}[var^b, d_1, \dots, d_n \setminus var, false, \dots, false]) \rangle \wedge \\
\Gamma_E(b) = \perp \rangle.
\end{aligned}$$

where  $d_1, \dots, d_n$  is the list of ghost variables of the form  $x_{def}$ .

**While loop** Speculation can also begin at each iteration of a while loop. Similarly to standard wp reasoning, we can soundly approximate the weakest precondition of a loop by finding invariants which imply our speculative and non-speculative postconditions. As with the if rule, a proof obligation  $\Gamma_E(b) = \perp$  must hold in the non-speculative case.

$$wp_s(\text{while}(b)\{p\}, \langle Q_s, Q \rangle) = \langle Inv_s, Inv \rangle$$

where  $Inv_s \Rightarrow Q_s$ ,  $Inv \Rightarrow \Gamma_E(b) = \perp \wedge Inv_s[var^b, d_1, \dots, d_n \setminus var, false, \dots, false]$  and  $Inv \wedge \neg b \Rightarrow Q$ , and given  $wp_s(p, \langle Inv_s, Inv \rangle) = \langle P_s, P \rangle$ , then  $Inv_s \Rightarrow P_s$  and  $Inv \wedge b \Rightarrow P$ . Like the if rule, the while rule copies the proof obligations in the speculative precondition to the non-speculative precondition, and maintains those in the speculative precondition in case the loop is reached within an existing speculative context.

## 2.2 Using $wp_s$

The property that  $wp_s$  verifies, when the calculated weakest precondition of a program holds, is *value-dependent non-interference* based on the definition in [25]. This property states that, given two initial states  $s_1$  and  $s_2$  which agree on the values of variables which are non-sensitive, after executing a prefix of instructions  $t$  of the program on each state, the resulting states will continue to agree on the values of variables which are non-sensitive. In other words, the values of variables which are sensitive have no effect on those that are non-sensitive (and hence the sensitive values cannot be deduced from observations of the non-sensitive values). Formally, given a program  $c$  with precondition  $P$  and postcondition  $Q$ <sup>4</sup>

$$\begin{aligned}
P \Rightarrow wp_s(c, Q) \Rightarrow \\
\forall s_1, s_2 \in P, t \leq c \cdot \forall s'_1 \cdot s_1 \sim s_2 \wedge s_1 \rightarrow_t s'_1 \Rightarrow \exists s'_2 \cdot s_2 \rightarrow_t s'_2 \wedge s'_1 \sim s'_2
\end{aligned}$$

where  $t \leq c$  denotes that  $t$  is a prefix of  $c$ ,  $s_1 \sim s_2$  denotes  $s_1$  and  $s_2$  agree on non-sensitive values, and  $s_1 \rightarrow_t s'_1$  denotes  $s'_1$  is reached from  $s_1$  by instructions  $t$ . Note that since the programming language is deterministic, the above property implies that all states reached from  $s_2$  by  $t$  agree with the non-sensitive values of  $s'_1$ .

To support its use in a concurrent setting,  $wp_s$  also supports rely/guarantee reasoning [21, 31]. To detect additional vulnerabilities that arise due to a processor's memory model, it is paired with a notion of reordering interference freedom

<sup>4</sup>  $\Rightarrow$  denotes logical entailment and binds less tightly than implication ( $\Rightarrow$ ).

(rif) [11, 12]. These techniques (see [10] for details) are independent of the details of the logic’s rules and can be equally applied to the extensions to  $wps$  in this paper.

### 3 Data Flow Spectre Variants

In addition to attacks related to speculation on control flow, such as Spectre-PHT of Section 1, attacks have been identified based on speculation on data flow; specifically speculation on dependencies between stores and subsequent loads. The most well-known of these is Spectre variant 4 (also known as Spectre-STL) [6]. This attack relies on a processor’s memory disambiguator mispredicting that a load is independent of an earlier store, and hence executing the load with a stale value.

#### 3.1 Spectre-STL

We illustrate Spectre-STL on Case 4 of the 13 litmus tests developed by Daniel et al. [13] and available at [https://github.com/binsec/haunted\\_bench/blob/master/src/litmus-stl/programs/spectrev4.c](https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c). The test is reexpressed in the language from Section 2. In the code below,  $idx$  is an input provided by the user who may be an attacker,  $secretarray$  is a publicly inaccessible array containing sensitive data and has length  $array\_size$ , and  $publicarray2$  is a publicly accessible array which has length  $512 \cdot 256$  (512 is the cache line size in bits, and 256 the number of integers representable using 8 bits).

```

r0 := idx;
r1 := array_size;
r0 := r0 & (r1 - 1);
secretarray[r0] := 0;    // This store may be bypassed
r1 := secretarray[r0];
r2 := publicarray2[r1 * 512];

```

The code begins by calculating the bitwise AND of  $idx$  and  $array\_size - 1$  to obtain a valid index of  $secretarray$ . This avoids an array bounds bypass as in the Spectre-PHT attack. The value at the calculated index is set to 0, a non-sensitive value. This value is then read and used to read a value from  $publicarray2$ . The multiplication by 512 in the final step allows the value read from  $secretarray$  to be deduced via a subsequent timing attack (by detecting the cache line affected by the read of  $publicarray2$ ).

This code is secure provided the value used to read  $publicarray2$  is the non-sensitive value 0. However, if it is run and the memory disambiguator mispredicts that the load of  $secretarray[r0]$  is independent of the prior store then the load can be executed first. In this case, a sensitive value will be used in the read from  $publicarray2$ . To prevent bypassing the store in this way, a typical mitigation is to insert a fence instruction after the store to  $secretarray$  [26].

### 3.2 Spectre-PSF

Spectre-PSF is a variant of Spectre-STL where, rather than mispredicting that a dependency does not exist between a load and earlier store, the memory disambiguator mispredicts that a dependency *does* exist [7, 26]. This behaviour has been confirmed as being possible on the AMD Zen 3 processor. We illustrate Spectre-PSF via an exploitable gadget from [26] (based on example code from AMD). The gadget is reexpressed in the language of Section 2. In the code below,  $idx$  is an input provided by the user,  $A$  is a public array of size 16,  $C$  is a public array of length  $C\_size=2$  initialised to  $[0,0]$ , and  $B$  is a public array of size  $512*256$ .

```

r0 := idx;
r1 := C_size;
if (r0 < r1) {
    C[0] := 64;
    r1 := C[r0];    // Value 64 may be forwarded to r1
    r1 := A[r1 * r0];
    r2 := B[r1 * 512];
}

```

Ignoring speculation on the branch, the code is secure provided that the value loaded from  $C[r0]$  is 64 only when  $idx$  (and hence  $r0$ ) is 0: the value loaded from  $A$  will be the publicly accessible value at index 0 when  $idx$  is either 0 or 1. However, if the processor mispredicts a dependency between the store to  $C[0]$  and the load from  $C[r0]$  when  $idx$  is 1 then the value 64 can be (incorrectly) forwarded to the load. That is,  $r1$  will be set to 64 and subsequently value  $A[64]$  will be used in the index of  $B$  in the final load. This access of  $A$  will be out of bounds and hence to potentially sensitive data. Again, a fence after the store can be used to mitigate the vulnerability.

## 4 Detecting Spectre-STL

The  $wp_s$  logic in Section 2 assumes that speculation starts only at branching points (of if statements or while loops). To detect the data flow variants of Spectre, we need to also allow speculation to start at stores. For Spectre-STL, when a store is reached during execution, we can begin speculating that it is not required for the following code, and hence can be bypassed (the store executing later after the following code).

Given the code  $s; c_1; c_2$  where  $s$  is a store and  $c_1$  and  $c_2$  are sequences of instructions, when the code of  $c_1$  is not dependent on  $s$ , speculation over  $c_1$  will lead to the execution  $c_1; s; c_2$ , where  $s$  has effectively been reordered after the instructions in  $c_1$ . When one or more instructions in  $c_1$  are dependent on  $s$ , speculating over  $c_1$  will lead to the execution  $\mathbf{spec}(c_1); s; c_1; c_2$ , where  $\mathbf{spec}(c_1)$  includes rolling back the speculation's effects and hence has no affect on the program, but may alter the processor's microarchitecture.

In practice, the number of instructions in  $c_1$  is limited by the processor’s *speculation window*, i.e., the upper bound on the number of instructions that can execute speculatively. This bound will depend on the microarchitectural components involved in the speculation. For Spectre-STL, it will depend on the size of the store buffer where bypassed store instructions wait to be executed, i.e., committed to memory. The size of this buffer can be up to 106 stores<sup>5</sup> and hence, in general, beyond the size of the single procedures we are targeting in our work. Hence, as in  $wp_s$  we assume speculation can continue to the end of our code and do not explicitly model a speculation window. This results in a logic that is sound (as we check vulnerabilities within *any* sized speculation window), but can lead to false positives in cases where the actual speculation window is shorter than the code remaining to be executed.

To extend  $wp_s$  to detect Spectre-STL vulnerabilities, we modify the store rule as follows.

