Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing

Oleksii Oleksenko
Microsoft Research

Marco Guarnieri
IMDEA Software Institute

Boris Köpf
Microsoft Research

Mark Silberstein
Technion

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179391

***Abstract*—Attacks like Spectre abuse speculative execution, one of the key performance optimizations of modern CPUs. Recently, several testing tools have emerged to automatically detect speculative leaks in commercial (black-box) CPUs. However, the testing process is still slow, which has hindered in-depth testing campaigns, and so far prevented the discovery of new classes of leakage.**

In this paper, we identify the root causes of the performance limitations in existing approaches, and propose techniques to overcome these limitations. With these techniques, we improve the testing speed over the state-of-the-art by up to two orders of magnitude.

These improvements enable us to run a testing campaign of unprecedented depth on Intel and AMD CPUs. As a highlight, we discover two types of previously unknown speculative leaks (affecting string comparison and division) that have escaped previous manual and automatic analyses.

***Index Terms*—Spectre, side-channel-attack, speculative-execution, random testing, constrained random verification**

I. Introduction

Attacks like Spectre [1] exploit speculative information leaks, which are a side-effect of performance optimizations in modern CPUs. These attacks can "trick" a CPU into leaving traces of secret information that make it accessible to an adversary, bypassing program-level security checks.

So far, most of the speculative leaks (e.g., those underlying Spectre [1], Meltdown [2], MDS [3], [4]) have been discovered in a manual effort, by analysing public documentation, patents, and by laborious experimentation.

Recently, several tools have emerged to *automate* this slow and costly process: *White-box* approaches [5]–[7] analyse the processor specification and detect leaks early in the design process. They have so far been successfully applied to smaller open-source CPUs. In contrast, *black-box* approaches [8]–[11] analyze fabricated chips. They have already been applied to full-scale x86 CPUs, and are the focus of this paper.

In the absence of a specification, black-box approaches rely on two kinds of random testing, with different scope:

(i) *Template-based* tools [8], [9] *mutate* code-templates known to trigger speculation. They can detect variants of known leaks rather than finding new ones.

(ii) *Model-based relational* tools [10], [11] generate *fully random* test cases (instruction sequences and inputs), and compare the leakage observed on the CPU with that specified by the leakage model (typically: an ISA-level specification of the permitted leaks).

Black-box approaches have successfully detected all known classes of speculative leaks, as well as previously unknown variants. However, so far they have *not discovered* any fundamentally new kind of speculative leakage. For template-based tools, this is explained by the scope of the approach. For model-based tools, this is due to the performance limitations of the existing approaches, which have been preventing in-depth testing—until now.

Our approach. In this paper, we present a model-based approach for detecting speculative leaks in black-box CPUs, which overcomes the performance limitations of state-of-the-art tools. We improve the speed of testing by two orders of magnitude, which enables us to significantly increase the breadth and depth of our testing campaigns. In our experiments on Intel and AMD CPUs, we detected two new classes of leaks that have escaped prior manual and automatic analysis.

Our key observation is that the vast majority of randomly-generated test cases are *ineffective* in that they have no chance of surfacing a speculative leak—yet existing tools [10], [11] still include them in the expensive leakage analysis, e.g., based on symbolic execution.

This observation motivated us to (i) identify the characteristics of effective test cases; (ii) enforce generation of such test cases where possible; and (iii) design lightweight methods for filtering ineffective test cases. Thus, we are able to dramatically speed up the testing campaigns by either generating test cases that are effective by design, or pruning ineffective test cases *before* they undergo any expensive leakage analysis.

Characteristics of effective test cases. Our goal is to identify programs that speculatively leak more information than what the CPU leakage model prescribes. Technically, this requires executing a program with *two* different inputs so that both executions *agree* on the leakage according to the leakage model (which we call *model trace*), but *differ* on the measurements made on the actual CPU (which we call *hardware trace*). The following conditions are necessary for this to happen:

1) The program misspeculates, i.e., there are *transient* instructions that are issued but never retire;
2) The transient instructions affect the hardware trace, i.e., transient leakage becomes observable;
3) The program inputs result in identical model traces, which makes it possible to compare hardware traces that witness transient leakage.

Our preliminary measurements show the potential of lever-

DOI 10.1109/SP46215.2023.00201

aging these conditions: with randomly generated programs and inputs, condition 1 is satisfied in only 13% of the cases, condition 2 is satisfied in only 7% of the cases, and less than 6% of the inputs satisfy condition 3. It means that, overall, less than 0.5% of program-input combinations are effective.

Improving test case effectiveness. We next explain how we check conditions 1 and 2, and how we ensure condition 3 by design, with the following techniques:

1) *Speculation filtering*: To check for misspeculation (condition 1) during execution of a test case, we leverage signals from the CPU's internal state by monitoring speculation-related performance counters. This technique is inspired by [12] where it was used for manual analysis; here we leverage it for automated test case pruning.

2) *Observation filtering*: To check for visibility in the hardware trace (condition 2), we modify the program by inserting a serialization fence after every instruction, and then compare the execution of the original program with the serialized version. A discrepancy between the hardware traces indicates that the original test case encountered speculation that is visible in the trace.

3) *Contract-driven input generation*: To generate input pairs with identical model traces (condition 3), existing tools rely on accidental matches during random sampling [11] or on symbolic execution [10], which are wasteful and slow, respectively. We propose a technique based on dependency tracking to identify registers and memory locations that can be varied while keeping the leakage model traces identical. Compared to prior work, this technique turns out to be effective *and* fast.

Implementation and experiments. We implement these techniques on top a model-based tool called Revizor [11]. Our implementation targets x86-64 CPUs.

To showcase the techniques, we run a testing campaign with 130 million test case executions, over 13 subsets of the x86 ISA. The highlights are:

- The techniques speed up testing by *two orders of magnitude*. With the current state-of-the-art [11], this campaign would have required over *2 months*. With the techniques we describe in this paper, the campaign took only *16 hours*.
- During this campaign we discovered *two new speculative leaks* based on undocumented kinds of speculation in 64-bit division and string comparison operations. We describe them in Section II.

We further perform an in-depth analysis of how our techniques contribute to this result:

- Speculation and observation filters consistently improve the testing speed. The largest speedups of an order of magnitude are achieved for programs generated from ISA subsets that *never* experience speculation, because virtually all test cases are filtered out. The smallest speed-ups are found in the instruction sets where speculation is ubiquitous (e.g., conditional branches that trigger Spectre V1), where a large portion of the test cases passes the filters.
- Contract-driven input generation reliably increases the number of detected leaks compared to random testing. This is because it increases the number of inputs satisfying condition 3 to over 99%, and achieves that with about 100x less performance overhead compared to symbolic execution-based input generation.
- For leaks that are already detected by the random baseline, our techniques *reduce the detection time by a factor of 10–50*. For the new speculative leaks, our techniques were instrumental for detecting them during our testing campaign.

x86-64 Subset	Information Leakage Detected?			
	Intel			AMD
	Core6	Core8	Xeon	Epyc
`cond`: Conditional branches	✓	✓	✓	✓
`strn`: String operations	✓	✓	✓	✓
`dmul`: Division and multiplication	✓	✓	✓	×
`flag`: Operations on flags	×	×	×	×
`lock`: Atomics w/ LOCK prefix	×	×	×	×
`atom`: Atomics w/o LOCK prefix	×	×	×	×
`dxfr`: Data transfer (load/store)	×	×	×	×
`setc`: Conditional byte set	×	×	×	×
`nop` : NOP instructions	×	×	×	×
`logi`: Logical operations	×	×	×	×
`conv`: Data type conversion	×	×	×	×
`cmov`: Conditional moves	×	×	×	×
`bit` : Bit test and bit scan	×	×	×	×
All except `cond`, `dmul`, `strn`	×	×	×	×

Fig. 1. Summary of the main testing campaign. The testing targets are: Intel i5-6500 (Core6); Intel i7-8650U (Core8); Intel E-2288G (Xeon); AMD EPYC 7543P (Epyc). All microcode patches were enabled. The campaign analyzed 13 subsets of the x86-64 ISA (details in Appendix A). Each subset was tested with 100'000 randomly generated programs, and each program executed with 100 inputs, amounting to 130 million test case executions per target.

The source code is publicly available under:

https://github.com/microsoft/sca-fuzzer

Responsible disclosure. We disclosed our findings to Intel and AMD. Both vendors acknowledged and confirmed the findings.

II. Discovered Speculative Leaks

In this section, we describe two new speculative leaks that we discovered with the techniques presented in this paper. The discoveries became possible due to the improved testing speed, which enabled us to dramatically increase the depth and breadth with which we scrutinize the ISA, and made feasible the testing campaign in Figure 1.

This section describes only the leaks themselves; a reader interested in the *process* of their discovery can find a detailed walk-through in Appendix B.

A. SCO: String Comparison Overrun

The first new speculative leak we found affects all four CPUs we tested (both Intel and AMD). It is caused by string instructions prefixed by `REPE` (repeat while equal) and `REPNE` (repeat while not equal)[1]. With these prefixes, the following instruction is repeated the number of times indicated in the count register, or until the termination condition is met.

