Verifying Secure Speculation in Isabelle/HOL

Chapter · November 2021

DOI: 10.1007/978-3-030-90870-6_3

Brijesh Dongol
University of Surrey
121 PUBLICATIONS **996** CITATIONS

Verifying Secure Speculation in Isabelle / HOL

Matt Griffin[0000−0003−2703−0368] and Brijesh Dongol[0000−0003−0446−3507]

University of Surrey, Guildford, Surrey, UK
{matt.griffin,b.dongol}@surrey.ac.uk

Abstract. Secure speculation is an information flow security hyperproperty that prevents transient execution attacks such as Spectre, Meltdown and Foreshadow. Generic compiler mitigations for secure speculation are known to be insufficient for eliminating vulnerabilities. Moreover, these mitigation techniques often overprescribe speculative fences, causing the performance of the programs to suffer. Recently Cheang et al. have developed an operational semantics of program execution capable of characterising speculative executions as well as a new class of information flow hyperproperties named TPOD that ensure secure speculation. This paper presents a framework for verifying TPOD using the Isabelle/HOL proof assistant by encoding the operational semantics of Cheang et al. We provide translation tools for automatically generating the required Isabelle/HOL theory templates from a C-like program syntax, which speeds up verification. Our framework is capable of proving the existence of vulnerabilities *and* correctness of secure speculation. We exemplify our framework by proving the existence of secure speculation bugs in 15 victim functions for the MSVC compiler as well as correctness of some proposed fixes.

Keywords: Isabelle/HOL · Secure Speculation · Formal Verification · Spectre · Transient Execution Vulnerabilities · Hyperproperties

1 Introduction

Transient execution vulnerabilities, especially Spectre, remain an active area of research since their disclosure in 2018 [17]. Even with the availability of many (now mature) solutions, there is seemingly no decrease in the discovery of new variants, including "next generation attacks" [3, 22]. Modern processors from nearly all vendors are susceptible, meaning that millions of devices are insecure. This problem is of high concern to cloud providers since vulnerabilities can leak sensitive information from system memory, including across hypervisors, breaking virtual machine boundaries. Moreover, attacks can be carried out using valid processor mechanisms, meaning that detecting an occurrence is extremely hard. A successful attack leaves almost no trace of its exploitation that could be discerned from normal operation.

The root of the problem lies within the design of modern microprocessors and the complexity of many interacting components. The combination of performance features such as speculative execution, branch prediction and out-of-order

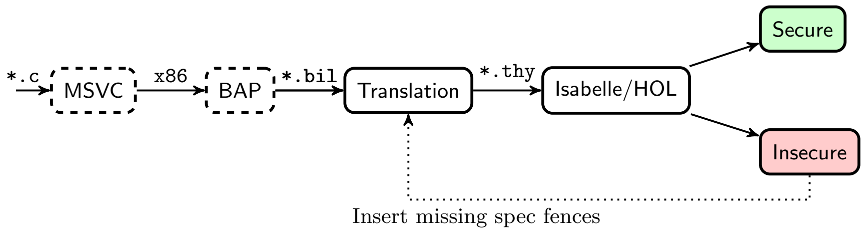

Fig. 1. Translation of C source programs to an Isabelle/HOL proof

execution are exploited/misused to extract information through side channels such as the CPU cache. Interestingly, processors that exhibit these vulnerabilities are correct, at least according to their vendor's specifications. Moreover, exploited mechanisms, namely branch prediction and speculative execution, are key to high performance computation. This indicates weaknesses in the specifications themselves, primarily their nondeterminism which affords flexibility at the permittance of undefined behaviour.

For some attacks (e.g., Meltdown [18] and Foreshadow [2]), the latest patches offer a combination of hardware- or micokernel-based fixes, but older architectures remain vulnerable. Moreover, many attacks (e.g., those based on Spectre [17, 22]) cannot be patched, even for the latest CPUs, since it is impossible to detect whether an arbitrary sequence of instructions is exploitable [16]. This suggests that a full solution to transient execution vulnerabilities will require a mix of hardware- and software-based solutions.

Existing mitigations (e.g., for Spectre) tend to be heavy handed, and thus, either incur performance penalties [20] or only target specific variants [14, 25]. A compiler-based approach, developed by Microsoft (MSVC), designed to work with new specifications of the `lfence` instructions in x86 processors is known to over-prescribe speculation barriers, leading to performance concerns [16]. More seriously, these mitigations are known to be incomplete — 15 victim functions were originally identified for Spectre variant 1 (see §2.1), capable of bypassing MSVC mitigations [16].

The above motivates the need for formal proofs of correctness of secure speculation for the compiled assembly-level representation of programs. A significant step towards such proofs were taken by Cheang et al. [5], who developed an operational semantics of program execution capable of characterising speculative executions. They have coupled this with a new class of information flow hyperproperties [6] named *trace property-dependent observational determinism (TPOD)* with mechanisation in the UCLID5 model checker [24]. TPOD generalises observational determinism [19, 23] to allow information flow from high to low states in the presence of speculation, succinctly characterising correctness of secure speculation.

Our paper comprises the following core contributions. **(1)** We build on the work of Cheang et al. [5] by encoding their operational semantics in the Isabelle/HOL proof assistant. **(2)** We couple our Isabelle/HOL mechanisation with a Hoare-style deductive verification technique, which we show is capable of reasoning about both the absence and presence of secure speculation. **(3)** We

```c
void victim_function_v01(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}
```

Listing 1.1. Spectre Bounds Check Bypass

prove the existence of vulnerabilities for all 15 victim functions originally identified for MSVC [16], and correctness of two of the most interesting fixes. **(4)** We develop an automatic translation tool that generates associated Isabelle/HOL theories for a given BIL[1] input, integrating into a verification workflow from C code to Isabelle/HOL proofs (see Fig. 1). **(5)** We develop theorems to identify programs that are guaranteed to satisfy TPOD, which reduces the proof burden.

Our mechanisation work uncovers minor issues in the operational semantics of Cheang et al. [5], which we revise in our operational semantics (see §3). Moreover, we treat speculative fences as dedicated instructions in program memory, as opposed to abstract barriers, more closely modelling a mitigated program. Furthermore, we clear speculative states on resolution for intuitive comparisons in proof conditions. Finally, we introduce a set of theorems to discharge trivial proof obligations, achieving similar verification scalability as the "havocing adversary" without the need for an explicit `havoc` instruction.

2 Transient Execution Vulnerabilities

Transient execution vulnerabilities target performance features such as speculative execution and branch prediction. Since the first discovered attacks [17, 18] many such vulnerabilities have been uncovered, targetting specific hardware architectures. Some variants have been fixed via hardware and microarchitecture updates, e.g. Foreshadow [2] no longer affects Intel architectures since 9th Generation Coffee Lake [13]. However, many variants of Spectre remain unpatched and require software-based mitigation.

2.1 Spectre Variant 1: Bounds Check Bypass

Spectre mistrains the branch predictor so that mispredictions are made on branching statements. One such mistraining strategy targets the pattern history table (PHT), which is used to decide the direction (taken/not-taken) of a conditional branch. This vulnerability is classified as Spectre-PHT [3] and was first introduced as Spectre Bounds Check Bypass (BCB), aka variant 1, by Kocher et al. [17]. Consider the simple C program in Listing 1.1, which demonstrates the Spectre BCB vulnerability.

