Secure Speculation: From Vulnerability to Assurances with UCLID5

Cameron Rasmussen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-95
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-95.html

May 24, 2019

Acknowledgement

I would like to thank my advisor Sanjit Seshia, from helping me in my first semester as a transfer student to now. His guidance has been immensely helpful, and I've grown a lot working with him. I owe Pramod Subramanyan for all his advice and direction during projects; credit goes to him for the inception of many of these formalizations (that I gloss over) as well as some of the figures. I appreciate and owe Kevin Cheang as he has been working with me since I started my MS degree, working together on homework and projects alike. This thesis is a culmination of joint work which is also reported in a paper that is jointly authored with Kevin Cheang, Sanjit Seshia, and Pramod Subramanyan. My contributions to this work are found largely in the verification choices and adversary modeling.

Secure Speculation: From Vulnerability to Assurances with Uclid5

Cameron Rasmussen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee

Sanjit Seshia
Research Advisor

(Date)

★ ★ ★ ★ ★ ★ ★

Krste Asanovic
Second Reader

(Date)

Abstract

Spectre and Meltdown represented a family of new vulnerabilities called transient execution attacks. They have shown that micro-architectural state was taken for granted and side channels were capable of exposing more than was previously believed possible. This thesis addresses the problem of determining whether a given program is vulnerable to transient execution attacks. Using an approach based on formal methods, we formalize a secure speculation property, adversary and platform models, and use the Uclid5 tool to verify whether a given program satisfies the secure speculation property.

Acknowledgements

I would like to thank my advisor Sanjit Seshia, from helping me in my first semester as a transfer student to now. His guidance has been immensely helpful, and I've grown a lot working with him. I owe Pramod Subramanyan for all his advice and direction during projects; credit goes to him for the inception of many of these formalizations (that I gloss over) as well as the figures in the § 1.1. I also appreciate and owe Kevin Cheang as he has been working with me since I started on my MS degree, working together on homework and projects alike. Many a late night was spent debugging work on the verge of a deadline with him. The following thesis is a culmination of joint work between the four of us over the course of my degree which is also reported in a paper [15] that is jointly authored with Kevin Cheang, Sanjit Seshia, and Pramod Subramanyan. My contributions to this work are found largely in the verification choices and adversary modeling.

This work was supported in part by the ADEPT Center, SRC tasks 2867.001, the iCyPhy Center, and NSF grants CNS-1739816 and CNS-1646208.

Contents

Chapter 1 Introduction

Recently discovered vulnerabilities Spectre [32] and Meltdown [37] have brought us to reevaluate many previously accepted hardware architectures and optimizations. They highlighted that there can be a fundamental disconnect between the ISA and micro-architectural state while proving that these side channels are more easily exploited than expected. This point was driven even further in the subsequent flood of variants [24, 25, 26, 27, 28, 34, 39] and new attack vectors that this class of vulnerabilities, coined transient execution attacks [40], opened up. Not to be left behind, the number of mitigations proposed has been steadily increasing [30, 42, 43, 55], but they lack soundness guarantees, and in a high security context, we need more than just informal scrutiny to trust that a vulnerability is no longer exploitable.

Transient execution attacks work by exposing micro-architectural state to a user through side-channels. Their existence is due to a number of optimizations in modern processors which exist at the micro-architectural level, but are abstracted away at the architectural level such as: out-of-order execution, branch prediction, caching, speculation, etc. Out-of-order allows processors to execute nonsequential instructions as they become ready to execute minimizing time spent idle, but branching control flow poses a question of which instruction comes next. The processor would normally have to first resolve branches before knowing which instruction to execute next. Instead of waiting for branch resolution, the processor uses a prediction structure to guess whether a branch will be taken or not and begins executing instructions speculatively. Once the branch condition is resolved, the processor can walk back the speculative changes when there is a misprediction, or commit the changes as valid execution if the prediction was correct. This accounts for great performance gains because we don't have to pause execution to wait for branch resolution when we guess correctly, and even mispredictions are only roughly equivalent to idling until branch resolution. Predicting whether a branch will be taken has been observed to be reliable given temporal consistency and locality; if a branch was taken consistently in the past, it will likely be taken again. What is important here for these attacks is what is possible during speculative execution and what state gets walked back. Only architectural state (registers, memory buffers) is considered when walking back from a misprediction, whereas micro-architectural state such as the cache or prediction structure are not. This is our side channel; if we can detect differences in this micro-architectural state, then we can learn about what happened during speculative execution. If an attacker can train the prediction structure to mispredict on sensitive code, using our side channel we can discern secret information.

The immediate fix is to disable speculation, but disabling this optimization wholesale would come with incredible performance costs (as will disabling any of the other optimizations mentioned). Luckily, the processor does enable instruction (or memory) serialization with special instructions which can selectively turn off speculation or act as a speculation fence. These fence instructions then face the new problem of finding where to place them. This exact question was approached by the Microsoft Visual Studio team, and their answer was to use static analysis to identify where these instructions would be necessary [42]. The problem was that their solution is not sound, as Paul Kocher was able to show using a number of different examples that tweaked the canonical Spectre variant 1 example [31]. This is hints at the essence of the problem; we cannot formally reason about these vulnerabilities and thus cannot prove anything about the proposed mitigations.

Some other approaches have attempted to prevent information leakage through side channels [30], but that becomes a cat and mouse game of patching side channels only to find new ones being used. Each newly secured side channel will come at a performance cost, and it is infeasible to remove all side channels. For these reasons, this is not the approach proposed by this work.

1.1. Spectre Attack

This section will describe in greater detail the spectre variant 1 attack. We focus on spectre variant 1 as a motivating example because it presents the most interesting transient execution attack that hasn't yet been readily solved like variant 2, Meltdown or Foreshadow, but it is important to point out that this work can be generalized beyond just variant 1.

```
1  void victim_function_v01(ull x) {
2       if (x < array1_size) {
3            temp &= array2[array1[x] * 512];
4       }
5  }
```

Figure 1.1.: The canonical example of Spectre variant 1 and appeared as example 1 in Kocher's follow up post

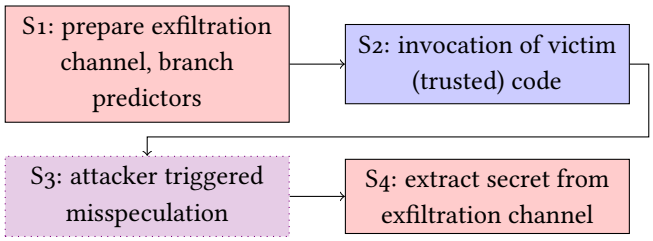

Figure 1.2.: Four Stages of a Speculative Execution Attack. Execution of untrusted code is shown in red, while trusted code is in blue. We show the attacker-triggered misspeculation in the trusted code in the violet dotted box.

The attack is carried out in 4 stages as shown in Figure 1.2. Stage 1 sees the attacker prepare the side channel and training the branch predictor. The commonly used side channel is the cache which can be primed by first flushing its contents then loading a number of sentinel values. The branch predictor can be trained by repeatedly forcing a branch to be taken so that it will expect to be taken in the future.

For Stage 2, the attacker invokes the vulnerable code with their malicious inputs. These inputs in concert with the trained branch prediction will force misspeculation. Here invocation does not necessarily require explicit API or functions calls, but any instance where an attacker can guarantee execution will follow stage 1 with their supplied inputs.

Stage 3 sees the vulnerable code intentionally trigger misspeculation which will cause changes in microarchitectural state. These changes will persist even after the branch is resolved, and misspeculation is walked back. In the case of the cache, that means we have evicted one of our sentinel values that was loaded into the cache previously, and which value was evicted will leak some secret information.

Stage 4 sees control return to the attacker where they then probe the side channel to learn any leakage caused by the misspeculation. For the cache, this involves reloading sentinel values and timing their accesses. The value that takes longer to load is the value that was evicted.

These stages are sketched out in Figure 1.3

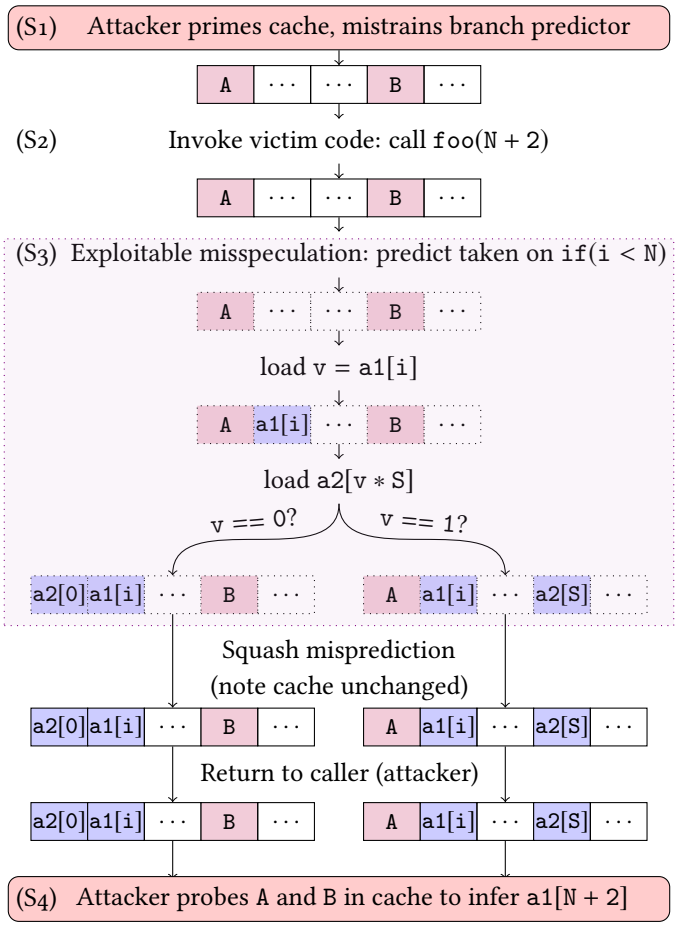

Figure 1.3.: Cache state evolution in Spectre variant 1. The rectangular boxes show the addresses that are cached. Untrusted accesses are red while accesses by trusted code are blue. For simplicity, we show the attack on a direct-mapped cache. The extension to set associative caches is straightforward.

1.2. Primer on UCLID5

We use formal methods to analyze whether a program is vulnerable to transient execution attacks like those described in the preceding section. In particular, we use UCLID5, a modeling and verification system for hardware-software systems [51]. UCLID5 is a verification language and toolkit. It supports modular abstraction which allows for easy compositional reasoning and handles both sequential code and transition systems. A model in UCLID5 consists of state variables, initial state and a transition relation. With these models, we can then prove assertions, invariants and/or properties. To solve these verification problems, UCLID5 uses satisfiability modulo theories (SMT) solvers [5].

Simple UCLID5 Example

```
module main {

    var a,b : integer;

    init {
        a = 0;
        b = 1;
    }

    next {
        a' = b;
        b' = a + b;
    }

    invariant a_lt_eq_b : a <= b;

    control {
        v = induction;
        check;
        print_results;
        v.print_cex();
    }

}
```

Figure 1.4.: UCLID5 5 model using induction to try to prove a property about the Fibonacci sequence

We illustrate the operation of UCLID5 with a very simple example. In Figure 1.4, we see two different ways to model the Fibonacci sequence. We have two state variables (`a,b`) both of which are integers. They are initialized in the `init` block. In the `next` block, we do primed assignments as part of the transition relationship when updating `a,b`. We then have an invariant that `a` is always less than or equal to `b`.