Our testing campaign revealed that the CPUs can speculatively compare (`CMPS`) or scan strings (`SCAS`) beyond the

[1] `REP` also speculates, but we found no information leaks caused by it.

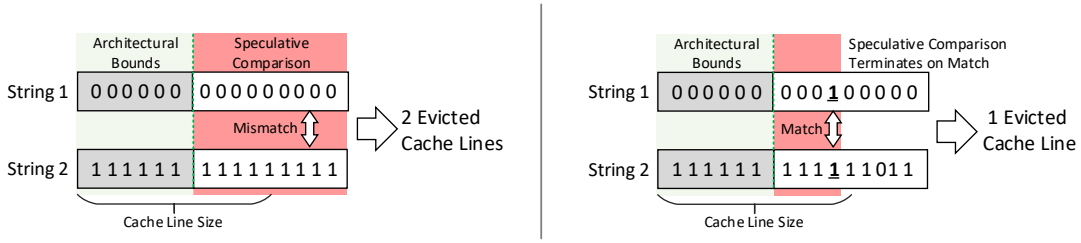

Fig. 2. Illustration of leakage with String Comparison Overrun.

termination condition and leak the outcome into the cache state. This new leak, which we call *String Comparison Overrun* (SCO), has two notable features:

(i) It does not require training: The CPU always goes beyond the bounds if the counter value is not available;

(ii) It does not require a software leakage gadget: A single instruction contains both the speculation trigger and the leaking memory access.

For illustration, consider the following snippet, which compares two strings word-by-word for inequality, starting from the addresses in `rdi` (String 1) and `rsi` (String 2). The comparison terminates when it either finds a matching word or reaches the bounds specified in the count register `rcx`.

```
1 rcx = slow_computation()
2 REPNE CMPSW
```

In the example, the value of `rcx` is not available, which triggers speculation, and the CPU compares memory beyond the specified bounds. Assuming the adversary controls String 1, it can perform equality checks on the memory following String 2 (see Figure 2). The outcome affects the number of evicted cache lines: If the strings disagree on all words (Figure 2a), comparison continues until the speculation window expires, and if the window is large enough, at least two cache lines will be evicted. If not (Figure 2b), comparison terminates with the earliest matching word, and if the match is within the first cache line, only one line gets evicted. The attacker can distinguish these two cases via a cache side-channel attack, thus learning the outcome of out-of-bounds comparison.

This leak enables an adversary to determine the memory content following String 2 as far as speculation lasts. The complexity of determining memory content depends on the granularity: Word-level checks (`CMPSW`) require only up to 2^{16} attempts to produce a match whereas quad-word checks (`CMPSQ`) require up to 2^{64} attempts. The lower complexity, however, comes with smaller reach, as the number of comparisons is bounded by the speculation window: In our prototype implementation, we managed to leak 22 out-of-bounds bytes with `CMPSW` compared to 88 bytes with `CMPSQ`.

B. ZDI: Zero Dividend Injection

The second speculative leak was discovered only in Intel CPUs, and it is triggered by division instructions. Specifically, we found that 64-bit division can speculate on the value of one of its source operands. We call it *Zero Dividend Injection* (ZDI), and to the best of our knowledge, this is the first documented case of value prediction in a commercial CPU.

For illustration, consider the following snippet, where `DIV` in line 2 divides a 128-bit operand in `rdx:rax` by a 64-bit operand in `rcx`.

```
1 rdx = slow_computation()
2 DIV rcx  // means rdx:rax/ rcx
3 MOV rsi, [array_base + rax]
4 // or MOV rsi, [array_base + rdx]
```

If values of registers `rcx` and `rax` are available but that of `rdx` is not, the CPU bypasses this dependency by speculatively assuming that `rdx` (i.e., the upper 64 bits of the dividend) is zero. That is, the CPU effectively computes `0:rax/rcx`, and the result is exposed through the memory access at line 3.

For example, without speculation, $2^{64}/4$ and $2^{63}/2$ yield the same result. With ZDI, the former speculatively yields 0 while the latter yields 2^{62}. An attacker can observe the result from line 3, and thus learn more information about the division operands than would be possible without speculation. Furthermore, line 3 speculatively executes an unexpected memory ready, which causes additional information leakage.

In our experiments, the predicted value was always zero, regardless of the processor state. Moreover, even though we tested several multi-register instructions (such as `MUL`), we only observed value speculation in `DIV` and `IDIV`.

In the next sections, we first discuss the basic mechanism of leakage detection, and then describe the techniques used to discover speculative leaks automatically.

III. Background: Testing for Speculative Leaks

This section provides the background on model-based testing for speculative leaks. We start by introducing speculation contracts [13], a form of a leakage model that captures speculative microarchitectural leaks at the ISA level. Next, we describe how contracts can be used to detect unknown speculative leaks. We conclude by presenting an overview of Revizor [11], a black-box random testing tool for detecting speculative leaks, which is the basis of our implementation.

A. Microarchitectural Leakage and Contracts

When a CPU executes a program, its execution changes the CPU's microarchitectural state. An attacker can observe some of these changes via side channels [14]–[16]. We call such observable changes a **hardware trace**.

The process of collecting a hardware trace can be described as a function $Measure$ that takes a program p, a program's input i, and a microarchitectural context μ, and returns a hardware trace:

$$HTrace = Measure(p, i, \mu)$$

A program leaks information when its hardware traces depend on the input i, as the attacker can distinguish different inputs (potentially containing secrets) by comparing traces.

The leak could be caused not just by the normal program execution but also by the instructions executed transiently; that is, by those instructions that the CPU executes but never retires. We call such leaks **speculative leaks**. For example, the hardware trace may expose secrets accessed after a branch misprediction, as demonstrated in Spectre [1].

To aid the development of defences against speculative leaks, **speculation contracts** [13] have been proposed as an abstract ISA-level specification of the expected microarchitectural leaks (i.e., a speculative leakage model). A contract describes the information expected to leak on a CPU while executing a victim program p with an input i.

A contract abstracts away the microarchitectural details of the function $Measure$, and provides a high-level (i.e., ISA-level) function $Contract$ that returns a trace of expected observations (**contract trace**):

$$CTrace = Contract(p, i)$$

A speculation contract annotates ISA instructions with (a) an *observation clause* that describes the information disclosed by the instruction, and (b) an *execution clause* that describes whether (and how) instructions trigger speculation.

Example 1. CT-SEQ is a contract that describes the leakage expected on a CPU with cache side channels and without speculative execution. To this end, the observation clause of CT-SEQ exposes the addresses of all memory accesses (loads and stores) and of all control-flow operations (jumps, calls, etc). Its execution clause is empty for all instructions, which means that no instruction is expected to trigger speculation.

Example 2. CT-COND is another contract, which describes the leakage expected on a CPU with a branch predictor. Its observation clause is identical to CT-SEQ; its execution clause prescribes that all conditional branches always speculatively take a wrong target. Thus, CT-COND exposes the information leaked by the transient instructions that were executed due to branch prediction.

B. Detecting Leaks via Relational Analysis

Speculation contracts have been applied to detect information leaks in CPUs [11]. A contract violation (i.e., an *unexpected* leakage) is discovered by comparing the leakage according to contract traces (i.e., the *expected* leakage) with the leakage in hardware traces (i.e., the *observed* leakage on the CPU under test).

```
1 CMP rax, 10  // compare rax with 10
2 JNE .END     // jump if not equal
3 MOV rax, [rbx]  // load from address in rbx
```

Fig. 3. A program that produces a counterexample to CT-SEQ.

Definition 1 (Violation). A CPU violates [13] a contract $Contract$ if there exists a program p, a pair of inputs (i, i'), and a microarchitectural state μ, such that $Contract(p, i) = Contract(p, i')$, and $Measure(p, i, \mu) \neq Measure(p, i', \mu)$.

An evidence to a contract violation is called a **counterexample**, and it consists of a program p, a pair of inputs i, i', and a microarchitectural state μ that match Def 1. A counterexample is an evidence to an unexpected leakage because the attacker can distinguish $Measure(p, i, \mu)$ from $Measure(p, i', \mu)$, while these executions are supposed to be indistinguishable according to the contract.

C. Model-based Relational Testing with Revizor

Model-based tools [11], [17], [18] apply the above approach to detect unexpected leaks in CPUs. In this paper, we base our work on one such tool, called Revizor [11].

Revizor searches for contract counterexamples by generating random test cases. A **test case** consists of a random program p and a sequence of random inputs $[i_0, i_1, ...]$. For each program-input combination Revizor collects the corresponding contract and hardware traces:

- Contract traces are collected by the **contract model**, an executable version of the contract implemented with an ISA emulator (QEMU). The emulator is modified to record observations according to the contract observation clause, and to implement speculation according to its execution clause. The model collects traces by executing the program p with each of the inputs i on this emulator, and then retrieving the recorded observations.
- Hardware traces are collected by the executor. It implements $Measure$ by executing the program p with each of the inputs i on the target CPU. The microarchitectural state μ is set indirectly, as it is not directly accessible on black-box CPUs: Each program execution inherently modifies the microarchitectural state, which sets μ for the following executions. Hardware traces are collected by monitoring the microarchitectural changes caused by each execution; specifically, Revizor collects traces by monitoring L1D cache state with performance counters.