The function takes some untrusted input `x` and performs a bounds check to ensure that it is less that the size of `array1` (`array1_size`). The value `x` indexes

[1] BIL is an assembly intermediate language [1].

```
00001bbd: RAX := mem[array2, el]:u64
00001bc4: RDX := mem[array1, el]:u64
00001bdc: ... // some processing on the index of array1 RDX
00001c1c: RDX := mem[RDX, el]:u64
00001c54: ... // some processing on the index of array2 RAX
00001c6d: RDX := mem[RAX, el]:u64
```

Listing 1.2. Concise translation of line 3 in Listing 1.1 to assembly in BIL. Full translation is given in Listing 1.6.

`array1` and subsequently `array2` through some transformation. The result is stored in `temp` before control is returned to the caller. The bounds check on line 2 is supposed to act as a guard between trusted and untrusted operations. Under normal execution, a caller would not be able to perform the memory reads and writes on line 3 if the guard does not hold. However, under speculative execution, the result of the branch condition on line 2 could be mispredicted, which can be reliably guaranteed if the branch predictor is mistrained.

If a misprediction occurs, the operations on line 3 will be executed speculatively with a value of `x` outside the prescribed bounds. This could lead to unconstrained reads from protected memory and differences in the speculative state across multiple executions. Note that the adversary cannot directly observe these differences as architectural changes are reverted when mispredictions are resolved. However, these changes persist in the *microarchitecture state* such as the L1 and L3 caches.[2]

We demonstrate how reads can be unconstrained, by examining a subset of the assembly corresponding to the program in Listing 1.1. An excerpt of the BIL[3] instructions (resulting from compiling line 3) in the program using Microsoft's Visual C++ Compiler (MSVC) are shown in Listing 1.2. Instructions are shown on the right with addresses (in program memory) on the left.

Assuming that the branch predictor has mispredicted the condition that `x` is less than the size of `array1`, the memory read at `00001c1c` may load an unconstrained value `v` from memory into the register `RDX`, where `v` is being used as an address. As instruction `00001c1c` is speculative, this read is permitted despite the fact that `v` is potentially a value in protected memory. This is the first indication of a vulnerability, even though the adversary cannot yet read `v` in the speculative state. However, the adversary can discover `v` through a side channel. In particular, the memory read at `00001c6d`, ultimately accesses `v`,[4] which means that `v` will be brought into the cache. Since the cache is microarchitectural, even when the branch predictor resolves this misspeculation on `x`, `v` remains in the cache making the value susceptible to a timing attack [16, 17]. Note that the

[2] For this to be exploited we must have already "poisoned" the cache [17]. In this paper, assume cache poisoning to have occurred prior to execution of each program.

[3] BIL is an assembly intermediate language [1]. In general, we *must* reason about secure speculation in assembly language since compilers may optimise branch statements in high-level languages.

[4] Technically speaking, the value being accessed is the transformed value `v * 512`.

purpose of `array2` is to provide a mechanism for indexing addresses; the attacker is not aiming to steal secrets from `array2` itself.

One way to fix this issue is via a *speculative fence*, `specfence`, which is an assembly instruction that resolves all branch predictions, in turn ensuring the absence of speculation at specific programmer-controlled locations. In Listing 1.2, a speculative fence is required prior to executing instruction `00001c6d`, which is the instruction that loads `v` into the cache as an address. After introduction of a speculative fence immediately prior to `00001c6d`, the misprediction on `x` will be resolved, preventing `v` from being loaded in the first place. However, determining precisely where speculative fences should be placed is difficult: underfencing leaves the code vulnerable, while overfencing negatively impact performance. Compilers alone cannot be trusted to reliably fence program code [16], pointing to a need for formal verification.

2.2 TPOD

Properties involving speculative execution cannot be formalised over a single trace. Instead we require a more general class of properties called *hyperproperties* [6], which are properties over sets of traces. An example is *observational determinism* [19, 23], which is a two-trace hyperproperty. It is well known that observational determinism allows one to establish *low equivalence*, which is an information flow property that holds iff from a *user state* (i.e., a *low state*) an observer cannot detect any difference in a *root state* (i.e., a *high state*). Operations are either untrusted and act on the low state, or are trusted and act on the high state.

It turns out *observational determinism* is not sufficient to characterise the types of properties described in § 2.1 [5]. Instead, we require a more general condition called *trace property-dependent observational determinism (TPOD)*, which is a four-trace hyperproperty. Unlike observational determinism, which precludes information flow from high to low states, TPOD allows information flow from a high state to a low state in the presence of misspeculation.

We describe TPOD directly in terms of its formal definition (see Fig. 2), but describe its components informally for now. The formal definitions of the components comprising TPOD are given in §3.3. We use $s_1 \approx_{\mathcal{L}} s_2$ to denote that state s_1 and s_2 are low-equivalent. For traces π_1 and π_2, we use $\pi_1 \approx_{\mathcal{L}} \pi_2$ to denote that for all indices i, $\pi_1^i \approx_{\mathcal{L}} \pi_2^i$, where π_k^i is the state indexed i in π_k. We assume that Tr is the set of all traces of a program, and $T \subseteq Tr$ is the set of traces of the program with no misprediction. Moreover, $op_{\mathcal{L}}(\pi)$ returns the sequence of low-state (i.e., untrusted) instructions executed for each state of the trace, which may be null if the instruction is trusted. $op_{\mathcal{H}}(\pi)$ is similar, but returns the sequence of trusted instructions. A difference between $op_{\mathcal{L}}(\pi)$ and $op_{\mathcal{H}}(\pi)$ is that $op_{\mathcal{H}}(\pi)$ also returns the observable memory after each high instruction (see §3.3 for details).

By (1) we assume that π_1 and π_2 are traces with no misprediction (and hence no speculation) and π_3 and π_4 are traces with misprediction, and by (2) all four traces execute the same sequence of low instructions. By (3), traces π_1 and π_3

$$
\begin{aligned}
\forall \pi_1, \pi_2, \pi_3, \pi_4 \in Tr.\ & \pi_1 \in T \wedge \pi_2 \in T \wedge \pi_3 \notin T \wedge \pi_4 \notin T \longrightarrow && (1)\\
& op_{\mathcal{L}}(\pi_1) = op_{\mathcal{L}}(\pi_2) = op_{\mathcal{L}}(\pi_3) = op_{\mathcal{L}}(\pi_4) \longrightarrow && (2)\\
& op_{\mathcal{H}}(\pi_1) = op_{\mathcal{H}}(\pi_3) \wedge op_{\mathcal{H}}(\pi_2) = op_{\mathcal{H}}(\pi_4) \longrightarrow && (3)\\
& \pi_1 \approx_{\mathcal{L}} \pi_2 \wedge \pi_3^0 \approx_{\mathcal{L}} \pi_4^0 \longrightarrow && (4)\\
& \pi_3 \approx_{\mathcal{L}} \pi_4 && (5)
\end{aligned}
$$

Fig. 2. Formal definition of TPOD

execute the same sequence of high instructions, as do π_2 and π_4. By (4), π_1 and π_2 are low equivalent, while π_3 and π_4 are initially low equivelent. Under these assumptions, we must show (5), which ensures that π_3 and π_4 (the traces with mispredication) are low equivalent. Note that π_1 and π_3 execute the same instructions with the same memory, but only π_3 can speculate. Similarly, for π_2 and π_4. Also note that there is no such constraint on π_1 and π_2 (and on π_3 and π_4), thus they could execute different instructions with different memory in the high state.