Finally the `control` block is our proof script for how to approach this problem. We will use single step induction, which will result in our invariants failing. This will happen because the solver starts in an arbitrary state which satisfy all invariants then checks if we can then transition from this arbitrary state to break any invariants which is possible when assigning negative values to `a` and `b`. To fix this, we would need to add strengthening invariants to constrain the state variables to be strictly non-negative, and with that, we can prove all invariants are true of the model.

Modular UCLID5 Example

The model in Figure 1.5 first implements `swap` which will swap the value of its two state variables `x,y` then tries to prove the correctness of `swap`'s implementation. To do this, it uses two new features: procedural code and module instantiation. Procedural code is handled in the procedure `update()` where we handle the addition swap between the state variables. Module instantiation let's us create an instance of a module, in this case `s`, which we can feed inputs and step through execution by using the `next` operator on it. This model satisfies the invariant that the addition swap works.

For a more in depth tutorial on how to use UCLID5 and what it can do, refer to [56].

```
module swap {
    input a, b : bv4;
    var x, y : bv4;

    init {
        x = a;
        y = b;
    }

    procedure update()
        modifies x, y;
    {
        x = x + y;
        y = x - y;
        x = x - y;
    }

    next {
        call update();
    }
}

module main {
    var one, two : bv4;
    var flag : boolean;

    instance s : swap(a : (one), b : (two));

    init {
        flag = true;
    }

    next {
        flag' = !flag;
        next(s);
    }

    invariant correct_swap :
        (flag ==> (one == s.x && two == s.y)) &&
        (!flag ==> (one == s.y && two == s.x));

    control {
        v = induction;
        check;
        print_results;
        v.print_cex();
    }
}
```

Figure 1.5.: UCLID5 model trying to prove that a swap using addition is correct

1.3. Overview

The two primary contributions of this thesis are first to generalize the property that accurately isolates transient execution vulnerabilities and second to create a formal verification methodology that would allow us to reason about them. The general property leverages a new class of information-flow security properties introduced in Cheang et. al [15] to capture the vulnerability. Figure 1.6 is our proposed workflow for reasoning about potentially vulnerable code using this generalized property.

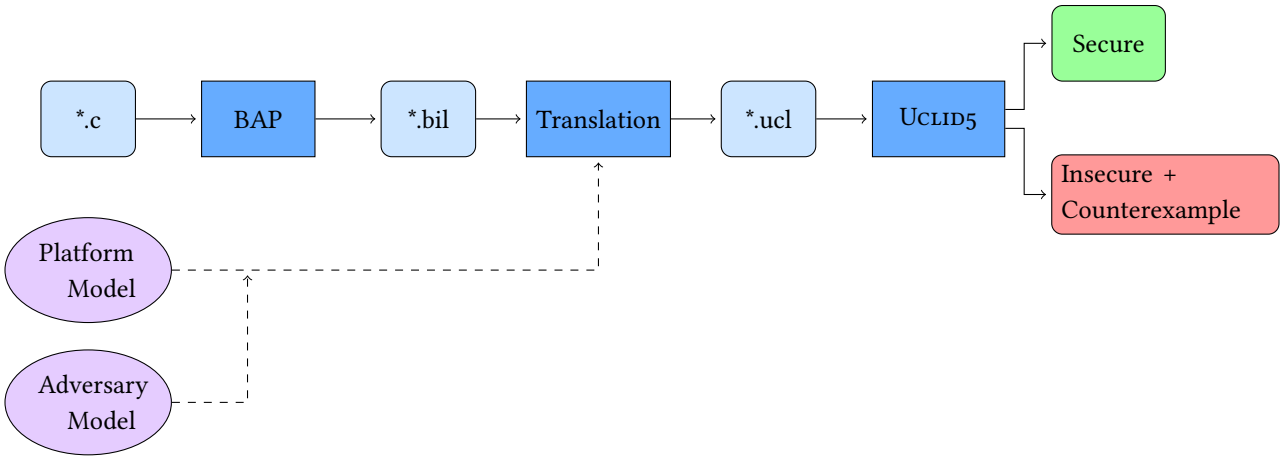

Figure 1.6.: Workflow from vulnerable C code to ultimately proving whether it is secure from transient execution attacks.

The Workflow

We start with three things: the vulnerable C code, the platform which we will be running on, and the threat model we are considering. The vulnerable code can be represented by the vulnerable snippet in Figure 1.1, while the adversary and platform model aren't explicit objects. The platform model takes into account that we are working on an optimized out-of-order machine with speculation. While the code may assume an in-order processor, Spectre relies on microarchitectural state, so how this platform works is a necessary component in order to model everything. The adversary model answers the question of what an attacker is capable of on our platform. While the platform model allows for microarchitectural state to be changed speculatively, the adversary model tells us that aspects of this state are observable by an attacker. These two models are composed together with the output of CMU's Binary Analysis Platform (BAP) [12] to translate the potentially vulnerable code into a Uclid5 [51] model. This model can then be handed off to Uclid5 to prove whether it is secure or not.

Chapter 2 Secure Speculation

It was the aim of this work to first define a generalized property that captures the class of vulnerabilities that Spectre represents, and then to provide a methodology which can be used to reason about code using this property. Before this can be done, it is necessary to construct a system model that is capable of speculation. Then the adversary must be considered with respect to the system model. Once we have a reasonable model of the platform and attacker, we can then formalize our property to isolate the vulnerability.

2.1. The Platform Model

Due to compiler optimizations and transformations, it is necessary that we reason about the vulnerability using assembly instructions. No specific instruction set architecture (ISA) lends itself any advantages to reasoning about the vulnerability, so we will use an abstract set of semantics that mimic BAP's instruction language (BIL). For instructions, we support:

- all basic ALU operations
- memory loads/stores
- direct branches
- jumps
- speculation fences

The instructions are standard with the exception of the speculation fence. The speculation fence prevents a processor from continuing to speculatively execute beyond the fence until all previous instructions are committed. This results in the fence instructions only finishing when not misspeculating. The other point to highlight is the exclusion of indirect branches. Their inclusion would increase the complexity of modeling substantially on a speculative processor due to the need for branch target buffers, indirect branch predictors, and the return call stack. This does mean we cannot reason about indirect jumps and returns.

Beyond the supported ISA, our processor must also support two optimizations: out-of-order execution and speculation. Through previous work on this project, we found that modeling an out-of-order processor that works on single instructions quickly becomes intractable in terms of runtime. Our decision was to approximate the effects of out-of-order execution. How this is done is further explained in 3.3.1.

For speculation, we focus only on conditional branches. An advantage the model has over hardware is that we only simulate stalling on instructions and speculation. This allows us to first evaluate the condition for a branch before we begin speculation which allows us to know when we are misspeculating because we know which branch *should* be taken. Speculation fences then work by disallowing further execution when misspeculating. We make speculation and branch prediction deterministic based on a combination of the processor model's current state and the condition being evaluated. This branch condition also causes the branch predictor's state to be updated. With our interpretation of out-of-order execution, when choosing to speculate on a branch, we will speculatively

execute the entire basic block before branch resolution or continuing to speculate further. This does end up being an overapproximation because we do not take into account some cases where data dependence may prevent certain instructions from being executed speculatively.

2.2. The Adversary Model

As outlined in the § 1.1, the attacker has two “modes." One where the attacker has control where he is able to prime the side channel, train the branch predictor, and probe the side channel. The other mode consists of the passive attacker where the victim executes the vulnerable code on the malicious input and setup. Because we choose to reason about vulnerable code, we focus on the passive attacker who is also capable of making observations and assume that any malicious activity before the execution of the vulnerable code has already been carried out such as priming the side channel or training the branch predictor.

The next necessary step is to define what the attacker is capable of observing. In § 1.1, the attacker depends on a side channel to exfiltrate information due to secret dependent memory accesses which results in changed cache state, but can only do so after the victim code is ran by detecting changes in the cache. To abstract away the exact implementation of the cache or even the specific side channel, we consider all memory addresses that are accessed during execution to be observable by the attacker. This considers a more powerful attacker who can consider all intermediate memory accesses rather than only the differences found in the cache (e.g. in the flush and reload or prime and probe attacks); more specifics are continued in § 3.3.1. Now, this alone is not sufficient; an attacker can still load a secret from an arbitrary address and taint other microarchitectural state using this secret. One such example would be to speculatively branch based on this potential secret:

```
1  void victim_function_nested_ifs(unsigned x) {
2        unsigned val1, val2;
3        if (x < array1_size) {
4            val1 = array1[x];
5            if (val1 & 1) {
6                val2 = 0;
7            }
8        }
9  }
```

Figure 2.1.: A Spectre variant 1 gadget that can result in changes to branch predictor state

In Figure 2.1, if misspeculation is triggered on line 3, then it would be possible to speculatively resolve the branch on line 5. If the branch on line 5 resolves to true, then it will be pulled into the branch predictor and cause future branches to be predicted true. Using similar methods that allow an attacker to train the branch predictor, an attacker could detect whether the branch predictor was updated. This will then leak information about the value loaded on line 5. To capture this, it is necessary to consider branch predictor state as observable.

As mentioned in the platform model, we do use an overappoximation of the system. Because of this, we expose ourselves to false positives, but our security guarantees remain sound when proving code is not vulnerable. In terms of our modeling, each basic block will be represented with a procedure.

2.3. The Property

We wish to formulate the property in a way that it isolates leaks that transient execution is solely responsible for. Given an otherwise secure program, we want to be able to reason about if a Spectre-like vulnerability can leak new information. To answer this dilemma, a new class of information flow properties was formulated, called trace property-dependent observational determinism (TPOD). It is a hyperproperty over four traces, or 4-safety property. Let the four traces be $\pi_1, \pi_2, \pi_3, \pi_4$ and assume that the following holds:

- π_1 and π_2 satisfy the trace property T
- π_3 and π_4 do not satisfy the trace property T
- All traces share the same low security input
- π_3 and π_4 share the same high security input as π_1 and π_2 respectively
- π_1 and π_2 have equivalent observable state

If all of this holds, then TPOD is satisfied if π_3 and π_4 have equivalent observable state. This property allows us to determine if the trace property T can allow for new observations, or leakages. If π_1 and π_2 make the same observations when given the same inputs as π_3 and π_4 respectively, then any new leaks must be directly attributable to trace property T being broken in π_3 and π_4.

The refinement to TPOD for this work is to consider T to be whether a trace is nonspeculative. This means that we will consider traces π_1 and π_2 to act as nonspeculative processors, whereas traces π_3 and π_4 are capable of speculative execution. Under this construction, if TPOD doesn't hold, then the vulnerability is directly attributable to speculative execution. This refinement is called secure speculation.

$$
\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4. \\
&\quad \text{nonspec}(\pi_1) \wedge \text{nonspec}(\pi_2) \wedge \text{spec}(\pi_3) \wedge \text{spec}(\pi_4) \implies \\
&\quad\quad \pi_1 \approx_{\mathcal{L}} \pi_2 \ \wedge \pi_1 \approx_{\mathcal{L}} \pi_3 \wedge \pi_1 \approx_{\mathcal{L}} \pi_4 \implies \\
&\quad\quad\quad \pi_1 \approx_{\mathcal{H}} \pi_3 \wedge \pi_2 \approx_{\mathcal{H}} \pi_4 \implies \\
&\quad\quad\quad\quad obs(\pi_1) = obs(\pi_2) \implies \\
&\quad\quad\quad\quad\quad obs(\pi_3) = obs(\pi_4)
\end{aligned}
$$

Figure 2.2.: Secure Speculation Property

2.4. Related Work

One of the closer related works to ours is CheckMate [54] which uses happens-before graphs to analyze transient execution vulnerabilities. CheckMate uses happens-before graphs to encode information about the orders in which instructions can be executed. By searching for patterns in the graph where branches are followed by dependent loads, an architectural model can be analyzed for susceptibility to Spectre/Meltdown. A key difference between CheckMate and our approach is that we first assume that the hardware is vulnerable and reason about whether a

program is can be exploited, whereas CheckMate reasons about whether a vulnerability is possible on a specific architecture through instruction orderings. For this reason, CheckMate's work results in a pattern based method whereas ours is semantic.

