Validation of Side-Channel Models via Observation Refinement

Pablo Buiras
KTH Royal Institute of Technology
Stockholm, Sweden
buiras@kth.se

Hamed Nemati
Stanford University
Stanford, United States
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
hnnemati@stanford.edu

Andreas Lindner
KTH Royal Institute of Technology
Stockholm, Sweden
andili@kth.se

Roberto Guanciale
KTH Royal Institute of Technology
Stockholm, Sweden
robertog@kth.se

ABSTRACT

Observational models enable the analysis of information flow properties against side channels. Relational testing has been used to validate the soundness of these models by measuring the side channel on states that the model considers indistinguishable. However, unguided search can generate test states that are too similar to each other to invalidate the model. To address this we introduce observation refinement, a technique to guide the exploration of the state space to focus on hardware features of interest. We refine observational models to include fine-grained observations that characterize behavior that we want to exclude. States that yield equivalent refined observations are then ruled out, reducing the size of the space. We have extended an existing model validation framework, Scam-V, to support refinement. We have evaluated the usefulness of refinement for search guidance by analyzing cache coloring and speculative leakage in the ARMv8-A architecture. As a surprising result, we have exposed SiSCLoak, a new vulnerability linked to speculative execution in Cortex-A53.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Hardware attacks and countermeasures*;

KEYWORDS

Testing, Side channels, Information flow security, Model validation, Microarchitectures

ACM Reference Format:
Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Validation of Side-Channel Models via Observation Refinement. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480130

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480130

1 INTRODUCTION

The complexity of modern processors has given rise to sophisticated side channels that may compromise security. Due to this complexity, information-flow analyses that handle such leaks rely on abstract models that overapproximate the attacker's view of the real system in terms of "observations", i.e. parts of the program state that may be leaked during execution. Such *observational models* are useful provided they do not miss any information flows: i.e., states that are *observationally equivalent* should be indistinguishable on the real hardware. Unfortunately, this assumption is often violated: side channels can arise from unforeseen interactions between microarchitectural features that are often hidden, undocumented, or underspecified. In recent years, speculative execution has emerged as a prime example of this type of vulnerability [27, 31, 33].

Previous work [37] has shown that testing can be used to identify these problems. Fig. 1 shows the general strategy for a single test case generation and execution in the Scam-V framework. (1) Scam-V generates a binary program; (2) it synthesizes a relation that identifies which states are observationally equivalent for the program according to the model under validation; (3) it generates an instance of this relation in terms of two input states; (4) it executes the test case on real hardware and measures the side-channel to reveal whether the instance is indistinguishable. Section 2 provides preliminaries required to elaborate on this process.

Validating observational models via relational testing can result in a state space explosion. Unguided search explores spurious states that are unlikely to invalidate the model, either because they are too similar to each other or because they fail to trigger microarchitectural behavior that can be measured by an attacker. It is therefore critical to define strategies to guide the search in order to identify vulnerabilities within the limited time available for testing. Also, in case of an incorrect model, it is important to collect enough counterexamples to get better insight and identify patterns that trigger microarchitectural features in unexpected ways. Moreover, we would like counterexamples to cover as many vulnerable programs as possible in order to avoid missing classes of vulnerabilities.

In this work, we extend the validation process of Scam-V with *observation refinement* (Section 3), a technique for guiding the exploration of the input state space in meaningful directions. Specifically, we refine observational models to capture behaviors that we would like to exclude, essentially adding more fine-grained observations

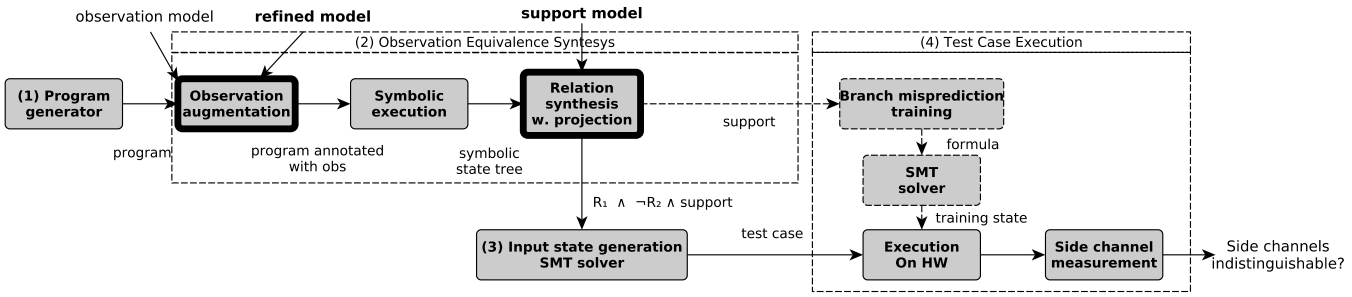

Figure 1: Test-case generation and evaluation with Scam-V. Components with thick borders are those that have been extended in this work. Components with dashed lines only apply to experiments involving speculative execution.

of the system state. States that yield equivalent refined observations can then be ruled out of the search, considerably reducing the size of the search space. In many cases these refined observations are meant to exclude states that are strongly suspected to be indistinguishable or irrelevant to the side channel of interest.

Section 4 presents two interesting applications of observation refinement: validation of a model for cache coloring in the presence of prefetching and validation in the presence of speculative execution. Section 5 presents an implementation of observation refinement for Scam-V (bold boxes in Fig. 1), including key optimizations that make the refinement search more tractable and support for speculative execution (dashed boxes in Fig. 1).

In Section 6 we present a series of conducted experiments. These results show that refinement substantially reduces the number of tests and time needed to find counterexamples (when the model is incorrect). Moreover our tests have produced an unexpected result. They exposed SiSCloak[1], a new vulnerability in Cortex-A53 processors. Finally, we conclude our paper with related work and concluding remarks in Section 7 and Section 8.

2 BACKGROUND

In the following we model the program running on the processor by a transition system $M = \langle S, \rightarrow \rangle$, where S is a set of states and $\rightarrow \subseteq S \times S$ a transition relation. This model reflects the processor architecture and abstracts from low-level behavior of the microarchitecture, such as caches, electric currents, timing, or power consumption.

We use the running example of Fig. 2 to illustrate the main concepts of our work. This program updates **x2** by dereferencing **x0** and, if **x0** is strictly lower than **x1**, it updates **x3** by further dereferencing **x2**.

2.1 Side Channels

Contemporary platforms provide a number of resources, such as caches and energy stored in batteries, that are limited and shared among several processes. While resource sharing is essential, great care must be taken to avoid unintended information channels that might leak secret data. Generally speaking, side channels are paths through which sensitive information can escape and that are not

[1]SIngle SpeCulative LOad AttacK. This vulnerability was responsibly disclosed to ARM by the authors on June 2020. ARM has confirmed that the Cortex-A53 is vulnerable to attacks based on single speculative memory loads.

Running Example	Observation
`ldr x2, [x0]`	load from **x0**
`add x1, x1, 1`	none
`if x0 < x1`	branch on **x0**<**x1**
`  ldr x3, [x2]`	load from **x2**

Figure 2: Example code

present in the abstract architectural model M. Absence of information flows due to side-channels is formalized as a variation of non-interference: two states of model M are *indistinguishable* if and only if a real-world attacker is not able to distinguish executions on real hardware that start from any states of the real system that correspond to the model state.

Most side channel attacks exploit the data cache (e.g. [6, 38, 43, 44, 49]). For instance when **x0** < **x1**, line four of our running example may leak the value of **x2** via the cache, since accessing different addresses can affect different cache lines. An approach to extracting information via the cache is Flush+Reload [48] where: (1) the attacker flushes the desired lines from the cache; (2) the victim executes its process or a fragment of it; (3) the attacker measures the time needed to access the flushed line. Depending on the reload time, the attacker decides whether the line was accessed.

2.2 Observational Models

When analyzing the resilience of software against side channel attacks, a model capturing the channel is required. Unfortunately, for complex microarchitectures it is infeasible to model all the relevant, complex, and intertwined processor features like cache hierarchies, cache replacement policies, branch prediction, as well as bus and memory features. This may require knowledge at a level of detail that is not even public for many processors.

Observational models solve this problem by overapproximating attacker capabilities [7, 19, 36]. An observational model extends the abstract processor model with a set of possible observations O and a transition relation $\rightarrow \subseteq S \times O \times S$. The observations represent the part of the (ISA) processor state that may affect the channel at each transition. For instance, in order to overapproximate the information leakage that may occur in Fig. 2 due to the presence of caches, the processor model may be extended with the observations shown in the right column. Intuitively, these observations capture

that the program execution time depends on the addresses of memory loads and instructions that are executed based on conditional constructs. For the first load instruction, the value of **x0** affects which cache line the processor accesses, which in turn decides whether the load is slow or fast. The addition instruction has no observations, since its execution time is constant and it does not access the cache. Whether or not line four executes depends on the condition **x0** < **x1** and it affects execution time. Line four contains another load instruction, which also affects and is affected by the cache state.