After collecting the traces, Revizor checks if any of them satisfies the definition of violation (Def 1). For this, it groups the inputs that produce the same contract trace into **equivalence classes**, and checks if all hardware traces corresponding to inputs in the same class match. If there is at least one pair of different traces (i.e., leakage according to Def 1), these constitute a counterexample to the contract, and Revizor reports the unexpected leakage to the user.

Example 3. Consider a round of a testing campaign where a CPU with branch prediction is tested against CT-SEQ.

The round begins by generating a random program shown in Figure 3, and a sequence of random inputs[2]:

i_1={rax=10,rbx=5 }, i_2={rax=10,rbx=20}
i_3={rax=40,rbx=10}, i_4={rax=20,rbx=70}

Revizor executes this test case on the contract model for CT-SEQ, which collects the jump targets (line 2) and the load addresses (line 3), producing the following traces:

ctrace_1={load *5 }, ctrace_2={load *20 }
ctrace_3={jump .END}, ctrace_4={jump .END}

Note that ctrace_3 and ctrace_4 do not include the load because it is skipped when rax==10.

Next, Revizor executes the test case on the target CPU while monitoring cache evictions with a side-channel attack, resulting in the following traces (CL stands for cache line):

htrace_1={evict CL *5 }, htrace_2={evict CL *20}
htrace_3={evict CL *10}, htrace_4={evict CL *70}

In the first two traces, the branch is not taken, and the loads from addresses 5 and 20 evict the corresponding cache lines (if the cache line size is 64 bytes, they evict from the same cache set). In addition, the first two executions train the branch predictor, so the next two executions experience mispredictions. For input i_3, the CPU speculatively executes a load from address 10, and for i_4, a load from address 70. With 64-byte cache lines, i_3 and i_4 evict different cache lines, which results in different hardware traces.

Accordingly, the last two inputs form a counterexample: $\text{ctrace}_3 = \text{ctrace}_4$ and $\text{htrace}_3 \neq \text{htrace}_4$. Revizor detects it and reports to the user.

Handling microarchitectural states. We now provide additional details on how Revizor handles the microarchitectural state μ when testing a program p with inputs $[i_0, i_1, \ldots]$. Before executing an input, Revizor partially resets the microarchitectural state μ by flushing caches, invalidating TLB, flushing microarchitectural buffers with VERW, and pushing memory fences into the pipeline. The rest of μ (e.g., the state of branch predictors and of other internal buffers) is set by the execution of the test case itself: Since the program is executed with each of the inputs in a sequence, without interruptions, the executions of $[i_0 ... i_{n-1}]$ indirectly set μ_n for i_n. Note that this mechanism is imperfect and parts of the state may not be randomized; see [11] for further discussion of this point.

Input Effectiveness. To form a counterexample, we need two inputs with the same contract trace but different hardware traces; we call such inputs *effective*. In other words, given a program p and inputs $I = [i_0, i_1, ...]$, an input $i \in I$ can surface leakage only if there is another input $i' \in I$ (different from i) such that $Contract(p, i) = Contract(p, i')$. Otherwise, if I contains no input with a contract trace identical to i, then this inputs becomes a wasted effort as it cannot, by definition, be used for leakage detection; we call it an *in*effective input.

Example 4. Consider the program in Figure 3, executed with the following inputs: i_1={rax=0,rbx=1}, i_2={rax=1,rbx=1}, i_3={rax=0,rbx=2}. If the target contract is CT-SEQ, the first two inputs are effective: They produce the same contract trace {load *1}. The input i_3, however, produces the trace {load *2}, and since it is the only input to produce this trace, i_3 is ineffective.

[2]In practice, an input assigns values to multiple registers and to several pages of memory. This example is simplified for clarity.

IV. Design and Implementation

This section presents an approach to increase the speed of testing for speculative leaks and its implementation for Intel and AMD CPUs.

Our key observations is that random testing tools produce very few **effective test cases**; that is, few test cases have potential for surfacing a speculative leak. Yet the existing tools spend just as much time on the effective cases as on the ineffective ones. Changing this balance can significantly increase the testing speed.

We identify the following conditions that make a test case effective:

1) *Misspeculation*: The test case must trigger speculation, and the speculation must be based on an incorrect prediction. This will lead to transient instruction that are issued but never retire.
2) *Trace of misspeculation*: When the misspeculation is triggered, some of the transient instructions must affect the hardware trace to make a leakage observable.
3) *Effective Inputs*: To detect leakage with relational analysis, the inputs that triggers misspeculation must be effective, as described in §III-C.

Next, we illustrate conditions 1 and 2 using examples (condition 3 is illustrated above in Example 4). In the following, we assume a CPU that *only* speculates over conditional branches and the traces are obtained via L1D cache side-channel.

Example 5. The following program is ineffective because it does not meet condition 1:

```
1 CMP rax, 10
2 MOV rax, rbx
```

The program has no instructions that would trigger speculation, thus it cannot produce a speculative leak.

Example 6. The following program is ineffective because it does not meet condition 2:

```
1 CMP rax, 10
2 JNE .END
3 ADD rax, rbx
```

Even if the branch (line 2) is mispredicted, the speculation will not affect the hardware trace, because the program has no memory accesses.

Example 7. The following program is ineffective because it does not meet condition 2:

```
1 CMP rax, 10
2 JNE .l1
3 MOV rax, [rbx]
4 .l1: MOV rax, [rbx]
```

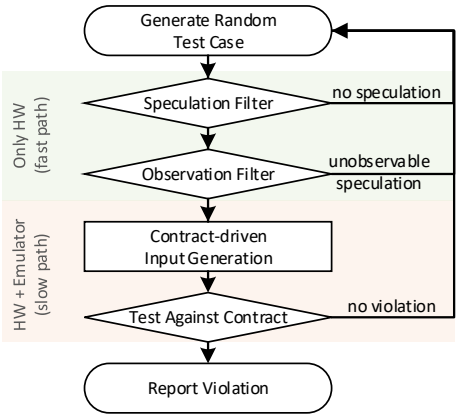

Fig. 4. Main testing algorithm.

Even though it has a conditional branch and a memory access after it, the program also has a non-speculative load (line 4), which uses the same address as the speculative one (line 3). Accordingly, whenever the program evicts a cache line speculatively, the eviction is hidden by the non-speculative access to the same address.

A. High-level Algorithm

With the above conditions, we optimize the testing process to focus primarily on the effective test cases. We identify that the most time-consuming stage of relational testing is the collection of contract traces, because it involves execution of test cases on an ISA-level emulator. Thus, our goal is to prune the test cases before reaching this stage.

We use the algorithm in Figure 4. A testing round begins by generating a random test case. First, the *speculation filter* (§IV-B) checks condition 1 by monitoring performance counters. Next, the *observation filter* (§IV-C) checks condition 2 by performing a serialization experiment. If the test case fails any of these filters, it is discarded. Otherwise, the test case is passed to the *contract-driven input generator* (§IV-D), which ensures condition 3 by creating additional inputs such that every input gets at least one other input in its equivalence class. Only then, when all three conditions are met, the test case is passed to the time-consuming relational analysis to check for contract violations.

Notably, the speculation and observation filters check the conditions conservatively, and they err on the side of permitting the test cases that produce uncertain results (§IV-E).

We next describe the ideas behind each of the techniques, as well as technical details of their implementation. Our implementation is based on Revizor (described in §III-C).

B. Speculation Filter

The task of the speculation filter is to find the test cases that trigger misspeculation. Given a test case (i.e., a program p and multiple inputs $[i_1, i_2, \ldots]$), the filter executes the program on the target CPU with each input and monitors speculation-related performance counters in the process. If at least one of the executions produces misspeculation (reflected in a change to the selected performance counters), then the test case is passed down to the next stages; otherwise, it is discarded.

Implementation. On Intel CPUs, the natural candidates for detecting misspeculations are the counters called `MACHINE_CLEARS.COUNT` (the number of issued machine clears) and `BR_MISP_RETIRED.ALL_BRANCHES` (the number of branch mispredictions). As noted by Ragab et al. [12], these counters are sufficient to detect all types of speculation.

In our experiments, however, we observed that `MACHINE_CLEARS.COUNT` occasionally misses a machine clear and thus, it cannot be used as a reliable signal. Instead, our implementation relies on `INT_MISC.RECOVERY_CYCLES`, which counts recovery cycles from both machine clears and branch mispredictions, and which in practice provides more reliable results. To double check its readings, we use another pair of counters—`UOPS_ISSUED.ANY` and `UOPS_RETIRED.SLOTS`—whose difference gives the number of transient (i.e., never retired) micro-operations. Concretely, the speculation filter detects a misspeculation if either the number of recovery cycles is non-zero (i.e., `INT_MISC.RECOVERY_CYCLES` is incremented during the execution), or the number of transient micro-operations is non-zero (`UOPS_ISSUED.ANY-UOPS_RETIRED.SLOTS > 0`).