Another closely related effort is by McIlroy et al. [40] who introduce a formal model of speculative execution in modern processors and analyze it for transient execution vulnerabilities. Similar to our work, they too introduce speculative operational semantics, but their model includes indirect jumps and a timer. They focus on specific micro-architectural models whereas we use a more abstract semantics that are capable of speculation. This gives us a model that lends itself more easily to automated verification. Further, we propose the novel contribution of information-flow properties TPOD and its refinement, secure speculation.

The Spectre vulnerability was first discovered by Kocher et al. [32, 33] while Meltdown was discovered by Lipp et al. [37]. Their public disclosure has triggered an avalanche of new transient execution vulnerabilities. Some of the more notable examples of which are Foreshadow [13] which attacked enclave platforms and virtual machine monitors, SpectreRSB [34] and Ret2Spec [39]. A thorough survey of transient execution vulnerabilities was done by Canella et al. [14]. These vulnerabilities build on the well researched field of micro-architectural side-channel attacks [1, 22, 23, 29, 36, 38, 44, 45, 46, 50, 52]. Verification of mitigations to these "traditional" side-channel attacks is well-studied as well [2, 3, 4, 6, 9, 11, 17, 18, 19, 20, 47].

TPOD and its refinement, secure speculation, are examples of hyperproperties [16]. A large body of work has studied hyperproperties that encode secure information flow. Influential examples of this line of work include noninterference [21], separability [49] and observational determinism [41, 48, 57]. Our verification method is based on self-composition which has been well-studied (e.g., [7, 8]. While we take a straightforward approach to using self-composition, more sophisticated approaches are also possible in some cases (e.g., [53]).

Chapter 3 Modeling and Verification

Given the formal property, we then sought to explore a methodology which could allow users to reason about their code using said property. There was a number of decisions here, as always arise when trying to determine what is necessary to model. To reduce the trusted computing base, we chose to work at an assembly level. Out-of-order execution and speculation would be required to simulate the vulnerability, and we would need to have a concept of observable state in our models.

Our process follows 4 major steps which will be expanded upon in subsequent sections:

1. Compile C source code into binaries
2. Using CMU's Binary Analysis Platform (BAP), convert the binary into an architecture independent assembly
3. Extract the potentially vulnerable segment and translate it into a Uclid5 model
4. Compose the model with the security property so that it can be verified with Uclid5

As we walk through each of these steps, we will illustrate them on an example of the canonical example of Spectre variant 1 (Figure 1.1, Appendix A).

3.1. From C

The choice of C as a starting point was easy. All examples of the original paper as well as Kocher's followup to Microsoft's Qspectre Compiler mitigations were written in C, and its ubiquity in system code makes it a ripe target for attackers. The problem with C code is it's expressiveness, and how difficult it would be to capture these semantics in a generalizable way. As the shortcomings of Qspectre shows, C style semantics and compiler tricks can be hard to reason about (for static analysis or otherwise). For these reasons, we knew we wanted to start with C, but actually reason with something simpler.

As this example was explained previously in § 1.1, the explanation of the code is omitted here.

3.2. Using BAP

One of the benefits of C is that it works across all systems, but if we were going to be working at an assembly level, then we would end up on a specific architecture. CMU's Binary Analysis Platform [10, 12] allows us to abstract the exact architecture that we target when compiling, and conveniently breaks our code down into basic blocks with bare bones semantics. The output of BAP is referred to as the BAP Intermediate Language (BIL), and because it can be quite long, only the victim function is listed here. The entire output for the compiled C can be found at Appendix B.

```
1  000001ba: sub victim_function_v01()
2  00000172:
3  00000173: v377 := RBP
4  00000174: RSP := RSP - 8
```

```
5  00000175: mem := mem with [RSP, el]:u64 <- v377
6  00000176: RBP := RSP
7  00000177: mem := mem with [RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 <- RDI
```

This part of the BIL handles the function prologue. The activation record is created by pushing the old base pointer (v377) to the stack, setting the new base pointer (RBP), and loading the argument onto the stack (RDI).

```
8  00000178: RAX := mem[0x601050, el]:u64
9  0000017a: CF := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 < RAX
10 00000180: when ~CF goto %000001b4
11 00000181: goto %00000182
```

```
if (x < array1_size)
```

Here array1_size gets loaded from memory and assigned to the accumulator before being compared to the argument x. Then branching is handled based on the result of the comparison. If it is false, then we jump to 0x1b4 (function epilogue/clean up), otherwise we fall through to 0x182 and begin executing the body of the if.

```
12 00000182:
13 00000183: RAX := mem[0x601040, el]:u64
14 00000184: RDX := mem[0x601038, el]:u64
15 00000185: RCX := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64
16 00000187: RCX := RCX << 3
17 0000018f: v274 := RCX
18 00000190: RDX := RDX + v274
19 00000197: RDX := mem[RDX, el]:u64
```

```
temp &= array2[array1[x] * 512];
```

This is a lengthy encoding for a seemingly short line. Lines 0x183-0x185, we load the pointer array2 into RAX, array1 into RDX, and the argument x into RCX. Lines 0x187-0x190, we make x double aligned before adding it to array1. At line 0x197, we load from this address (RDX = array1[x]).

```
20 00000199: RDX := RDX << 0xC
21 000001a1: v284 := RDX
22 000001a2: RAX := RAX + v284
23 000001a9: RDX := mem[RAX, el]:u64
24 000001aa: RAX := mem[0x601058, el]:u64
25 000001ab: RAX := RAX & RDX
26 000001b2: mem := mem with [0x601058, el]:u64 <- RAX
27 000001b3: goto %000001b4
```

```
temp &= array2[array1[x] * 512];
```

Next we multiply array1[x] by 512 and make it double aligned before adding it as an offset to array2 to load from. Then the remaining lines, 0x1aa-0x1b2, handle loading the value of temp before doing a bitwise AND operation and storing it back to temp. This basic block finishes by falling through to the function epilogue.

```
28 000001b4:
29 000001b5: RBP := mem[RSP, el]:u64
30 000001b6: RSP := RSP + 8
31 000001b7: v295 := mem[RSP, el]:u64
32 000001b8: RSP := RSP + 8
33 000001b9: return v295
```

This is the function epilogue where we clean up the activation record and restore the saved based pointer.

3.3. Modeling with UCLID5

This section will detail the exact steps taken in translating the BIL output, adversary model, and platform model as well as the rationale behind these choices. The BIL and the models will be composed together to create a UCLID5 model of the program that simulates the its execution on a speculative processor while exposing certain state to the adversary's observations.

3.3.1. Abstractions for Scalability

As previously mentioned in § 2.1, we choose to approximate the execution of an out-of-order processor with speculation. We approximate out-of-order execution by capturing its effects in the "worst case" scenario where it executes all instructions in a basic block before speculation is resolved. This scenario models a superset of all observations an attacker can make, so the security guarantees remain sound under this abstraction. To do this, we will treat each basic block as an atomic operation on the state variables. This will mean our `pc` values will refer to the current basic block to be executed rather than individual instructions. Speculation will be handled when conditional branching is possible at the end of a basic block.

Memory was modeled with a word level array, where loading/storing from memory will correspond to reading from/storing to the array. We do not explicitly model caching (hits/misses, evictions), but consider every memory address accessed to be visible to an adversary. The adversary will log all memory accesses in order when loading/storing from memory, and these observations will be compared after each basic block.

The BIL utilizes a stack machine, so to avoid unnecessary stack use, we optimize out some of the unnecessary operations and remove the function prologue/epilogue from our model. This helps reduce the amount of reasoning with the theory of arrays that is offloaded to the back-end SMT solver to speed up our proofs.

3.3.2. Verification Choices

As was previously mentioned, secure speculation is a 4-safety hyperproperty, or a property that is quantifying over four traces of execution. We rely on the technique called self-composition to verify these properties, which sees us instantiate separate instances of a module for each trace that we are quantifying over. These instances are then sequentially executed in lock-step with the verification of a hyperproperty transformed to the verification of a safety property over the four-way self-composition. This is handled in UCLID5 through module construction and instantiation similar to Figure 1.5, except for secure speculation we will create four separate instances.

For the models capable of speculation, we know when we are misspeculating or speculatively executing valid instructions and simplify reasoning by only considering two different cases: speculative execution during misspeculation and valid execution. We do not consider the case where we are speculatively executing valid instructions because these instructions are guaranteed to be committed. This will be explained further when discussing the `branch()` procedure in § 3.3.4.

We approached this verification problem with two different methods: bounded model checking (BMC) and induction. Bounded model checking was used to identify whether code was insecure, and induction was used to

prove our mitigation satisfied the property. For BMC, we used a 32b memory, unlike BIL which assumes a 64b memory, for smaller models, while the induction models use uninterpreted types to scale.

We limit the speculation window by the number of basic blocks that can be executed. The window only needs to be as large as the number of basic blocks in the code we are analyzing when unrolling to ensure soundness if there are no loops. When the code contains loops, then BMC cannot be guaranteed to find the vulnerability, and induction will be required. For k-induction, the speculation window only needs to be large enough for k blocks to be sound.

Finally, we consider the inputs of these programs to not only include the arguments but the data memory as well. Because of this, we need to identify what would be considered secret information for the concept of low/high equivalence. To make this easier, we consider the initial data memories identical across traces except for a single address which contains a secret.

3.3.3. Common Module

What follows is the model for the induction proofs. We identified a number of type definitions, symbolic constants, and uninterpreted functions that were being reused across all model instances. The module abstraction in Uclid5 allows us to collect these in a single module which can then be reused for different models.

```
module common {
    type byte_t;
    type word_t;
    type addr_t = word_t;
    type mem_t = [addr_t]word_t;

    type pc_t = enum {
        block1,
        block2,
        block3,
        halt
    };
```

We are define the uninterpreted types for the models. Because the pc values can only jump between basic blocks, we restrict it to an enum type.

```
    type obs_mem_t;
    function update_obs_mem(mem_obs : obs_mem_t, addr : word_t) : obs_mem_t;
```

One of the most important aspects of the secure speculation property and adversary model is the definition of the observation function. In the canonical example, the attacker relies on the cache as a timing side channel, but we don't want to focus on specific side channels for exfiltration. We instead consider a more powerful attacker for the purposes of these models, one where the attacker can observe any memory address that gets accessed (instruction and data memory). This `obs_mem_t` type will be updated deterministically based on the sequence of memory accesses within each instance using the uninterpreted update function.

```
    type br_pred_state_t;
    function update_br_pred(state : br_pred_state_t, cond : boolean) : br_pred_state_t;
    function br_pred(state : br_pred_state_t, pc : pc_t) : boolean;
    function br_resolve(state : br_pred_state_t, pc : pc_t) : boolean;
```

The speculation decision needs to be deterministic between the speculative processors as long as it does not depend on secret inputs. If the speculative processors were allowed to diverge in their choices arbitrarily, then our observations would be trivially allowed to diverge. This data type will handle branch predictor state which will be used when deciding whether to misspeculate (`br_pred`) or not and when to walk back from misspeculation (`br_resolve`).

```
1      type spec_idx_t = bv2;
2      const spec_idx0 : spec_idx_t = 0bv2;
3      const spec_idx1 : spec_idx_t = 1bv2;
4      const spec_idx2 : spec_idx_t = 2bv2;
5      const spec_idx_max : spec_idx_t = 3bv2;
6      function walk_back(state : br_pred_state_t, pc : pc_t, spec_idx : spec_idx_t) : spec_idx_t;
7
8      type spec_mem_t      = [spec_idx_t]mem_t;   // Stores memory across speculation so it can be
       restored
9      type spec_reg_t      = [spec_idx_t]word_t;  // Stores shadow registers as we deepen speculation
10     type spec_flag_reg_t = [spec_idx_t]boolean; // Stores flag registers values for speculation
       checkpoints
11     type spec_pc_t       = [spec_idx_t]pc_t;    // Stores the PC value that would have been correct
```

This is how we will store the current level of speculation in the speculation window. We are not misspeculating when at `spec_idx0`. The uninterpreted function determines what state to walk back to when branch resolution happens. It is important to note that this will be something an attacker could observe because speculative branch resolution can affect persistent state in the branch predictor.