Definition 1 (Observational equivalence). *States $s, s' \in S$ are* observationally equivalent*, denoted $s \sim_M s'$, iff for every possible trace $s_1 \xrightarrow{o_1} s_1^1 \ldots \xrightarrow{o_1^n} s_1^n$ of M there is a trace $s_2 \xrightarrow{o_2} s_2^1 \ldots \xrightarrow{o_2^n} s_2^n$ such that $[o_1, \ldots, o_1^n] = [o_2, \ldots, o_2^n]$.*

For instance, all states in $\mathbf{false}_v = \{s \mid \mathbf{x0} = v \wedge \neg(\mathbf{x0} < \mathbf{x1} + 1)\}$ lead to the sequence of observations [load v, none, branch **false**] and are observationally equivalent. Also, all states in $\mathbf{true}_{v,v'} = \{s \mid \mathbf{x0} = v \wedge \mathbf{x0} < \mathbf{x1} + 1 \wedge \mathtt{mem}[\mathbf{x0}] = v'\}$ lead to the sequence of equivalent observations [load v, none, branch **true**, load v'].

Definition 2 (Soundness). *An observational model M is* sound *if observationally equivalent states (i.e., $s_1 \sim_M s_2$) lead to executions that are* indistinguishable *for an attacker on real hardware.*

Observational equivalence is, in principle, different from *indistinguishability*. Unsound models can overlook some information flows, i.e., two states that are observationally equivalent according to the model can lead to executions on the real hardware that can be distinguished by measuring a side channel. Sound observational models can be used as reliable foundations for side-channel analyses in terms of non-interference [23, 24], where we consider a program secure if given two states that have the same public variables it produces the same observations.

2.3 Validation of Observational Models

Scam-V validates observation models (see Fig. 1) by generating random binary programs, and pairs of input states (called test cases) per program. The programs are first augmented with annotations that indicate observations for each statement, as shown on the right column in Fig. 2. Then, to generate test cases for the program (i.e., two observationally equivalent states), Scam-V synthesizes a relation which characterizes the space of observationally equivalent states for the program according to the model under validation. The generated test cases, together with the test program, are then executed and Scam-V evaluates indistinguishability of test cases by measuring the side channel. Each distinguishing test case on the hardware is a counterexample to the soundness of the model.

Scam-V supports multiple architectures[2] by translating binary programs to an intermediate language. Supporting a new architecture requires implementing a new binary translator and the bare-metal code that executes experiments on the new platform. While we focus on cache side channels here, the tool supports several types of side channels. To analyze a new channel (e.g., caused by TLB state, variable time arithmetic operations, variable time DRAM accesses, or power consumption) it is necessary to implement a new module for augmenting input programs with the relevant observations (which represent the expected ISA state leakage) and to extend the test case executor to measure the channel.

To synthesize the equivalence relation, Scam-V relies on symbolic execution of the annotated program. This means executing the program with symbolic inputs, exploring all possible execution paths, and collecting the execution effects along each path in symbolic expressions in a terminating symbolic state $\sigma \in \Sigma$ per path. A further adaption of standard symbolic execution allows handling the annotated observations. Each symbolic state consists of the path condition $\mathbf{p}_\sigma$ (expressing which states take the corresponding execution path) and the list of symbolic expressions $\mathbf{l}_\sigma$ that correspond to the encountered observations (accounting for effects of assignments along the path). For instance, we obtain two terminating states for the example in Fig. 2. The state that results when the condition in the third line holds has the path condition **x0** < **x1** + 1 and symbolic observation list [load **x0**, none, branch **true**, load mem(**x0**)]. Note that the last observation in the list shows the propagation of the symbol for the initial value of **x0** through the last line of the program, which leads to observation of the value located at **x0** in the initial memory mem. The other state has the path condition $\neg(\mathbf{x0} < \mathbf{x1} + 1)$ and the symbolic observation list [load **x0**, none, branch **false**].

The relation $\sim_M$ is synthesized using the result of symbolic execution. The formula considers each pair of execution paths and ensures equal observation sequences:

$$s_1 \sim_M s_2 \triangleq \bigwedge_{(\sigma_1,\sigma_2)\in\Sigma\times\Sigma} \left(\begin{array}{l} \mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2) \\ \quad \Rightarrow \mathbf{l}_{\sigma_1}(s_1) = \mathbf{l}_{\sigma_2}(s_2) \end{array} \right) \quad (1)$$

For the example in Fig. 2, let $\__1$ and $\__2$ be variables of s_1 and s_2 respectively. Instantiating Equation 1 for this example generates the formula:

$$\begin{array}{l} (\mathbf{x0}_1 < \mathbf{x1}_1 + 1 \wedge \mathbf{x0}_2 < \mathbf{x1}_2 + 1) \\ \qquad \Rightarrow (\mathbf{x0}_1 = \mathbf{x0}_2 \wedge \mathtt{mem}_1[\mathbf{x0}_1] = \mathtt{mem}_2[\mathbf{x0}_2]) \wedge \\ (\neg(\mathbf{x0}_1 < \mathbf{x1}_1 + 1) \wedge \neg(\mathbf{x0}_2 < \mathbf{x1}_2 + 1)) \Rightarrow (\mathbf{x0}_1 = \mathbf{x0}_2) \wedge \\ (\mathbf{x0}_1 < \mathbf{x1}_1 + 1 \wedge \neg(\mathbf{x0}_2 < \mathbf{x1}_2 + 1)) \Rightarrow \mathbf{false} \wedge \\ (\neg(\mathbf{x0}_1 < \mathbf{x1}_1 + 1) \wedge \mathbf{x0}_2 < \mathbf{x1}_2 + 1) \Rightarrow \mathbf{false} \end{array}$$

Note that the right hand side of the implication in each conjunct $(\mathbf{l}_{\sigma_1}(s_1) = \mathbf{l}_{\sigma_2}(s_2))$ has been partially evaluated for brevity. For example, the last two are trivially **false** because the observation lists do not agree in length since the two states take different execution paths.

inline]RG: If we have space we can extend this

3 OBSERVATION REFINEMENT

For a program p, an observational model M_1 induces a partition of the input states into observation equivalence classes. For instance, the observations of the program in Fig. 2 lead to the equivalence classes $\mathbf{false}_v$ and $\mathbf{true}_{v,v'}$ for $v, v' \in \{0 \ldots 2^{64} - 1\}$ exemplified after Definition 1 (see Fig. 3.a). In order to validate a model, we should test "relevant" pairs of states that belong to the same class. The search of these pairs faces two challenges.

The first challenge is to identify a notion of test coverage that can drive the tests. The equivalence classes of the model under validation are seldom suitable for this purpose, since their number is usually large. Blindly testing many different classes may simply

[2]Currently ARMv8, CortexM0, and RISC-V

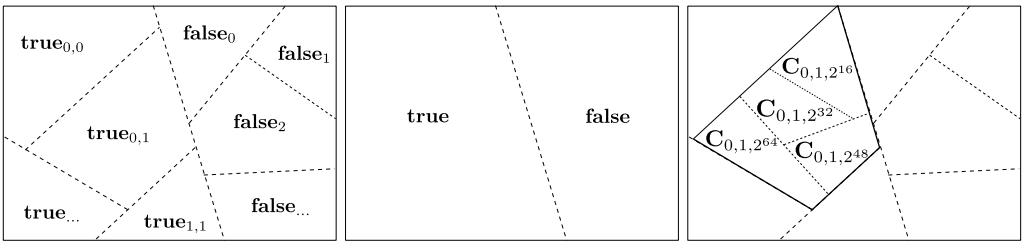

Figure 3: Input state space partitioned via observation equivalence classes

expose the same hardware behavior. For instance in Fig. 2, simply iterating over every equivalence class $\mathbf{false}_v$ requires 2^{64} tests, all probably exercising the same hardware features. For this reason, it is usually convenient to define supporting observational models that induce coarser equivalence classes and use these classes for a notion of test coverage, provided that they can be easily enumerated. This consists in taking successive test cases from different partitions, ensuring that we systematically explore the entire space. For instance, we may desire to guarantee that the tests for Fig. 2 cover all possible paths, which are just two. In this case the supporting observational model only observes the program counter for every instruction and produces the equivalence classes $\mathbf{false} = \{s \mid \neg(\mathtt{x0} < \mathtt{x1} + 1)\}$ and $\mathbf{true} = \{s \mid \mathtt{x0} < \mathtt{x1} + 1\}$, which are depicted in Fig. 3.b.