- (i) The speculative postcondition  $Q_s$  is added to the non-speculative precondition. By transferring the speculative *post*condition, we effectively ignore the store, reflecting that it does not occur as part of the speculation. As in the if statement rule, all ghost variables  $y_{def}$  are replaced by false (to leave just the predicates in terms of the base variables) and each base variable  $y^b$  is replaced by  $y$  (to identify these variables with variables in the non-speculative predicate).
- (ii) The speculative postcondition is also added to the speculative precondition. This reflects the case where the speculation on the store occurs in the context of an ongoing speculative execution. In this case, the store (being bypassed) will have no effect on the ongoing execution. For example, the rule will not cause a proof obligation  $\Gamma_x = \perp$  to be resolved by a store  $x := 0$  (where the literal 0 is a non-sensitive value).

The resulting rule is formalised below (where the additions to the original store rule from Section 2, corresponding to (i) and (ii) above, are underlined).

$$\begin{aligned}
 wp_{STL}(x := e, \langle Q_s, Q \rangle) = & \langle \underline{Q_s} \wedge Q_s[x, \Gamma_x, x_{def} \setminus e, \Gamma_E(e), true], \\
 & \underline{Q[x, \Gamma_x \setminus e, \Gamma_E(e)]} \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\
 & (\forall y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \setminus e]) \wedge \\
 & \underline{Q_s[var^b, d_1, \dots, d_n \setminus var, false, \dots, false]} \rangle
 \end{aligned} \tag{1}$$

where  $d_1, \dots, d_n$  is the list of ghost variables of the form  $y_{def}$ .

To illustrate the utility of this rule, we apply it (along with other rules of  $wp_s$ ) to the litmus test from Section 3.1 in Figure 1, and to the same litmus test with a fence inserted to prevent speculation in Figure 2. In both cases, a leak instruction is added before the access to `publicarray2`.

The introduced leak instruction adds proof obligations in both the speculative and non-speculative states that  $\Gamma_{r1}$  is  $\perp$ . Preceding backwards through the proof of Figure 1, these obligations are transformed by the load to `r1` to conditions on

<sup>5</sup> <https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2>

```

⟨⟨idxdef ⇒
  (secretarray[idx & (array_size - 1)]def ⇒ Γsecretarray[idx & (array_size - 1)] = ⊥) ∧
  (¬secretarray[idx & (array_size - 1)]def ⇒ Γsecretarray[idx & (array_size - 1)]b = ⊥)⟩ ∧
  (¬idxdef ⇒
    (secretarray[idxb & (array_size - 1)]def ⇒ Γsecretarray[idxb & (array_size - 1)] = ⊥) ∧
    (¬secretarray[idxb & (array_size - 1)]def ⇒ Γsecretarray[idxb & (array_size - 1)]b = ⊥)),
  Γsecretarray[idx & (array_size - 1)] = ⊥
  r0 := idx;
  ⟨⟨secretarray[r0 & (array_size - 1)]def ⇒ Γsecretarray[r0 & (array_size - 1)] = ⊥) ∧
  (¬secretarray[r0 & (array_size - 1)]def ⇒ Γsecretarray[r0 & (array_size - 1)]b = ⊥),
  Γsecretarray[r0 & (array_size - 1)] = ⊥
  r1 := array_size; // array_size = array_sizeb since array_size is a constant
  ⟨⟨secretarray[r0 & (r1 - 1)]def ⇒ Γsecretarray[r0 & (r1 - 1)] = ⊥) ∧
  (¬secretarray[r0 & (r1 - 1)]def ⇒ Γsecretarray[r0 & (r1 - 1)]b = ⊥),
  Γsecretarray[r0 & (r1 - 1)] = ⊥
  r0 := r0 & (r1 - 1);
  ⟨⟨secretarray[r0]def ⇒ Γsecretarray[r0] = ⊥) ∧
  (¬secretarray[r0]def ⇒ Γsecretarray[r0]b = ⊥), Γsecretarray[r0] = ⊥
  secretarray[r0] := 0; // This store may be bypassed
  ⟨⟨secretarray[r0]def ⇒ Γsecretarray[r0] = ⊥) ∧
  (¬secretarray[r0]def ⇒ Γsecretarray[r0]b = ⊥), Γsecretarray[r0] = ⊥
  r1 := secretarray[r0];
  ⟨Γr1 = ⊥, Γr1 = ⊥
  leak r1;
  ⟨true, true⟩
  r2 := publicarray2[r1*512];
  ⟨true, true⟩

```

**Fig. 1.** Spectre-STL litmus test (code highlighted in gray).

$secretarray[r0]$ ; in the speculative case this condition is dependent on whether  $secretarray[r0]$  is defined during the speculation.

The interesting step is the store to  $secretarray[0]$ . Since the value stored is non-sensitive, the proof obligation is satisfied in the non-speculative case (assuming  $secretarray[0]$  is not used in the security classification  $\mathcal{L}$  of another variable). Hence, the non-speculative precondition of the store includes only the transferred condition from the speculative postcondition, i.e.,  $\Gamma_{secretarray[r0]} = \perp$ . The speculative precondition is equivalent to the speculative postcondition: the second conjunct of the precondition in rule (1) evaluates to true when  $secretarray[r0]$  is defined and  $secretarray[r0]$  is non-sensitive.

Proceeding further backwards through the proof, the index used to access  $secretarray$  is replaced with  $idx \ \& \ (array\_size - 1)$ . Thus, the final non-speculative precondition is  $\Gamma_{secretarray[idx \ \& \ (array\_size - 1)]} = \perp$ , indicating that the code is secure provided that this condition holds initially. This is more precise than a

```

⟨true, true⟩
r0 := idx;
⟨true, true⟩
r1 := array_size;
⟨true, true⟩
r0 := r0 & (r1 -1);
⟨true, true⟩
secretarray[r0] := 0;    // This store may no longer be bypassed
⟨true,  $\Gamma_{secretarray[r0]} = \perp$ ⟩
fence;
⟨( $secretarray[r0]_{def} \Rightarrow \Gamma_{secretarray[r0]} = \perp$ )  $\wedge$ 
( $\neg secretarray[r0]_{def} \Rightarrow \Gamma_{secretarray^b[r0]} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
r1 := secretarray[r0];
⟨ $\Gamma_{r1} = \perp, \Gamma_{r1} = \perp$ ⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1*512];
⟨true, true⟩
    
```

**Fig. 2.** Spectre-STL litmus test with fence mitigation applied.

simple syntactic analysis which identifies the gadget, but does not define the conditions under which it can be successfully exploited.

The proof in Figure 2 is identical before the fence instruction is reached (i.e., below the fence instruction). At this point, the speculative predicate becomes true and hence no condition is transferred to the non-speculative precondition at the store instruction. The result is that the final non-speculative precondition is true, indicating that the code is always secure.

To further validate our rule, we applied it (along with other required rules from  $wp_s$ ) to the remaining 12 litmus tests of Daniel et al. [13] (see Appendix A) and for each of the 9 litmus tests with a vulnerability, we applied it to a version of the litmus test with a fence added as a mitigation. All vulnerabilities were detected and all tests with mitigations showed the vulnerability could no longer occur. However, there is one test where we detect a vulnerability and Daniel et al. do not. This test, Case 9, is the same as Case 4 but includes a loop after the store which is intended to fill the reorder buffer<sup>6</sup>, forcing the store to be evaluated and take effect in memory before the load from *publicarray2*. Since our logic supports detection of Spectre-PHT (as well as Spectre-STL), it allows the loop to speculatively exit early. In general, our logic detects multiple variants of Spectre including, as in this case, vulnerabilities that arise due to their combination.

<sup>6</sup> The reorder buffer contains *all* speculated instructions and provides an upper limit on the number of instructions that can be speculatively executed.

## 5 Detecting Spectre-PSF

Store forwarding refers to using the value of a store instruction in a subsequent load instruction before the store has taken effect in memory. This can be done safely when the store and load are to the same address. On some processors, store forwarding can be done speculatively based on a prediction that a store and subsequent load are to the same address. This leads to the Spectre-PSF vulnerability described in Section 3.2.