Finally the common module contains uninterpreted functions for our ALU operations because we are using uninterpreted types, and symbolic constants used to initialize values across the models. It is also explicitly assumed that all addresses referenced are distinct from each other, e.g. we do not consider `array1` as the secret address.

3.3.4. Program Module

Now that we have some of the basic types out of the way, we can begin understanding the construction of model. The model is primarily composed of somewhat optimized translations of the BIL basic blocks as procedures and a number of procedures that were reused across different models. The reason that these reused procedures couldn't be collected in the common module is because they both rely and can modify state of the module.

```
1  module program {
2      type * = common.*;
3
4      input speculative : boolean;
5      input lfence      : boolean;
6      var pc : pc_t;
7      var mem : mem_t;
8
9      // Registers
10     var RDI,
11         RAX,
12         RDX,
13         RCX: word_t;
14     var CF : boolean;
15
16     // Observable states
17     var br_pred_state : br_pred_state_t;        // Branch predictor state
```

```
18      var obs_mem       : obs_mem_t;
```

Here the model imports type definitions from common on line 2 and declares most of the state variables. The inputs `speculative` and `lfence` respectively determine if the model is capable of speculation and whether we have enabled the mitigation.

```
1       var spec_level : spec_idx_t;
2       var spec_mem : spec_mem_t;
3       var spec_pc : spec_pc_t;
4       var spec_RAX,
5           spec_RDX,
6           spec_RCX : spec_reg_t;
7       var spec_CF : spec_flag_reg_t;
```

The speculation stacks will store state before beginning a new level of misspeculation so that we can walk back state. It is important to note that the spec_pc will store the correct pc value to jump to if a branch gets resolved.

```
1       procedure do_block1()
2           modifies pc, mem,
3                    RDI, RAX, RDX, RCX, CF,
4                    br_pred_state, obs_mem,
5                    spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
6       {
7           // Ignore prologue
8           // 00000177: mem := mem with [RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 <- RDI
9           // 00000178: RAX := mem[0x601050, el]:u64
10
11          call (RAX) = load_mem(array1_size_addr);
12          // 0000017a: CF := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 < RAX
13          CF = common.lessthan(RDI, RAX);
14          // 00000180: when ~CF goto %000001b4
15          // 00000181: goto %00000182
16          call branch(!CF, block3, block2);
17      }
```

This is the first basic block translated into a Uclid5 procedure. The other blocks are translated similarly, with the inclusion of the final branch or assignment to pc directly dependent on the basic block. In this block, we see the use of two important procedures: `load_mem` and `branch`.

```
1       procedure load_mem(addr : word_t)
2           returns (value : word_t)
3           modifies obs_mem;
4       {
5           value = mem[addr];
6           obs_mem = common.update_obs_mem(obs_mem, addr);
7       }
8
9       procedure store_mem(addr : word_t, value : word_t)
10          modifies mem, obs_mem;
11      {
12          mem[addr] = value;
13          obs_mem = common.update_obs_mem(obs_mem, addr);
14      }
```

The memory functions read/write to memory while logging all accesses into the `obs_mem` variable. Because of the use of an uninterpreted function, if this procedure is ever called with different inputs between instances or a

different number of times between instances, then our observations will diverge between these instances. When our observations diverge, that means an attacker has found an observable difference during speculative execution that is secret dependent, and we have proven the model is insecure. We know that observable differences must be secret dependent because all initial state is equal between all instances except for the secrets, and the model is deterministic.

```
    procedure branch(cond : boolean, pc_if : pc_t, pc_else : pc_t)
        modifies pc, mem, br_pred_state,
                 RAX, RCX, RDX, CF,
                 spec_level, spec_mem, spec_RAX, spec_RCX, spec_RDX, spec_CF, spec_pc;
    {
        var pred : boolean;

        br_pred_state = common.update_br_pred(br_pred_state, cond);
        pred = common.br_pred(br_pred_state, pc);

        if (cond) {
            if (speculative && pred) {
                call save_reg_states(pc_if);
                spec_level = spec_level + common.spec_idx1;
                pc = pc_else;
            } else {
                pc = pc_if;
            }
        } else {
            if (speculative && pred) {
                call save_reg_states(pc_else);
                spec_level = spec_level + common.spec_idx1;
                pc = pc_if;
            } else {
                pc = pc_else;
            }
        }
    }
```

The branch procedure is more interesting by far. It first updates the branch predictor with the evaluation of the branch condition; if a branch is secret dependent, our branch predictors will diverge due to this update (and subsequent pc values will also diverge). After, we make a choice whether to misspeculate (possibly again) or not with our updated branch predictor, storing the result in `pred`. We then identify which branch is valid based on the branch condition and decide whether to misspeculate. If we are on a speculative processor and `pred` was true, then we push onto the speculative stack (notice how we pass the correct pc value) before choosing the wrong branch. If we are on a non-speculative processor or `pred` was false, we can continue onto the "valid" execution path. The word valid here is ambiguous because this we can currently be executing speculatively.

```
    procedure do_block2()
        modifies pc, mem,
                 RDI, RAX, RDX, RCX, CF,
                 br_pred_state, obs_mem,
                 spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
    {
        var v274, v284 : word_t;

        if (lfence && spec_level != common.spec_idx0) {
            call do_resolve();
```

```
        } else {
            // 00000183: RAX := mem[0x601040, el]:u64
            call (RAX) = load_mem(array2_addr);
            // 00000184: RDX := mem[0x601038, el]:u64
            call (RDX) = load_mem(array1_addr);
```

While the translations from basic blocks to BIL is formulaic, the beginning of `block2()` illustrates the implementation of the `lfence`, or speculation fence. The implementation first checks whether the `lfence` option is turned on and whether we are currently misspeculating. If these are both true, we force the model to walk back misspeculation without executing any further. The placement of the speculation fence here was chosen due to inspection of the vulnerable code snippet in Figure 1.1 and corroborated by counterexamples generated by BMC proving the existence of the vulnerability.

```
    next {
        // If speculating and resolving
        if (spec_level != common.spec_idx0 && common.br_resolve(br_pred_state, pc)) {
            call do_resolve();
        } else {
            case
                (pc == block1) : { call do_block1(); }
                (pc == block2) : { call do_block2(); }
                (pc == block3) : { call do_block3(); }
                (pc == halt)   : {}
            esac
        }
    }
```

The next block represents each transition for our model. If we are misspeculating and the branch predictor wants to resolve a branch, we do so. Otherwise, we execute the next block according to the pc.

```
    // Handles walking back misspeculation
    procedure do_resolve()
        modifies pc, mem, RAX, RDX, RCX, CF, spec_level;
    {
        var prev_spec_level : spec_idx_t;
        // Non deterministic choice of walkback level
        assume (prev_spec_level == common.walk_back(br_pred_state, pc, spec_level));
        assume (common.spec_idx0 <=_u prev_spec_level && prev_spec_level <_u spec_level);
        // Walkback
        spec_level = prev_spec_level;
        call restore_state();
    }
```

When walking back, we select any level of speculation which is greater than or equal to o (nonspeculative execution) and less than our current level. Once we've decided, state is walked back by popping off the speculation stacks.

Not explicitly shown are the translations of the other blocks, the model initialization, and pushing/popping from the speculative stack. The other block's procedures follow directly from the BIL and don't differ greatly from the first block shown. Model initialization largely assigns to the symbolic constants in the common module, but these initializations are largely unimportant in the inductive models. Finally pushing/popping from the speculative stack just assigns to either the stack or the variable it tracks and is unnecessary to show.

3.4. Composing the Model with the Property

With a model that is capable of speculation, it is now possible to compose the secure speculation property. In UCLID5 that entails instantiating 4 instances of the model (or traces), and we encode the implication chain as a number of assumptions and ultimately the tested invariants. Due to the nature of inductive proofs, we have a number of auxiliary invariants which are used to strengthen the proof to remove spurious counterexamples.

```
module main {
    type * = common.*;

    var lfence : boolean;

    instance t1 : program(speculative : (false), lfence : (lfence));
    instance t2 : program(speculative : (false), lfence : (lfence));
    instance t3 : program(speculative : (true),  lfence : (lfence));
    instance t4 : program(speculative : (true),  lfence : (lfence));

    assume (lfence == true);

    init {
        assume (t1.mem == t3.mem);
        assume (t2.mem == t4.mem);
    }
```

We instantiate the four traces, following the naming structure of the property: traces 1 & 2 are nonspeculative while 3 & 4 are speculative. The `lfence` is fed to them as an input; when it is false we should see the property broken, while all invariants should pass when true. Because state variables are initialized to the symbolic constants in the model's `init` block with the exception of the secrets, we need only ensure that the traces pairwise share secrets.

```
    assume (t1.pc == t2.pc);
    assume (t1.obs_mem == t2.obs_mem);
    assume (t1.br_pred_state == t2.br_pred_state);
```

Some of the implications are encoded here. The nonspeculative processors should be deterministic as should their observations because if they weren't then spectre style attacks wouldn't be required to exploit them.

```
    next {
        // If we are not speculating, assume we have seen the same things up to this point
        // If we are speculating, wait until the speculative model walks back to continue
        if (t3.spec_level == common.spec_idx0) {
            assume (t1.obs_mem == t3.obs_mem);
            next(t1);
        }
        if (t4.spec_level == common.spec_idx0) {
            assume (t2.obs_mem == t4.obs_mem);
            next(t2);
        }
        next(t3); next(t4);
    }
```

This is how we step execution of the traces. It is important to note that the non-speculative processors wait idle until the corresponding speculative processor finishes misspeculating. This makes it an invariant that the speculative processor's valid state is the same as the nonspeculative processor's current state. This is a vital aspect of leveraging the previous assumptions on the nonspeculative processors so that they constrain the search space for the speculative

processors. It is also necessary we assume that initial observations between corresponding nonspeculative and speculative processors are the same. When this is the case, we ensure that all calls to `update_mem_obs` by the speculative processor when not speculating will have the same inputs as its corresponding nonspeculative processor, and thus will have the same observations at the end of this step.