The second challenge is to identify interesting pairs of states within an equivalence class, since the size of each class can be enormous. There are several states within an equivalence class that cannot have different effects on the side channels. For instance for a given v, testing every pair of states in $\mathbf{false}_v$ that only differ for the value of register **x1**, in addition to being infeasible, is unlikely to find any distinguishable execution: the states lead the program to access exactly the same memory locations and to follow the same execution path. The same should happen (**but it does not**, see Section 6.4) for pairs of $\mathbf{false}_v$ that only differ in the contents of memory at address **x0**, since if the condition is not met then the result of the first load does not affect any memory access. Similar arguments hold for every register or memory location that does not affect the execution or that cannot affect the side channel: e.g., registers **x2** and **x3** for the example. This search is seldom successful without proper guidance.

We use properties of observation equivalence to guide our search in meaningful directions. Intuitively, for every architecture there is an observational model that is trivially sound: the model that observes the complete ISA state after each instruction. For this model the observational equivalence relation is the identity and its soundness is guaranteed vacuously as the identity relation implies absence of any leakage – there is nothing that could be leaked in this case. At the opposite side we have the model that produces no observations, that considers all states as equivalent, and whose observational equivalence relation is the whole cartesian product of the input spaces. Given two models M_1 and M_2, we say that M_2 is *more-restrictive* than M_1 if $\sim_p^{M_2} \subseteq \sim_p^{M_1}$ for every program p, i.e. if observational equivalence w.r.t. M_2 entails observational equivalence w.r.t. M_1.

We will use this to drive model validation. Intuitively, if the model under validation M_1 is unsound then there must be a *refined* (i.e., more restrictive) model that is sound. In order to drive test-case generation for validating M_1, we consider a refined model M_2, which has to be chosen so as to capture the observations that might arise from the side channel under scrutiny. The model M_2 enables us to repartition each equivalence class C_i of M_1 into M_2-observation equivalence classes $C_i^1 \ldots C_i^n$(see Fig. 3.c). For instance for the example in Fig. 2, we may observe the highest two bits of **x1** in lines 2 for checking if time needed for additions depends on the size of the arguments. In this case, the class $\mathbf{true}_{0,1}$ is repartitioned into four classes $C_{0,1,2^{16*i}} = \{s \mid \mathtt{x0} = v \wedge \mathtt{x0} < \mathtt{x1} + 1 \wedge \mathtt{mem}[\mathtt{x0}] = v' \wedge 2^{16*i-1} \leq \mathtt{x1} < 2^{16*i}\}$.

If M_1 is unsound and M_2 adds useful observations then there must be two states s_1 and s_2 that are in the same M_1-equivalence class C_i, but that are in the two different M_2-equivalence classes C_i^j and C_i^k and are distinguishable. For this reason we guide our exploration of the state space using the relation $s_1 \sim^{M_1 \wedge \neg M_2} s_2$, which is defined as $s_1 \sim^{M_1} s_2$ and $s_1 \not\sim^{M_2} s_2$. This ensures that the generated states are "interesting" with respect to M_2. On the other hand, the inability of finding such pairs of states that are distinguishable on the hardware would suggest that M_2 does not add useful observations to M_1. Once we fix M_1 and M_2, and assuming we start from program P, refinement-guided exploration proceeds as follows:

(1) **Add observations.** Program P is instrumented with observations for models M_1 and M_2, yielding programs P_1 and P_2, respectively.
(2) **Symbolic execution.** Each of P_1 and P_2 are symbolically executed, yielding symbolic states σ_1 and σ_2. These states are data structures that represent all possible program paths as well as the list of symbolic observations produced in each path.
(3) **Relation synthesis.** Observational equivalence relations $\sim^{M_1}$ and $\sim^{M_2}$ are synthesized. These are constructed such that for all states s_1, s_2 we have that $s_1 \sim^{M_i} s_2$ only if s_1 and s_2 produce identical observation lists in M_i, i.e. $\mathbf{l}_{\sigma_i}(s_1) = \mathbf{l}_{\sigma_i}(s_2)$.
(4) **Generate test case.** When generating test cases, we require that the input states s_1 and s_2 be such that $s_1 \sim^{M_1} s_2$ and $s_1 \not\sim^{M_2} s_2$, that is, they should be observationally equivalent in M_1 but distinct in M_2. The refined model provides guidance by allowing us to skip test cases that are not relevant to the model under validation, thereby improving the ability to find counterexamples.

For performance reasons, in our tool we do not generate the two separate relations explicitly in this manner, but we introduce an optimization that is functionally identical to this process (see Section 5).

4 APPLICATIONS

4.1 Supporting models for coverage

We present two simple models that are useful to identify coverage measures for models of cache side-channels.

4.1.1 Path enumeration: M_{pc}. A basic application of support models is to ensure path coverage, i.e., to require that all execution paths of a program be explored. For this, the supporting model M_{pc} observes the program counter for every instruction. Let Σ be the set of states resulting from the symbolic execution, then this model produces $|\Sigma|^2$ equivalence classes, where $C^{pc}_{\sigma_1,\sigma_2 \in \Sigma\times\Sigma} = \{\mathbf{p}_{\sigma_1}(s_1) \wedge \mathbf{p}_{\sigma_2}(s_2)\}$.

4.1.2 Cache line enumeration: M_{line}. For cache side channels we may desire to guarantee that the addresses of the tests cover all possible cache lines of the real hardware, which may be a few hundred. In this case, the supporting observational model M_{line} only observes a few bits (the ones identifying the cache set index) of the accessed addresses. This approach reflects the intuition that the hardware behavior could be different for some cache lines. Let L be the number of cache set indexes, for our running example this model produces L^2 equivalence classes, where $\mathbf{C}^{line}_{l_1,l_2} = \{s \mid line(\mathbf{x0}) = l_1 \wedge (\mathbf{x0} < \mathbf{x1} + 1) \Rightarrow line(\mathtt{mem}[\mathbf{x0}]) = l_2\}$.

4.2 Models to validate and refinements

We present application that exposed behaviors of hardware that invalidate observational models: i.e., prefetching for models of cache partitioning, and branch prediction for models that do not take into account speculation.

4.2.1 Cache Partitioning (M_{part}) vs prefetching. On many processors, the replacement policy for a cache set does not depend on previous accesses performed to other cache sets. The resulting isolation among cache sets leads to the development of an efficient countermeasure against access-driven attacks: cache coloring [22, 41]. This consists in partitioning the cache sets into multiple regions and ensuring that memory pages accessible by the attacker are mapped to a specific region of the cache. In this case, accesses to other regions do not affect the state of cache sets that an attacker can examine.

We dub the observational model for cache coloring M_{part}. In this model, observations include the address of every memory access within the attacker-accessible region. For instance, the model would annotate the observations **if** AR(**x0**) then **x0 else** none and **if** AR(**x2**) then **x2 else** none for lines 1 and 5 of the example 2, where AR(_) is a predicate that identifies the addresses belonging to the attacker accessible region.

As it has been shown in previous work [37], automatic cache prefetching can invalidate this partitioned-cache observational model. In fact, a memory access to a set close to the boundary of the two cache regions can trigger prefetching of a stride pattern that may include sets in the attacker-observable region. For instance in example 2, let the cache consist of 128 lines of 64 bytes and let the highest 64 lines be accessible by the attacker. In this case $\mathsf{AR}(v) \triangleq 64 \leq line(v) \leq 127$. State $s_1 = \{\mathbf{x0} = 0, \mathbf{x1} = 1, \mathtt{mem}[\mathbf{x0}] = 0\}$ satisfies the branch condition and accesses cache line 0 twice. Similarly, $s_2 = \{\mathbf{x0} = 62 * 64, \mathbf{x1} = 62 * 64 + 1, \mathtt{mem}[\mathbf{x0}] = 63 * 64\}$ satisfies the branch condition and accesses cache lines 62 and 63. Since none of the memory accesses satisfies $\mathsf{AR}(v)$, the observations for these two states are [none, branch **true**, none], hence the states are observationally equivalent according to M_{part}. However, the stride of two accesses in consecutive cache lines in s_2 can trigger an automatic cache prefetcher to load the memory block that starts at $64 * 64$ and that is mapped to cache line 64, affecting that state of the portion of the cache that is accessible by the attacker.

Finding a pair of input states that invalidates this model is rare without adequate guidance. Intuitively, a counterexample should lead to memory accesses in different cache sets that are inaccessible by the attacker, otherwise prefetching would not cause distinguishable access patterns in the accessible region. However, the observational equivalence relation deriving from M_{part} places no constraints that would force the input states to involve addresses on different inaccessible cache sets. Specifically, for the example the first part of the observational equivalence relation is:

$$\left(\begin{array}{l}(\mathbf{x0}_1 < \mathbf{x1}_1 + 1 \wedge \mathbf{x0}_2 < \mathbf{x1}_2 + 1)\\ \Rightarrow (\mathsf{AR}(\mathbf{x0}_1) = \mathsf{AR}(\mathbf{x0}_2)) \wedge (\mathsf{AR}(\mathbf{x0}_1) \Rightarrow (\mathbf{x0}_1 = \mathbf{x0}_2))\end{array}\right) \wedge$$

$$\left(\begin{array}{l}(\mathsf{AR}(\mathtt{mem}_1[\mathbf{x0}_1]) = \mathsf{AR}(\mathtt{mem}_2[\mathbf{x0}_2])) \wedge (\mathsf{AR}(\mathtt{mem}_1[\mathbf{x0}_1])\\ \Rightarrow (\mathtt{mem}_1[\mathbf{x0}_1] = \mathtt{mem}_1[\mathbf{x0}_2]))\end{array}\right)$$

Note that for the cases in which the addresses are outside the accessible region, the relation imposes no extra constraints. Therefore, a naive exploration of the search space is likely to yield many redundant test cases.