To see that TPOD does indeed capture the secure speculation properties of interest, consider a speculative execution of the program in Listing 1.2 up to instruction `00001c6d`. We construct the four traces π_1-π_4 in parallel. Since we are considering a speculative execution, π_1 and π_2 will be *stuttering*, i.e., waiting at the mispredicted branch (see §3 below), while π_3 and π_4 are executing the program speculatively. Thus (1) holds. Further assume that conditions (2) and (3) hold, meaning that the operations executed until this point are equivalent as defined by (2) and (3). By (4), π_1 and π_2 are low equivalent, and since TPOD is prefix-closed [5] π_3 and π_4 must also be low equivalent. Upon executing `00001c6d`, it will be possible for the system to load from different memory addresses, which violates low equivalence. This models the phenomenon where the adversary can identify a difference in the memory addresses accessed.

3 Operational semantics

3.1 Syntax and semantics

We model secure speculation using operational semantics for *speculative in-order* processors similar to Cheang et al. [5] with some modifications. We keep the proposed assembly intermediate representation (AIR), whose syntax for programs (*Prog*), instructions (*Instr*) and expressions (*Exp*) are given below. We let *Reg* and *Const* be the set of all registers and constants, respectively, and $\lozenge_u$ and $\lozenge_b$ be unary and binary operators.

$$
\begin{aligned}
& \mathit{Prog} ::= \mathit{Instr}^* \qquad \mathit{Exp} ::= \mathit{Const} \mid \mathit{Reg} \mid \lozenge_u\ \mathit{Exp} \mid \mathit{Exp}\ \lozenge_b\ \mathit{Exp}\\
& \mathit{Instr} ::= \mathit{Reg} := \mathit{Exp} \mid \mathsf{mem} := \mathsf{mem}[\mathit{Exp} \to \mathit{Exp}] \mid \mathit{Reg} := \mathsf{mem}[\mathit{Exp}]\\
& \qquad \mid \mathsf{if}\ \mathit{Exp}\ \mathsf{goto}\ \mathit{Addr}_{\Pi} \mid \mathsf{goto}\ \mathit{Addr}_{\Pi} \mid \mathsf{specfence}
\end{aligned}
$$

We let $\mathit{Addr}_{\Pi}, \mathit{Addr}_{\mu} \subseteq \mathit{Const}$ be the set of all program and memory addresses, respectively. The machine state is represented by $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n \rangle$, where

$\mathit{\Pi} : \mathbb{N} \to \mathit{Addr}_{\Pi} \to \mathit{Instr}$ is the *program memory* (mapping program addresses to instructions), $\mathit{\Delta} : \mathbb{N} \to \mathit{Reg} \to \mathit{Val}$ and $\mu : \mathbb{N} \to \mathit{Addr}_{\mu} \to \mathit{Val}$ are the *register* and *memory* states, $pc : \mathbb{N} \to \mathit{Addr}_{\Pi}$ is the *program counter*, $\omega \in (\mathit{Addr}_{\Pi} \times \mathit{Addr}_{\mu})^*$ is the trace of *accessed program* and *memory addresses*, β is the *branch predictor*, and finally $n \in \mathbb{N}$ is the *speculation level*. Note that $\mathit{\Delta}$, μ and pc are functions over the current speculation level, which discuss in detail below. We use $\mathit{\Delta}_n$ for $\mathit{\Delta}(n)$, where $\mathit{\Delta}_0$ refers to the architectural state with no speculation (similarly μ_n, pc_n). We use dot notation (e.g., $s.\mathit{\Pi}$) to refer to components of s.

A program is *speculating* in state s (denoted in *speculating*(s) iff $s.n > 0$). Moreover, we use $\rho \doteq pc_n$ to refer to the current program counter, $\iota \doteq \mathit{\Pi}(\rho)$ to the current instruction. Following Cheang et al., we assume an evaluation function, where $[\![e]\!]_{\mathit{\Delta}_n}$ evaluates the expression e in the register state $\mathit{\Delta}_n$. Semantics of expression evaluation are shown below.

$$\textsc{Const}\,\frac{}{c = [\![c]\!]_{\mathit{\Delta}_n}} \qquad \textsc{Reg}\,\frac{\mathit{\Delta}_n(r) = v}{v = [\![r]\!]_{\mathit{\Delta}_n}} \qquad \textsc{UnOp}\,\frac{v' = [\![e]\!]_{\mathit{\Delta}_n} \qquad v = \Diamond_u v'}{v = [\![\Diamond_u e]\!]_{\mathit{\Delta}_n}}$$

$$\textsc{BinOp}\,\frac{v_1 = [\![e_1]\!]_{\mathit{\Delta}_n} \qquad v_2 = [\![e_2]\!]_{\mathit{\Delta}_n} \qquad v = v_1 \Diamond_b v_2}{v = [\![e_1 \Diamond_b e_2]\!]_{\mathit{\Delta}_n}}$$

We now define the transition relation, which are shown in Fig. 3. Each state s already contains a mapping from s to the next instruction ($s.\iota$) to be executed. Thus, the transition relation is of the form $s \rightsquigarrow s'$, which advances the machine state from s to s' by either executing $s.\iota$ or resolving a misprediction in s. The rules use *uninterpreted predicates*, *mispred*(n, β, pc) and *resolve*(n, β, pc), which model branch misprediction and resolution, respectively, and an *uninterpreted function*, *update*(n, β, pc), which models branch prediction. The values of uninterpreted predicates and functions are non-deterministically selected and updated. Thus, in our verification, for any uninterpreted predicate or predicate, we must check both the true and false cases.

For a sequence $\sigma \in X^*$ and an element $x \in X$, we use $\sigma \cdot x$ for the sequence σ with x appended to the end. Thus $\omega \cdot \langle x, y\rangle$ denotes ω with the pair $\langle x, y\rangle$ appended to the end.

For space reasons, we only discuss the most important aspects of the transition relation here and ask the interested reader to consult the original paper for full details [5]. When executing a branch instruction, there are *four possible outcomes*, determined by the combination of the condition evaluation, $[\![e]\!]_{\mathit{\Delta}_n}$, and *mispred*$(n, \beta, pc)$. If *mispred*$(n, \beta, pc)$ holds we increment n, copy the values at n in the current $\mathit{\Delta}$ and μ to the next n and finally update pc.

An *execution* of a program is a sequence π generated using the transition rules in Fig. 3.

3.2 Adversary model

In addition to the rules above, we require a model of the adversary's capabilities. Again, we follow Cheang et al. [5] and formalise an adversary model capable of

$$\textsc{RegUpdate}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = r := e \qquad D = \varDelta_n[r \mapsto [\![e]\!]_{\varDelta_n}] \\ \mathit{pc}' = \mathit{pc}[n \mapsto \mathit{pc}_n + 1] \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta[n \mapsto D], \mu, \mathit{pc}', \omega', \beta, n\rangle}$$

$$\textsc{Load}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = r := \mathsf{mem}[e] \qquad a = [\![e]\!]_{\varDelta_n} \\ D = \varDelta_n[r \mapsto \mu_n(a)] \qquad \mathit{pc}' = \mathit{pc}[n \mapsto \mathit{pc}_n + 1] \qquad \omega' = \omega \cdot \langle \rho, a \rangle\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta[n \mapsto D], \mu, \mathit{pc}', \omega', \beta, n\rangle}$$

$$\textsc{Store}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{mem} := \mathsf{mem}[e_1 \to e_2] \qquad a = [\![e_1]\!]_{\varDelta_n} \\ D = \mu_n[a \mapsto [\![e_2]\!]_{\varDelta_n}] \qquad \mathit{pc}' = \mathit{pc}[n \mapsto \mathit{pc}_n + 1] \qquad \omega' = \omega \cdot \langle \rho, a \rangle\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta, \mu[n \mapsto D], \mathit{pc}', \omega', \beta, n\rangle}$$