Abstracting from how the prediction is made, our rule reflects that the value of a store instruction, can be used speculatively in *any* subsequent load. When there is no leak or a given load does not cause a leak, the misprediction is benign and does not manifest in our reasoning. When the load does cause a leak, the variable associated with the load will appear in the postcondition of the store. For each subset of such variables, we replicate the speculative proof obligation with the variables replaced by the value of the store. This captures all possible predictions including those in which the value of the store is forwarded to more than one subsequent load. These additional proof obligations are also transferred to the non-speculative precondition of the store, reflecting that speculation may have begun at the store. The rule is formalised below (with the additions to the Spectre-STL store rule from Section 4 underlined).

$$\begin{aligned}
 wp_{PSF}(x := e, \langle Q_s, Q \rangle) = & \\
 & \frac{\langle \forall \{y_1, \dots, y_m\} \subseteq vars(Q_s). \\
 & (Q_s \wedge Q_s[x, \Gamma_x, x_{def} \setminus e, \Gamma_E(e), true])[y_1, \dots, y_m \setminus e, \dots, e], \\
 & Q[x, \Gamma_x \setminus e, \Gamma_E(e)] \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\
 & (\forall y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \setminus e]) \wedge \\
 & \forall \{y_1, \dots, y_m\} \subseteq vars(Q_s). \\
 & Q_s[var^b, d_1, \dots, d_n \setminus var, false, \dots, false][y_1, \dots, y_m \setminus e, \dots, e] \rangle}{\langle \forall \{y_1, \dots, y_m\} \subseteq vars(Q_s). \\
 & (Q_s \wedge Q_s[x, \Gamma_x, x_{def} \setminus e, \Gamma_E(e), true])[y_1, \dots, y_m \setminus e, \dots, e], \\
 & Q[x, \Gamma_x \setminus e, \Gamma_E(e)] \wedge \Gamma_E(e) \sqsubseteq \mathcal{L}(x) \wedge \\
 & (\forall y \cdot \Gamma_y \sqcap \mathcal{L}(y) \sqsubseteq \mathcal{L}(y)[x \setminus e]) \wedge \\
 & \forall \{y_1, \dots, y_m\} \subseteq vars(Q_s). \\
 & Q_s[var^b, d_1, \dots, d_n \setminus var, false, \dots, false][y_1, \dots, y_m \setminus e, \dots, e] \rangle}
 \end{aligned} \tag{2}$$

where  $vars(\mathcal{L}(x))$  denotes the list of variables occurring free in  $\mathcal{L}(x)$ , and  $d_1, \dots, d_n$  is the list of ghost variables of the form  $y_{def}$ . Note that when the set  $\{y_1, \dots, y_m\}$  is the empty set, the predicate in both the speculative and non-speculative preconditions are equivalent to those of the STL rule. Hence, this rule will detect vulnerabilities to both Spectre-STL and Spectre-PSF.

To illustrate rule (2), we apply it to the litmus test from Section 3.2 in Figure 3. Each load of an array value ( $B[r1]$ ,  $A[r1 * r0]$  and  $C[r0]$ ) introduces a potential leak. Note that the leak due to the load of  $C[r0]$  does not change the state tuple  $\langle Q_s, Q \rangle$  since both the non-speculative and speculative predicates already imply  $\Gamma_{r0} = \perp$ . For the non-speculative precondition of the store  $C[0] := 64$ , the postcondition  $\Gamma_{r0} = \perp$  is unchanged, the postcondition  $\Gamma_{C[r0]} = \perp$  is transformed to true, and the postcondition  $\Gamma_{A[C[r0]*r0]} = \perp$  is transformed to  $\Gamma_{A[C[0 \mapsto 64][r0]*r0]} = \perp$  where  $C[0 \mapsto 64]$  is array  $C$  with element 0 equal to 64.

In addition, for each subset of global variables in the non-speculative postcondition, we need to transfer the required predicate to the non-speculative precondition. There are two global variables,  $A[C[r0]*r0]$  and  $C[r0]$ , and hence four subsets including the empty set. The speculative postcondition with variables of the form  $y_{def}$  set to false, and variables of the form  $y^b$  replaced by  $y$  is

```

    ⟨..., idx < C_size ⇒ ΓA[C[idx]*idx] = ⊥ ∧ ΓA[64*idx] = ⊥ ∧ ΓC[idx] = ⊥ ∧
      ΓA[C[0→64][idx]*idx] = ⊥) ∧ (idx ≥ C_size ⇒ ...)⟩
    r0 := idx;
    ⟨..., Γr0 = ⊥ ∧ (r0 < C_size ⇒ ΓA[C[r0]*r0] = ⊥ ∧ ΓA[64*r0] = ⊥ ∧ ΓC[r0] = ⊥ ∧
      ΓA[C[0→64][r0]*r0] = ⊥) ∧ (r0 ≥ C_size ⇒ ...)⟩
    r1 := C_size;
    ⟨..., Γr0 = ⊥ ∧ Γr1 = ⊥ ∧
      (r0 < r1 ⇒ ΓA[C[r0]*r0] = ⊥ ∧ ΓA[64*r0] = ⊥ ∧ ΓC[r0] = ⊥ ∧
        ΓA[C[0→64][r0]*r0] = ⊥) ∧ (r0 ≥ r1 ⇒ ...)⟩
    if (r0 < r1){
        ⟨..., ΓA[C[r0]*r0] = ⊥ ∧ ΓA[64*r0] = ⊥ ∧ Γr0 = ⊥ ∧ ΓC[r0] = ⊥ ∧
          ΓA[C[0→64][r0]*r0] = ⊥)
        C[0] := 64; // Value 64 may be forwarded to r1
        (as below)
        leak r0;
        ⟨(A[C[r0]*r0]def ∧ C[r0]def ⇒ ΓA[C[r0]*r0] = ⊥ ∧ Γr0 = ⊥ ∧ ΓC[r0] = ⊥) ∧
          (A[C[r0]*r0]def ∧ ¬C[r0]def ⇒ ΓA[C[r0]b*r0] = ⊥ ∧ Γr0 = ⊥ ∧ ΓC[r0]b = ⊥) ∧
          (¬A[C[r0]*r0]def ∧ C[r0]def ⇒ ΓA[C[r0]*r0]b = ⊥ ∧ Γr0 = ⊥ ∧ ΓC[r0] = ⊥) ∧
          (¬A[C[r0]*r0]def ∧ ¬C[r0]def ⇒ ΓA[C[r0]b*r0]b = ⊥ ∧ Γr0 = ⊥ ∧ ΓC[r0]b = ⊥),
          Γr0 = ⊥ ∧ ΓC[r0] = ⊥ ∧ ΓA[C[r0]*r0] = ⊥)
        r1 := C[r0];
        ⟨Γr0 = ⊥ ∧ Γr1 = ⊥ ∧ (A[r1*r0]def ⇒ ΓA[r1*r0] = ⊥) ∧
          (¬A[r1*r0]def ⇒ ΓA[r1*r0]b = ⊥), Γr0 = ⊥ ∧ Γr1 = ⊥ ∧ ΓA[r1*r0] = ⊥)
        leak r1*r0;
        ⟨(A[r1*r0]def ⇒ ΓA[r1*r0] = ⊥) ∧ (¬A[r1*r0]def ⇒ ΓA[r1*r0]b = ⊥),
          ΓA[r1*r0] = ⊥)
        r1 := A[r1*r0];
        ⟨Γr1 = ⊥, Γr1 = ⊥)
        leak r1;
        ⟨true, true⟩
        r1 := B[r1];
        ⟨true, true⟩
    }
    ⟨true, true⟩
    
```

**Fig. 3.** Spectre-PSF litmus test

$\Gamma_{A[C[r0]*r0]} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{C[r0]} = \perp$ . Hence, the required predicates for each subset of global variables are as follows.

- For the empty set  $\{\}$ , we have  $\Gamma_{A[C[r0]*r0]} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{C[r0]} = \perp$ .
- For  $\{A[C[r0]*r0]\}$ , we have  $\Gamma_{r0} = \perp \wedge \Gamma_{C[r0]} = \perp$  since  $\Gamma_{A[C[r0]*r0]} = \perp$  is true when  $A[C[r0]*r0]$  is 64.
- For  $\{C[r0]\}$ , we have  $\Gamma_{A[64*r0]} = \perp \wedge \Gamma_{r0} = \perp$  since  $\Gamma_{C[r0]} = \perp$  is true when  $C[r0]$  is 64.

- For  $\{A[C[r0] * r0], C[r0]\}$ , we have  $\Gamma_{r0} = \perp$ .

Conjoining the four predicates above gives us the condition required for the leaks not to be exploitable via either Spectre-STL (the empty-set case) or Spectre-PSF:  $\Gamma_{A[C[r0]*r0]} = \perp \wedge \Gamma_{A[64*r0]} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{C[r0]} = \perp$ .

The overall non-speculative precondition for the program is derived under the assumptions that  $C\_size$  and the input  $idx$  are non-sensitive, and that  $idx \geq 0$  and all elements of arrays  $A$  and  $C$  are non-sensitive. To keep the presentation simple, we elide the speculative precondition above the store  $C[0] := 64$  and the non-speculative proof obligation due to Spectre-PHT, i.e., the non-speculative proof obligation when  $idx \geq C\_size$ .

When  $idx = 0$ , the precondition simplifies to true since each array index evaluates to 0 which is in the range of the respective arrays (recall from Section 3.2 that  $C$  is of size 2 and  $A$  of size 16). When  $idx = 1$ ,  $A[64 * idx]$  accesses a memory location beyond the end of array  $A$  and hence data which is potentially sensitive. Hence, the code is not provably secure: the Spectre-PSF vulnerability discussed in Section 3.2 is detected.

Adding a fence instruction after the store will prevent speculative store forwarding. This situation is also correctly evaluated by our logic. The fence’s speculative precondition is true and hence no proof obligations are transferred to the non-speculative precondition of the store. This results in the precondition of the program when  $idx$  is 0 or 1 evaluating to true.

The above litmus test (in both fenced and unfenced form) constitutes the only litmus test for Spectre-PSF in the literature. Other approaches for detecting Spectre-PSF are based on an explicit semantics of the microarchitectural features that give rise to the vulnerability [20] or, like us, rely on this single litmus test for validation [26].