Observation refinement allows us to guide this search toward suitable counterexamples. The refined model $M_{part'}$ would observe the addresses in all memory operations, independently of the attacker accessibility. By following the algorithm in Section 3, we require our test cases to differ in their refined observations. The corresponding formula would include a constraint of the form:

$$(\mathbf{x0}_1 < \mathbf{x1}_1 + 1 \wedge \mathbf{x0}_2 < \mathbf{x1}_2 + 1) \Rightarrow \left(\begin{array}{l}(\neg\mathsf{AR}(\mathbf{x0}_1) \Rightarrow (\mathbf{x0}_1 \neq \mathbf{x0}_2)) \ \wedge\\ \neg\mathsf{AR}(\mathtt{mem}_1[\mathbf{x0}_1]) \Rightarrow (\mathtt{mem}_1[\mathbf{x0}_1] \neq \mathtt{mem}_1[\mathbf{x0}_2])\end{array}\right)$$

This adds the constraint that the cache set indices for addresses outside the observable region should be different, which is exactly what we need to guide the search. It can be useful to use a further supporting model M_{line} that simply observes the set index of all memory accesses. If the cache consists of 64 lines, M_{line} generates 64^{2*n} equivalence classes, where n is the number of memory accesses. Enumerating these classes guarantees coverage of cache line set, which could be useful if the behavior of the system is different for different cache lines: i.e. across the border of the cache partitions. In case that the number of memory accesses n is large, one can use a coarser supporting model, which observes only a few bits of the cache set index.

4.2.2 Constant time programming (M_{ct}) vs branch prediction. A widely used strategy to prevent cache side channels is to respect the "constant time" policy: memory accesses and branches should only be dependant of public information. This policy relyes on the assumption that cache state is only affected by the addresses of memory accesses. This assumption can be formalized by the observational model M_{ct}, which observes the program counter of every instruction and every address being accessed. This is equivalent to the model in our running example, in which for simplicity we observe the boolean value of each branch condition. In this example, the branch condition fully determines the control flow of the program, so it is not necessary to observe the program counter at every step.

The *Spectre* family of attacks showed that speculative execution and its microarchitectural effects can be used to leak secret data from the L1 data cache [31] in programs that are traditionally considered secure. The original Spectre-v1 exploited branch prediction. In this case the processor uses *Pattern History Tables* (PHT) to record patterns of past executions of conditional branches, i.e., whether the **true** or the **false** branch was executed, and then use it to predict the outcome of that branch. In case of misprediction then the work that was executed speculatively is discarded and the processor is redirected to execute the correct instruction path. However, mispredictions are not completely transparent, since speculatively executed instructions can affect the microarchitectural state, such as state of the data caches.

To demonstrate problems related to speculation we use our running example. We assume that the memory of the victim process is split into two areas, where the high addresses contain confidential information and the low addresses contain public information. Moreover, we assume that the highest address of public data is stored in register **x1**, and that the attacker controls the input **x0**. The program would usually be considered free of side channels since the memory accesses and branches are only dependant of public information: the branch condition and the first load is addressed by the attacker input **x0** and the second load is executed only if the address has been loaded from the public memory area. However, if the microarchitecture supports speculative execution, an attacker can fool the prediction mechanism by first supplying low values of **x0** and then a value that points to the confidential memory area. This causes the CPU to access sensitive data, which can then be leaked by the subsequent access to **x2**.

In order to check if a microarchitecture is not affected by Spectre-v1-like vulnerabilities we should validate M_{ct}. Note that a counterexample due to the above misprediction would consist of two states satisfying $\mathbf{x0} \geq \mathbf{x1} + 1$ (i.e., taking the same branch), having same **x0** (i.e., accessing the same address before branching), but different mem[**x0**] (i.e., accessing different addresses in the misspeculated branch). Unfortunately, the observational equivalence relation for M_{ct} does not provide enough guidance to find these states, since for states that do not satisfy the if condition the relation imposes no extra constraints on mem[**x0**].

We address this problem by relying on a refined observational model (M_{spec}) that observes speculative memory accesses. This requires speculative symbolic execution, which in case of speculative branch execution (i.e., the Spectre-v1 type) can be implemented on top of the standard symbolic execution, by transforming the program and inlining shadow statements that represent (wrongly) speculated instructions. First, our instrumentation constructs the control-flow graph of the program. Then, for every pair of mutually exclusive branches A and B in the control-flow graph, we prepend to the statements in A the shadow statements from B and vice versa. The shadow statements operate on a *shadow state* represented by variables marked with * such as $\mathbf{x1}^\star$ or $\mathbf{x2}^\star$. This state is essentially a copy of the real state at the time of branch prediction. It represents the transient state, i.e. the state of the CPU while it is performing speculative execution, and it allows us to compute transient observations without interfering with the actual computations in the branch. Fig. 4 shows instrumentation in action using our running example. Since in this case the else branch was initially empty, the instrumentation of the if branch has no effect.

Running Example	Observation
`ldr x2, [x0]`	load from **x0**
`add x1, x1, 1`	none
`if x0 < x1`	branch on **x0**<**x1**
`  ldr x3, [x2]`	load from **x2**
`else`	
`  x2* = x2`	none
`  x3* = x3`	none
`  ldr x3*, [x2*]`	load from **x2***

Figure 4: The running example instrumented via M_{spec}

Using this model to guide our search, we end up with a relation that includes useful constraints for cases when the branch is *not* taken, such as

$$\neg(\mathbf{x0}_1 < \mathbf{x1}_1 + 1) \wedge \neg(\mathbf{x0}_2 < \mathbf{x1}_2 + 1) \Rightarrow (\mathbf{x0}_1 = \mathbf{x0}_2) \ \wedge$$
$$\neg(\mathbf{x0}_1 < \mathbf{x1}_1 + 1) \wedge \neg(\mathbf{x0}_2 < \mathbf{x1}_2 + 1) \Rightarrow$$
$$((\mathbf{x0}_1 \neq \mathbf{x0}_2) \vee (\mathtt{mem}[\mathbf{x0}_1] \neq \mathtt{mem}[\mathbf{x0}_2]))$$

This means the states must have the same ISA behavior, but they must access different addresses speculatively.

5 IMPLEMENTATION

Scam-V [3] is part of the binary analysis platform HolBA [32], which transpiles binary code to an architecture-independent binary intermediate representation with explicit observation statements. In this work, we extended and modified the Scam-V pipeline implementation to support observation refinement. Fig. 1 illustrates the updated architecture to generate test cases for a given observation model. Scam-V runs this process for a given number of programs, and for each program it generates a given number of test cases, as configured by the user. For performance reasons, while generating test cases for the same program the results of symbolic execution remain cached, and only the latter phases are executed per test case. The main changes to the pipeline with respect to previous work are in the observation augmentation phase and the relation synthesis phase, and the addition of a branch misprediction training module. The rest of this section discusses each of these extensions in detail.

[3] Scam-V is available at https://github.com/kth-step/HolBA.

5.1 Observation augmentation

Observation refinement involves two observational models: the model under validation M_1 and the refined model M_2 for guiding the state space exploration.

A naive implementation of observation refinement would require a large part of the Scam-V pipeline, including symbolic execution and relation synthesis, to run twice per program – once for each model. The resulting relations would then be used in the rest of the pipeline to generate test cases. While conceptually simple, this process would be prohibitively expensive.

We implement observation refinement with an optimization that allows us to run symbolic execution only once per program. The optimization relies on the assumption that there exists a projection function to obtain the symbolic observation list of the model under validation from the observation list of the refined model.

Projection Assumption. Let M_2 be more restrictive than M_1. For all programs P and symbolic paths $\mathbf{p}$ of P, if

- symbolic execution of P with M_2 observations on path $\mathbf{p}$ yields observation list $\mathbf{l}^{M_2}$; and
- symbolic execution of P with M_1 observations on path $\mathbf{p}$ yields observation list $\mathbf{l}^{M_1}$;

then there exists a function π such that $\pi(\mathbf{l}^{M_2}) = \mathbf{l}^{M_1}$.

This assumption holds for our two proposed applications. For simplifying the pipeline we enrich the observation type with a tag that indicates if an observation is exclusively from M_2 or from both. The projection function then simply removes all the observations with the wrong tag. For example, for validation against speculative leaks, these tags would distinguish between observations in real and transient branches. Under this assumption, we can run the Scam-V pipeline only on M_2. Then, when synthesizing the observational equivalence relation, we use the projection π to distinguish between the base observations (those that are required to be the same on both input states) and the refined ones (those that are required to be different on the input states).