It means that for any observations to diverge, the speculative processors must be (mis)speculating, otherwise observations will be the same across all 4 traces. This allows the property to hone in on leakages that only happen as a result of speculation.

```
        // -------------- 4-Safety Properties ----------------------
    invariant same_pc : t3.pc == t4.pc; // Same PC
    invariant same_mem_obs : t3.obs_mem == t4.obs_mem;
    invariant same_br_pred_state : t3.br_pred_state == t4.br_pred_state;
```

These are the invariants that are testing the property for this program. Given all the assumes to satisfy the implication chain, these invariants must be true for the program to be secure. The speculative traces must follow the same pc values, make the same memory accesses, and branch prediction cannot have depended on secrets (even speculatively).

```
        // --------------- Auxiliary Invariants ----------------------
    // Same
    invariant same_RDI : t1.RDI == t2.RDI && t1.RDI == t3.RDI && t3.RDI == t4.RDI;
    // Start with the same speculation levels and return pcs (for resolution)
    invariant same_spec_level : t3.spec_level == t4.spec_level;
    invariant same_spec_pc : t3.spec_pc == t4.spec_pc;

    // Nonspeculative models should never speculate
    assume (t1.spec_level == t2.spec_level && t1.spec_level == common.spec_idx0);
```

This is the beginning of the auxiliary invariants used to prove that this model is secure when implementing the speculative fence mitigation. `same_RDI` just states that the attacker controlled input is the same across all traces. The next two invariants state that the speculative stacks are at the same level of misspeculation and that we've been making the same choices on each branch. If this wasn't true, then at some point our branch prediction already diverged and the property wouldn't hold. Finally, the last invariant asserts that the nonspeculative traces never speculate.

```
    // When not speculating, the speculative models should have identical state to the nonspeculative
    models
    invariant same_when_not_speculating :
        (t3.spec_level == common.spec_idx0 ==> (
            t1.spec_level == t3.spec_level &&
            t1.pc  == t3.pc &&
            t1.mem == t3.mem &&
            t1.RAX == t3.RAX &&
            t1.RCX == t3.RCX &&
            t1.RDX == t3.RDX &&
            t1.CF  == t3.CF)) &&
        (t4.spec_level == common.spec_idx0 ==> (
            t2.spec_level == t4.spec_level &&
            t2.pc  == t4.pc &&
            t2.mem == t4.mem &&
            t2.RAX == t4.RAX &&
            t2.RCX == t4.RCX &&
            t2.RDX == t4.RDX &&
```

```
18                t1.CF   == t3.CF));
```

This invariant states that the pairwise traces maintain the exact same state when not speculating.

```
    // When we completely undo misspeculation, we should be identical to the nonspeculative models
    invariant eventually_the_same :
        (t3.spec_level != common.spec_idx0 ==> (
            t1.pc  == t3.spec_pc[common.spec_idx0] &&
            t1.mem == t3.spec_mem[common.spec_idx0] &&
            t1.RAX == t3.spec_RAX[common.spec_idx0] &&
            t1.RCX == t3.spec_RCX[common.spec_idx0] &&
            t1.RDX == t3.spec_RDX[common.spec_idx0] &&
            t1.CF  == t3.spec_CF[common.spec_idx0])) &&
        (t4.spec_level != common.spec_idx0 ==> (
            t2.pc  == t4.spec_pc[common.spec_idx0] &&
            t2.mem == t4.spec_mem[common.spec_idx0] &&
            t2.RAX == t4.spec_RAX[common.spec_idx0] &&
            t2.RCX == t4.spec_RCX[common.spec_idx0] &&
            t2.RDX == t4.spec_RDX[common.spec_idx0] &&
            t2.CF  == t4.spec_CF[common.spec_idx0]));
```

This invariant is a small tweak on the previous one; the state is pairwise the same between the nonspeculative traces and the bottom of the speculative stack of their corresponding speculative trace while speculating. The bottom of the speculative stack represents the correct state assignment for nonspeculative execution, so this should hold as well.

```
    define same_mem_entry(mem : mem_t, addr : addr_t) : boolean =
        mem[addr] == common.mem_init[addr];

    // All memory locations that are not secret or written to are constant/read-only
    invariant same_mem_mostly : (forall (addr : addr_t) ::
        (addr != common.secret && addr != common.temp_addr_init) ==> (
            same_mem_entry(t1.mem, addr) &&
            same_mem_entry(t2.mem, addr) &&
            same_mem_entry(t3.mem, addr) &&
            same_mem_entry(t4.mem, addr)));
```

This is where the invariants become much more model specific. First we define a macro to evaluate whether a memory address contains the same value as the symbolic constant memory in common. Then, we identify all memory that must be the same among the traces. This invariant serves to identify both what addresses are defined as secret and what can be written to which is why this section differs so much depending on the model. Up to this point, the invariants and composition were not unique to this model. In this example, we only ever write back to the variable `temp` once, so it is an easy version of this invariant. More complicated models have required a concept of when they can be written to.

3.5. Experiments

We used our verifier for a proof-of-concept demonstration to detect whether or not an arbitrary snippet of C code is vulnerable to the Spectre class of attacks. As benchmarks, we rely on Paul Kocher's list of 15 example victim functions vulnerable to the Spectre attack [31].

In particular, we show here results on Examples 1, 5, 7, 8, 10, 11, and 15 from Paul Kocher's list, along with the example from Figure 2.1. We chose these based on what we believe are illustrative of a wide range of victim functions that are not easily detectable using the currently existing static analysis tools such as Qspectre [42], which was only able to detect the first two examples in Paul Kocher's list. The shortened version of Kocher's list we rely

on is primarily due to our modeling relying on assembly instruction semantics and multiple examples compiling to the same assembly. We begin with a brief explanation of some of the benchmark examples and then discuss the results from applying bounded model checking and induction with our secure speculation property on our UCLID5 5 models. Figure 3.1 lists all benchmarks we discuss here.

Example 1 (Figure 3.1a): This example was explained in detail in §1.1.

Example 5 (Figure 3.1b): This example is similar to the first variant but implemented within a for loop. The low-security argument `x` may be larger than the array size, which causes the attack to occur as in example 1, but if `x` is within bounds of the array, note that condition `i > 0` is also potentially vulnerable to the attack.[1]

Example 7 (Figure 3.1c): This example is interesting because it depends on the value of a static variable updated from a previous call of the function. Thus every call to the function should not make the second array access unless `x` is equal to `last_x`.

Example 8 (Figure 3.1d): The ternary operator is interesting because the program counter is allowed to jump to two different basic blocks for the computation of the second array memory access in the BIL as opposed to one block as in Example 1.

Example 10 (Figure 3.1e): This is the first example where a second load dependent on a secret is not required for a leak. Knowing whether or not `array2[0]` was accessed is enough to leak the secret at `array1[x]`.

Example 11 (Figure 3.1f): This example uses a call to memcpy to leak the secret, but because of the single byte access, it gets optimized out in the assembly to a single load and store.

Example 15 (Figure 3.1g): This example is interesting because it passes a pointer instead of an integer as the attacker controlled input. We assume that the value stored at the pointer is constant across traces to ignore cases where the attacker forces a secret dependent branch during non-speculative execution.

Example NI (Figure 3.1h) In this example, nested if statements cause the attack to occur without a second address load dependent on a secret. If the programs speculatively choose not to execute the second if statement, but only one program eventually executes the second if as a result of a resolution, then a leak can occur.

	ex1	ex5	ex7	ex8	ex10	ex11	ex15	Fig. 2.1	NI
BMC	6.558	8.957	10.243	5.688	9.604	6.366	5.783	6.599	12.896
Ind	4.991	5.066	5.693	4.647	5.812	5.864	4.838	4.792	5.433

Table 3.1.: Runtime (sec.) of each example using 5 steps for bounded model checking to find vulnerabilities and 1 step induction to prove correctness after inserting a memory fence. These experiments were run on a 64-bit Little Endian machine with a Intel(R) Core(TM) i7-2670QM CPU at 2.20GHz with 5737MiB of RAM.

Table 3.1 lists the run-time (in seconds) required for each verification task with the memory fences implemented. As can be seen, the verifier is able to prove the correctness of these programs within a few seconds. Although these programs are small, this exercise gives us confidence that the method could be useful on larger programs. Scaling to larger programs will need to adopt a stronger software model checking engine and property specific abstractions. We assert that with these improvements, it will be possible to prove secure speculation for larger programs.

[1] In Paul Kocher's list, the condition is `x >= 0`, but this introduces an infinite loop.

```
1  void victim_function_v01(unsigned x)
       {
2        if (x < array1_size) {
3              temp &= array2[array1[x] *
                   512];
4        }
5  }
```

(a) Example 1: Original Spectre BCB (bounds check bypass) example.

```
1  void victim_function_v07(unsigned x)
       {
2        static unsigned last_x = 0;
3        if (x == last_x) {
4              temp &= array2[array1[x] *
                   512];
5        }
6        if (x < array1_size)
7              last_x = x;
8  }
```

(c) Example 7: BCB with unsafe static variable check.

```
1  void victim_function_v10(unsigned x,
       unsigned k) {
2        if (x < array1_size) {
3              if (array1[x] == k)
4                    temp &= array2[0];
5        }
6  }
```

(e) Example 10: BCB using an additional attacker controlled input.

```
1  void victim_function_v15(unsigned *x)
       {
2        if (*x < array1_size) {
3              temp &= array2[array1[*x] *
                     512];
4        }
5  }
```

(g) Example 15: BCB using attacker controlled pointer.

```
1  void victim_function_v05(unsigned x)
       {
2        size_t i;
3        if (x < array1_size) {
4              for (i = x - 1; i > 0; i--)
5                    temp &= array2[array1[
                         i] * 512];
6        }
7  }
```

(b) Example 5: BCB with a for loop.

```
1  void victim_function_v08(unsigned x)
       {
2        result = (x < array1_size);
3        temp &= array2[array1[result ? (
             x + 1) : 0] * 512];
4  }
```

(d) Example 8: BCB with the ternary conditional operator.

```
1  void victim_function_v11(unsigned x)
       {
2        if (x < array1_size) {
3              temp = memcmp(&temp, array2
                     + (array1[x] * 512),
                   1);
4        }
5  }
```

(f) Example 11: BCB using the memory comparison function.

```
1  void victim_function_nested_ifs(
       unsigned x) {
2        unsigned val1, val2;
3        if (x < array1_size) {
4              val1 = array1[x];
5              if (val1 & 1) {
6                    val2 = 0;
7              }
8        }
9  }
```

(h) Example NI: BCB with nested if statements.

Figure 3.1.: Examples that were verified for secure speculation with speculation fences implemented. In all code snippets assume that that arguments x and k are untrusted (low-security) inputs to the trusted (high-security) victim functions.

Chapter 4 Conclusion

The focus of this thesis is the exact abstractions and decisions made in the modeling process that made it possible to reason about secure speculation. This thesis set out to start with Spectre and chronicle the process of arriving at a proven sense of security. We did this by delving through the formal verification methodology to explain each step of the process and the rationale behind the numerous decisions made when creating the program models. The result should be a general strategy which could be used to further reason about these vulnerabilities or other applications of trace property-dependent observational determinism.

Future Work

This work focused on modeling a specific threat model with Spectre variant 1 in mind, but could be extended. It would be possible to consider other vulnerabilities/variants through changes in the adversary and system model. Some of the areas to explore in an expanded threat model would be including the stack accesses (function prologue/epilogue), indirect jumps, and psuedoinstructions for resource power/congestion (half-warm on states/floating point operations). For stack accesses, it would be interesting to investigate if new stack smashing techniques could be found by utilizing transient execution. Indirect jumps have already been found abusable by variant 2 [32] and SpectreRSB [34], so the ability to reason about them would be invaluable (though these mitigations may rely on hardware fixes). The immediate problem with indirect jumps is that they would break the basic block abstraction we have used here, and would require that we reconsider some modeling choices. Another area that hasn't been fully addressed is some of these alternative side-channels which have been historically used. Some examples of these side-channels could be whether a unit gets powered on through a speculative instruction (power consumption optimization frequently cycles power off of unused components on the processor), or the floating point unit stalls on an instruction because of an operation that began speculatively. These new avenues can be addressed by introducing psuedoinstructions similar to how we currently handle reading/writing to memory, but would require more architecture specific support. In this way, we would be able to capture more subtle tricks and provide an even stronger security guarantee.