$$\textsc{BranchT}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{if}\ e\ \mathsf{goto}\ c \qquad \neg\ \mathit{mispred}(n,\beta,\mathit{pc}) \qquad [\![e]\!]_{\varDelta_n} \\ \mathit{pc}' = \mathit{pc}[n \mapsto c] \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle \qquad \beta' = \mathit{update}(n,\beta,\mathit{pc})\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta, \mu, \mathit{pc}', \omega', \beta', n\rangle)}$$

$$\textsc{BranchF}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{if}\ e\ \mathsf{goto}\ c \qquad \neg\ \mathit{mispred}(n,\beta,\mathit{pc}) \qquad \neg[\![e]\!]_{\varDelta_n} \\ \mathit{pc}' = \mathit{pc}[n \mapsto \mathit{pc}_n + 1] \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle \qquad \beta' = \mathit{update}(n,\beta,\mathit{pc})\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta, \mu, \mathit{pc}', \omega', \beta', n\rangle}$$

$$\textsc{MisPredT}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{if}\ e\ \mathsf{goto}\ c \qquad \mathit{mispred}(n,\beta,\mathit{pc}) \qquad [\![e]\!]_{\varDelta_n} \\ \mathit{pc}' = \mathit{pc}[n \mapsto c, n' \mapsto \mathit{pc}_n + 1] \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle \\ \beta' = \mathit{update}(n,\beta,\mathit{pc}) \qquad n' = n + 1\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta[n' \mapsto \varDelta_n], \mu[n' \mapsto \mu_n], \mathit{pc}', \omega', \beta', n'\rangle}$$

$$\textsc{MisPredF}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{if}\ e\ \mathsf{goto}\ c \qquad \mathit{mispred}(n,\beta,\mathit{pc}) \qquad \neg[\![e]\!]_{\varDelta_n} \\ \mathit{pc}' = \mathit{pc}[n \mapsto \mathit{pc}_n + 1, n' \mapsto c] \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle \\ \beta' = \mathit{update}(n,\beta,\mathit{pc}) \qquad n' = n + 1\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta', \mu', \mathit{pc}', \omega', \beta', n'\rangle}$$

$$\textsc{Goto}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{goto}\ c \qquad \mathit{pc}' = \mathit{pc}[n \mapsto c] \\ \omega' = \omega \cdot \langle \rho, \bot \rangle \qquad \beta' = \mathit{update}(n,\beta,\mathit{pc})\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta, \mu, \mathit{pc}', \omega', \beta', n\rangle}$$

$$\textsc{SpecFence}\ \dfrac{\begin{array}{c}\neg\ \mathit{resolve}(n,\beta,\mathit{pc}) \qquad \iota = \mathsf{specfence} \qquad \varDelta' = \varDelta \restriction 0 \qquad \mu' = \mu \restriction 0 \\ \mathit{pc}' = \mathsf{if}\ n = 0\ \mathsf{then}\ \mathit{pc}[n \mapsto \mathit{pc}_n + 1]\ \mathsf{else}\ \mathit{pc} \restriction 0 \qquad \omega' = \omega \cdot \langle \rho, \bot \rangle\end{array}}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta', \mu', \mathit{pc}', \omega', \beta, 0\rangle}$$

$$\textsc{Resolve}\ \dfrac{\mathit{resolve}(n,\beta,\mathit{pc}) \qquad \beta' = \mathit{update}(n,\beta,\mathit{pc}) \qquad n' = n - 1}{\langle \varPi, \varDelta, \mu, \mathit{pc}, \omega, \beta, n\rangle \rightsquigarrow \langle \varPi, \varDelta[n \mapsto \bot], \mu[n \mapsto \bot], \mathit{pc}[n \mapsto \bot], \omega, \beta', n'\rangle}$$

Fig. 3. Transition rules of the operational semantics, recall that ρ is defined to be pc_n. We define $A \restriction a \doteq \lambda x.\ \mathsf{if}\ x = a\ \mathsf{then}\ A(a)\ \mathsf{else}\ \bot$.

reading from architectural registers and non-secret memory. To this end, we use a tuple $\mathcal{A} = \langle \mathcal{T}_\rho, \mathcal{EP}, \mathcal{S}_\mathcal{T}, \mathcal{U}^\mu_{rd}, \mathcal{U}^\mu_{wr} \rangle$, where $\mathcal{T}_\rho \subseteq \mathit{Addr}_\varPi$ refers to the *set of trusted instruction memory addresses*, $\mathcal{EP} : \mathit{Addr}_\varPi \in \mathcal{T}_\rho$ is the *trusted program's entrypoint*, $\mathcal{S}_\mathcal{T} \subseteq \mathit{Addr}_\mu$ is the *secret memory addresses*, $\mathcal{U}^\mu_{rd} \subseteq \mathit{Addr}_\mu$ and $\mathcal{U}^\mu_{wr} \subseteq \mathit{Addr}_\mu$ are the *adversary readable* and *writable addresses*.

In our development, we must define some restrictions on the adversary's capabilities. We assume that that adversary can only read from the addresses in the set $\mathcal{U}^\mu_{rd}$ and write to addresses in $\mathcal{U}^\mu_{wr}$. To enforce this we define the following state predicates. Note that we leave the state s implicit in the definitions.

$$\begin{aligned} &\mathit{conformantLoad}_{\mathcal{A}} \doteq (\iota = \ r := \mathsf{mem}[e]) \longrightarrow [\![e]\!]_{\Delta_n} \in \mathcal{U}^{\mu}_{rd} \\ &\mathit{conformantStore}_{\mathcal{A}} \doteq (\iota = \ \mathsf{mem} := \mathsf{mem}[e_1 \to e_2]) \longrightarrow [\![e_1]\!]_{\Delta_n} \in \mathcal{U}^{\mu}_{wr} \end{aligned}$$

Both predicates are only required to hold if the program is not speculating or executing an untrusted instruction. Thus, we define $\mathit{conformantLS}_{\mathcal{A}} \doteq \rho \notin \mathcal{T}_\rho \wedge \neg\ \mathit{speculating} \longrightarrow \mathit{conformantLoad}_{\mathcal{A}} \wedge \mathit{conformantStore}_{\mathcal{A}}$.

A further constraint of the program memory is that any transition to an address in $\mathcal{T}_\rho$ must target an element in $\mathcal{EP}$. This is to prevent the bypass of speculative fences. The entrypoint from $\mathcal{EP}$ must exist at the boundary between an untrusted and trusted instruction. This is a property of an execution π, and is formalised by the following predicate:

$$\mathit{conformantEP}_{\mathcal{A}}(\pi) \doteq \forall i.\ \pi^i.\rho \notin \mathcal{T}_\rho \wedge \pi^{i+1}.\rho \in \mathcal{T}_\rho \longrightarrow \pi^{i+1}.\rho = \mathcal{EP}$$

The definitions above allow us to formalise the notion of a *conformant trace*, which is designed to remove spurious counterexamples. We specify that the initial state must not be speculating and satisfy some predicate $\mathit{init}(\pi^0)$. This is shown in the equation below:

$$\begin{aligned} \mathit{conformant}_{\mathcal{A}}(\pi) \doteq{} & \neg \mathit{speculating}(\pi^0) \wedge \mathit{init}(\pi^0)\ \wedge \\ & \qquad \mathit{conformantEP}_{\mathcal{A}}(\pi)\ \wedge (\forall i.\ \mathit{conformantLS}_{\mathcal{A}}(\pi^i)) \end{aligned}$$

We say any execution π that satisfies $\mathit{conformant}_{\mathcal{A}}(\pi)$ is a *trace* of the program.