## 6 Related Work

Cauligi et al. [8] provide a detailed comparison of 24 formal semantics and tools for detecting Spectre vulnerabilities. While all approaches support Spectre-PHT, only 5 out of 24 [7, 20, 13, 5, 26] support Spectre-STL (and only 2 of these [20, 26] have support for Spectre-PSF). Three of the five are based on explicit models of a processor’s microarchitecture [7, 20, 13] and two on more abstract semantics [5, 26].

Of the former approaches, Cauligi et al. [7] and Guanciale et al. [20] model program instructions by translation to sequences of fetch, execute and commit microinstructions. Additional state information and associated microinstructions provide the prediction and rollback facilities required to model speculative execution. This level of detail has the potential to detect more vulnerabilities than abstract approaches, and in fact Guanciale et al. [20] independently discover Spectre-PSF (which they call Spectre-STL-D). However, such detailed models also add complexity to analysis.

Cauligi et al.’s approach [7] is supported by symbolic execution as is the approach of Daniel et al. [13]. The latter work addresses scalability issues inherent

with symbolic execution by removing redundant execution paths, and representing aspects of the microarchitectural execution symbolically rather than explicitly. These optimisations are validated using a set of litmus tests including those for Spectre-STL that we adopt in this paper. Later work by the authors [14] looks at modelling and implementing a hardware taint-tracking mechanism to mitigate vulnerabilities to Spectre, including Spectre-STL.

Fabian et al. [17] (not included in the above comparison) also employ symbolic execution for detecting Spectre vulnerabilities, including Spectre-STL. They define a framework for composing semantics of different variants of Spectre allowing to detect leaks due to a combination of, for example, Spectre-PHT and Spectre-STL. Our approach also allows the detection of such vulnerabilities as the proof obligations for each of the different Spectre variants is checked. We have confirmed this by applying the approach to Listing 1 of [17] (see Appendix B).

Barthe et al. [5] provide a higher-level semantics of speculative execution for a simple while language (similar to the language in this paper). Rather than modelling speculation via microinstructions, the semantics includes high-level directives which, for example, force a particular branch to be taken, or indicate which store is to be used by a load. The approach is implemented in the Jasmin verification framework [2, 3].

Ponce de León and Kinder [26] provide an axiomatic semantics for speculative execution (based on the work of Alglave et al. [1]) which is significantly less complex than the operational semantics of other approaches. The semantics defines which executions are valid via constraints on various relations between loads and stores in a program. Their approach is validated for Spectre-STL and Spectre-PSF using the same litmus tests as in this paper, and supported by bounded model checking.

Our work differs from the existing approaches by having its basis in weakest precondition (wp) reasoning. This opens the opportunity to adapt existing program analysis tools such as Boogie [4] or Why3 [18] which automate such reasoning (see [28] for work in this direction). Such tooling requires the user to provide annotations, particularly loop invariants, to programs but is able to handle greater nondeterminism than symbolic execution or model checking where nondeterminism can adversely affect scalability.

Our work is also based on an approach [10] which can be combined with rely/guarantee reasoning for analysis of concurrent programs [21, 31] and the proof technique, reordering interference freedom (rif), for taking into account processor weak memory models [11]. Its underlying logic can also be extended to support controlled release of sensitive information via declassification [27].

## 7 Conclusion

This paper has presented a weakest precondition-based approach for detecting vulnerabilities to the major data flow variants of Spectre, Spectre-STL and Spectre-PSF. The approach extends an existing approach for Spectre-PHT and can detect vulnerabilities to all three attacks including when the attacks occur

in combination. The approach has been validated with a set of litmus test used to validate related approaches and tools in the literature. A deeper evaluation of the approach, including its use on concurrent programs, requires automated tool support which is left to future work. Since it is based on weakest precondition reasoning, such support can be built on an existing auto-active program analyser such as Boogie or Why3.

**Acknowledgements** Thanks to Kirsten Winter, Robert Colvin and Mark Beaumont for feedback on this paper.

## References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014). <https://doi.org/10.1145/2627752>, <http://doi.acm.org/10.1145/2627752>
2. Almeida, J., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.: Jasmin: High-assurance and high-speed cryptography. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*. pp. 1807–1823. ACM (2017). <https://doi.org/10.1145/3133956.3134078>
3. Almeida, J., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., Strub, P.: The last mile: High-assurance and high-speed cryptographic implementations. In: *2020 IEEE Symposium on Security and Privacy, SP 2020*. pp. 965–982. IEEE (2020). <https://doi.org/10.1109/SP40000.2020.00028>
4. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005. Lecture Notes in Computer Science*, vol. 4111, pp. 364–387. Springer (2005). [https://doi.org/10.1007/11804192\\_17](https://doi.org/10.1007/11804192_17)
5. Barthe, G., Cauligi, S., Grégoire, B., Koutsos, A., Liao, K., Oliveira, T., Priya, S., Rezk, T., Schwabe, P.: High-assurance cryptography in the spectre era. In: *42nd IEEE Symposium on Security and Privacy, SP 2021*. pp. 1884–1901. IEEE (2021). <https://doi.org/10.1109/SP40001.2021.00046>
6. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtvushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: Heninger, N., Traynor, P. (eds.) *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. pp. 249–266. USENIX Association (2019)
7. Cauligi, S., Disselkoen, C., von Gleissenthall, K., Tullsen, D.M., Stefan, D., Rezk, T., Barthe, G.: Constant-time foundations for the new Spectre era. In: Donaldson, A.F., Torlak, E. (eds.) *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. pp. 913–926. ACM (2020). <https://doi.org/10.1145/3385412.3385970>
8. Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., Stefan, D.: SoK: Practical foundations for software Spectre defenses. In: *43rd IEEE Symposium on Security and Privacy, SP 2022*. pp. 666–680. IEEE (2022). <https://doi.org/10.1109/SP46214.2022.9833707>

9. Colvin, R.J., Winter, K.: An abstract semantics of speculative execution for reasoning about security vulnerabilities. In: Sekerinski, E., et al. (eds.) *Formal Methods. FM 2019 International Workshops - Revised Selected Papers, Part II. Lecture Notes in Computer Science*, vol. 12233, pp. 323–341. Springer (2019). [https://doi.org/10.1007/978-3-030-54997-8\\_21](https://doi.org/10.1007/978-3-030-54997-8_21)
10. Coughlin, N., Lam, K., Smith, G., Winter, K.: Detecting speculative execution vulnerabilities on weak memory models. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) *Formal Methods - 26th International Symposium, FM 2024. Lecture Notes in Computer Science*, vol. 14933, pp. 482–500. Springer (2024). [https://doi.org/10.1007/978-3-031-71162-6\\_25](https://doi.org/10.1007/978-3-031-71162-6_25)
11. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) *Formal Methods - 24th International Symposium, FM 2021. Lecture Notes in Computer Science*, vol. 13047, pp. 292–310. Springer (2021). [https://doi.org/10.1007/978-3-030-90870-6\\_16](https://doi.org/10.1007/978-3-030-90870-6_16)
12. Coughlin, N., Winter, K., Smith, G.: Compositional reasoning for non-multicopy atomic architectures. *Formal Aspects Comput.* **35**(2), 8:1–8:30 (2023). <https://doi.org/10.1145/3574137>
13. Daniel, L., Bardin, S., Rezk, T.: Hunting the haunter - efficient relational symbolic execution for Spectre with Haunted RelSE. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021. The Internet Society* (2021)
14. Daniel, L., Bognar, M., Noorman, J., Bardin, S., Rezk, T., Piessens, F.: Prospect: Provably secure speculation for the constant-time policy. In: Calandrino, J.A., Troncoso, C. (eds.) *32nd USENIX Security Symposium, USENIX Security 2023*. pp. 7161–7178. USENIX Association (2023)
15. Dijkstra, E.W.: *A Discipline of Programming*. Prentice-Hall (1976), <https://www.worldcat.org/oclc/01958445>
16. Dijkstra, E.W., Scholten, C.S.: *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Heidelberg (1990)
17. Fabian, X., Guarnieri, M., Patrignani, M.: Automatic detection of speculative execution combinations. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*. pp. 965–978. ACM (2022). <https://doi.org/10.1145/3548606.3560555>
18. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013. Lecture Notes in Computer Science*, vol. 7792, pp. 125–128. Springer (2013). [https://doi.org/10.1007/978-3-642-37036-6\\_8](https://doi.org/10.1007/978-3-642-37036-6_8)
19. Goguen, J.A., Meseguer, J.: Security policies and security models. In: *1982 IEEE Symposium on Security and Privacy, 1982*. pp. 11–20. IEEE Computer Society (1982). <https://doi.org/10.1109/SP.1982.10014>
20. Guanciale, R., Balliu, M., Dam, M.: InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*. pp. 1853–1869. ACM (2020). <https://doi.org/10.1145/3372297.3417246>
21. Jones, C.B.: Specification and design of (parallel) programs. In: *IFIP Congress*. pp. 321–332 (1983)
22. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Ex-