Going beyond this, we could consider other vulnerabilities that are not in the Spectre family. This could be done by considering other optimization techniques and refining TPOD with different trace properties. This area has proven itself ripe for further inspection given that speculative execution was first introduced over 28 years ago, proposed by Kung and Robinson [35], and only recently was Spectre discovered.

One of the greatest pain points in our modeling was the fact that TPOD is a 4-safety property and thus relies on very careful abstractions and optimizations. It is future work to see if the verification can be further optimized for these sort of hyperproperties using intelligent construction of the models without loss of generality of the property. This hurts us especially because of the limitation on the size of code we can reason about, placing large programs out of reach. One optimization that could address this is to explore the application of taint analysis as a preprocessing step to first identify minimal code portions which need to be reasoned about. This area would be a joint effort between both how the modeling is done and UCLID5's backend.

Bibliography

1. O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. "Predicting secret keys via branch prediction". In: *Cryptographers' Track at the RSA Conference*. Springer. 2007, pp. 225–242.
2. J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. "Formal verification of side-channel countermeasures using self-composition". In: *Science of Computer Programming* 78:7, 2013, pp. 796–812.
3. J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. "Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC". In: *International Conference on Fast Software Encryption*. Springer. 2016, pp. 163–184.
4. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. "Verifying constant-time implementations". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 53–70.
5. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by A. Biere, H. van Maaren, and T. Walsh. Vol. 4. IOS Press, 2009. Chap. 8.
6. G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. "System-level non-interference for constant-time cryptography". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1267–1279.
7. G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure Information Flow by Self-Composition". In: *17th IEEE Computer Security Foundations Workshop, (CSFW-17)*. 2004, pp. 100–114.
8. G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21:6, 2011, pp. 1207–1252.
9. G. Barthe, B. Grégoire, and V. Laporte. "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE. 2018, pp. 328–343.
10. Binary Analysis Platform (BAP) Repository. *Available at* https://github.com/BinaryAnalysisPlatform/bap. 2019.
11. B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. "Vale: Verifying high-performance cryptographic assembly code". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 917–934.
12. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT, 2011, pp. 463–469.
13. J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 2018, 991–1008. ISBN: 978-1-931971-46-1.
14. C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *CoRR* abs/1811.05441, 2018. arXiv: 1811.05441. URL: http://arxiv.org/abs/1811.05441.
15. K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. "A Formal Approach to Secure Speculation". In: *Computer Security Foundations Symposium, 32nd IEEE*. IEEE. 2019.
16. M. R. Clarkson and F. B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18:6, 2010, pp. 1157–1210. ISSN: 0926-227X.

17. G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, Washington, D.C., 2013, pp. 431–446. ISBN: 978-1-931971-03-4.

18. G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. “CacheAudit: A tool for the static analysis of cache side channels”. In: *ACM Transactions on Information and System Security (TISSEC)* 18:1, 2015, p. 4.

19. H. Eldib, C. Wang, and P. Schaumont. “Formal verification of software countermeasures against side-channel attacks”. In: *ACM Transactions on Software Engineering and Methodology* 24:2, 2014, p. 11.

20. H. Eldib, C. Wang, and P. Schaumont. “SMT-based verification of software countermeasures against side-channel attacks”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2014, pp. 62–77.

21. J. A. Goguen and J. Meseguer. “Security Policies and Security Models”. In: *IEEE Symposium on Security and Privacy*. 1982, pp. 11–20.

22. B. Gras, K. Razavi, H. Bos, and C. Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 955–972.

23. D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. 2016, pp. 368–379.

24. Intel. *Deep Dive: Analyzing Potential Bounds Check Bypass Vulnerabilities*. 2018. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-analyzing-potential-bounds-check-bypass-vulnerabilities.

25. Intel. *Deep Dive: Managed Runtime Speculative Execution Side Channel Mitigations*. 2018. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations.

26. Intel. *Deep Dive: Mitigation Overview for Side Channel Exploits in Linux*. 2018. URL: https://software.intel.com/security-software-guidance/insights/deep-dive-mitigation-overview-side-channel-exploits-linux.

27. Intel. *Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115*. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read.

28. Intel. *Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115*. 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass.

29. G. Irazoqui, T. Eisenbarth, and B. Sunar. “S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES”. In: *IEEE Symposium on Security and Privacy*. 2015, pp. 591–604.

30. V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. *DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors*. Cryptology ePrint Archive, Report 2018/418. https://eprint.iacr.org/2018/418. 2018.

31. P. Kocher. *Spectre Mitigations in Microsoft’s C/C++ Compiler*. 2018. URL: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

32. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *ArXiv e-prints*, 2018. arXiv: 1801.01203.

33. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proceedings of the IEEE Symposium on Security and Privacy*, 2019, pp. 19–37.

34. E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. “Spectre Returns! Speculation Attacks using the Return Stack Buffer”. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD, 2018.

35. H. T. Kung and J. T. Robinson. “On Optimistic Methods for Concurrency Control”. In: *ACM Trans. Database Syst.* 6:2, 1981, pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567. URL: http://doi.acm.org/10.1145/319566.319567.

36. S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing”. In: *CoRR* abs/1611.06952, 2016. URL: http://arxiv.org/abs/1611.06952.

37. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. “Meltdown”. In: *ArXiv e-prints*, 2018. arXiv: 1801.01207.

38. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. “Last-Level Cache Side-Channel Attacks Are Practical”. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 605–622. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.43. URL: http://dx.doi.org/10.1109/SP.2015.43.

39. G. Maisuradze and C. Rossow. “Ret2Spec: Speculative Execution Using Return Stack Buffers”. In: *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. 2018.

40. R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. “Spectre is here to stay: An analysis of side-channels and speculative execution”. In: *arXiv preprint arXiv:1902.05178*, 2019.

41. J. Mclean. “Proving Noninterference and Functional Correctness Using Traces”. In: *Journal of Computer Security* 1, 1992, pp. 37–58.

42. Microsoft. */Qspectre*. 2018. URL: https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2017.

43. Microsoft. *Spectre mitigations in MSVC*. 2018. URL: https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/.

44. Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. “The Spy in the Sandbox - Practical Cache Attacks in Javascript”. In: *CoRR* abs/1502.07373, 2015. URL: http://arxiv.org/abs/1502.07373.

45. C. Percival. *Cache missing for fun and profit*. 2005.

46. P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 565–581.

47. J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, et al. “Verified low-level programming embedded in F”. In: *Proceedings of the ACM on Programming Languages* 1:ICFP, 2017, p. 17.

48. A. W. Roscoe. “CSP and Determinism in Security Modelling”. In: *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*. 1995, pp. 114–127.

49. J. M. Rushby. “Proof of separability: A verification technique for a class of a security kernels”. In: *Proceedings of the International Symposium on Programming, 5th Colloquium, Torino, Italy*. 1982, pp. 352–367.

50. M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *CoRR* abs/1702.08719, 2017. URL: http://arxiv.org/abs/1702.08719.

51. S. A. Seshia and P. Subramanyan. “UCLID5: Integrating Modeling, Verification, Synthesis and Learning”. In: *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018.

52. Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur. “Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage”. In: *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. 2018, pp. 131–145.

53. M. Sousa and I. Dillig. “Cartesian Hoare Logic for Verifying K-safety Properties”. In: *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. Santa Barbara, CA, USA, 2016, pp. 57–69. ISBN: 978-1-4503-4261-2.

54. C. Trippel, D. Lustig, and M. Martonosi. “CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018*. 2018, pp. 947–960.

55. P. Turner. *Retpoline: a software construct for preventing branch-target-injection*. 2018. URL: https://support.google.com/faqs/answer/7625886.

56. UCLID5 Verification and Synthesis System. *Available at* http://github.com/uclid-org/uclid/.

57. S. Zdancewic and A. C. Myers. “Observational Determinism for Concurrent Program Security”. In: *Proc. of the 16th IEEE Computer Security Foundations Workshop*. IEEE. 2003, pp. 29–43.

Appendix A Spectre Variant 1 C Source

```
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

typedef unsigned long long int ull;
extern ull array1_size, array2_size, array_size_mask;
extern ull *array1, *array2, temp;
ull array1_size, array2_size, array_size_mask, temp;
ull *array1;
ull *array2;

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

int main() {
        ull x = 3;
        victim_function_v01(x);
        return 0;
}
```

Appendix B BIL of Spectre Variant 1

```
000001bb: program
00000006: sub __libc_csu_fini()
00000002:
00000003: v331 := mem[RSP, el]:u64
00000004: RSP := RSP + 8
00000005: return v331


00000083: sub __libc_csu_init()
00000007:
00000008: v241 := R15
00000009: RSP := RSP - 8
0000000a: mem := mem with [RSP, el]:u64 <- v241
0000000b: v243 := R14
0000000c: RSP := RSP - 8
0000000d: mem := mem with [RSP, el]:u64 <- v243
0000000e: R15 := pad:64[low:32[RDI]]
0000000f: v245 := R13
00000010: RSP := RSP - 8
00000011: mem := mem with [RSP, el]:u64 <- v245
00000012: v247 := R12
00000013: RSP := RSP - 8
00000014: mem := mem with [RSP, el]:u64 <- v247
00000015: R12 := 0x600E10
00000016: v249 := RBP
00000017: RSP := RSP - 8
00000018: mem := mem with [RSP, el]:u64 <- v249
00000019: RBP := 0x600E18
0000001a: v251 := RBX
0000001b: RSP := RSP - 8
0000001c: mem := mem with [RSP, el]:u64 <- v251
0000001d: R14 := RSI
0000001e: R13 := RDX
00000021: RBP := RBP - R12
0000002a: RSP := RSP - 8
00000032: RBP := RBP ~>> 3
00000039: RSP := RSP - 8
0000003a: mem := mem with [RSP, el]:u64 <- 0x400581
0000003b: call @_init with return %0000003c

0000003c:
0000003d: v337 := RBP
00000043: ZF := 0 = v337
00000044: when ZF goto %0000006a
00000045: goto %00000046