3.3 Formalising TPOD

In this section, we formalise the components used in the definition of TPOD as given in §2.2. The presentation in §2.2 leaves the adversary implicit. Following §3.2, we have an explicit adversary model, thus formalise the components of TPOD in terms of this adversary.

The set of traces with no misprediction is defined as follows: $T \doteq \{\pi \in \mathit{Tr} \mid \forall i.\ \neg\ \mathit{mispred}(\pi^i.n, \pi^i.\beta, \pi^i.pc)\}$.

Next we define the low and high operations. Recall that $\mathcal{T}_\rho$ is the set of trusted instruction addresses with respect to an adversary $\mathcal{A}$. The low operations are given by $op^{\mathcal{A}}_{\mathcal{L}}(s) \doteq \mathsf{if}\ s.pc_0 \notin \mathcal{T}_\rho\ \mathsf{then}\ \Pi(s.pc_0)\ \mathsf{else}\ \bot$, where $s.pc_0$ is the architectural program counter for the current non-speculative instruction.

The high operations return both an instruction and the memory read. The instruction itself is given by $\mathrm{inst}^{\mathcal{A}}_{\mathcal{T}}(s) \doteq \mathsf{if}\ s.pc_0 \in \mathcal{T}_\rho\ \mathsf{then}\ \Pi(pc_0)\ \mathsf{else}\ \bot$ and the memory by $\mathcal{P}^{\mathcal{A}}_{\mathcal{T}}(s) \doteq \lambda a.\ \mathsf{if}\ a \notin \mathcal{S}_{\mathcal{T}}\ \mathsf{then}\ \mu_0(a)\ \mathsf{else}\ \bot$, recalling that $\mathcal{S}_{\mathcal{T}}$ denotes the secret memory addresses of $\mathcal{A}$. Moreover, we can determine the architectural memory using μ_0, which is the memory where no speculation is taking place. Formally, a high operation returns a tuple $op^{\mathcal{A}}_{\mathcal{H}}(s) \doteq \langle \mathrm{inst}_{\mathcal{T}}(s), \mathcal{P}_{\mathcal{T}}(s)\rangle$

We define low equivalence of two states s_1 and s_2 as follows:

$$\begin{aligned} s_1 \approx_{\mathcal{L}} s_2 \doteq{} & (\neg \mathit{speculating}(s_1) \vee \neg \mathit{speculating}(s_2)) \wedge (op_{\mathcal{L}}(s_1) \neq \bot) \longrightarrow \\ & (\forall a \in \mathcal{U}^{\mu}_{rd}.\ s_1.\mu_0(a) = s_2.\mu_0(a)) \wedge \\ & s_1.\Delta_0 = s_2.\Delta_0 \wedge s_1.\beta = s_2.\beta \wedge s_1.\omega = s_2.\omega \end{aligned}$$

Low-equivalence of traces is defined by pointwise lifting as discussed in §2.2.

4 Mechanisation Techniques

Our proofs are constructed using the proof assistant Isabelle/HOL, which overcomes the limitations of finite traces and the specific properties present in prior works, which used the UCLID5 model checker [24]. Our Isabelle/HOL theories act as a library, offering re-usability. The proofs themselves require very little human interaction.

Our workflow (see Fig. 1) is similar to that of Rasmussen [21], but we generate Isabelle/HOL theories instead of a UCLID5 representation. First, the C code is compiled to assembly. This step could target any compiler and architecture compatible with BAP [1]. In our work, we use the MSVC compiler to translate the C code into x86 assembly. Second the assembly code is fed into the Binary Analysis Platform (BAP) to generate the corresponding BIL [1]. Finally, we generate the optimised Isabelle/HOL theory for the given BIL. For this final translation step, we have developed an automatic translation tool from BIL to Isabelle/HOL that instantiates the theorems necessary to prove TPOD.

4.1 Program specification

In our development, we provide an Isabelle/HOL representation of each BIL instruction (see Fig. 4 for an example). All data types are represented as uninterpreted 64-bit words, with expression evaluation handled by Isabelle/HOL's built-in `Word_Lib` theories. We define a well-formed predicate which ensures that the program memory is valid and verifiable. Well-formed predicates extend to other components of the system state, such as the program counter, trusted set of addresses, and entry point.

We have developed a set of theorems to discharge trivial cases automatically. Below, we describe the most interesting of such trivial programs for which a well formed TPOD proof obligation can be discharged without a complex stepwise, inductive proof. We ask the interested reader to consult our Isabelle/HOL theories [10] for other examples.

Theorem 1. *TPOD holds for any well-formed Π if any of the following hold:*

1. $\forall \iota \in \Pi.\ \iota \neq \mathsf{if}\ e\ \mathsf{goto}\ c$,
2. $\forall \iota \in \Pi.\ \iota \notin \{r := \mathsf{mem}[e], \mathsf{mem} := \mathsf{mem}[e_1 \to e_2]\}$,
3. $\forall \rho \in \Pi.\ \rho \in \mathcal{T}_\rho$.

This theorem has been verified in Isabelle/HOL.

Intuitively speaking, condition 1 ensures that Π contains no branch instructions. TPOD assumes that we start in a state without speculation and branching introduces speculation. Therefore, if Π contains no branch instructions, there can be no speculation and no violation of TPOD. Condition 2 ensures that Π contains no load or store instructions. Recall that (4) in Fig. 2 assumes that

```
definition "ex01_Π_vulnerable ≡ [
  0 ↦ RAX := mem[Const array1_size_addr, el]:u64,
  1 ↦ CF := (BinOp (Reg RDI) (air_lt) (Reg RAX)),
  2 ↦ when (Reg CF) goto (ProgramAddress 4),
  3 ↦ goto (ProgramAddress 19),

  4 ↦ RAX := mem[Const array2_addr, el]:u64,
  5 ↦ RDX := mem[Const array1_addr, el]:u64,
  6 ↦ RCX := (Reg RDI),
  7 ↦ RCX := (BinOp (Reg RCX) (air_shiftl) (Const 3)),
  8 ↦ (V 274) := (Reg RCX),
  9 ↦ RDX := (BinOp (Reg RDX) (+) (Reg (V 274))),
  10 ↦ RDX := mem[Reg RDX, el]:u64,
  11 ↦ RDX := (BinOp (Reg RDX) (air_shiftl) (Const 12)),
  12 ↦ (V 284) := (Reg RDX),
  13 ↦ RAX := (BinOp (Reg RAX) (+) (Reg (V 284))),
  14 ↦ RDX := mem[Reg RAX, el]:u64,
  15 ↦ RAX := mem[Const temp_addr, el]:u64,
  16 ↦ RAX := (BinOp (Reg RAX) (AND) (Reg RDX)),
  17 ↦ mem := mem with [Const temp_addr, el]:u64 <- (Reg RAX),
  18 ↦ goto (ProgramAddress 19),

  19 ↦ goto Halt
]"
```

Fig. 4. Isabelle/HOL program memory for Example 1

the speculative traces are initially architecturally equivalent, thus the absence of memory operations prevents these traces from diverging. Condition 3 ensure that all instructions are trusted. Any two high states are trivially low equivalent to the adversary even if their states differ and would otherwise violate TPOD.