- plotting speculative execution. In: 2019 IEEE Symposium on Security and Privacy, SP 2019. pp. 1–19. IEEE (2019). <https://doi.org/10.1109/SP.2019.00002>
23. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy, SP 2015. pp. 605–622. IEEE Computer Society (2015). <https://doi.org/10.1109/SP.2015.43>
  24. Murray, T.C., Sison, R., Engelhardt, K.: COVERN: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018. pp. 16–30. IEEE (2018). <https://doi.org/10.1109/EuroSP.2018.00010>
  25. Murray, T.C., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016. pp. 417–431. IEEE Computer Society (2016). <https://doi.org/10.1109/CSF.2016.36>
  26. Ponce de León, H., Kinder, J.: Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. In: 43rd IEEE Symposium on Security and Privacy, SP 2022. pp. 235–248. IEEE (2022). <https://doi.org/10.1109/SP46214.2022.9833774>
  27. Smith, G.: Declassification predicates for controlled information release. In: Riesco, A., Zhang, M. (eds.) 23rd International Conference on Formal Engineering Methods (ICFEM 2022). Lecture Notes in Computer Science, Springer (2022)
  28. Smith, G.: A Dafny-based approach to thread-local information flow analysis. In: 11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023. pp. 86–96. IEEE (2023). <https://doi.org/10.1109/FormaliSE58978.2023.00017>
  29. Smith, G., Coughlin, N., Murray, T.: Value-dependent information-flow security on weak memory models. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019. Lecture Notes in Computer Science, vol. 11800, pp. 539–555. Springer (2019). [https://doi.org/10.1007/978-3-030-30942-8\\_32](https://doi.org/10.1007/978-3-030-30942-8_32)
  30. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021. pp. 1–16. IEEE (2021). <https://doi.org/10.1109/CSF51468.2021.00017>
  31. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing* **9**(2), 149–174 (1997). <https://doi.org/10.1007/BF01211617>

## A Spectre-STL Litmus tests

Below we apply our logic to the 13 litmus tests for Spectre-STL developed by Daniel et al. [13] and available at [https://github.com/binsec/haunted\\_bench/blob/master/src/litmus-stl/programs/spectrev4.c](https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c). The tests are reexpressed in the language from Section 2. In the tests,  $idx$  is an input provided by the user who may be an attacker. This value is not sensitive, i.e.,  $\mathcal{L}(idx) = \perp$ .  $secretarray$  is a publicly inaccessible array which may contain sensitive data and has length  $array\_size = 16$ . This value is not sensitive, i.e.,  $\mathcal{L}(array\_size) = \perp$ .  $publicarray$  is a publicly accessible array which has length  $array\_size$ . For all indices  $i$  of  $publicarray$ ,  $\mathcal{L}(publicarray[i]) = \perp$ .  $publicarray2$  is a publicly accessible array which has length  $512 \cdot 256$  (512 is the cache line size in bits, and 256 the number of integers representable using 8 bits). For all indices  $i$  of  $publicarray2$ ,  $\mathcal{L}(publicarray2[i]) = \perp$ . Hence, the only sensitive data is in  $secretarray$  or parts of memory outside of the defined arrays and variables.

The speculative precondition is elided whenever there are no points where speculation can start (stores or branches) earlier in the code.

### Case 1

$data$ ,  $data\_slowptr$  and  $data\_slowslowptr$  are local (pointer) variables and can point to sensitive data. To express this test in our simple programming language, all expressions involving referencing (&) and dereferencing (\*) of pointers have been resolved.

The code is insecure since the non-speculative precondition requires an element of  $secretarray$  to have security level  $\perp$ .

$$\begin{aligned}
 &\langle \dots, \Gamma_{secretarray[idx \& (array\_size - 1)]} = \perp \rangle \\
 &r0 := idx; \\
 &\langle \dots, \Gamma_{secretarray[r0 \& (array\_size - 1)]} = \perp \rangle \\
 &r1 := array\_size; \\
 &\langle \dots, \Gamma_{secretarray[r0 \& (r1 - 1)]} = \perp \rangle \\
 &r0 := r0 \& (r1 - 1); \\
 &\langle \dots, \Gamma_{secretarray[r0]} = \perp \rangle \\
 &r1 := secretarray; \\
 &\langle \dots, \Gamma_{secretarray[r0]} = \perp \rangle \\
 &data := r1; \\
 &\langle (\neg secretarray[r0]_{def} \Rightarrow \Gamma_{secretarray[r0]^b} = \perp) \wedge \Gamma_{secretarray[r0]} = \perp, \\
 &\quad \Gamma_{secretarray[r0]} = \perp \rangle \\
 &data\_slowptr := r1;
 \end{aligned}$$

```

⟨⟨¬secretarray[r0]def ⇒ Γsecretarray[r0]b = ⊥⟩ ∧ Γsecretarray[r0] = ⊥,
  Γsecretarray[r0] = ⊥⟩
data_slowslowptr := r1;
⟨⟨¬secretarray[r0]def ⇒ Γsecretarray[r0]b = ⊥⟩ ∧ Γsecretarray[r0] = ⊥,
  Γsecretarray[r0] = ⊥⟩
secretarray[r0] := 0 // This store may be bypassed
⟨⟨secretarray[r0]def ⇒ Γsecretarray[r0] = ⊥⟩ ∧
  (¬secretarray[r0]def ⇒ Γsecretarray[r0]b = ⊥), Γsecretarray[r0] = ⊥⟩
r1 := secretarray[r0];
⟨Γr1 = ⊥, Γr1 = ⊥⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩

```

**Case 2**

The code is insecure since  $idx$  may be greater than the length of  $publicarray$ .

```

⟨..., Γpublicarray[idx] = ⊥⟩
r0 := idx;
⟨..., Γpublicarray[r0&(array_size-1)] = ⊥ ∧ Γr0 = ⊥ ∧ Γpublicarray[idx] = ⊥⟩
r1 := array_size;
⟨..., Γpublicarray[r0&(r1-1)] = ⊥ ∧ Γr0 = ⊥ ∧ Γr1 = ⊥ ∧ Γpublicarray[idx] = ⊥⟩
r0 := r0 & (r1 - 1);
⟨..., Γpublicarray[r0] = ⊥ ∧ Γr0 = ⊥ ∧ Γpublicarray[idx] = ⊥⟩
idx := r0 // This store may be bypassed
⟨⟨publicarray[idx]def ⇒ Γpublicarray[idx] = ⊥⟩ ∧
  (¬publicarray[idx]def ⇒ Γpublicarray[idx]b = ⊥), Γpublicarray[idx] = ⊥⟩
leak idx; // note that Γidx = ⊥ is true since ℒ(idx) = ⊥
⟨⟨publicarray[idx]def ⇒ Γpublicarray[idx] = ⊥⟩ ∧
  (¬publicarray[idx]def ⇒ Γpublicarray[idx]b = ⊥), Γpublicarray[idx] = ⊥⟩
r1 := publicarray[idx];
⟨Γr1 = ⊥, Γr1 = ⊥⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩

```

**Case 3**

The code is secure since there is no store to bypass.

```

⟨..., true⟩
r0 := idx;

```

```

⟨...,  $\Gamma_{r0} = \perp$ ⟩
 $r1 := \text{array\_size}$ ;
⟨...,  $\Gamma_{r0} = \perp \wedge \Gamma_{r1} = \perp \wedge \Gamma_{\text{publicarray}[r0\&(r1-1)]} = \perp$ ⟩
 $r0 := r0 \ \& \ (r1 - 1)$ ;
⟨...,  $\Gamma_{r0} = \perp \wedge \Gamma_{\text{publicarray}[r0]} = \perp$ ⟩
leak  $r0$ ;
⟨...,  $\Gamma_{\text{publicarray}[r0]} = \perp$ ⟩
 $r1 := \text{publicarray}[r0]$ ;
⟨...,  $\Gamma_{r1} = \perp$ ⟩
leak  $r1$ ;
⟨...,  $\text{true}$ ⟩
 $r2 := \text{publicarray2}[r1 * 512]$ ;
⟨ $\text{true}, \text{true}$ ⟩
    
```

#### Case 4

See Section 4.

#### Case 5

$\text{case5\_ptr}$  and  $\text{toleak}$  are local variables.  $\text{case5\_ptr}$  is initially set to  $\text{secretarray}$ .