The observation augmentation for M_{spec} implements the instrumentation described in Section 4.2.2, which reuses the existing symbolic execution engine to compute speculative observations. Our implementation allows us to bound the number and type of instructions that can be speculated. This enables us to model executions with both partial and without mis-speculation: the model M_{pc} (i.e. without instrumentation) formalizes executions without mis-speculation, model M_{spec_1} of Section 6.5 corresponds to executing up to the first load (included) of the mis-speculated branch, and M_{spec} corresponds to execution of the whole mis-speculated branch.

5.2 Relation synthesis with projection

A test case for a program P is a pair of initial states s_1 and s_2 such that P produces the same observations in model M_1 when executed from either state but different observations in refined model M_2 (see Section 2.3). We synthesize such a relation for each generated program, using the tags in the combined observations to project M_1 and M_2 observations, as they have to be treated differently. Moreover, the relation includes any additional constraints that arise from a support model to guide state exploration, which usually includes at least path coverage. Finally, Scam-V queries an SMT solver, namely Z3 [18], for a model of this relation to generate test-cases. Each test case corresponds to a concrete valuation of the registers and memory locations used by program P in the states s_1 and s_2.

5.3 Branch Misprediction Training

To validate resilience against Spectre-like attacks we extended the pipeline to produce states that train the branch predictor to mispredict. Notice that while validating M_{pc}, two states s_2 and s_2 that belong to the same equivalence class follow the same execution path because the model observes the program counter. Therefore, both states satisfy the same symbolic path condition $\mathbf{p}$ (represented by the formula support in Fig. 1). To generate a further state s_t that takes a different path, it suffices to find a satisfying assignment for a path condition $\mathbf{p}' \neq \mathbf{p}$ from the symbolic execution tree, which we do using the SMT solver.

When executing the experiment, in order to train the branch predictor we execute the program multiple times using input s_t. Then, we execute and measure the execution with inputs s_1 and s_2. Due to the training, the branch will be mispredicted, and the wrong branch will be executed speculatively. Since the states have been chosen to be observationally equivalent on the correct branch and observationally distinct on the other branch, any distinguishable behaviors would invalidate the observational model and point to a speculative leak. Notice that we do not test the case when only one of the two executions occurs in the mispredicted branch because in this case the effects on the cache are supposed to be different and do not represent counterexamples to the observational model.

5.4 Additional changes to the pipeline

The experiments conducted in Section 6 required several further extensions to Scam-V. The original Scam-V pipeline did not handle programs where observations depend on previous loads. For these programs, it is insufficient to generate only register assignments to create the initial states (as it was the case in the previous version of Scam-V) and memory content should be specified. In order to support this type of programs, we have extended the SMT interface and the evaluation platform to properly translate encodings of memory contents between Scam-V and Z3, and for initializing the initial memory of experiments on the evaluation platform.

We introduced new program generators for the templates in Fig. 5. We reused the existing infrastructure, which allows to define monadic generators and follows a grammar-driven approach in the style of QuickCheck [17]. The generators are implemented in SML and can be composed to generate more complex programs to fit different attack scenarios.

Scam-V uses M_{pc} of Section 4.1.1 as supporting model to cover of all execution paths. This allows us to optimize the synthesis of the equivalence relation. Instead of generating the whole relation $s_1 \sim s_2$, we can split the relation into one formula per pair of execution paths that s_1 and s_2 can take. These relations are smaller than the full observation equivalence, since each of them covers only one of the conjuncts of Eq. 1. Scam-V explores these relations in round-robin fashion, systematically covering pairs of execution paths.

6 CASE STUDIES AND RESULTS

To demonstrate the effectiveness of observation refinement and supporting models for coverage, we conducted experiments on Raspberry Pi 3, which is a widely available ARMv8 embedded system. We first tried to validate a model for cache partitioning. Unsurprisingly, this model is unsound due to prefetching. The second set of experiments attempts the validation of a standard model for constant time programming. These experiments lead to a surprising result: even if the processor is claimed to be immune against attacks based on speculative execution, we were able to identify speculative leakage. This allowed us to formulate a new variant of the original Spectre attack [31].

6.1 Evaluation Platform

Our experiments use Raspberry Pi 3 boards, which use a Cortex-A53 processor. The processor is 8-stage pipelined with a 2-way superscalar and in-order execution pipeline. It also supports speculative execution based on control flow prediction for performance reasons. Using such a technique the processor relies on a *branch prediction engine* to predict the control flow of a running program and determine operations which are needed to execute, rather than waiting for all branch instructions to resolve. Furthermore the processor uses a cache prefetcher to load instructions or data into respective caches before it is actually needed. The prefetcher is activated when a stride of at least three loads (default setting) accesses addresses that are equidistant.

The board also has ARM TrustZone. To execute the generated experiments, Scam-V uses a *platform module* which runs in TrustZone. The module configures page tables to setup *cacheable* and *uncacheable* memory, clears the cache before every execution of the program, inserts memory barriers around the experiment code, and inspects the cache state after execution. Executing the platform module inside TrustZone allows us to use privileged debug instructions to obtain the cache state directly for comparison after experiment execution.

In a more realistic setting, an attacker can use the *performance monitor counter* (PMC) for timing analysis and derive exploit code from identified vulnerabilities. The PMC consists of a number of special-purpose registers built into the processor which track the counts of specific hardware-related activities like the processor cycles and cache hits.

In order to guarantee repeatability of our results we execute each experiment 10 times and check for discrepancies in the final state of the data cache. Experiments not giving the same results in all runs are classified as *inconclusive* and excluded from further analysis.

6.2 Validation of M_{part}

To study cache partitioning as in Section 4.2.1 (M_{part}), we used the Stride Template in Fig. 5 to generate simple programs that may trigger the automatic cache prefetcher: a stride of three to five loads starting from the base address r_0 and distance v. We use ? to show that an instruction is optional and we annotate the constraints used for the allocation of registers. Notice that the length of a cache line of Cortex-A53 is 64 bytes; therefore, the template ensures that the accesses lie in different cache set indexes. The framework instantiates this template by randomly assigning registers to $r_0, \ldots, r_5$, ensuring that the register chosen for r_0 is different than the registers chosen for $r_1, \ldots, r_5$, and choosing a constant v. In order to steer test case generation we used M_{line} for coverage and $M_{part'}$ for refinement.

Table 1 reports the results of 450 generated test programs for validation of M_{part} model. The data cache of Cortex-A53 has 128 set indexes. In these experiments we assume that the highest 67 indexes are accessible by the attacker: i.e., $\mathtt{AR}(v) \triangleq 61 \leq line(v) \leq 127$. Our results show that prefetching violates cache partitioning and that observational refinement increases the probability of finding a counterexample by a factor of 20 and halves the Time To first Counterexample (i.e., the time needed to invalidate the model). Moreover, refinement has no substantial impact on time needed to generate input states and time to execute the experiments.

Since cache partitioning is usually implemented via virtual memory, we repeated our experiments assuming that the attacker-accessible memory is page aligned. One page of memory in Cortex-A53 spawns 64 lines, hence we assumed the highest 64 cache set indexes to be accessible: i.e., $\mathtt{AR}(v) \triangleq 64 \leq line(v) \leq 127$. Interestingly, neither the experiments with nor without refinement have discovered a counterexample. This suggests that prefetching in Cortex-A53 stops at page boundary and that cache partitioning may be secure in the presence of prefetching if the attacker region is page aligned. Clearly, since our validation is based on testing, the absence of counterexamples does not guarantee this property. However, since refinement has proven to give more direction to the search of experiments, we have more confidence in the conclusion we draw from the experiments with refinement.

6.3 Validation of M_{ct}

Fig. 6 illustrates the original Spectre-PHT [31], where the victim uses two arrays `A` and `B` that start at address `#A` and `#B` respectively. The arrays do not contain confidential data, and we assume every element of `A` is a valid index into the array `B`. We also assume that the attacker controls the value of register **`x0`** and that the size of the array `A` is stored at address `#A-size`. This program is usually considered secure as it ensures that **`x0`** lies within the bounds of `#A` and the three memory accesses are only dependent on public information: the location `#A-size`, the locations of `#A` and `#B`, the attacker input `R0`, and `#A[R0]`. In fact, the program respects the standard constant-time policy [8, 11]. However, in the presence of speculation this program may be insecure. Misprediction can cause an out-of-bounds memory read from address `#A+x0` (when index **`x0`** is greater than the size of `A`) that reads sensitive data, which is later used as index for a second memory read from `#B+x2`.

Some microarchitectures, including Cortex-A53, have been previously claimed to be immune to these types of vulnerabilities as they allow speculative fetching but limited speculative or out-of-order execution of the fetched instructions. The informal argument was that mispredictions cannot cause buffer overreads or leave any footprint on the cache due to restricted speculative loads.