4.2 Operational semantics

We represent the operational semantics defined in §3 in our Isabelle/HOL theories using Hoare-style triples ($\{P\}\ S\ \{Q\}$), similar to those introduced in the Isabelle/HOL proofs for the seL4 kernel [15]. In our model, we define the precondition P and postcondition Q as state predicates and the statement S as a state transformer. Moreover, since the instructions to be executed can be determined from the pre-state s using $s.\iota$, we consider predicate transformers of the form $\{P\} \rightsquigarrow \{Q\}$.

We introduce halting, defined by $\mathit{halting} \doteq pc_n \notin \mathit{\Pi}$ to describe a state with a program counter that does not point to an instruction in $\mathit{\Pi}$. At this point the program cannot advance and must resolve if speculating. We say a state s has terminated ($\mathit{terminates}(s)$) iff it is halting and not speculating such that $\mathit{terminates}(s) \doteq \mathit{halting}(s) \wedge \neg \mathit{speculating}(s)$. If the system terminates it will stutter, at which point we can trivially infer $\{P\} \rightsquigarrow \{P\}$.

$$\frac{\textit{speculating } s_s \qquad s_s \rightsquigarrow s'_s}{(s_{ns}, s_s) \rightsquigarrow_2 (s_{ns}, s'_s)} \qquad \frac{\neg \textit{ speculating } s_s \qquad s_s \rightsquigarrow s'_s \qquad s_{ns} \rightsquigarrow s'_{ns}}{(s_{ns}, s_s) \rightsquigarrow_2 (s'_{ns}, s'_s)}$$

$$\frac{(s_1, s_3) \rightsquigarrow_2 (s'_1, s'_3) \qquad (s_2, s_4) \rightsquigarrow_2 (s'_2, s'_4)}{(s_1, s_2, s_3, s_4) \rightsquigarrow_4 (s'_1, s'_2, s'_3, s'_4)}$$

Fig. 5. Transition rules for the states in the four traces of TPOD, where s_s and s_{ns} are states with and without speculation, respectively

As TPOD is a four-trace hyperproperty, we are required to transition each of the four system states simultaneously. This means that for a system with k transition rules, we must consider k^4 cases across all the four traces which quickly becomes intractable. Many of theses cases are spurious and cannot occur within a well-formed TPOD execution. For example, state pairs (s_1, s_3) and (s_2, s_4) are *operationally equivalent* if the execute the same low and high operations and therefore maintain the same architectural state. Operationally equivalent state pairs can be transitioned synchronously using a new transition rule $\rightsquigarrow_2$, which is defined using $\rightsquigarrow$ (see Fig. 5). This reduces the quadratic complexity of checking two traces to a linear check. This reduction can be performed on both speculating and non-speculating states, leading to a 4-way synchronous check $\rightsquigarrow_4$, which reduced k^4 interleavings to a linear check as well.

By the definition of low equivalence of states $s_a \approx_{\mathcal{L}} s_b$, s_a and s_b may have architectural differences iff these are high states or the program is speculating. Transitions only ever append to the set of program and memory addresses ω, an architectural system component verified in low equivalence. If at any point the predicate $\textit{violation}_{\approx_{\mathcal{L}}}(s_a, s_b) \doteq s_a.\omega \preceq s_b.\omega \vee s_b.\omega \preceq s_a.\omega$ does not hold (where $\preceq$ denotes subsequence) then there exists no future transition in which $s_a.\omega = s_b.\omega$. Given a future transition in which we terminate in the low state s.t. $\textit{terminates}(s) \wedge \textit{op}_{\mathcal{L}}(s) \neq \perp$, we violate $s_a \approx_{\mathcal{L}} s_b$. In the case of the non speculative states s_1 and s_2 this will lead to an invalid trace, for the speculative states s_3 and s_4 this will violate TPOD.

We discuss execution traces in §4.3, and how we apply these predicates to catch invalid traces and violations of TPOD early.

4.3 Program execution

Using concatenation rules for Hoare triples ($\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\} \implies \{P\}\ S_1; S_2\ \{R\}$) we construct an inductive predicate that defines a partial execution across four traces $\textit{execute}(\pi_1, \pi_2, \pi_3, \pi_4)$ from any given system state given below. An execution is valid iff its traces are well-formed and each contain a single state, or the last two states in each trace are a valid transition. These execution traces are not required to satisfy the initial state requirements of the $\textit{conformant}_{\mathcal{A}}(\pi)$ predicate introduced in §3.2.

We use a predicate *wfs* to indicate that a state is well-formed. By showing that $\{\textit{wfs}\} \rightsquigarrow \{\textit{wfs}\}$ we build invariants that minimize spurious transitions and discharge trivially unreachable executions. This is extended to $\rightsquigarrow_2$ and $\rightsquigarrow_4$ for two and four-trace hyperproperties.

$$\begin{array}{lll}
\mathit{execute}(\pi_1, \pi_2, \pi_3, \pi_4) \doteq & & \\
\text{if } \pi_i = [s_i], i \in \{1,2,3,4\} & \text{then} & \mathit{wfs4}(s_1, s_2, s_3, s_4) \land \mathit{violation}_{\approx_{\mathcal{L}}}(s_1, s_2) \\
\text{if } \pi_i = (\pi'_i \cdot s_i) \cdot s'_i, i \in \{1,2,3,4\} & \text{then} & (s_1, s_2, s_3, s_4) \rightsquigarrow_4 (s'_1, s'_2, s'_3, s'_4) \land \\
& & \mathit{execute}(\pi'_1 \cdot s_1, \pi'_2 \cdot s_2, \pi'_3 \cdot s_3, \pi'_4 \cdot s_4) \land \\
& & \mathit{violation}_{\approx_{\mathcal{L}}}(s_1, s_2)
\end{array}$$

The theory below, proven in Isabelle/HOL describes the trivial cases which when satisfied can discharge a proof of low equivalence $\pi_3 \approx_{\mathcal{L}} \pi_4$ across partial executions.

Theorem 2. *TPOD holds for any partial execution $\mathit{execute}(\pi_1, \pi_2, \pi_3, \pi_4)$ given any of the following hold:*

1. $\neg \mathit{speculating}(\pi_3^0) \land \neg \mathit{speculating}(\pi_4^0) \land \ \pi_3^0 \approx_{\mathcal{L}} \pi_4^0 \ \land$ $((\forall i.\ \neg \mathit{speculating}(\pi_3^i)) \lor (\forall i.\ \neg \mathit{speculating}(\pi_4^i)))$,
2. $\forall i.\ \mathit{op}_{\mathcal{H}}(\pi_3^i) \neq \bot$,
3. $\forall i.\ \mathit{speculating}(\pi_3^i) \land \mathit{speculating}(\pi_4^i)$.

This theorem has been verified in Isabelle/HOL. Condition 1 inherently ensures that traces π_3 or π_4 do not speculate. Without speculation in traces π_3 and π_4 the architectural state remains constant and we satisfy low equivalence. Given we assume the initial states are not speculating and are architecturally equivalent we can infer for any i that $\mathit{speculating}(\pi_3^i)$ iff $\mathit{speculating}(\pi_4^i)$ as branch predictor will make the same prediction across all four traces. Therefore, we satisfy TPOD if π_3 or π_4 do not speculate. Conditions 2 and 3 ensures all states in a trace execute high operations or are speculative respectively. As the adversary cannot observe the trusted 'high' state or any speculative state then low equivalence will trivially hold, even if the architectural state is not equivalent.