The code is insecure since it requires that elements of  $\text{secretarray}$  (the initial value of  $\text{case5\_ptr}$ ) have security level  $\perp$ .

```

⟨...,  $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{\text{case5\_ptr}[\text{idx}\&(\text{array\_size}-1)]} = \perp$ ⟩
 $r0 := \text{idx}$ ;
⟨...,  $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{\text{case5\_ptr}[r0\&(\text{array\_size}-1)]} = \perp$ ⟩
 $r1 := \text{array\_size}$ ;
⟨...,  $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{\text{case5\_ptr}[r0\&(r1-1)]} = \perp \wedge \Gamma_{\text{publicarray}[r0\&(r1-1)]} = \perp$ ⟩
 $r0 := r0 \ \& \ (r1 - 1)$ ;
⟨...,  $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{\text{case5\_ptr}[r0]} = \perp \wedge \Gamma_{\text{publicarray}[r0]} = \perp$ ⟩
 $r1 := \text{publicarray}$ ;
⟨...,  $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{\text{case5\_ptr}[r0]} = \perp \wedge \Gamma_{r1[r0]} = \perp$ ⟩
 $\text{case5\_ptr} := r1$ ; // This store may be bypassed
⟨( $\text{case5\_ptr}[r0]_{\text{def}} \Rightarrow \Gamma_{\text{case5\_ptr}[r0]} = \perp$ )  $\wedge$  ( $\neg \text{case5\_ptr}[r0]_{\text{def}} \Rightarrow \Gamma_{\text{case5\_ptr}[r0]^b} = \perp$ )  $\wedge$ 
 $\Gamma_{\text{toleak}} = \perp, \Gamma_{\text{case5\_ptr}[r0]} = \perp \wedge \Gamma_{\text{toleak}} = \perp$ ⟩
 $r1 = \text{case5\_ptr}[r0]$ ;
⟨ $\Gamma_{\text{toleak}} = \perp \wedge \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \wedge \Gamma_{\text{toleak}} = \perp$ ⟩
 $\text{toleak} = r1$ ;
⟨ $\Gamma_{\text{toleak}} = \perp, \Gamma_{\text{toleak}} = \perp$ ⟩
 $r1 := \text{toleak}$ ; // if  $\neg \text{toleak}_{\text{def}}$  then  $\text{toleak} = \text{toleak}^b$  since  $\text{toleak}$  is local
⟨ $\Gamma_{r1} = \perp, \Gamma_{r1} = \perp$ ⟩
leak  $r1$ ;
⟨ $\text{true}, \text{true}$ ⟩
 $r2 := \text{publicarray2}[r1 * 512]$ ;
⟨ $\text{true}, \text{true}$ ⟩
    
```

**Case 6**

$case6\_idx$ ,  $case6\_array$  and  $toleak$  are local variables.  $case6\_idx$  is initially set to 0, and  $case6\_array$  is initially set to  $[secretarray, publicarray]$ .

The code is insecure since it requires elements of  $secretarray$  (the initial value of  $case6\_array[case6\_idx]$ ) to have security level  $\perp$ .

$$\begin{aligned} & \langle \dots, \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_array[case6\_idx][idx \& (array\_size - 1)]} = \perp \rangle \\ & r0 := idx; \\ & \langle \dots, \Gamma_{case6\_array[1][r0 \& (array\_size - 1)]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_idx} = \perp \wedge \\ & \quad \Gamma_{r0 \& (array\_size - 1)} = \perp \wedge \Gamma_{case6\_array[case6\_idx][r0 \& (array\_size - 1)]} = \perp \rangle \\ & r1 := array\_size; \\ & \langle \dots \Gamma_{case6\_array[1][r0 \& (r1 - 1)]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_idx} = \perp \wedge \\ & \quad \Gamma_{r0 \& (r1 - 1)} = \perp \wedge \Gamma_{case6\_array[case6\_idx][r0 \& (r1 - 1)]} = \perp \rangle \\ & r0 := r0 \& (r1 - 1); \\ & \langle \dots \Gamma_{case6\_array[case6\_idx][r0]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_idx} = \perp \wedge \\ & \quad \Gamma_{r0} = \perp \rangle \\ & case6\_idx := 1; \quad // \text{ This store may be bypassed} \\ & \langle (case6\_array[case6\_idx][r0]_{def} \Rightarrow \Gamma_{case6\_array[case6\_idx][r0]} = \perp) \wedge \\ & \quad (\neg case6\_array[case6\_idx][r0]_{def} \Rightarrow \Gamma_{case6\_array[case6\_idx][r0]^b} = \perp) \wedge \\ & \quad \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_idx} = \perp \wedge \Gamma_{r0} = \perp, \\ & \quad \Gamma_{case6\_array[case6\_idx][r0]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{case6\_idx} = \perp \wedge \Gamma_{r0} = \perp \rangle \\ & r1 := case6\_idx; \\ & \langle (case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]} = \perp) \wedge \\ & \quad (\neg case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]^b} = \perp) \wedge \\ & \quad \Gamma_{toleak} = \perp \wedge \Gamma_{r1} = \perp \wedge \Gamma_{r0} = \perp, \\ & \quad \Gamma_{case6\_array[r1][r0]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{r1} = \perp \wedge \Gamma_{r0} = \perp \rangle \\ & leak\ r0; \\ & \langle (case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]} = \perp) \wedge \\ & \quad (\neg case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]^b} = \perp) \wedge \\ & \quad \Gamma_{toleak} = \perp \wedge \Gamma_{r1} = \perp, \Gamma_{case6\_array[r1][r0]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{r1} = \perp \rangle \\ & leak\ r1; \\ & \langle (case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]} = \perp) \wedge \\ & \quad (\neg case6\_array[r1][r0]_{def} \Rightarrow \Gamma_{case6\_array[r1][r0]^b} = \perp) \wedge \\ & \quad \Gamma_{toleak} = \perp, \Gamma_{case6\_array[r1][r0]} = \perp \wedge \Gamma_{toleak} = \perp \rangle \\ & r2 := (case6\_array[r1])[r0]; \\ & \langle \Gamma_{toleak} = \perp \wedge \Gamma_{r2} = \perp, \Gamma_{r2} = \perp \wedge \Gamma_{toleak} = \perp \rangle \\ & toleak = r2; \\ & \langle \Gamma_{toleak} = \perp, \Gamma_{toleak} = \perp \rangle \\ & r1 := toleak; \quad // \text{ if } \neg toleak_{def} \text{ then } toleak = toleak^b \text{ since } toleak \text{ is local} \\ & \langle \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \rangle \\ & leak\ r1; \\ & \langle true, true \rangle \\ & r2 := publicarray2[r1 * 512]; \\ & \langle true, true \rangle \end{aligned}$$

**Case 7**

$case7\_mask$  a local variable which is initially set to  $MAXINT$ .

The code is insecure since  $idx \ \& \ MAXINT$  (where  $MAXINT$  is the initial value of  $case7\_mask$ ) may be greater than the length of  $publicarray$ .

$$\begin{aligned}
 & \langle \dots, \Gamma_{toleak} = \perp \wedge \Gamma_{publicarray[idx \& case7\_mask]} = \perp \\
 & r0 := array\_size; \\
 & \langle \dots, \Gamma_{publicarray[idx \& (r0-1)]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{idx} = \perp \wedge \Gamma_{r0} = \perp \wedge \\
 & \quad \Gamma_{publicarray[idx \& case7\_mask]} = \perp \wedge \Gamma_{case7\_mask} = \perp \rangle \\
 & r0 := r0 - 1; \\
 & \langle \dots \Gamma_{publicarray[idx \& r0]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{idx} = \perp \wedge \Gamma_{r0} = \perp \wedge \\
 & \quad \Gamma_{publicarray[idx \& case7\_mask]} = \perp \wedge \Gamma_{case7\_mask} = \perp \rangle \\
 & case7\_mask := r0; \quad // This store may be bypassed \\
 & \langle (publicarray[idx \& case7\_mask]_{def} \Rightarrow \Gamma_{publicarray[idx \& case7\_mask]} = \perp) \wedge \\
 & \quad (\neg publicarray[idx \& case7\_mask]_{def} \Rightarrow \Gamma_{publicarray[idx \& case7\_mask]^b} = \perp) \wedge \\
 & \quad \Gamma_{toleak} = \perp \wedge \Gamma_{idx} = \perp \wedge \Gamma_{case7\_mask} = \perp, \\
 & \quad \Gamma_{publicarray[idx \& case7\_mask]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{idx} = \perp \wedge \Gamma_{case7\_mask} = \perp \rangle \\
 & r0 := idx; \quad // if  $\neg idx_{def}$  then  $idx = idx^b$  since  $idx$  is local \\
 & \langle (publicarray[r0 \& case7\_mask]_{def} \Rightarrow \Gamma_{publicarray[r0 \& case7\_mask]} = \perp) \wedge \\
 & \quad (\neg publicarray[r0 \& case7\_mask]_{def} \Rightarrow \Gamma_{publicarray[r0 \& case7\_mask]^b} = \perp) \wedge \\
 & \quad \Gamma_{toleak} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{case7\_mask} = \perp, \\
 & \quad \Gamma_{publicarray[r0 \& case7\_mask]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{case7\_mask} = \perp \rangle \\
 & r1 := case7\_mask; \quad // if  $\neg case7\_mask_{def}$  then  $case7\_mask = case7\_mask^b$  since local \\
 & \langle (publicarray[r0 \& r1]_{def} \Rightarrow \Gamma_{publicarray[r0 \& r1]} = \perp) \wedge \\
 & \quad (\neg publicarray[r0 \& r1]_{def} \Rightarrow \Gamma_{publicarray[r0 \& r1]^b} = \perp) \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{r1} = \perp, \\
 & \quad \Gamma_{publicarray[r0 \& r1]} = \perp \wedge \Gamma_{toleak} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{r1} = \perp \rangle \\
 & leak \ r0 \ \& \ r1; \\
 & \langle (publicarray[r0 \& r1]_{def} \Rightarrow \Gamma_{publicarray[r0 \& r1]} = \perp) \wedge \\
 & \quad (\neg publicarray[r0 \& r1]_{def} \Rightarrow \Gamma_{publicarray[r0 \& r1]^b} = \perp) \wedge \Gamma_{toleak} = \perp, \\
 & \quad \Gamma_{publicarray[r0 \& r1]} = \perp \wedge \Gamma_{toleak} = \perp \rangle \\
 & r1 := publicarray[r0 \ \& \ r1]; \\
 & \langle \Gamma_{toleak} = \perp \wedge \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \wedge \Gamma_{toleak} = \perp \rangle \\
 & toleak := r1; \\
 & \langle \Gamma_{toleak} = \perp, \Gamma_{toleak} = \perp \rangle \\
 & r1 := toleak; \quad // if  $\neg toleak_{def}$  then  $toleak = toleak^b$  since  $toleak$  is local \\
 & \langle \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \rangle \\
 & leak \ r1; \\
 & \langle true, true \rangle \\
 & r2 := publicarray2[r1 * 512]; \\
 & \langle true, true \rangle
 \end{aligned}$$
**Case 8**

$case8\_mult$  is a local variable which is initially set to 200.