In order to validate this claim for ARM Cortex-A53 we have conducted experiments using the observation models M_{ct} of Section 4.2.2, which consider speculative execution to be completely transparent and constant-time programs to be side-channel free.

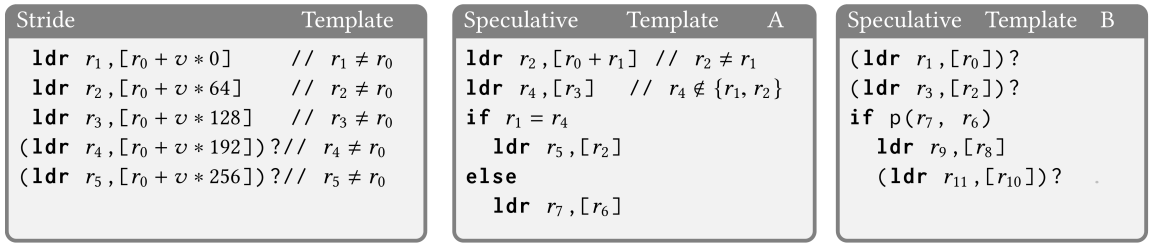

Figure 5: Template (in pseudo-code) for M_{part} and M_{ct}.

Model	M_{part}		M_{part} Page Aligned		M_{ct} Template A		M_{ct} Template B	
Refinement	No	$M_{part'}$	No	$M_{part'}$	No	M_{spec}	No	M_{spec}
Coverage	M_{pc}	M_{pc} & M_{line}	M_{pc}	M_{pc} & M_{line}	M_{pc}	M_{pc}	M_{pc}	M_{pc}
Programs	450	450	425	425	655	652	942	941
Prog. w. Count.	21	89	0	0	6	626	0	498
Experiments	13752	18000	12860	17000	26200	25737	37680	37640
- Counterexample	21	447	0	0	6	12462	0	4838
- Inconclusive	1096	4709	612	4245	5	506	2	352
- Avg. Gen. time (s)	4.3	1.8	8.5	4.6	4.7	3.1	2.6	5.0
- Avg. Exe. time (s)	3.3	3.4	3.3	3.3	10.5	10.3	10.7	10.7
- T.T.C. (s)	8892	2070	-	-	102600	13	-	681

Table 1: Result of the experiments

We used Template A in Fig. 5 to generate simple programs [4] that may lead to wrongly-predicted memory accesses. Notice that for states such that $r_1 \neq r_4$ the program does not use the value of r_2 in any memory access. Therefore, two of these states should be indistinguishable if they only differ for the content of memory at $r_0 + r_1$. The framework instantiates this template by randomly assigning registers to r_i while satisfying the side constraints (i.e., $r_2 \neq r_1$ and $r_4 \notin \{r_1, r_2\}$). To steer test case generation we used M_{pc} for coverage and M_{spec} for refinement.

Table 1 reports the results for 655 generated test programs. Execution of experiments is more costly w.r.t. M_{part} since we must run multiple times the same program to train the branch predictor to mispredict the outcome of the branch. First, despite the existing claims, the counterexamples identified by Scam-V invalidate immunity of Cortex-A53 to speculative leakage and shows that the core is affected by a new variant of Spectre dubbed SiSCloak (see Section 6.4). Second, our results substantiate the effectiveness of observational refinement to drive test generation. When the tests are driven by the refined model, they identify 12462 counterexamples and at least one counterexample for the majority (i.e. 626) of programs. The time needed to find the first counterexample to invalidate the test is substantially smaller with refinement: 13 seconds vs 29 hours (i.e. 102600 seconds) and refinement has no sensible impact on time needed to generate and execute experiments. When tests are generated without any guidance, they can only identify 6 counterexamples. Moreover, these 6 counterexamples cover only a specific subclass of programs, where the generator has selected r_6 to be the same register of either r_0 or r_1. In this case, when $r_1 = r_4$, the transient branch leaks the value of a register (by loading from r_6) that has been already partially leaked (by loading from $r_0 + r_1$). Therefore, executing tests without refinement may lead to the wrong conclusion that speculation introduces only this type of leakage.

We also executed experiments for a more general template, Template B in Fig. 5, which allows from zero to two loads before the branch, one or two loads in the if body, and a comparison predicate p randomly chosen by the program generator. In this case, there is no constraint for register allocation, therefore in some cases a machine register is used for multiple $r_0, \ldots, r_{11}$, and in other cases each register placeholder is assigned to a different machine register. The tests driven by refinement identify 4838 counterexamples and the first one was discovered after 11 minutes. Programs generated with this template are harder to test without refinement, since there are many variables that can be altered without affecting the side channel. In fact, tests without refinement were not able to identify any counterexamples during 138 hours.

6.4 SiSCloak

Our experiments show that Cortex-A53 is capable of executing loads in speculation, but it prevents using the result of a speculated instruction for subsequent operations, probably due to the absence of register renaming and the short CPU pipeline. However, this limitation does not completely prevent speculative leakage.

We use the counterexamples from Fig. 6 to explain SiSCloak. The first counterexample (second column of Fig. 6), which was also previously presented but not experimented in [29], is a variation of Spectre-PHT. Compared to Spectre-PHT, the access ldr x2, [#A+x0] has been anticipated by the programmer or the compiler. In this case, a Cortex-A53 CPU may mispredict the condition x0 < x1 and speculatively access #B+x2, which may contain data that has been read out-of-bound.

[4] For clarity we present the speculative templates as pseudocode, but the real code uses comparison and branch instructions instead of a structured *if* statement.

Spectre-PHT
```
ldr x1,[#A_size]
if(x0 < x1)
  ldr x2,[#A+x0]
  ldr x3,[#B+x2]
```

Example 1
```
ldr x1,[#A_size]
ldr x2,[#A+x0]
if(x0 < x1)
  ldr x3,[#B+x2]
```

Example 2
```
ldr x1,[#A_size]
if(x0 < x1)
  ldr x2,[#A+x0]
  if(x2 & #0x80000000 < #0)
    ldr x3,[#B+x2]
```

Figure 6: Spectre-PHT and SiSCloak counterexamples.

The second counterexample (third column of Fig. 6) demonstrates leakage when the classification of elements of an array is stored in few bits of the array itself. As usual we assume that every element of A is a valid index into the array B. We also assume that the highest bit of each element of A identifies if the element itself is public. Let the attacker control the value of register x0. This program is considered secure at the ISA level as it ensures that x0 always lies within the bounds of #A and that no memory access depends of non-public information. However, Cortex-A53 may mispredict condition x2 & 0x80000000 which leads to considering a confidential element to be public. In this case, the CPU may speculatively access #B+x2, making the confidential data in x2 affect the cache.

To collect evidence of SiSCloak vulnerabilities we initially used ARM TrustZone and privileged debug instructions to directly inspect the cache state. Later we mounted a real attack that recovers bits of **x2** for both programs of Fig. 6 by using a Flush+Reload attack (see Section 2.1) and the cycle counter of core's PMC.

6.5 Scope of speculation in Cortex-A53

Speculation can cause different leakage on different microarchitectures, depending on their pipeline depth, prediction strategy, etc. It is therefore useful to test observational models that are tailored for a specific architecture. In fact, while a model that observes all loads of arbitrarily nested mispredicted branches is probably sound, it is also too coarse and may lead to consider insecure many side-channel free programs, imposing unnecessary fences and overhead.

We first focus on the depth of speculation. For Cortex-A53, the short pipeline and the absence of proper register renaming suggest that only **one** memory load can be done speculatively and that its result cannot be used for further computations. We tested this via an observational model that we dubbed M_{spec_1}. The model includes the observations of M_{ct} and in case of branches the first load of the transient branch. We used both Template-B and Template-C of Fig. 7, where two loads are causally dependent, are executed only if the register comparison succeeds, and can be interleaved by an arithmetic operation. Both M_{spec_1} and M_{spec} instrument this program by adding to the empty else branch a copy of the if-body in that operates over the shadow state. The shadow load ldr $r_6^\star, [r_5^\star + r_3^\star]$ is observable in both model, while the shadow load ldr $r_8^\star, [r_7^\star + r_6^\star]$ is only observable in M_{spec}.

We comment on the results of table in Fig. 7. Similarly to the templates of Section 6.3, the model M_{ct} is unsound for Template-C due to speculative leakage. This is an example of a leaking program that cannot be detected without refinement. Model M_{ct} imposes no constraints on registers used in the branch if the branch is not taken, so Scam-V cannot generate test cases that differ in those registers unless the refinement M_{spec} is enabled. Driving testing via refinement produces higher-quality counterexamples that no amount of unguided testing can uncover. Moreover, we were able to identify counterexamples for M_{spec_1} in Template-B, which indicate that in some circumstances (i.e. when loads have no causal dependencies) Cortex-A53 can execute more than one transient load. Finally, our results support the ARM claim that Cortex-A53 is not vulnerable to Spectre-PHT [5], which is an instance of Template-C. In the original Spectre, the address of the first speculative load is considered to be known (and controlled) by the attacker. This is equivalent to the observations in M_{spec_1}, which uncovered no counterexamples for Template-C.