```

```
00000046:
00000047: RBX := 0
0000004e: goto %0000004f

0000004f:
00000050: RDX := R13
00000051: RSI := R14
00000052: RDI := pad:64[low:32[R15]]
00000053: v321 := mem[R12 + (RBX << 3), el]:u64
00000054: RSP := RSP - 8
00000055: mem := mem with [RSP, el]:u64 <- 0x40059D
00000056: call v321 with return %00000057

00000057:
0000005a: RBX := RBX + 1
00000061: v401 := RBX - RBP
00000067: ZF := 0 = v401
00000068: when ~ZF goto %0000004f
00000069: goto %0000006a

0000006a:
0000006d: RSP := RSP + 8
00000074: RBX := mem[RSP, el]:u64
00000075: RSP := RSP + 8
00000076: RBP := mem[RSP, el]:u64
00000077: RSP := RSP + 8
00000078: R12 := mem[RSP, el]:u64
00000079: RSP := RSP + 8
0000007a: R13 := mem[RSP, el]:u64
0000007b: RSP := RSP + 8
0000007c: R14 := mem[RSP, el]:u64
0000007d: RSP := RSP + 8
0000007e: R15 := mem[RSP, el]:u64
0000007f: RSP := RSP + 8
00000080: v317 := mem[RSP, el]:u64
00000081: RSP := RSP + 8
00000082: return v317


00000086: sub __libc_start_main()
00000084:
00000085: goto mem[0x601018, el]:u64


000000ac: sub _init()
00000087:
0000008a: RSP := RSP - 8
00000091: RAX := mem[0x600FF8, el]:u64
00000092: v347 := RAX
00000098: ZF := 0 = v347
```

```
00000099: when ZF goto %0000009f
0000009a: goto %0000009b

0000009b:
0000009c: RSP := RSP - 8
0000009d: mem := mem with [RSP, el]:u64 <- 0x4003A5
0000009e: call @sub_4003d0 with return %0000009f

0000009f:
000000a2: RSP := RSP + 8
000000a9: v393 := mem[RSP, el]:u64
000000aa: RSP := RSP + 8
000000ab: return v393


000000fa: sub _start()
000000ad:
000000ae: RBP := 0
000000b6: RSI := mem[RSP, el]:u64
000000b7: RSP := RSP + 8
000000b8: RDX := RSP
000000b9: RSP := RSP & 0xFFFFFFFFFFFFFFF0
000000c0: v325 := RAX
000000c1: RSP := RSP - 8
000000c2: mem := mem with [RSP, el]:u64 <- v325
000000c3: v327 := RSP
000000c4: RSP := RSP - 8
000000c5: mem := mem with [RSP, el]:u64 <- v327
000000c8: RDI := 0x400525
000000c9: RSP := RSP - 8
000000ca: mem := mem with [RSP, el]:u64 <- 0x400409
000000cb: call @__libc_start_main with return %000000cc

000000cc:
000000cd: RAX := 0x601037
000000ce: v297 := RBP
000000cf: RSP := RSP - 8
000000d0: mem := mem with [RSP, el]:u64 <- v297
000000d3: RAX := RAX - 0x601030
000000da: v305 := RAX - 0xE
000000db: CF := RAX < 0xE
000000e0: ZF := 0 = v305
000000e1: RBP := RSP
000000e2: when CF | ZF goto %000000ef
000000e3: goto %000000e4

000000e4:
000000e5: RAX := 0
000000e6: v333 := RAX
000000ec: ZF := 0 = v333
```

```
000000ed: when ZF goto %000000ef
000000ee: goto %000000f5

000000ef:
000000f0: RBP := mem[RSP, el]:u64
000000f1: RSP := RSP + 8
000000f2: v319 := mem[RSP, el]:u64
000000f3: RSP := RSP + 8
000000f4: return v319

000000f5:
000000f6: RBP := mem[RSP, el]:u64
000000f7: RSP := RSP + 8
000000f8: RDI := 0x601030
000000f9: goto RAX


00000117: sub main()
000000fb:
000000fc: v405 := RBP
000000fd: RSP := RSP - 8
000000fe: mem := mem with [RSP, el]:u64 <- v405
000000ff: RBP := RSP
00000102: RSP := RSP - 0x10
00000109: mem := mem with [RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 <- 3
0000010a: RAX := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64
0000010b: RDI := RAX
0000010c: RSP := RSP - 8
0000010d: mem := mem with [RSP, el]:u64 <- 0x400541
0000010e: call @victim_function_v01 with return %0000010f

0000010f:
00000110: RAX := 0
00000111: RSP := RBP
00000112: RBP := mem[RSP, el]:u64
00000113: RSP := RSP + 8
00000114: v309 := mem[RSP, el]:u64
00000115: RSP := RSP + 8
00000116: return v309


0000011a: sub sub_4003d0()
00000118:
00000119: goto mem[0x600FF8, el]:u64


00000171: sub sub_4004ca()
0000011b:
0000011c: v383 := RBP
0000011d: RSP := RSP - 8
```

```
197  0000011e: mem := mem with [RSP, el]:u64 <- v383
198  0000011f: RBP := RSP
199  00000120: v385 := RAX
200  00000121: RSP := RSP - 8
201  00000122: mem := mem with [RSP, el]:u64 <- 0x4004D0
202  00000123: call v385 with return %00000124
203
204  00000124:
205  00000125: RBP := mem[RSP, el]:u64
206  00000126: RSP := RSP + 8
207  00000127: goto %00000128
208
209  00000128:
210  00000129: RSI := 0x601030
211  0000012a: v351 := RBP
212  0000012b: RSP := RSP - 8
213  0000012c: mem := mem with [RSP, el]:u64 <- v351
214  0000012f: RSI := RSI - 0x601030
215  00000137: RSI := RSI ~>> 3
216  0000013e: RBP := RSP
217  0000013f: RAX := RSI
218  00000141: RAX := RAX >> 0x3F
219  00000149: v368 := RAX
220  0000014a: RSI := RSI + v368
221  00000152: RSI := RSI ~>> 1
222  00000155: ZF := 0 = RSI
223  00000159: when ZF goto %00000166
224  0000015a: goto %0000015b
225
226  0000015b:
227  0000015c: RAX := 0
228  0000015d: v291 := RAX
229  00000163: ZF := 0 = v291
230  00000164: when ZF goto %00000166
231  00000165: goto %0000016c
232
233  00000166:
234  00000167: RBP := mem[RSP, el]:u64
235  00000168: RSP := RSP + 8
236  00000169: v329 := mem[RSP, el]:u64
237  0000016a: RSP := RSP + 8
238  0000016b: return v329
239
240  0000016c:
241  0000016d: RBP := mem[RSP, el]:u64
242  0000016e: RSP := RSP + 8
243  0000016f: RDI := 0x601030
244  00000170: goto RAX
245
246
```

```
000001ba: sub victim_function_v01()
00000172:
00000173: v377 := RBP
00000174: RSP := RSP - 8
00000175: mem := mem with [RSP, el]:u64 <- v377
00000176: RBP := RSP
00000177: mem := mem with [RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 <- RDI
00000178: RAX := mem[0x601050, el]:u64
0000017a: CF := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 < RAX
00000180: when ~CF goto %000001b4
00000181: goto %00000182

00000182:
00000183: RAX := mem[0x601040, el]:u64
00000184: RDX := mem[0x601038, el]:u64
00000185: RCX := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64
00000187: RCX := RCX << 3
0000018f: v274 := RCX
00000190: RDX := RDX + v274
00000197: RDX := mem[RDX, el]:u64
00000199: RDX := RDX << 0xC
000001a1: v284 := RDX
000001a2: RAX := RAX + v284
000001a9: RDX := mem[RAX, el]:u64
000001aa: RAX := mem[0x601058, el]:u64
000001ab: RAX := RAX & RDX
000001b2: mem := mem with [0x601058, el]:u64 <- RAX
000001b3: goto %000001b4

000001b4:
000001b5: RBP := mem[RSP, el]:u64
000001b6: RSP := RSP + 8
000001b7: v295 := mem[RSP, el]:u64
000001b8: RSP := RSP + 8
000001b9: return v295
```

Appendix C Uclid5 Common Module

```
module common {
    type byte_t;
    type word_t;
    type addr_t = word_t;
    type mem_t = [addr_t]word_t;

    type pc_t = enum {
        block1,
        block2,
        block3,
        halt
    };

    type obs_mem_t;
    function update_obs_mem(mem_obs : obs_mem_t, addr : word_t) : obs_mem_t;

    type br_pred_state_t;
    function update_br_pred(state : br_pred_state_t, cond : boolean) : br_pred_state_t;
    function br_pred(state : br_pred_state_t, pc : pc_t) : boolean;
    function br_resolve(state : br_pred_state_t, pc : pc_t) : boolean;

    type spec_idx_t = bv2;
    const spec_idx0 : spec_idx_t = 0bv2;
    const spec_idx1 : spec_idx_t = 1bv2;
    const spec_idx2 : spec_idx_t = 2bv2;
    const spec_idx_max : spec_idx_t = 3bv2;
    function walk_back(state : br_pred_state_t, pc : pc_t, spec_idx : spec_idx_t) : spec_idx_t;

    type spec_mem_t      = [spec_idx_t]mem_t;   // Stores memory across speculation so it can be
    restored
    type spec_reg_t      = [spec_idx_t]word_t;  // Stores shadow registers as we deepen speculation
    type spec_flag_reg_t = [spec_idx_t]boolean; // Stores flag registers values for speculation
    checkpoints
    type spec_pc_t       = [spec_idx_t]pc_t;    // Stores the PC value that would have been correct

    function multiply(x : word_t, y : word_t) : word_t;
    function add(x : word_t, y : word_t) : word_t;
    function sub(x : word_t, y : word_t) : word_t;
    function left_shift(shift : word_t, x : word_t) : word_t;
    function mask(x : word_t, num_bits : word_t) : word_t;

    function lessthan(x : word_t, y : word_t) : boolean;
    function and(x : word_t, y : word_t) : word_t;

    const val0x0 : word_t;
    const val0x1 : word_t;
    const val0xC : word_t;

    const last_x_addr_init : addr_t;
    const array1_addr_init : addr_t;
    const array2_addr_init : addr_t;
    const array1_size_addr_init : addr_t;
    const temp_addr_init : addr_t;
```

```
52      const N_addr_init : addr_t;
53      const S_addr_init : addr_t;
54
55      const obs_mem_init : obs_mem_t;
56      const RAX_init : word_t;
57      const RCX_init : word_t;
58      const RDX_init : word_t;
59      const RDI_init : word_t;
60      const CF_init  : boolean;
61      const ret_init : word_t;
62      const mem_init : mem_t;
63      const br_pred_init : br_pred_state_t;
64      const spec_mem_init : spec_mem_t;
65      const spec_reg_init : spec_reg_t;
66      const spec_flag_reg_init : spec_flag_reg_t;
67      const spec_pc_init  : spec_pc_t;
68
69
70      const secret : addr_t;
71
72      assume distinct(secret, array1_addr_init, array2_addr_init, array1_size_addr_init, temp_addr_init)
        ;
73  }
```