The code is insecure since  $idx * 200$  (where 200 is the initial value of  $case8\_mult$ ) may be greater than the length of  $publicarray$ .

```

⟨...,  $\Gamma_{toleak} = \perp \wedge \Gamma_{publicarray[idx * case8\_mult]} = \perp$ ⟩
case8\_mult := 0; // This store may be bypassed
⟨ $\Gamma_{idx} = \perp \wedge \Gamma_{case8\_mult} = \perp \wedge$ 
  ( $publicarray[idx * case8\_mult]_{def} \Rightarrow \Gamma_{publicarray[idx * case8\_mult]} = \perp$ )  $\wedge$ 
  ( $\neg publicarray[idx * case8\_mult]_{def} \Rightarrow \Gamma_{publicarray[idx * case8\_mult]}^b = \perp$ )  $\wedge \Gamma_{toleak} = \perp$ ,
   $\Gamma_{idx} = \perp \wedge \Gamma_{case8\_mult} = \perp \wedge \Gamma_{publicarray[r0 * case8\_mult]} = \perp \wedge \Gamma_{toleak} = \perp$ ⟩
r0 := idx; // if  $\neg idx_{def}$  then  $idx = idx^b$  since  $idx$  is local
⟨ $\Gamma_{r0} = \perp \wedge \Gamma_{case8\_mult} = \perp \wedge$ 
  ( $publicarray[r0 * case8\_mult]_{def} \Rightarrow \Gamma_{publicarray[r0 * case8\_mult]} = \perp$ )  $\wedge$ 
  ( $\neg publicarray[r0 * case8\_mult]_{def} \Rightarrow \Gamma_{publicarray[r0 * case8\_mult]}^b = \perp$ )  $\wedge \Gamma_{toleak} = \perp$ ,
   $\Gamma_{r0} = \perp \wedge \Gamma_{case8\_mult} = \perp \wedge \Gamma_{publicarray[r0 * case8\_mult]} = \perp \wedge \Gamma_{toleak} = \perp$ ⟩
r1 := case8\_mult; // if  $\neg case8\_mult_{def}$  then  $case8\_mult = case8\_mult^b$  since local
⟨ $\Gamma_{r0} = \perp \wedge \Gamma_{r1} = \perp \wedge (publicarray[r0 * r1]_{def} \Rightarrow \Gamma_{publicarray[r0 * r1]} = \perp) \wedge$ 
  ( $\neg publicarray[r0 * r1]_{def} \Rightarrow \Gamma_{publicarray[r0 * r1]}^b = \perp$ )  $\wedge \Gamma_{toleak} = \perp$ ,
   $\Gamma_{r0} = \perp \wedge \Gamma_{r1} = \perp \wedge \Gamma_{publicarray[r0 * r1]} = \perp \wedge \Gamma_{toleak} = \perp$ ⟩
leak r0 * r1;
⟨( $publicarray[r0 * r1]_{def} \Rightarrow \Gamma_{publicarray[r0 * r1]} = \perp$ )  $\wedge$ 
  ( $\neg publicarray[r0 * r1]_{def} \Rightarrow \Gamma_{publicarray[r0 * r1]}^b = \perp$ )  $\wedge \Gamma_{toleak} = \perp$ ,
   $\Gamma_{publicarray[r0 * r1]} = \perp \wedge \Gamma_{toleak} = \perp$ ⟩
r0 := publicarray[r0 * r1];
⟨ $\Gamma_{toleak} = \perp \wedge \Gamma_{r0} = \perp$ ,  $\Gamma_{r0} = \perp \wedge \Gamma_{toleak} = \perp$ ⟩
toleak := r0;
⟨ $\Gamma_{toleak} = \perp$ ,  $\Gamma_{toleak} = \perp$ ⟩
r1 := toleak; // if  $\neg toleak_{def}$  then  $toleak = toleak^b$  since  $toleak$  is local
⟨ $\Gamma_{r1} = \perp$ ,  $\Gamma_{r1} = \perp$ ⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩

```

### Case 9

The code is insecure since it requires elements of  $secretarray$  to have security level  $\perp$ .

```

⟨...,  $\Gamma_{secretarray[idx \& (array\_size - 1)]} = \perp$ ⟩
r0 := idx;
⟨...,  $\Gamma_{secretarray[r0 \& (array\_size - 1)]} = \perp$ ⟩
r1 := array\_size;
⟨...,  $\Gamma_{secretarray[r0 \& (r1 - 1)]} = \perp$ ⟩
r0 := r0 & (r1 - 1);

```

```

⟨...,  $\Gamma_{secretarray[r0]} = \perp$ ⟩
secretarray[r0] := 0;    //This store may be bypassed
⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
(¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
i := 0;    // if ¬idef then i = ib since i is local
⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
(¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_i = \perp \wedge \Gamma_{secretarray[r0]} = \perp$ ⟩
while(i < 200){
    ⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
    (¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
    r1 := i;    // if ¬idef then i = ib since i is local
    ⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
    (¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
    i := r1 + 1;
    ⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
    (¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
}
⟨(secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]} = \perp$ ) ∧
(¬secretarray[r0]def ⇒  $\Gamma_{secretarray[r0]^b} = \perp$ ),  $\Gamma_{secretarray[r0]} = \perp$ ⟩
r1 := secretarray[r0];
⟨ $\Gamma_{r1} = \perp$ ,  $\Gamma_{r1} = \perp$ ⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩
    
```

### Case 10

The remaining cases involve function calls. We model these by inlining the function code which is followed by a fence. The fence ensures that all stores in the function are executed before it returns. We use the suffix `_ret` to denote the return value of the function.

The code is insecure since `fidx` may be greater than the length of `publicarray`.

```

⟨...,  $\Gamma_{publicarray[fidx]} = \perp$ ⟩
r0 := fidx;
⟨...,  $\Gamma_{r0 \& (array\_size - 1)} = \perp \wedge \Gamma_{publicarray[r0 \& (array\_size - 1)]} = \perp \wedge \Gamma_{publicarray[fidx]} = \perp$ ⟩
r1 := array_size;
⟨...,  $\Gamma_{r0 \& (r1 - 1)} = \perp \wedge \Gamma_{publicarray[r0 \& (r1 - 1)]} = \perp \wedge \Gamma_{publicarray[fidx]} = \perp$ ⟩
r0_ret := r0 & (r1 - 1);
⟨...,  $\Gamma_{r0\_ret} = \perp \wedge \Gamma_{publicarray[r0\_ret]} = \perp \wedge \Gamma_{publicarray[fidx]} = \perp$ ⟩
fence;
⟨...,  $\Gamma_{r0\_ret} = \perp \wedge \Gamma_{publicarray[r0\_ret]} = \perp \wedge \Gamma_{publicarray[fidx]} = \perp$ ⟩
fidx := r0_ret;    //This store may be bypassed
    
```

$$\begin{aligned}
&\langle \Gamma_{\mathit{fix}} = \perp \wedge (\mathit{fix}_{\mathit{def}} \Rightarrow \\
&\quad (\mathit{publicarray}[\mathit{fix}]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[\mathit{fix}]} = \perp) \wedge \\
&\quad (\neg \mathit{publicarray}[\mathit{fix}]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[\mathit{fix}]^b} = \perp)) \wedge \\
&\quad (\neg \mathit{fix}_{\mathit{def}} \Rightarrow \\
&\quad (\mathit{publicarray}[\mathit{fix}]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[\mathit{fix}]^b} = \perp) \wedge \\
&\quad (\neg \mathit{publicarray}[\mathit{fix}]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[\mathit{fix}]^b} = \perp)), \\
&\quad \Gamma_{\mathit{fix}} = \perp \wedge \Gamma_{\mathit{publicarray}[\mathit{fix}]} = \perp \rangle \\
&r0 := \mathit{fix}; \quad // \text{ if } \neg \mathit{fix}_{\mathit{def}} \text{ then } \mathit{fix} = \mathit{fix}^b \text{ since } \mathit{fix} \text{ is local} \\
&\langle \Gamma_{r0} = \perp \wedge (\mathit{publicarray}[r0]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[r0]} = \perp) \wedge \\
&\quad (\neg \mathit{publicarray}[r0]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[r0]^b} = \perp), \Gamma_{r0} = \perp \wedge \Gamma_{\mathit{publicarray}[r0]} = \perp \rangle \\
&\mathit{leak } r0; \\
&\langle (\mathit{publicarray}[r0]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[r0]} = \perp) \wedge \\
&\quad (\neg \mathit{publicarray}[r0]_{\mathit{def}} \Rightarrow \Gamma_{\mathit{publicarray}[r0]^b} = \perp), \Gamma_{\mathit{publicarray}[r0]} = \perp \rangle \\
&r1 := \mathit{publicarray}[r0]; \\
&\langle \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \rangle \\
&\mathit{leak } r1; \\
&\langle \mathit{true}, \mathit{true} \rangle \\
&r2 := \mathit{publicarray}2[r1 * 512]; \\
&\langle \mathit{true}, \mathit{true} \rangle
\end{aligned}$$