Also, these experiments confirm that driving testing via refinement is critical to identify counterexamples. We also investigated the source of speculation. In fact, conditional branches are not the only instructions activating prediction. We focused on the case of straight-line speculation [10], where unconditional branches can lead to speculative leakage if the processor transiently executes loads that come after unconditional branches. ARM claims that their processors are not affected by straight-line speculation in case of direct branches [10]. We analysed this claim via Template-D of Fig. 7. Notice that the code after `jmp end` is not supposed to be executed in absence of speculation. In order to stride the tests, we also implemented a new refined model, dubbed $M_{spec'}$, by transforming unconditional branches to tautologically true conditional branches. This enables to reuse the instrumentation of M_{spec} for unconditional branches. Clearly, in absence of unconditional branches, $M_{spec'}$ and M_{spec} are equivalent. For these programs our experiments support the ARM claim that there is no speculative leakage in case of unconditional direct branches. The fact that experiments with refinement were able to find counterexamples for the majority of other programs increases confidence in this conclusion.

7 RELATED WORK

The validation of information flow properties regarding side-channels is the distinguishing feature of Scam-V. In contrast to Scam-V, other works validate models of functional processor behavior by either white-box approaches [12, 20] or black-box testing [21, 30]. Similarly, the recent tool CacheQuery [45] applies testing and measuring execution times on hardware to learn functional behavior of unknown cache replacement policies.

Observational models allow abstracting from the mechanisms used by an attacker such as profiling of side channels. For instance, the program counter security model [36] abstracts time channels when execution time depends on victim control flow. Almeida et.

Template C

```
ldr r1, [r2]
if r3 < r4
  ldr r6, [r5 + r3]
 (lsl r6, r6, #c)?
  ldr r8, [r7 + r6]
```

Template D

```
(ldr r1, [r2]|nop)^n
jmp end
  ldr r2, [r3]
 (ldr r4, [r5])^n
end: ...
```

Model	M_{ct}		M_{spec_1}		M_{ct}
Template	C		C	B	D
Refinement	No	M_{spec}	M_{spec}	M_{spec}	$M_{spec'}$
Programs	8	8	8	915	478
Experiments	8000	8000	8000	36600	47800
Counter.	0	3423	0	206	0
Inconclusive	87	87	87	303	0
Avg. Gen. (s)	8.5	9.0	9.8	7.2	4.0
Avg. Exe. (s)	10.5	12.4	10.8	10.8	4.0
T.T.C. (s)	-	21	-	16268	-

Figure 7: Further experiments

al. [9] present a model that also includes memory addresses accessed by the victim and is used for cache trace-driven attacks [40]. Consumers of such observation models are information flow analysis tools like Ct-verif [7] and CacheAudit [19], which rely on model soundness.

Spectre attacks [14] have exposed the problem of observational model unsoundness. Several works have recently addressed the formal foundations of different forms of speculation to capture Spectre-like vulnerabilities [15, 16, 26, 28, 34]. Thus, sound observational models should approximate the information flows as prescribed by these works. For instance, Guarnieri et al. [28] have shown that an observation model, which observes memory accesses assuming that the CPU always mispredicts, is a valid overapproximation for microarchitectures that support only branch prediction. This corresponds to the refined model M_{spec} that we have used in our experiments.

Other lines of work analyze application code for vulnerabilities directly, like the tool SpecFuzz [39] does for speculation leakage. To generate test cases, it applies fuzzing and prioritizes among possible speculation paths to make its search for speculative memory safety violations tractable. While this approach focuses on saving mitigation overheads for specific code and avoiding overapproximation of attackers for a given architecture, we aim to generalize to the most secure model for a specific processor implementation and thus consider more narrow attack surfaces.

Several prototypes have been developed to reproduce and detect known Spectre-PHT attacks [16, 28, 46]. Checkmate [42] synthesizes proof-of-concept attacks by using models of pipelines with speculative and out-of-order execution. However, existing tools have been designed to reproduce a specific type of attack or to verify resilience of a piece of software against a specific attack. Instead, Scam-V is designed to identify new types of information flows and can potentially discover new vulnerabilities and new information channels. In a similar vein, other works [25, 35, 47] use fuzzing to identify new side channels.

SiSCloak is different from the "straight-line speculation" [10] vulnerability. Straight-line speculation involves the processor speculatively executing the next instructions past an unconditional change in control flow. It is shown [10] that existing ARM processors are affected by this vulnerability for unconditional indirect branches and function returns, but are not affected for unconditional direct branches. We believe both SiSCloak and straight-line speculation are caused by the ability of ARM Cortex-A53 to speculatively issue memory requests when the control flow cannot be statically determined.

8 CONCLUDING REMARKS

We introduced observation refinement, a novel technique to guide state space exploration for validation of observation models. Our notion of refinement is flexible enough to encode techniques like path and line enumeration. We extend Scam-V with our observation refinement technique and show that it can considerably increase the chance of spotting counterexamples. We show the significance of our approach by using Scam-V to validate soundness of a model for cache partitioning and to evaluate the Cortex-A53 processor for vulnerabilities due to speculative execution. Our experiments led to the discovery of SiSCloak, a new class of speculative execution vulnerability that affects the ARM Cortex-A53 processor.

Our approach is not specific to the evaluated models and can be extended to set up experiments for other observational models and microarchitectural features, e.g. other variants of the spectre attack, with relative ease. As a future direction, we plan to investigate techniques to refine unsound observation models to automatically restore their soundness, e.g., by adding state observations or using techniques similar to Eclipse Repairnator [1].

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for helpful discussions of this work. This work has been supported by the TrustFull project financed by the Swedish Foundation for Strategic Research, the KTH CERCES Center for Resilient Critical Infrastructures financed by the Swedish Civil Contingencies Agency, as well as the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

A ARTIFACT APPENDIX

A.1 Abstract

The artifact for this paper is the prototype implementation of the introduced *observation refinement* technique together with the experimental data used in the presented evaluation results. It is embedded in the existing model validation framework, Scam-V, which we have also extended to support our refinement technique.

In our evaluation, we showed that observation refinement is an efficient technique to guide the exploration of the state space and to find vulnerabilities on real-world hardware platforms. Additional to the source code and experimental data, we provide a replication package implemented as a VirtualBox VM to make reproducibility of our results convenient. We also provide an introduction to our framework implementation and detailed descriptions of how to replicate the results presented in the paper.

We provide instructions to support the evaluation of generated test cases on hardware. Alternatively, the artifact can be configured to connect to an existing, compatible experiment platform.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC ARM cross-compiler (tested under *gcc-arm8-8.2-2018.08-aarch64-elf*).
- **Data set:** We have generated our own data-sets. All are included in the provided VM.
- **Run-time environment:** Our framework has been developed and tested under the Linux environment, more specifically the Debian and Ubuntu distributions. The framework depends on the HOL4 theorem prover, Z3 SMT solver, SQLite DB manager, OpenOCD debugger, which are all cross-platform software, included and pre-installed in the package.
- **Hardware:** A compatible hardware benchmark platform with Raspberry Pi 3 boards (Cortex-A53 processor) is required. We include instructions on how to set up such a platform and remote access to existing platforms is an option.
- **Metrics:** Execution time and time to first counterexample, number of counterexamples to the model soundness and inconclusive cases.
- **Output:** The framework stores generated test-cases in database files and the results per experiment will be displayed in the system console.
- **Experiments:** The process is automated by a number of scripts described in the corresponding README documents. For validating the results, see Section A.6.1.
- **How much disk space required?:** The provided VM image requires around 15 GB.
- **How much time is needed to prepare workflow?:** We have deployed the framework in the replication package, no installation is required.
- **How much time is needed to complete experiments?:** Using 4 Raspberry Pi 3 boards, it approximately takes 7 days to execute all experiments.
- **Publicly available?:** Our framework is a well-documented and open-source software publicly available on GitHub [4].
- **Code licenses:** BSD licensed
- **Archived:** https://doi.org/10.6084/m9.figshare.15086895.v3

A.3 Description

A.3.1 How to access. In our paper, we describe a method to generate test inputs to validate side-channel models. The implementation thereof is called Scam-V and consists of a test input generation tool and a number of models together with a hardware benchmark infrastructure. The tool has been packaged as plain source code and preinstalled in a VM [13] together with our evaluation results. However, the actual evaluation of the tool-generated test inputs requires a benchmark platform that can be used by the benchmark infrastructure.

A benchmark platform can be built with a bit of special hardware and wiring according to the documentation in the repository we have included in the replication package and made available through the GitHub project EmbExp-Box [2]. Among other things, this requires Raspberry Pi 3 boards and OpenOCD JTAG probes. Alternatively, a remote connection to an existing benchmark platform can be used. Both cases require a configuration like the file `config/networks.json.example` illustrates.