Similarly, on AMD CPUs, we used the counters `PMCx0C1` (retired micro-operations) and `PMCx0AB` (micro-operations dispatched from the decoder).

C. Observation Filter

The task of the observation filter is to find the test cases that produce observable speculation traces. Given a program p, the observation filter first creates a serialized version of the program p_{ser} by injecting a serialization fence—`lfence`—after every instruction. Then, the filter executes both p and p_{ser} over all inputs $[i_1, i_2, \ldots]$ in the test case, and collects the hardware traces. Since `lfence`s stop speculative execution, differences in the hardware traces of p and p_{ser} over an input i are the result of transiently executed instructions. The filter admits a test case if p and p_{ser} produce different traces over at least one of the inputs; otherwise, the test case is discarded.

Implementation. Our implementation relies on Revizor's mechanism for collecting hardware traces (see §III-C). This mechanism relies on a side-channel attack to observe changes in the L1D cache state. Accordingly, the observation filter looks for differences in the L1D cache state between p and p_{ser}. These differences are caused by transient memory accesses during p's execution that evict a cache line that is *not evicted* by the (non-speculative) memory accesses in p_{ser}.

D. Contract-driven Input Generator

The task of the contract-driven input generator (CIG) is to ensure that all inputs are effective. Random generation is rarely successful in this task, as the likelihood of two random inputs producing the same contract trace can be extremely low (e.g., in the experiment §V-D, only 6% of random inputs were effective). CIG solves this problem by creating additional inputs based on the feedback from the contract model.

Given a program p and an input i, CIG generates additional (and different) inputs $[i_1, i_2 \ldots]$ such that $Contract(p, i) = Contract(p, i_j)$:

1) CIG tracks contract-level dependencies to compute the set $Dep_{Contract}(p, i)$, which contains all dependencies of the contract trace $Contract(p, i)$. Concretely, $Dep_{Contract}(p, i)$ contains all parts of i (registers and memory locations) that may affect the trace.
2) CIG generates new inputs $[i_1, i_2 \ldots]$ by mutating all the values in i that do not belong to $Dep_{Contract}(p, i)$. Since parts of the input not in $Dep_{Contract}(p, i)$ do not affect the contract trace, all generated inputs are contract-equivalent.

Correctness of effective input generation. Contract-level dependency tracking guarantees that $Dep_{Contract}(p, i)$ contains *all* dependencies that may influence the contract trace. As a result, for all inputs i, i', if i and i' agree on the values of all dependencies in $Dep_{Contract}(p, i)$, then $Contract(p, i) = Contract(p, i')$. Contract-driven input generation leverages this guarantee to produce contract-equivalent inputs by mutating the parts of i that are not in $Dep_{Contract}(p, i)$.

Implementation. We instrumented the Revizor's contract model to track dependencies between memory locations and registers at each step of program execution. The instrumentation keeps track of a dependency map $DMap$ that assigns to each location (a register, a memory address, or a flag) its set of dependencies, i.e., the set of initial values that might have contributed to the location's current value. The map also tracks dependencies for the program counter `pc`. The map is updated throughout the execution as follows:

- The initial dependency map $DMap_0$ is such that $DMap_0(l) = \{l\}$ for each location l, where a location can be a register, a memory address, or a flag.
- For each instruction `instr`, we compute the read set $read(\texttt{instr})$ (containing all locations read by `instr`) and the write set $write(\texttt{instr})$ (containing all locations that `instr` writes to). Note that for control-flow instructions, the program counter `pc` is a part of the write set $write(\texttt{instr})$.
- Whenever we execute an instruction `instr`, we update the dependency map $DMap$ to $DMap'$ by tracking the new dependencies from `instr`'s read set and from the program counter to `instr`'s write set. That is, for each $l \in write(\texttt{instr})$, the new dependency set $DMap'(l)$ is $DMap(\texttt{pc}) \cup \bigcup_{l \in read(\texttt{instr})} DMap(l)$.
- Whenever the contract model explores a speculative path, we create a copy of the dependency map which is used (and updated) throughout the speculative transaction (similarly to how the contract model tracks the speculative program state). When the speculative transaction terminates, the speculative dependency map is discarded and the old dependency map is restored.

We compute the set $Dep_{Contract}(p, i)$ of dependencies associated with the trace $Contract(p, i)$ as follows. Whenever the contract produces an observation o and the current dependency map is $DMap$, we compute the locations L that influence o's values and we update $Dep_{Contract}(p, i)$ by adding the program counter's dependencies $DMap(\texttt{pc})$ and the dependencies of L's locations $\bigcup_{l \in L} DMap(l)$. For instance, if the observation exposes the accessed memory address, then the set L contains the operands determining the address.

Example 8. Consider the following program:

```
1 CMP rax, 10
2 JNE .l1
3 MOV rax, [rbx]
4 .l1: MOV rbx, [rax]
```

It is executed with the input i={`rax=20,rbx=5`}, and the target contract is CT-SEQ. During dependency tracking, the dependency map is updated as follows (here, $DMap_j$ denotes the map after executing line j):

Initially, the map $DMap_0$ is such that each location l is mapped to $DMap_0 = \{l\}$: e.g.,

$$DMap_0(\texttt{rax}) = \{\texttt{rax}\}$$

At line 1: The comparison sets the dependencies of all flags to those of `rax` and of `pc`, since all instruction read `pc` by default. For example, the resulting dependencies of `ZF` are:

$$DMap_1(\texttt{ZF}) = \{\texttt{pc}, \texttt{rax}\}$$

At line 2: Jump propagates the dependencies of `ZF` to `pc`:

$$DMap_2(\texttt{pc}) = \{\texttt{pc}, \texttt{rax}\}.$$

(Line 3 is not executed for this input.)

At line 4: The dependency set $Dep_{\text{CT-SEQ}}(p, i)$ is updated, because loads are exposed in CT-SEQ. Specifically, the load address depends on `rax` and on `pc` (by default), therefore:

$$Dep_{\text{CT-SEQ}}(p, i_1) = \\ = DMap_2(\texttt{pc}) \cup DMap_2(\texttt{rax}) = \{\texttt{pc}, \texttt{rax}\}$$

After line 4, the execution finishes, and the resulting dependency set is passed down to the input generator.

The generator takes the input i_1, and randomly mutates those parts of it that are not included into $Dep_{\text{CT-SEQ}}(p, i)$. In this (simplified) example, the input consists of only two registers `rax` and `rbx`, and only `rbx` is not included in $Dep_{\text{CT-SEQ}}(p, i)$. Accordingly, the generator mutates `rbx` and leaves `rax` unchanged. The resulting input i' is:

$$i'=\{\texttt{rax=10,rbx=70}\}$$

The input pair i, i' is effective because they produce the same contract trace. Should speculation occur during the measurement on the target CPU, the (speculative) load at line 3 will access different addresses for inputs i and i', thereby surfacing a speculative leak.

E. Practical Issues

The filtering techniques described in §IV-B and §IV-C rely on empirical measurements, which are inherently imprecise on a black-box CPU. Such imprecision occasionally leads to false negatives and false positives.

False negatives. Both speculation and observation filters implicitly assume that speculative execution is deterministic. This assumption is not always true on a black-box CPU, where we cannot directly control the microarchitectural state. Even if the test case contains the necessary instructions and data to

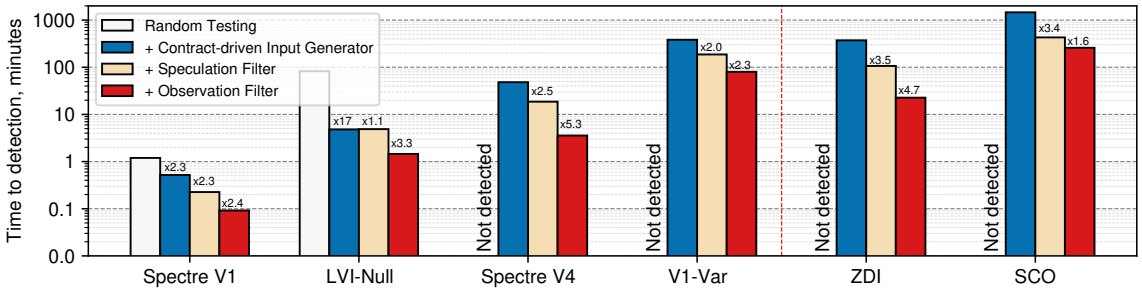

Fig. 5. Detection Time: The average amount of time required to detect a leak. The numbers above the bars are the ratio between the given bar and the one left from it, which corresponds to the relative speedup between the measurements.

trigger speculation, the microarchitectural state may prevent a misspeculation (e.g., if a branch predictor is not trained). Accordingly, the speculation filter may discard an effective test case because a predictor did not produce a misprediction, or the observation filter may discard the test case because the speculation was not long enough to leave a trace.