Appendix D Uclid5 Model of Spectre Variant 1

```
module program {
    type * = common.*;

    input speculative : boolean;
    input lfence      : boolean;
    var pc : pc_t;
    var mem : mem_t;

    // Registers
    var RDI,
        RAX,
        RDX,
        RCX: word_t;
    var CF : boolean;

    // Observable states
    var br_pred_state : br_pred_state_t;        // Branch predictor state
    var obs_mem       : obs_mem_t;

    // Speculation stack for saving register values at branch checkpoints
    var spec_level : spec_idx_t;
    var spec_mem : spec_mem_t;
    var spec_pc : spec_pc_t;
    var spec_RAX,
        spec_RDX,
        spec_RCX : spec_reg_t;
    var spec_CF : spec_flag_reg_t;

    const array1_addr      : word_t;
    const array2_addr      : word_t;
    const array1_size_addr : word_t;
    const temp_addr        : word_t;

    assume (array1_addr == common.array1_addr_init);
    assume (array2_addr == common.array2_addr_init);
    assume (array1_size_addr == common.array1_size_addr_init);
    assume (temp_addr == common.temp_addr_init);

    procedure do_block1()
        modifies pc, mem,
                 RDI, RAX, RDX, RCX, CF,
                 br_pred_state, obs_mem,
                 spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
    {
        // Ignore prologue
        // 00000177: mem := mem with [RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 <- RDI
        // 00000178: RAX := mem[0x601050, el]:u64

        call (RAX) = load_mem(array1_size_addr);
        // 0000017a: CF := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64 < RAX
        CF = common.lessthan(RDI, RAX);
        // 00000180: when ~CF goto %000001b4
        // 00000181: goto %00000182
```

```
54          call branch(!CF, block3, block2);
55      }
56
57      procedure do_block2()
58          modifies pc, mem,
59                   RDI, RAX, RDX, RCX, CF,
60                   br_pred_state, obs_mem,
61                   spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
62      {
63          var v274, v284 : word_t;
64
65          if (lfence && spec_level != common.spec_idx0) {
66              call do_resolve();
67          } else {
68              // 00000183: RAX := mem[0x601040, el]:u64
69              call (RAX) = load_mem(array2_addr);
70              // 00000184: RDX := mem[0x601038, el]:u64
71              call (RDX) = load_mem(array1_addr);
72              // 00000185: RCX := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:u64
73              RCX = RDI;
74              // 00000187: RCX := RCX << 3
75              RCX = common.left_shift(common.val0x3, RCX);
76              // 0000018f: v274 := RCX
77              v274 = RCX;
78              // 00000190: RDX := RDX + v274
79              RDX = common.add(RDX, v274);
80              // 00000197: RDX := mem[RDX, el]:u64
81              call (RDX) = load_mem(RDX);
82              // 00000199: RDX := RDX << 0xC
83              RDX = common.left_shift(common.val0xC, RDX);
84              // 000001a1: v284 := RDX
85              v284 = RDX;
86              // 000001a2: RAX := RAX + v284
87              RAX = common.add(RAX, v284);
88              // 000001a9: RDX := mem[RAX, el]:u64
89              call (RDX) = load_mem(RAX);
90              // 000001aa: RAX := mem[0x601058, el]:u64
91              call (RAX) = load_mem(temp_addr);
92              // 000001ab: RAX := RAX & RDX
93              RAX = common.and(RAX, RDX);
94              // 000001b2: mem := mem with [0x601058, el]:u64 <- RAX
95              call store_mem(temp_addr, RAX);
96              // 000001b3: goto %000001b4
97              pc = block3; br_pred_state = common.update_br_pred(br_pred_state, true);
98          }
99      }
100
101     procedure do_block3()
102         modifies pc, mem,
103                  RDI, RAX, RDX, RCX, CF,
104                  br_pred_state, obs_mem,
105                  spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
106     {
107         // Ignore epilogue
108         pc = halt; br_pred_state = common.update_br_pred(br_pred_state, true);
109     }
110
111     assume (spec_level == common.spec_idx0 || spec_level == common.spec_idx1);
```

```
112
113     init {
114         var secret_value : word_t;
115         // Initiallze memory with 1 secret value in the secret address
116         mem = common.mem_init;
117         mem[secret] = secret_value;
118         // Initialize registers
119         RDI = common.RDI_init;
120         RAX = common.RAX_init;
121         RDX = common.RDX_init;
122         RCX = common.RCX_init;
123         CF  = common.CF_init;
124         // Initialize observation datastructures
125         obs_mem = common.obs_mem_init;
126         br_pred_state = common.br_pred_init;
127         // Initialize speculation metadata
128         spec_level = common.spec_idx0;
129         spec_mem = common.spec_mem_init;
130         assume (forall (addr_ : addr_t, spec_level_ : spec_idx_t) :: common.read(common.mem_init,
    addr_) == common.read(spec_mem[spec_level_], addr_));
131         spec_pc  = common.spec_pc_init;
132         // Initialize speculation register checkpoint arrays
133         spec_RAX = common.spec_reg_init;
134         spec_RDX = common.spec_reg_init;
135         spec_RCX = common.spec_reg_init;
136         spec_CF  = common.spec_flag_reg_init;
137         // Start PC at the first blcok of victim function
138         pc = block1;
139     }
140
141     next {
142         // If speculating and resolving
143         if (spec_level != common.spec_idx0 && common.br_resolve(br_pred_state, pc)) {
144             call do_resolve();
145         } else {
146             case
147                 (pc == block1) : { call do_block1(); }
148                 (pc == block2) : { call do_block2(); }
149                 (pc == block3) : { call do_block3(); }
150                 (pc == halt)   : {}
151             esac
152         }
153     }
154
155     procedure branch(cond : boolean, pc_if : pc_t, pc_else : pc_t)
156         modifies pc, mem, br_pred_state,
157                  RAX, RCX, RDX, CF,
158                  spec_level, spec_mem, spec_RAX, spec_RCX, spec_RDX, spec_CF, spec_pc;
159     {
160         var pred : boolean;
161
162         br_pred_state = common.update_br_pred(br_pred_state, cond);
163         pred = common.br_pred(br_pred_state, pc);
164
165         if (cond) {
166             if (speculative && pred) {
167                 call save_reg_states(pc_if);
168                 spec_level = spec_level + common.spec_idx1;
```

```
169                 pc = pc_else;
170             } else {
171                 pc = pc_if;
172             }
173         } else {
174             if (speculative && pred) {
175                 call save_reg_states(pc_else);
176                 spec_level = spec_level + common.spec_idx1;
177                 pc = pc_if;
178             } else {
179                 pc = pc_else;
180             }
181         }
182     }
183
184     // Add speculation checkpoint to speculation checkpoint stack
185     procedure save_reg_states(resolvePC : pc_t)
186         modifies spec_level, spec_mem, spec_pc, spec_RAX, spec_RDX, spec_RCX, spec_CF;
187     {
188         spec_RAX[spec_level] = RAX;
189         spec_RDX[spec_level] = RDX;
190         spec_RCX[spec_level] = RCX;
191         spec_CF[spec_level]  = CF;
192
193         spec_pc[spec_level] = resolvePC;
194         spec_mem[spec_level] = mem;
195     }
196
197     procedure restore_state()
198         modifies pc, mem, RAX, RDX, RCX, CF;
199     {
200
201         RAX = spec_RAX[spec_level];
202         RDX = spec_RDX[spec_level];
203         RCX = spec_RCX[spec_level];
204         CF = spec_CF[spec_level];
205
206         pc  = spec_pc[spec_level];
207         mem = spec_mem[spec_level];
208     }
209
210     // Handles walking back misspeculation
211     procedure do_resolve()
212         modifies pc, mem, RAX, RDX, RCX, CF, spec_level;
213     {
214         var prev_spec_level : spec_idx_t;
215         // Non deterministic choice of walkback level
216         assume (prev_spec_level == common.walk_back(br_pred_state, pc, spec_level));
217         assume (common.spec_idx0 <=_u prev_spec_level && prev_spec_level <_u spec_level);
218         // Walkback
219         spec_level = prev_spec_level;
220         call restore_state();
221     }
222
223     procedure load_mem(addr : word_t)
224         returns (value : word_t)
225         modifies obs_mem;
226     {
```

```
227         value = mem[addr];
228         obs_mem = common.update_obs_mem(obs_mem, addr);
229     }
230
231     procedure store_mem(addr : word_t, value : word_t)
232         modifies mem, obs_mem;
233     {
234         mem[addr] = value;
235         obs_mem = common.update_obs_mem(obs_mem, addr);
236     }
237 }
```

Appendix E Composition of Spectre Variant 1 with Secure Speculation

```
module main {
    type * = common.*;

    var lfence : boolean;

    instance t1 : program(speculative : (false), lfence : (lfence));
    instance t2 : program(speculative : (false), lfence : (lfence));
    instance t3 : program(speculative : (true),  lfence : (lfence));
    instance t4 : program(speculative : (true),  lfence : (lfence));

    assume (lfence == true);

    init {
        assume (t1.mem == t3.mem);
        assume (t2.mem == t4.mem);
    }

    assume (t1.pc == t2.pc);
    assume (t1.obs_mem == t2.obs_mem);
    assume (t1.br_pred_state == t2.br_pred_state);

    next {
        // If we are not speculating, assume we have seen the same things up to this point
        // If we are speculating, wait until the speculative model walks back to continue
        if (t3.spec_level == common.spec_idx0) {
            assume (t1.obs_mem == t3.obs_mem);
            next(t1);
        }
        if (t4.spec_level == common.spec_idx0) {
            assume (t2.obs_mem == t4.obs_mem);
            next(t2);
        }
        next(t3); next(t4);
    }

        // -------------- 4-Safety Properties ----------------------
    invariant same_pc : t3.pc == t4.pc; // Same PC
    invariant same_mem_obs : t3.obs_mem == t4.obs_mem;
    invariant same_br_pred_state : t3.br_pred_state == t4.br_pred_state;

        // --------------- Auxiliary Invariants ----------------------
    // Same
    invariant same_RDI : t1.RDI == t2.RDI && t1.RDI == t3.RDI && t3.RDI == t4.RDI;
    // Start with the same speculation levels and return pcs (for resolution)
    invariant same_spec_level : t3.spec_level == t4.spec_level;
    invariant same_spec_pc : t3.spec_pc == t4.spec_pc;

    // Nonspeculative models should never speculate
    assume (t1.spec_level == t2.spec_level && t1.spec_level == common.spec_idx0);
```

```
    // When not speculating, the speculative models should have identical state to the nonspeculative models
    invariant same_when_not_speculating :
        (t3.spec_level == common.spec_idx0 ==> (
            t1.spec_level == t3.spec_level &&
            t1.pc  == t3.pc &&
            t1.mem == t3.mem &&
            t1.RAX == t3.RAX &&
            t1.RCX == t3.RCX &&
            t1.RDX == t3.RDX &&
            t1.CF  == t3.CF)) &&
        (t4.spec_level == common.spec_idx0 ==> (
            t2.spec_level == t4.spec_level &&
            t2.pc  == t4.pc &&
            t2.mem == t4.mem &&
            t2.RAX == t4.RAX &&
            t2.RCX == t4.RCX &&
            t2.RDX == t4.RDX &&
            t1.CF  == t3.CF));

    // When we completely undo misspeculation, we should be identical to the nonspeculative models
    invariant eventually_the_same :
        (t3.spec_level != common.spec_idx0 ==> (
            t1.pc  == t3.spec_pc[common.spec_idx0] &&
            t1.mem == t3.spec_mem[common.spec_idx0] &&
            t1.RAX == t3.spec_RAX[common.spec_idx0] &&
            t1.RCX == t3.spec_RCX[common.spec_idx0] &&
            t1.RDX == t3.spec_RDX[common.spec_idx0] &&
            t1.CF  == t3.spec_CF[common.spec_idx0])) &&
        (t4.spec_level != common.spec_idx0 ==> (
            t2.pc  == t4.spec_pc[common.spec_idx0] &&
            t2.mem == t4.spec_mem[common.spec_idx0] &&
            t2.RAX == t4.spec_RAX[common.spec_idx0] &&
            t2.RCX == t4.spec_RCX[common.spec_idx0] &&
            t2.RDX == t4.spec_RDX[common.spec_idx0] &&
            t2.CF  == t4.spec_CF[common.spec_idx0]));

    define same_mem_entry(mem : mem_t, addr : addr_t) : boolean =
        mem[addr] == common.mem_init[addr];

    // All memory locations that are not secret or written to are constant/read-only
    invariant same_mem_mostly : (forall (addr : addr_t) ::
        (addr != common.secret && addr != common.temp_addr_init) ==> (
            same_mem_entry(t1.mem, addr) &&
            same_mem_entry(t2.mem, addr) &&
            same_mem_entry(t3.mem, addr) &&
            same_mem_entry(t4.mem, addr)));


    control {
        // v = unroll(3);
        v = induction;

        check;
        print_results;
        v.print_cex(t3.pc, t4.pc,
                    t3.spec_level, t4.spec_level,
                    t3.mem, t4.mem,
```

```
107                     t3.spec_mem, t4.spec_mem,
108                     common.read(t3.mem, t3.array1_size_addr), common.read(t4.mem, t4.array1_size_addr)
        ,
109                     t3.obs_mem, t4.obs_mem,
110                     t3.RAX, t4.RAX,
111                     (t3.RAX == t4.RAX),
112                     t3.RDX, t4.RDX,
113                     t3.lfence && t3.spec_level != common.spec_idx0,
114                     t4.lfence && t4.spec_level != common.spec_idx0);
115     }
116 }
```