With a benchmark platform and connection configuration in place, a user is able to generate test inputs and execute them on the actual hardware via a network connection. Figure 8 shows the overall architecture of Scam-V. The dashed boxes group the components of the pipeline into two categories: experiment generation and experiment platform. Components in the box labeled experiment generation run locally on the user machine, while those in the box labeled experiment platform represent the network-connected hardware experiment platform the evaluation of experiments depends on.

A.4 Installation

No installation is needed. The execution environment is deployed in a VM.

A.5 Experiment workflow

When starting the VM, a terminal and browser open automatically. The browser loads a README document with detailed instructions on how to generate and execute experiments. All paths given below are relative to `~/scamv` in the VM.

A.5.1 Premade Scam-V configurations. The VM includes scripts that configure Scam-V to generate experiments with the same settings as those in the paper. Using these configurations is the most straightforward way of invoking Scam-V in order to reproduce our results, and it should work out-of-the-box by using the scripts in the `introduction` directory following the steps described in the README document.

A.6 Evaluation and expected results

The whole process of validating individual experiments and whole sets is described in the EmbExp-Logs [3] README document. In order to simplify this process, we provide a script in the replication VM to support high level operation and ease the introduction to Scam-V.

Notice that it may happen that the experiment execution process stalls due to run-time issues as indicated in the EmbExp-Logs [3] README file. In this case many experiments execute without a result, which is indicated with the warning `unsuccessful`. This requires either to issue a complete restart or, better yet, to cancel the running experiments and resume by manually orchestrating the scripts in Scam-V examples or EmbExp-Logs [3] according to the documentation. We do not provide a high level script for this purpose.

A.6.1 Evaluation Checklist: Having executed experiments, in order to validate the obtained results the following points must be checked for each experiment output. Notice that the numbers in our checklist are approximate and the results of experiments can be slightly affected by different factors. For example, the execution time can be affected by latency and throughput of the connection to the experiment board server (hardware benchmark platform), the number of counterexamples might be affected by unforeseen and hidden microarchitectural interactions, etc.

Model M_{part}. With refinement in place:

- Number of programs with counterexamples is ~4 times more

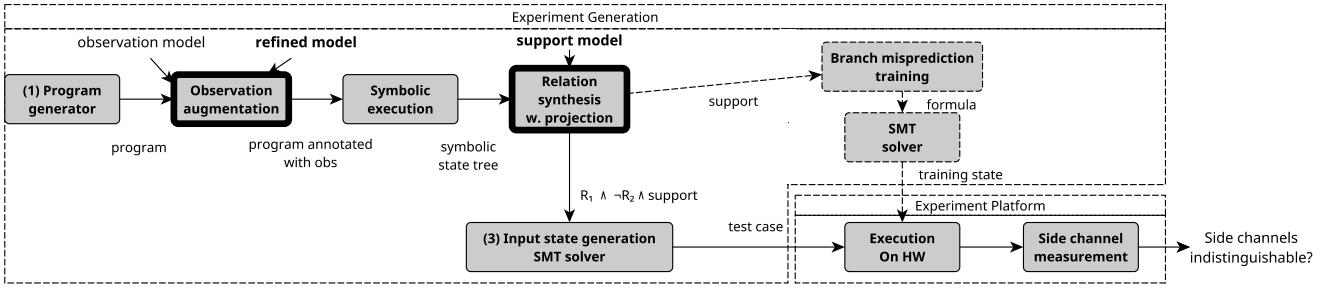

Figure 8: Test-case generation and evaluation with Scam-V.

- Number of counterexamples is ∼20 times greater
- Time to reach the first counterexample is ∼4 times faster

Model M_{ct} Template A.. With refinement in place:

- Number of programs with counterexamples is ∼100 times more
- Number of counterexamples is ∼2000 times greater
- Time to reach the first counterexample is ∼7000 times faster

Model M_{ct} Template B.. Without refinement we do not expect to find any counterexample, while with refinement in place:

- ∼50% of all programs will have at least one counterexample
- ∼13% of all experiments will be counterexamples
- Time to reach the first counterexamples is ∼15 minutes

Model M_{ct} Template C.. Without refinement we do not expect to find a counterexample, while with refinement in place:

- ∼42% of all experiments will be counterexamples
- Time to reach the first counterexamples is less than a minute

Model M_{spec_1} Template C and B and with refinement. While with Template C we do not expect to get any counterexample, with Template B:

- ∼0.6% of all experiments will be counterexamples
- Time to reach the first counterexamples is ∼4.5 hours

REFERENCES

[1] [n. d.]. Eclipse Repairnator. https://projects.eclipse.org/proposals/eclipse-repairnator Accessed: 2021-01-29.

[2] [n. d.]. EmbExp-Box - Embedded Experiments In a Box. https://github.com/kth-step/EmbExp-Box Accessed: 2021-07-29.

[3] [n. d.]. EmbExp-Logs - Embedded Experiments Logs. https://github.com/kth-step/EmbExp-Logs Accessed: 2021-07-29.

[4] [n. d.]. Scam-V. https://github.com/kth-step/HolBA/tree/master/src/tools/scamv Accessed: 2021-09-09.

[5] [n. d.]. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability Accessed: 2021-09-09.

[6] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS)*. Springer-Verlag, 112–121.

[7] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. 53–70.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 53–70.

[9] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. 2013. Formal Verification of Side-Channel Countermeasures Using Self-Composition. *Sci. Comput. Program.* 78, 7 (2013), 796–812.

[10] ARM. 2020. Straight-line Speculation. *Whitepaper* (2020).

[11] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-Level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1267–1279. https://doi.org/10.1145/2660267.2660283

[12] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. 2006. Putting it all together - Formal verification of the VAMP. *STTT* 8, 4-5 (2006), 411–430.

[13] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Scam-V MICRO 2021 artifact. https://doi.org/10.6084/m9.figshare.15086895.v3

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 249–266.

[15] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2019. Towards Constant-Time Foundations for the New Spectre Era. *arXiv preprint arXiv:1910.01755* (October 2019).

[16] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *CSF 2019*.

[17] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[19] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18, 1, Article 4 (2015), 32 pages.

[20] Anthony C. J. Fox. 2003. Formal Specification and Verification of ARM6. In *TPHOLs (Lecture Notes in Computer Science)*, Vol. 2758. Springer, 25–40.

[21] Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP (Lecture Notes in Computer Science)*, Vol. 6172. Springer, 243–258.

[22] Michael Misiu Godfrey and Mohammad Zulkernine. 2014. Preventing Cache-Based Side-Channel Attacks in a Cloud Environment. *IEEE transactions on cloud computing* 2, 4 (2014), 395–408.

[23] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 11–20.

[24] Joseph A. Goguen and José Meseguer. 1984. Unwinding and Inference Control. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 75–87.

[25] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. https://www.ndss-symposium.org/ndss-paper/absynthe-automatic-blackbox-side-channel-synthesis-on-commodity-microarchitectures/

[26] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1853–1869.

[27] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 38–55.

[28] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. SPECTECTOR: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE.

[29] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2020. Hardware-Software Contracts for Secure Speculation. arXiv:cs.CR/2006.03841

[30] Zhe Hou, David Sanán, Alwen Tiu, Yang Liu, and Koh Chuen Hoa. 2016. An Executable Formalisation of the SPARCv8 Instruction Set Architecture: A Case Study for the LEON3 Processor. In *FM (Lecture Notes in Computer Science)*, Vol. 9995. 388–405.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 1–19. https://doi.org/10.1109/SP.2019.00002

[32] Andreas Lindner, Roberto Guanciale, and Roberto Metere. 2019. TrABin: Trustworthy Analyses of Binaries. *Science of Computer Programming* 174 (2019), 72 – 89.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[34] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).

[35] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 1427–1444. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

[36] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*. 156–168.

[37] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. *CoRR* abs/2005.05254 (2020). arXiv:2005.05254 https://arxiv.org/abs/2005.05254

[38] Michael Neve and Jean-Pierre Seifert. 2007. Advances on Access-driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography (SAC'06)*. Springer-Verlag, 147–162.

[39] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium, USENIX Security*.

[40] Dan Page. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive* 2002, 169 (2002).

[41] George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB Slice-a Low-Cost High-Speed Address Translation Mechanism. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 355–363.

[42] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. 947–960.

[43] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 2 (Jan. 2010), 37–71.

[44] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES'03, LNCS)*. Springer, 62–76.

[45] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2020. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 519–532. https://doi.org/10.1145/3385412.3386008

[46] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2019. KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *arXiv preprint arXiv:1909.00647* (2019).

[47] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. https://www.ndss-symposium.org/ndss-paper/speechminer-a-framework-for-investigating-and-measuring-speculative-execution-vulnerabilities/

[48] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*. 719–732.

[49] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 305–316.