Fortunately, such cases are rare. We tested both filters on several instances of known leaks, and they produced correct results. We further evaluate false negatives in detail in §V-C.

False positives. The observation filter occasionally produces false positives that are caused by external noise. When the filter executes p and p_{ser} with the same input and collects the corresponding hardware traces, one of the traces could get corrupted by the measurement noise. This will lead to an apparent mismatch between the traces, even though it is not caused by transient instructions. These false positives result in some ineffective test cases not being discarded early on. The impact is minor, however, since we observed a corrupted measurement on average only once in a million executions.

V. Evaluation

This section evaluates the impact of the speculation and the observation filters, and of the contract-driven input generator.

We evaluate the impact across three metrics: *Detection time* is the time required to detect a leak; we consider it the key metric for our leakage-detection tool. *Testing speed* is the amount of time required to test N test cases. *Detection rate* is the number of contract violations detected within N test cases. Detection time is a compound metric, as it depends on both the testing speed and the detection rate.

In §V-B, we measure the impact of our techniques on the detection time. Then, we break it down into its components. In §V-C we measure the changes in the testing speed and the detection rate caused by speculation and observation filters; and in §V-D, we evaluate the changes caused by CIG.

A. Experimental Setup

We perform the experiments on an Intel Xeon E-2288G CPU[3] with all microcode patches against speculative leaks enabled; the only exception was the V4 target in §V-B, where the corresponding patch was disabled.

All the experiments used the same configuration of the test case generator (unless mentioned otherwise): program size—32 instructions; number of memory accesses per program—8; input generation entropy—16 bits.

B. Detection Time

Our first research question is: *Do our techniques reduce the leak detection time?*

The answer is affirmative, yet the extent of the reduction depends on the leak. To evaluate it, we consider Spectre V1, V4, and LVI, a variant of V1 (called V1-Var), as well as the new speculative leaks—ZDI and SCO—described in §II. We perform the evaluation on several testing configurations (described in detail in Appendix C), which we specifically pick such that our tool detects only one type of leaks in each configuration, and any other leaks are unlikely to surface.

For example, in the configuration "Spectre V1", we test a combination of subsets `nop`, `bit`, `cond`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi` (see Appendix A) against the CT-SEQ contract. This configuration can surface V1 because the instruction set contains conditional branches, but it cannot surface the other leaks because the firmware patch is enabled (prevents V4), microcode assists are not permitted (prevents LVI), the subsets do not include divisions (prevents ZDI) and string operations (prevents SCO).

For each of these configurations, we perform an experiment where we execute 100'000 test cases with 100 inputs each; for ZDI and SCO, we execute 500'000 test cases (these leaks are harder to detect, so we need more test cases). We measure the overall execution time, count the number of detected contract violations, and hence calculate the average time to violation.

Each experiment is performed with each of our techniques: (i) fully random testing (baseline), (ii) with contract-driven input generation, (iii) with speculation filtering, and (iv) with observation filtering. We fix the seed for random generation to ensure that the different techniques are evaluated on the exact same sequence of test cases.

Results. Figure 5 shows the results. Note that the vertical axis is in log-scale. We highlight the following findings:

- In *isolation*, speculation filtering, observation filtering, and contract-driven generation already lead to significantly

[3]We also performed the measurements on the other machines from Figure 1, but the results were very close to Intel Xeon, hence we do not present them. The only notable difference was in Figure 6 where the AMD CPU did not show any signs of speculation in `dmul` and `flag`.

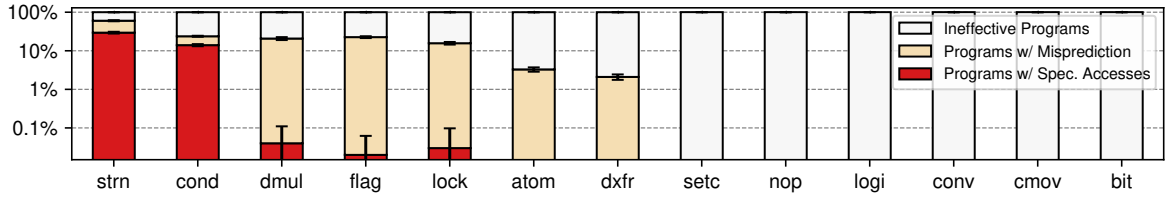

Fig. 6. Share of random test cases with misspeculation and speculative memory accesses in different x86 subsets. The subsets are described in Appendix A.

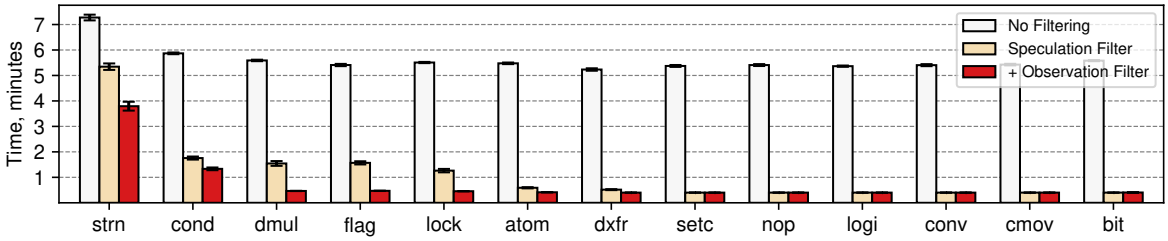

Fig. 7. Change of the testing speed caused by speculation and observation filtering, for subsets of x86 ISA. The subsets are described in Appendix A.

faster detection of almost all leaks. In *combination*, they can reduce the detection time by a factor of 10–50.

- An exception is the configuration LVI-Null, where the speculation filter had almost no impact on the detection time. This is because this speculation type is easy to trigger: any access to a page with an unset Dirty bit suffices. Thus, most of the test cases experienced a machine clear, and the speculation filter was not useful.
- Without the techniques presented in this paper (i.e. in the baseline experiment), only V1 and LVI-Null are detected. V4 and ZDI are not detected because the effectiveness of random inputs is particularly low and speculation is hard to trigger, which is why contract-driven generation is instrumental for detection. Detection of SCO requires a pair of inputs such that in one of the inputs the strings match and in the other do not; finding a randomly-generated pair that both matches this description *and* is effective, is very unlikely. V1-Var requires a microarchitectural race condition, which is hard to trigger with random programs.

Sources of lower detection time. To investigate the source of such a drastic improvement in the detection time, we break it down into its components. Detection time is effectively a ratio between testing speed and detection rate. Our techniques impact these parameters differently: Speculation and observation filters improve the testing speed (because they discard ineffective test cases), but they also decrease the detection rate (because of false negatives, see §IV-E). On the other hand, contract-driven generation increases the detection rate (because all inputs can be used for leakage detection), but decreases the testing speed (because dependency tracking takes additional time). We evaluate these effects in §V-C and §V-D.

C. Impact of Speculation and Observation Filters

Testing speed: *How much does speculation and observation filtering change the testing speed?*

The answer to this question depends on the instructions from which test cases are randomly generated: they govern the likelihood of triggering speculation and hence the effect of filtering out test cases that neither speculate nor perform speculative memory accesses.

We perform experiments on the subsets of the x86 ISA from Figure 1. For each subset, we generate 1000 test cases with 100 inputs each, and we measure (i) the fraction of test cases that passes each filter, and (ii) the time required for processing all test cases with none, one, or both filters enabled. The measurement is repeated 10 times with different seed values of the test case generator. We report the mean values and the standard deviations.

Results. The results for (i) are given in Figure 6, for (ii) in Figure 7. We highlight the following findings:

- Speculation and observation filtering consistently improve the testing speed for all subsets of the x86 ISA we considered (Figure 7). The reason for this improvement is that most of the test cases experience no speculation, and neither do they trigger speculative memory accesses (Figure 6), which is why pruning them early is beneficial.
- The largest speedups of an order of magnitude are achieved in the ISA subsets that *never* experience speculation (e.g., `cmov`, `nop`) because virtually all test cases are filtered out. The smallest speed-ups are found in the instruction sets that have a true violation (e.g., `cond` could trigger Spectre V1). There speculation is ubiquitous, and a large portion of the test cases passes the filter. (A similar behavior is observed if the contract permits speculation, see §VI-C.)
- The impact of false positives on speculation filtering (§IV-B) is noticeable in `flag` and `lock`, see Figure 6: Many ineffective test cases pass the first filter, which is why the speed-up from speculation filtering is less significant. However, these cases do not pass through the observation filter, so they have almost no effect on the testing speed when both filters are enabled.
- Notably, the baseline testing speed was somewhat different between the instruction sets, because certain instructions take longer to execute on the contract model. For

	No Filtering	Speculation Filter	+ Observation Filter
Spectre V1	1577	1585	1537
Spectre V4	15	15	14
LVI-Null	167	168	159
V1-Var	3	3	3
ZDI	11	11	11
SCO	3	4	4

Fig. 8. Number of detected violations within a batch of test cases, tested with different filtering levels. In the first 4 rows, the batch size is 100k cases; in the last 2, the size is 500k.