We can join two partial executions given the first execution ends in four parallel states that transition (via $\rightsquigarrow_4$) to the initial four states in the second execution. This allows us to simplify TPOD proof obligations by joining trivially low equivalent partial executions with complex partial executions that require stepwise proofs. Using overloading, we also use $\cdot$ notation to mean sequence concatenation and sequence prepending.

$$\frac{\begin{array}{c}\mathit{execute}(\pi_1 \cdot s_1, \pi_2 \cdot s_2, \pi_3 \cdot s_3, \pi_4 \cdot s_4) \\ \mathit{execute}(s'_1 \cdot \pi'_1, s'_2 \cdot \pi'_2, s'_3 \cdot \pi'_3, s'_4 \cdot \pi'_4) \\ (s_1, s_2, s_3, s_4) \rightsquigarrow_4 (s'_1, s'_2, s'_3, s'_4)\end{array}}{\mathit{execute}(\pi_1 \cdot [s_1, s'_1] \cdot \pi'_1, \pi_2 \cdot [s_2, s'_2] \cdot \pi'_2, \pi_3 \cdot [s_3, s'_3] \cdot \pi'_3, \pi_4 \cdot [s_4, s'_4] \cdot \pi'_4)}$$

Finally, we define the predicate $\mathit{execute}_{TPOD}(\pi_1, \pi_2, \pi_3, \pi_4)$ which ensures a *full* and *conformant* (according to §3.2) execution of a program which eventually terminates in a low state. Given this, we can infer that if $\exists i.\mathit{violation}_{\approx_{\mathcal{L}}}(\pi_1^i, \pi_2^i)$ holds then our execution is invalid and if $\exists i.\mathit{violation}_{\approx_{\mathcal{L}}}(\pi_3^i, \pi_4^i)$ holds we violate $\pi_3 \approx_{\mathcal{L}} \pi_4$. It is sufficient to verify that TPOD holds for all execution traces that satisfy $\mathit{execute}_{TPOD}(\pi_1, \pi_2, \pi_3, \pi_4)$ to show that the program is secure in the context of TPOD.

```
00000336: when x >= array1_length goto %00000330
00001bb8: specfence // compiler generated (sub-optimal)
00001bbd: RAX := mem[array2, el]:u64
00001bc4: RDX := mem[array1, el]:u64
00001bdc: ... // some processing on the index of array1 RDX
00001c1c: RDX := mem[RDX, el]:u64
00001c54: ... // some processing on the index of array2 RAX
00001c6d: RDX := mem[RAX, el]:u64
00001c54: ... // some processing on the index of array2 RAX
00000330: ... // end of program
```

Listing 1.3. Excerpt of MSVC solution in BIL. The placement of the `specfence` is sub-optimal and can be moved so that it is executed immediately before `00001c6d`

5 Case studies

Paul Kocher provided 15 victim functions [16] to test the effectiveness of MSVC's Spectre mitigations and placement of speculative fences, realised as `lfence` in Intel x86 and `CSDB` in ARM. These examples are variations on the vulnerable code discussed in §2.1. We have used our framework to show that all 15 examples contain secure speculation vulnerabilities.

We also verify correctness Examples 1, 2 and 8 from [16] (see §A). Of these, examples 1 and 2 compile to the same BIL, thus only require one proof in Isabelle/HOL. Example 8 uses a ternary operator to perform a bounds check, which changes the logical flow such that `array1` and `array2` are always indexed even if the program is out-of-bounds.

MSVC correctly identifies and fixes the secure speculation vulnerability in examples 1 and 2 by placing a speculative fence immediately after the branch statement shown in Listing 1.3. However, placement of the speculative fence is not optimal. It occurs *prior* to either of the memory reads and any of the other non-vulnerable instructions. A more optimal solution is to move the `specfence` so that it is executed immediately before `00001c6d`. This optimised version of the program has also been proven correct using our Isabelle/HOL framework.

MSVC is unable to correctly place a `specfence` for the remaining 13 examples. We manually insert the necessary fences in example 8, and prove that this modified program satisfies secure speculation.

6 Related Work

Proving correctness of secure speculation has received a lot of attention in recent years. Abstract models capable of describing Spectre-like attacks have been developed using CSP [7] and pomsets [8]. Such models are further removed from the original programs, and hence, additional work is required to link proofs with the programs themselves. In contrast, our workflow (Fig. 1) ensures that we verify the compiled assembly generated from the original program.

Correctness of secure speculation is of particular interest in the context of cryptographic code [4, 11, 27]. The properties of interest for cryptographic code are stronger than TPOD, making these proofs simpler since violations are easier to detect. We are interested in general programs making TPOD more applicable.

Many of the analysis techniques have associated tools. For hardware, this includes SAT-based approaches [26] and Unique Program Execution Checking (UPEC) [9]. For languages, tools include those based on static analysis [4], static typing [27], concolic analysis [12] and model checking [5]. Such tools are fine-tuned to handle a specific property (often more restrictive than TPOD) with a fixed execution semantics, and there is no guarantee that tools themselves are correct. Our theorem proving-based approach is more transparent. Moreover, we also have flexibility to openly change the operational semantics (to incorporate other architectural features) and the properties being verified independently.

Our proofs do not yet consider more sophisticated behaviours, e.g., out-of-order executions, thus we do not yet check the full range of Spectre variants. However, the introduction of out-of-order executions introduces a large amount of non-determinism meaning existing tools (including [4, 12, 28]) become infeasible [12]. It will be interesting to see the impact of these (more permissive) behaviours for our current proof technique in future work.

7 Conclusions

This paper has presented a mechanisation of a recently developed operational semantics by Cheang et al. [5] in Isabelle/HOL. The mechanisation integrates with an existing workflow (Fig. 1) that allows one to trace the Isabelle/HOL theories back to original C programs. One of our core contributions is a translation tool that generates Isabelle/HOL theories for BIL representations of the C program. As discussed in §6 existing tools on verifying secure speculation are generally based on symbolic or static analysis of the programs, in contrast to our methods, which are based on Hoare-logic encodings within a deductive proof environment. Our mechanisation closely follows Cheang et al. [5], but it also reveals some minor issues in their presentation, and alternative characterisations for some aspects of the operational semantics, as discussed earlier.

The most challenging aspect of our development has been the state space explosion caused by the fact that for a state transition with k possible transition rules, since TPOD is a four-trace hyperproperty, a naive expansion would required one to consider k^4 cases for each step, which is infeasible. Our solution is described in §4.3. Further proof optimisation has been achieved by identifying programs and partial executions that that trivially satisfy TPOD, which allows one to discharge proofs of large sections of code automatically (see §4.3).

In future work, we will extend our approach to understand and verify the requirements of a Spectre-safe library as well as considering next generation Spectre vulnerabilities mitigated by recompilation. We also aim to move our approach forward with the modern web, e.g., WebAssembly, into our model and verifying that this too is secure against Spectre.