### Case 11

The code is insecure since it requires that *toleak* is initially non-sensitive.

$$\begin{aligned}
&\langle \dots, \Gamma_{\mathit{toleak}} = \perp \rangle \\
&r0 := \mathit{id}; \\
&\langle \dots, \Gamma_{r0 \& (\mathit{array\_size} - 1)} \wedge \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{publicarray}[r0 \& (\mathit{array\_size} - 1)]} = \perp \rangle \\
&r1 := \mathit{array\_size}; \\
&\langle \dots, \Gamma_{r0 \& (r1 - 1)} \wedge \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{publicarray}[r0 \& (r1 - 1)]} = \perp \rangle \\
&r0 := r0 \& (r1 - 1); \\
&\langle \dots, \Gamma_{r0} \wedge \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{publicarray}[r0]} = \perp \rangle \\
&\mathit{leak } r0; \\
&\langle \dots, \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{publicarray}[r0]} = \perp \rangle \\
&r1 := \mathit{publicarray}[r0]; \\
&\langle \dots, \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{r1} = \perp \rangle \\
&\mathit{toleak\_ret} := r1; \\
&\langle \mathit{true}, \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{toleak\_ret}} = \perp \rangle \\
&\mathit{fence}; \\
&\langle \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{toleak\_ret}} = \perp, \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{\mathit{toleak\_ret}} = \perp \rangle \\
&r0 := \mathit{toleak\_ret}; // \text{ if } \neg \mathit{toleak\_ret}_{\mathit{def}} \text{ then } \mathit{toleak\_ret} = \mathit{toleak\_ret}^b \text{ since } \mathit{toleak\_ret} \text{ local} \\
&\langle \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{r0} = \perp, \Gamma_{\mathit{toleak}} = \perp \wedge \Gamma_{r0} = \perp \rangle \\
&\mathit{toleak} := r0; \quad // \text{ This store may be bypassed} \\
&\langle \Gamma_{\mathit{toleak}} = \perp, \Gamma_{\mathit{toleak}} = \perp \rangle \\
&r1 := \mathit{toleak}; \quad // \text{ if } \neg \mathit{toleak}_{\mathit{def}} \text{ then } \mathit{toleak} = \mathit{toleak}^b \text{ since } \mathit{toleak} \text{ is local} \\
&\langle \Gamma_{r1} = \perp, \Gamma_{r1} = \perp \rangle \\
&\mathit{leak } r1;
\end{aligned}$$

```

⟨true, true⟩
r2 := publicarray[r1 * 512];
⟨true, true⟩
    
```

### Case 12

The code is secure since there is no store to bypass.

```

⟨..., true⟩
r0 := idx;
⟨..., Γr0&(array_size-1) = ⊥ ∧ Γpublicarray[r0&(array_size-1)] = ⊥⟩
r1 := array_size;
⟨..., Γr0&(r1-1) = ⊥ ∧ Γpublicarray[r0&(r1-1)] = ⊥⟩
r0_ret := r0 & (r1 - 1);
⟨..., Γr0_ret = ⊥ ∧ Γpublicarray[r0_ret] = ⊥⟩
fence;
⟨..., Γr0_ret = ⊥ ∧ Γpublicarray[r0_ret] = ⊥⟩
r0 := r0_ret;
⟨..., Γr0 = ⊥ ∧ Γpublicarray[r0] = ⊥⟩
leak r0;
⟨..., Γpublicarray[r0] = ⊥⟩
r1 := publicarray[r0];
⟨Γr1 = ⊥, Γr1 = ⊥⟩
leak r1;
⟨..., true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩
    
```

### Case 13

The code is secure since the only store is followed by a fence (and hence cannot be bypassed).

```

⟨..., true⟩
r0 := idx;
⟨..., Γr0&(array_size-1) = ⊥ ∧ Γpublicarray[r0&(array_size-1)] = ⊥⟩
r1 := array_size;
⟨..., Γr0&(r1-1) = ⊥ ∧ Γpublicarray[r0&(r1-1)] = ⊥⟩
r0 := r0 & (r1 - 1);
⟨..., Γr0 = ⊥ ∧ Γpublicarray[r0] = ⊥⟩
leak r0;
⟨..., Γpublicarray[r0] = ⊥⟩
r1 := publicarray[r0];
⟨..., Γr1 = ⊥⟩
toleak_ret := r1;
    
```

```

⟨true,  $\Gamma_{toleak\_ret} = \perp$ ⟩
fence;
⟨ $\Gamma_{toleak\_ret} = \perp, \Gamma_{toleak\_ret} = \perp$ ⟩
r1 := toleak_ret; // if  $\neg toleak_{def}$  then  $toleak\_ret = toleak\_ret^b$  since  $toleak\_ret$  local
⟨ $\Gamma_{r1} = \perp, \Gamma_{r1} = \perp$ ⟩
leak r1;
⟨true, true⟩
r2 := publicarray2[r1 * 512];
⟨true, true⟩

```

## B Combination of Spectre-PHT and Spectre-STL

Below we apply our logic to Listing 1 from Fabian et al. [17] which involves a vulnerability arising from a combination of Spectre-PHT and Spectre-STL. The tests are reexpressed in the language from Section 2. In the test, *secret* is a pointer to a sensitive value, and *public* a pointer to a non-sensitive value. *p* is a local variable and *A* a publicly accessible array.

The speculative precondition is elided whenever there are no points where speculation can start (stores or branches) earlier in the code.

To express this test in our simple programming language, pointers are modelled as arrays of length 1.

The code is insecure since it requires that the value at *secret* is non-sensitive.

```

⟨...,  $\Gamma_{public[0]} = \perp \wedge \Gamma_{secret[0]} = \perp \wedge \Gamma_{p[0]} = \perp$ ⟩
r0 := 0;
⟨...,  $\Gamma_{public[0]} = \perp \wedge \Gamma_{secret[0]} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{p[0]} = \perp$ ⟩
r1 := secret;
⟨...,  $\Gamma_{public[0]} = \perp \wedge \Gamma_{r1[0]} = \perp \wedge \Gamma_{r0} = \perp \wedge \Gamma_{p[0]} = \perp$ ⟩
p := r1;
⟨( $p[0]_{def} \Rightarrow \Gamma_{p[0]} = \perp$ )  $\wedge$  ( $\neg p[0]_{def} \Rightarrow \Gamma_{p[0]^b} = \perp$ )  $\wedge \Gamma_{public[0]} = \perp,$ 
 $\Gamma_{public[0]} = \perp \wedge \Gamma_{p[0]} = \perp \wedge \Gamma_{r0} = \perp$ ⟩
r1 := public; // if  $\neg public_{def}$  then  $public = public^b$  since  $public$  is local
⟨( $p[0]_{def} \Rightarrow \Gamma_{p[0]} = \perp$ )  $\wedge$  ( $\neg p[0]_{def} \Rightarrow \Gamma_{p[0]^b} = \perp$ )  $\wedge \Gamma_{r1} = \perp,$ 
 $\Gamma_{r1[0]} = \perp \wedge \Gamma_{p[0]} = \perp \wedge \Gamma_{r0} = \perp$ ⟩
p := r1; // This store may be bypassed
⟨( $p[0]_{def} \Rightarrow \Gamma_{p[0]} = \perp$ )  $\wedge$  ( $\neg p[0]_{def} \Rightarrow \Gamma_{p[0]^b} = \perp$ ),  $\Gamma_{p[0]} = \perp \wedge \Gamma_{r0} = \perp$ ⟩
if (r0! = 0)
  ⟨( $p_{def} \Rightarrow \Gamma_{p[0]} = \perp$ )  $\wedge$  ( $\neg p_{def} \Rightarrow \Gamma_{p[0]^b} = \perp$ ),  $\Gamma_{p[0]} = \perp$ ⟩
  r1 := p[0];
  ⟨ $\Gamma_{r1} = \perp, \Gamma_{r1} = \perp$ ⟩
  leak r1;
  ⟨true, true⟩
  r2 := A[r1 * 512];
  ⟨true, true⟩
⟨true, true⟩

```