	Random Input Generation	Contract-driven Input Generation
Spectre V1	272	1577
Spectre V4	0	15
LVI-Null	4	167
V1-Var	0	3
ZDI	0	11
SCO	0	3

Fig. 9. Number of detected violations within a batch of programs, where their inputs are generated either randomly or with CIG. In the first 4 rows, the batch size is 100k programs; in the last 2, the size is 500k.

example, string operations (`strn`) produce many memory accesses, which slows down contract tracing.

Detection rate: *How many contract violations are missed due to false negatives of filtering?*

The inherent limitation of speculation and observation filtering is that some of the effective test case could be mistakenly discarded (see §IV-E). We seek to estimate the amount of such wrongly discarded cases by comparing the numbers of violations detected in the first experiment §V-B.

Figure 8 presented the number of violations detected within the same sequence of test cases, but with different filtering levels. As we see, the number of violations changed very little when we applied the filtering techniques. It implies that the rate of false negatives was low and the techniques had little impact on the detection rate. Curiously, in some of the measurements, more violations were detected with filtering rather than without. We attribute it to the instability of speculation in some of the test cases, which introduces a measure of randomness into this experiment.

D. Impact of Contract-driven Input Generation

Detection rate: *How many more contract violations are found due to contract-driven generation?*

While random input generation may result in ineffective inputs, contract-driven input generation (CIG) produces effective inputs by construction (§IV-D). Thus, each test case is tested more thoroughly with CIG-produced inputs, and we expect to detect more contract violations.

We check if this expectation by looking at the total number of violations detected in §V-B, shown in Figure 9. The columns *Random Testing* and *CIG* are the number of violations detected with the corresponding input generation method, given the same sequence of programs. With random generation, there are only a few violations, and some leaks are not detected at all. Meanwhile, with CIG, the number of violations increases drastically, which indicates that CIG helps in finding leaks.

To illustrate the source of such a significant improvement, we perform an additional experiment that shows just how ineffective random generation is. For this, we randomly generate 1000 programs and for each of them we generate 100 random inputs. For each program, we execute the inputs on the CT-SEQ contract model, and we count the number of effective inputs (i.e., those for which there is at least another input producing the same contract trace). For CT-SEQ, the likelihood of randomly generating an effective input depends on the number of memory accesses and of control-flow paths. Thus, we repeat the experiment in four configurations: with 1 and 4 conditional branches, and with 4 and 16 memory accesses.

Configuration	BB=1 Mem=4	BB=1 Mem=16	BB=4 Mem=4	BB=4 Mem=16
Effective inputs	4.5%	0%	5.7%	0%

Fig. 10. Share of effective inputs produced by random input generation, for different program generator configurations. *BB* stands for the number of basic blocks in a program; *Mem* stands for the number of memory accesses.

The results are in Figure 10. The share of effective inputs produced by the random generator is very low (less than 6% in our results). Moreover, the number of effective inputs decreases with the increase of memory accesses in the program (so, when the contract trace exposes more information). These results indicate the key limitation of random generation for relational testing, which CIG overcomes.

Testing speed: *How much does CIG cost in terms of the testing speed?*

Contract-driven generation, with its contract-level dependency tracking, is considerably more time-consuming than random generation, as the following experiment shows.

For each of the configurations from the previous experiment (Figure 10), we execute 1000 test cases, each with 100 inputs, and we measure the execution time. We repeat the measurement with random input generation and with CIG.

The results are in Figure 11. As we can see, CIG has a noticeable cost: It increases the execution time (i.e., reduces the testing speed) by over 2x. Nevertheless, this cost is acceptable, because CIG allows us to detect many more contract violations. As we saw in Figure 5, even with this high cost, CIG produces a net benefit w.r.t. the detection time.

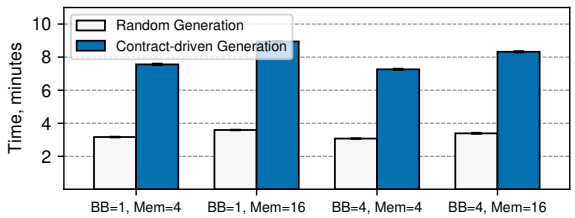

Fig. 11. Performance impact of contract-driven input generation, in different program generator configurations. *BB* stands for the number of basic blocks in a program; *Mem* stands for the number of memory accesses.

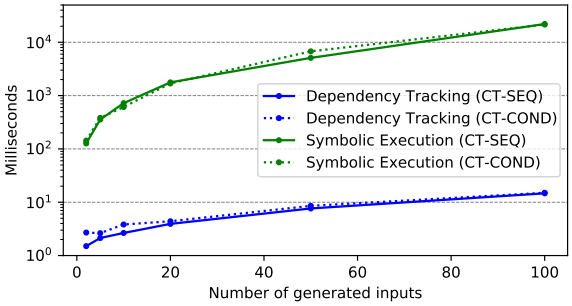

Fig. 12. Execution time of different input-generation techniques—contract-driven input generation (§IV-D, in blue) and symbolic execution (in green)—for the CT-SEQ and CT-COND contracts.

E. Dependency Tracking vs Symbolic Execution

Effective inputs can be generated using different techniques. For instance, our contract-driven input generator (§IV-D) relies on dependency tracking whereas Scam-V [17] (another model-based testing tool) uses symbolic execution. Here, we compare these two techniques.

Generating contract-equivalent inputs with symbolic execution. Directly comparing the input generator from §IV-D with Scam-V symbolic execution approach is impossible because Revizor targets x86 whereas Scam-V targets ARM. Instead, we implemented a contract-driven input generator based on symbolic execution by extending SPECTECTOR [19], [20]. This is a state-of-the-art symbolic analysis tool for verifying the absence of leaks in x86 programs with respect to the CT-SEQ and CT-COND contracts. Given a program p and concrete input i, we generate the fresh j-th contract-equivalent input by keeping track of the inputs $[i_1, \ldots, i_{j-1}]$ already generated and executing a symbolic query asking a new input i' such that (a) i and i' produce the same contract traces (i.e., $Contract(p, i) = Contract(p, i')$) and (b) i' is different from i and from all previously generated inputs $[i_1, \ldots, i_{j-1}]$. If the symbolic query is satisfiable, SPECTECTOR generates a fresh contract-equivalent input. We use SPECTECTOR's symbolic relational reasoning capabilities, which support expressing symbolic constraints over pairs of executions, to encode the aforementioned symbolic query.

Experiment. To compare the cost of CIG and symbolic execution, we measure the time needed to compute n contract-equivalent inputs $[i_1, \ldots, i_n]$, with n taking values in $\{2, 5, 10, 20, 50, 100\}$, given a program p and a concrete initial input i_0. For each value of n, we generate 100 random programs with 10 concrete initial inputs each and we measure the average input generation time for both approaches. We repeat these experiments for two different contracts: CT-SEQ and CT-COND. The generated programs consist of 24 instructions with 1 basic block and 12 memory accesses.

In all experiments, the symbolic memory initially allocated for SPECTECTOR's inputs is limited to 20 bytes (rather than the full memory pages used by CIG). This was necessary to limit the symbolic analysis' execution time.

Results. Figure 12 depicts the average input generation time for CIG (with dependency tracking) and symbolic execution. We highlight the following findings:

- The input generation time increases (for both techniques) with the number of inputs to generate.
- Generating contract-equivalent inputs using dependency tracking is significantly faster than with symbolic execution. For CT-SEQ, for instance, CIG takes between 1.5 ms ($n = 2$) and 14.6 ms ($n = 100$) on average, whereas symbolic execution takes between about 142.7 ms ($n = 2$) and 22.15 s ($n = 100$).
- Generating inputs for the CT-COND contract is slightly more expensive than for the CT-SEQ contract, for both techniques. This is due to the COND execution clause resulting in the execution of more instructions than SEQ.
- Increasing the size of the generated programs by duplicating the number of instructions and memory accesses (not shown in Figure 12), results in an increase in the input generation time for both techniques. For CT-SEQ, CIG takes between 2.3 ms ($n = 2$) and 15.6 ms ($n = 100$) on average, whereas symbolic execution takes between about 183.4 ms ($n = 2$) and 42.2 s ($n = 100$).

VI. DISCUSSION

A. Impact of New Speculative Leaks

To understand the potential security impact of SCO and ZDI, we counted the corresponding instructions in three common applications: (i) In Linux kernel v5.14 we found 320 `CMPS`, 276 `SCAS`, and 28 64-bit divisions. (ii) In GLibC v2.27 we found 131 `CMPS`, 17 `SCAS`, and 4 64-bit divisions. (iii) In OpenSSL v1.1.1 we found 18 `CMPS`, no `SCAS`, and 2 64-bit divisions. Even though we do not know how many of these instances are exploitable, the numbers show us that both SCO and ZDI could potentially affect critical applications and libraries. We leave the investigation of software vulnerabilities caused by SCO and ZDI to future work, since the goal of this paper is to present the techniques for finding speculative leaks rather than ways to exploit them.