References

1. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806, pp. 463–469. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_37
2. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: Enck, W., Felt, A.P. (eds.) USENIX Security Symposium. pp. 991–1008. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/bulck
3. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Bulck, J.V., Yarom, Y.: Fallout: Leaking data on Meltdown-resistant CPUs. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) CCS. pp. 769–784. ACM (2019), https://doi.org/10.1145/3319535.3363219
4. 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.) PLDI. pp. 913–926. ACM (2020), https://doi.org/10.1145/3385412.3385970
5. Cheang, K., Rasmussen, C., Seshia, S.A., Subramanyan, P.: A formal approach to secure speculation. In: CSF. pp. 288–303. IEEE (2019), https://doi.org/10.1109/CSF.2019.00027
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010), https://doi.org/10.3233/JCS-2009-0393
7. Colvin, R.J., Winter, K.: An abstract semantics of speculative execution for reasoning about security vulnerabilities. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) FM 2019 International Workshops. Lecture Notes in Computer Science, vol. 12233, pp. 323–341. Springer (2019), https://doi.org/10.1007/978-3-030-54997-8_21
8. Disselkoen, C., Jagadeesan, R., Jeffrey, A., Riely, J.: The code that never ran: Modeling attacks on speculative evaluation. In: IEEE S&P. pp. 1238–1255. IEEE (2019), https://doi.org/10.1109/SP.2019.00047
9. Fadiheh, M.R., Müller, J., Brinkmann, R., Mitra, S., Stoffel, D., Kunz, W.: A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In: IEEE DAC. pp. 1–6. IEEE (2020), https://doi.org/10.1109/DAC18072.2020.9218572
10. Griffin, M., Dongol, B.: Isabelle files for "Verifying Secure Speculation in Isabelle/HOL" (2021), https://gitlab.eps.surrey.ac.uk/mg00634-phd/formal-secure-spec
11. 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. pp. 1853–1869. ACM (2020), https://doi.org/10.1145/3372297.3417246
12. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: Principled detection of speculative information flows. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1–19. IEEE (2020), https://doi.org/10.1109/SP40000.2020.00011

13. Intel: Transient execution attacks & related security issues by cpu. Tech. rep., Intel (2019), https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model, accessed 5 May, 2021
14. Kiriansky, V., Lebedev, I.A., Amarasinghe, S.P., Devadas, S., Emer, J.S.: DAWG: A defense against cache timing attacks in speculative execution processors. In: MICRO. pp. 974–987. IEEE Computer Society (2018), https://doi.org/10.1109/MICRO.2018.00083
15. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) SOSP. pp. 207–220. ACM (2009), https://doi.org/10.1145/1629575.1629596
16. Kocher, P.: Spectre mitigations in microsoft's c/c++ compiler (2018), https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, accessed 5 May, 2021
17. 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: Exploiting speculative execution. In: IEEE S&P. pp. 1–19. IEEE (2019), https://doi.org/10.1109/SP.2019.00002
18. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: Enck, W., Felt, A.P. (eds.) USENIX Security Symposium. pp. 973–990. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
19. McLean, J.: Proving noninterference and functional correctness using traces. J. Comput. Secur. **1**(1), 37–58 (1992), https://doi.org/10.3233/JCS-1992-1103
20. Prout, A., Arcand, W., Bestor, D., Bergeron, B., Byun, C., Gadepally, V., Houle, M., Hubbell, M., Jones, M., Klein, A., Michaleas, P., Milechin, L., Mullen, J., Rosa, A., Samsi, S., Yee, C., Reuther, A., Kepner, J.: Measuring the impact of spectre and meltdown. In: IEEE HPEC. pp. 1–5. IEEE (2018), https://doi.org/10.1109/HPEC.2018.8547554
21. Rasmussen, C.: Secure Speculation: From Vulnerability to Assurances with UCLID5. Master's thesis, EECS Department, University of California, Berkeley (May 2019), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-95.html
22. Ren, X., Moody, L., Taram, M., Jordan, M., Tullsen, D.M., Venkat, A.: I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In: ISCA (2021), https://www.cs.virginia.edu/venkat/papers/isca2021a.pdf
23. Roscoe, A.W.: CSP and determinism in security modelling. In: IEEE S&P. pp. 114–127. IEEE Computer Society (1995), https://doi.org/10.1109/SECPRI.1995.398927
24. Seshia, S.A., Subramanyan, P.: UCLID5: integrating modeling, verification, synthesis and learning. In: MEMOCODE. pp. 1–10. IEEE (2018), https://doi.org/10.1109/MEMCOD.2018.8556946
25. Taram, M., Venkat, A., Tullsen, D.M.: Context-sensitive fencing: Securing speculative execution via microcode customization. In: Bahar, I., Herlihy, M., Witchel, E., Lebeck, A.R. (eds.) ASPLOS. pp. 395–410. ACM (2019), https://doi.org/10.1145/3297858.3304060

26. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The checkmate approach. IEEE Micro **39**(3), 84–93 (2019), https://doi.org/10.1109/MM.2019.2910010
27. Vassena, M., Disselkoen, C., von Gleissenthall, K., Cauligi, S., Kici, R.G., Jhala, R., Tullsen, D.M., Stefan, D.: Automatically eliminating speculative leaks from cryptographic code with blade. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021), https://doi.org/10.1145/3434330
28. Wang, G., Chattopadhyay, S., Biswas, A.K., Mitra, T., Roychoudhury, A.: Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. ACM Trans. Softw. Eng. Methodol. **29**(3), 14:1–14:31 (2020), https://doi.org/10.1145/3385897

A Verified victim functions

A.1 Example 1 and 2

```
void victim_function_v01(size_t x) {
     if (x < array1_size) {
          temp &= array2[array1[x] * 512];
     }
}
```

Listing 1.4. Example 1 in C

```
void leakByteLocalFunction_v02(uint8_t k) {
     temp &= array2[(k)* 512];
}

void victim_function_v02(size_t x) {
     if (x < array1_size) {
          leakByteLocalFunction(array1[x]);
     }
}
```

Listing 1.5. Example 2 in C

```
0000030d: RAX := mem[0x4050, el]:u64
0000031d: CF := mem[RBP - 8, el]:u64 < RAX
00000336: when ~CF goto %00000330
00001e19: goto %00001bb8

00001bb8:
00001bbd: RAX := mem[0x4040, el]:u64
00001bc4: RDX := mem[0x4038, el]:u64
00001bcb: RCX := mem[RBP - 8, el]:u64
00001bdc: RCX := RCX << 3
00001c00: #313 := RCX
00001c03: RDX := RDX + #313
00001c1c: RDX := mem[RDX, el]:u64
00001c2d: RDX := RDX << 0xC
00001c51: #318 := RDX
00001c54: RAX := RAX + #318
00001c6d: RDX := mem[RAX, el]:u64
00001c74: RAX := mem[0x4058, el]:u64
00001c81: RAX := RAX & RDX
00001c9a: mem := mem with [0x4058, el]:u64 <- RAX
00001e1a: goto %00000330

00000330:
```

Listing 1.6. Example 1 and 2 in BIL, secure after placing a specfence immediately prior to the memory load at program address 00001c6d.

A.2 Example 8

```
void victim_function_v08(size_t x) {
    temp &= array2[array1[x < array1_size ? (x + 1) : 0] *
  512];
}
```

Listing 1.7. Example 8 in C

```
0000083d: RDX := mem[0x4040, el]:u64
00000844: RCX := mem[0x4038, el]:u64
0000084b: RAX := mem[0x4050, el]:u64
0000085b: CF := mem[RBP - 8, el]:u64 < RAX
00000874: when ~CF goto %0000086e
00001e2e: goto %00001681

0000086e:
0000087e: RAX := 0
00001e2f: goto %00000880

00001681:
00001686: RAX := mem[RBP - 8, el]:u64
00001697: RAX := RAX + 1
000016ba: RAX := RAX << 3
000016d3: goto %00000880

00000880:
00000890: #92 := RCX
00000893: RAX := RAX + #92
000008ac: RAX := mem[RAX, el]:u64
000008bd: RAX := RAX << 0xC
000008e1: #97 := RDX
000008e4: RAX := RAX + #97
000008fd: RDX := mem[RAX, el]:u64
00000904: RAX := mem[0x4058, el]:u64
00000911: RAX := RAX & RDX
0000092a: mem := mem with [0x4058, el]:u64 <- RAX
00000935: goto %00000330

00000330:
```

Listing 1.8. Example 8 in BIL, secure after placing a specfence immediately prior to the memory load at program address 000008fd.