Mitigation. SCO can be (partially) patched at the software level by adding an `lfence` before the string comparison, which shortens the speculation window, although it does not prevent the speculation entirely. ZDI can be patched by inserting an `lfence` after 64-bit division operations, thereby stopping the speculative execution. At the time of writing, Intel and AMD have not announced any hardware/firmware mitigations for SCO or ZDI.

B. Filtering vs Targeted Generation

Our strategy for improving the effectiveness of testing relies on generating test cases randomly and then filtering out the ineffective ones. Alternatively, one could target program generation towards test cases that are effective by design. For example, adding a load after every instruction in a test case would trivially address Condition 2 in §IV.

We refrained from this approach because targeted generation can have unintended consequences: Every speculative leak has

its own unique set of requirements, and unknown leaks may have unknown requirements (e.g., ZDI requires a division with a dividend larger than 2^{64}). Targeted program generation that satisfy Conditions 1 and 2 might violate these unknown requirements, thereby making it less likely (or even impossible) to detect unknown leaks. For example, by adding a load after every instruction, we increase the probability that a speculative value produced by LVI or by V4 will be overwritten, and it can reduce the chances of detecting these leaks.

C. Limitations

Not detected leaks. There are known leaks that have not been detected in our experiments (e.g., Meltdown [2], FPVI [12]). The main reason is coverage: We tested a small subset of the x86-64 ISA, and most of the non-detected leaks require instructions/events outside of this subset. For example, FPVI requires floating-point or SIMD instructions, and Meltdown requires page faults that were intentionally avoided during our experiments. To increase the coverage and to detect such leaks, we would have to enhance the test case generator to produce more complex instructions and to handle faults.

The only exceptions (that we known of) are Straight Line Speculation (SLS) [21] and Phantom JMPs [22]. Our testing campaign covered all the instructions necessary for these leaks, yet they were not detected. The reason behind it is imperfect randomization of the microarchitectural state: These leaks require mistraining of the BTB by executing diverse (e.g., random) binaries. However, Revizor executes each test case multiple times, which means that the BTB state is not randomized enough, resulting in not exploring such "mispredicting" BTB states. We consider solving this issue as future work.

Detection of non-speculative leaks. Even though speculation and observation filters are instrumental in increasing test effectiveness for detecting speculative leaks, they reduce the chances of detecting non-speculative leaks (e.g., those described by Sanchez Vicarte et al. [23]). Indeed, a test case that detects a non-speculative leak might be discarded because it does not leave traces of misspeculation (i.e., it does not meet conditions 1 or 2 from section IV).

Dependency tracking. Our dependency tracking is coarse-grained: it only tracks which locations influence the trace, but it does not track the relationships between locations and trace observations. For instance, if an observation *o* depends on the sum of registers `rax` and `rbx` being greater than 10, our dependency tracking determines that `rax` and `rbx` influence the trace. Hence, CIG will not mutate the values of `rax` and `rbx`. However, any mutation satisfying `rax+rbx>10` would not change the trace. Finer-grain (but more expensive) dependency analyses might capture some of these relationships and allow more input mutations.

Testing against contracts allowing speculation. The performance improvement of speculation and observation filters can be reduced by testing against contracts (like CT-COND) that allow selected speculative leaks. Indeed, some of the tests that pass our filters (i.e., they leave traces of misspecultion) might simply represent leaks that are allowed at contract-level. In our testing campaign, we avoided this issue by splitting the tested instruction set into subsets, which we then tested against the simple CT-SEQ contract (that forbids all speculative leak).

D. Generality

Speculation and observation filters. Our filtering techniques can be applied to any testing approach for detecting speculative leaks using randomly generated programs. The benefits of filtering, however, depend on the specific testing approach: As our results show, early filtering significantly improves the effectiveness of model-based testing for black-box CPUs by preventing ineffective test cases to be executed on the contract model. In other settings (e.g., white-box testing of RTL processor designs), the execution of the contract model is unlikely to be the performance bottleneck, so the impact of filtering might be reduced.

Implementing filters for other architectures. We implemented the filtering techniques for Intel and AMD processors using (a) performance counters for reliably detecting speculation (for the speculation filter), and (b) instructions that reliably stop speculation (for the observation filter). Regarding point (a), other architectures provide counters tracking the effects of speculative execution. For instance, ARM provides the `BR_MIS_PRED` counter tracking the number of retired mispredicted branch instructions. Additionally, ARM also supports the `OP_RETIRED` and `OP_SPEC` counters counting respectively the number of retired and speculatively executed microoperations. Regarding point (b), many other architectures have instructions that act as speculation barrier, e.g., `SB` for ARM.

Contract-driven Input Generation. CIG can be integrated into relational testing approaches for detecting leaks with respect to a given leakage contract. In this context, CIG can be a lightweight, faster alternative to input generation techniques based on symbolic execution (Section V-E). Note that CIG is not restricted to the detection of speculative leaks. For instance, applying CIG to the CT-SEQ contract can generate inputs witnessing (architectural) leaks due to variable-timing instructions.

Filters and hardware traces. Revizor obtains hardware traces through cache side-channel attacks on the L1D cache, so this is what we used in our evaluation (§V). While our filtering techniques are largely agnostic of how hardware traces are obtained, different kinds of hardware trace may influence the outcome of the filters. For instance, if hardware traces only expose whether a block is currently in the L1D cache (as currently implemented in Revizor), then the observation filter rejects the program in Example 7. In contrast, hardware traces exposing information about cache metadata (like the age of each block [24], [25]) may result in Example 7 being accepted by the observation filter.

Distributed testing. Given that Revizor's testing process is random and the generated tests are independent, the performance issues of contract modeling could be addressed by

distributing the workload. Indeed, both the model and the executor could be distributed to an arbitrary number of nodes with a central node analyzing traces. The filters proposed in this paper could be applied within such an architecture too. We consider a distributed implementation as future work.

VII. Related work

For an overview and taxonomy of speculative attacks see [26] and the references provided throughout the paper. For an overview of software-based defenses, see [27]. Here we focus on work related to automatic detection of speculative leaks and side-channels in hardware.

Post-manufacture Detection of Speculative Leaks. We begin with the tools targeting fabricated CPUs.

Revizor [11] provides the baseline for the techniques presented in this paper. Revizor is based on relational testing against a leakage model, and it relies on random unguided generation. In contrast, speculation and observation filters use hardware signals to prune ineffective test cases, whereas contract-driven input generation uses the contract feedback to produce effective inputs.

Scam-V [17], [18] also relies on relational model-based testing, which is implemented by instrumenting test cases to record observations. It targets ARM ISA, and it uses symbolic execution for generating inputs with identical leakage. In contrast, we target x86, and we rely on a lightweight approach for input generation based on dependency tracking (see §V-E).

Transynther [8] is a tool that can detect variants of microarchitectural data sampling (MDS) attacks. To this end, it fills CPU buffers with buffer-specific nonces, mutates and executes code-templates that are known to trigger speculation, and it tries to detect leakage of the nonces. In contrast, our approach searches for violations of an abstract leakage model, which enables the discovery of entirely new classes of speculation.

SpeechMiner [9] is a framework to quantitatively evaluate CPUs for their vulnerability to known kinds of speculative attacks. The framework is scriptable and enables users to combine different kinds of primitives used in known attacks. In contrast, our tool generates fully random programs, which enables discovery of new leaks.

Ragab et al. [12] and Li et al. [28] use counters of machine clears to discover new types of speculative leakage, albeit in a manual fashion. Speculation filtering is inspired by the idea of using machine clear counters, and it uses the counters to prune test cases during automatic analysis.

Pre-silicon Detection of Speculative Leaks. The methods for white-box pre-silicon detection speculative leaks rely on access to a detailed CPU design.

CheckMate [5] is a tool that synthesizes exploits against microarchitectural leaks such as Spectre and Meltdown. It relies on a specification of the microarchitecture in terms of happens-before relations. In contrast, our approach relies only on minimal signals from the hardware and can be applied to commercial off-the-shelf CPUs.

IntroSpectre [6] is an RTL-based framework to detect Meltdown-type leaks. It relies on inserting nonces into a security domain, executing randomly mutated leakage templates, and searching for the nonces in other security domains. In contrast, our approach uses relational analysis and can identify information flows beyond direct data transfer.

Fadiheh et al. [7] propose to detect covert channels by model-checking an RTL model of a RISC-V CPU for violations of non-interference.

Detection of Side-channels. Several approaches automatically detect side-channels, without considering speculation.

Osiris [29] discovers novel non-speculative side-channels. For this, it generates random code snippets and measures if their execution time is secret-dependent. In contrast, we specifically focus on speculative leaks.

ABSynthe [30] creates leakage maps capturing interactions between different x86 instructions with SMT enabled, which it uses to synthesize attacks on a given target program.

In principle, the result of these tools could be used to refine the hardware traces used by our proposed techniques.

Leakage models and information flow checking. The work presented in this paper speeds up testing CPUs against leakage models. Specifically, we rely on a model for speculative leakage that is formalized as *speculation contracts* [13]. There are alternative leakage models emerging, for example those based on ideas from weak memory models [31], [32].

SecVerilog [33] is a hardware design language with information-flow control built in. It uses static type checking to ensure that a system complies with an information-flow control policy by design. Connecting these policies to ISA-level leakage model [13], [31], [32] and enforcing them by design is a promising avenue for future work.

VIII. Conclusion

In this paper, we showed how to overcome the performance limitations of model-based relational tools for speculative leaks. This allowed us to drastically speed up black-box testing for speculative leaks and to run a testing campaign of a previously-infeasible depth (130 million test executions) on Intel and AMD CPUs. Our testing campaign discovered two new leaks that have escaped previous analyses, which demonstrates the potential for automated detection of speculative leaks.

Acknowledgment

We would like to thank Saar Amar, Amaury Chamayou, Manuel Costa, Cédric Fournet, Istvan Haller, and Stavros Volos for discussions and support. We'd like to thank the anonymous reviewers and Scott. D. Constable for feedback on the paper. This work was partially supported by the Madrid regional government under the project S2018/TCS-4339 BLOQUES-CM, by the Spanish Ministry of Science, Innovation, and University under the projects RTI2018-102043-B-I00 SCUM and TED2021-132464B-I00 PRODIGY and under the Ramón y Cajal grant RYC2021-032614-I, and by a gift from Intel Corporation.

REFERENCES

[1] 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 *S&P*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Usenix Security*, 2018.

[3] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*. IEEE, 2019.

[4] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad : Cross-privilege-boundary data sampling," in *CCS*, 2019.

[5] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated Exploit Program Generation for Hardware Security Verification," in *MICRO*, 2018.

[6] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[7] M. R. Fadiheh, D. Stoffel, C. W. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *DATE*, 2019.

[8] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis Background Superscalar Memory Architecture," in *Usenix Security*, 2020.

[9] Y. Xiao, Y. Zhang, and R. Teodorescu, "SpeechMiner: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *NDSS*, 2020.

[10] H. Nemati, R. Guanciale, P. Buiras, and A. Lindner, "Speculative Leakage in ARM Cortex-A53," *arXiv*, 2020.

[11] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *27th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, 2022.

[12] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.

[13] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *S&P*. IEEE, 2021.

[14] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *Journal of Cryptology*, 2010.

[15] Y. Yarom and K. Falkner, "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *Usenix Security*, 2014.

[16] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA*, 2006.

[17] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of Abstract Side-Channel Models for Computer Architectures," in *CAV*, 2020.

[18] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *MICRO-54*. ACM, 2021.

[19] M. Guarnieri, B. Köpf, J. Morales, J. Reineke, and A. Sanchez, "Spectector: Principled Detection of Speculative Information Flows," in *S&P*. IEEE, 2020.

[20] X. Fabian, M. Patrignani, and M. Guarnieri, "Automatic Detection of Speculative Execution Combinations," in *CCS*, 2022.

[21] Pawel Wieczorkiewicz, "The AMD Branch (Mis)predictor: Just Set it and Forget it!" https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it, 2022, accessed: November, 2022.

[22] J. Wikner and K. Razavi, "Addendum to RETBLEED: Arbitrary Speculative Code Execution with Return Instructions," in *Usenix Security*, 2020.

[23] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data," in *ISCA*, 2021.

[24] W. Xiong, S. Katzenbeisser, and J. Szefer, "Leaking information through cache lru states in commercial processors and secure caches," *IEEE Transactions on Computers*, vol. 70, no. 4, 2021.

[25] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke, "Flushgeist: Cache Leaks from Beyond the Flush," *arXiv*, 2020.

[26] C. Canella, J. Van 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[27] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical Foundations for Spectre Defenses," in *S&P*. IEEE, 2022.

[28] C. Li and J.-L. Gaudiot, "Detecting spectre attacks using hardware performance counters," *IEEE Transactions on Computers*, 2022.

[29] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," in *Usenix Security*, 2021.

[30] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.

[31] H. P. de León and J. Kinder, "Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks," in *S&P*. IEEE, 2022.

[32] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *ISCA*. ACM, 2022.

[33] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *ASPLOS*, 2017.

APPENDIX

A. Subsets of x86-64 Used in Evaluation

In the main testing campaign (§II) and in the evaluation (§V), most of the experiments were performed on subsets of the x86-64 ISA. These particular subsets were selected because together they form a complete base x86-64 instruction set, excluding those instructions that are not (yet) supported by Revizor. Specifically, we excluded system instructions (e.g., `SYSCALL`), instructions incorrectly emulated by Unicorn (e.g., `ROR`), and control-flow instructions for which no contracts are available (`RET`, `CALL`, and indirect jumps). We omitted ISA extensions like AVX or x87 because they require more complex test case generation algorithms to avoid faults, also not yet supported by Revizor.

Each of the tested subsets constituted of several basic arithmetic instructions (including their versions with memory operands) and of several instructions that are unique to the given subset. The exact instructions are as follows:

- `cond` (conditional branches): *`ADC`*, *`ADD`*, *`CMP`*, *`DEC`*, *`INC`*, *`NEG`*, *`SBB`*, *`SUB`*, `J*`, `LOOP*`, `JMP` (unconditional direct jump).
- `strn` (string operations): *`ADC`*, *`ADD`*, *`CMP`*, *`DEC`*, *`INC`*, *`NEG`*, *`SBB`*, *`SUB`*, `CLC`, `CLD`, `CMC`, `LAHF`, `LOCK`, `REPE`, `REPNE`, `SAHF`, `SCASB`, `SCASD`, `SCASW`, `STC`, `STD`.
- `dmul` (division and multiplication): *`ADC`*, *`ADD`*, *`CMP`*, *`DEC`*, *`INC`*, *`NEG`*, *`SBB`*, *`SUB`*, `DIV`, `IMUL`, `MUL`.
- `flag` (operations on flags): *`ADC`*, *`ADD`*, *`CMP`*, *`DEC`*, *`INC`*, *`NEG`*, *`SBB`*, *`SUB`*, `CLC`, `CLD`, `CMC`, `LAHF`, `SAHF`, `STC`, `STD`.
- `lock` (atomics with LOCK prefix): *`ADC`*, *`ADD`*, *`CMP`*, *`DEC`*, *`INC`*, *`NEG`*, *`SBB`*, *`SUB`*, `LOCK ADC`, `LOCK ADD`, `LOCK CMP`, `LOCK DEC`, `LOCK INC`, `LOCK NEG`, `LOCK SBB`, `LOCK SUB`, `LOCK BSF`, `LOCK BSR`, `LOCK BT`, `LOCK BTC`, `LOCK BTR`, `LOCK BTS`, `LOCK NOT`, `LOCK OR`, `LOCK TEST`, `LOCK XOR`.

- `atom` (atomics without LOCK prefix): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, CMPXCHG, XADD, LOCK CMPXCHG, LOCK XADD.
- `dxfr` (data transfer): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, BSWAP, MOV, MOVSX, MOVZX, XCHG.
- `setc` (conditional byte set): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, SET*.
- `nop` (NOP instructions): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, NOP.
- `logi` (logical operations): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, AND, NOT, OR, TEST, XOR.
- `conv` (data type conversion): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, CBW, CDQ, CWD, CWDE.
- `cmov` (conditional moves): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, CMOV*.
- `bit` (bit test and bit scan): *ADC, ADD, CMP, DEC, INC, NEG, SBB, SUB*, BSF, BSR, BT, BTC, BTR, BTS.

B. Example: A walk through the discovery of SCO

In this example, we demonstrate the process of discovering a speculative leak with our tool by showing how we found SCO.

As a part of the main fuzzing campaign (Figure 1), we tested the `strn` subset of x86-64 on Intel Xeon E-2288G against the `CT-SEQ` contract. This contract permits side-channel information leakage through caches, but only for non-speculatively executed instructions. Accordingly, if Revizor finds an instance of *any* speculative leakage while testing against this contract, Revizor will report it as a violation.

To set this experiment up, we wrote the following configuration file:

```
1 instruction_categories:
2   - BASE-BINARY
3   - BASE-STRINGOP
4 contract_observation_clause: ct
5 contract_execution_clause:
6   - seq
7 enable_speculation_filter: true
8 enable_observation_filter: true
9 inputs_per_class: 2
```

Here, lines 1–3 select the `strn` subset; lines 4–6 tell Revizor to test against `CT-SEQ` contract; lines 7 and 8 enable the speculation and the observation filters (§IV-B and §IV-C); and line 9 tells CIG (§IV-D) to create two inputs for each input class.

We passed this configuration file to our modified version of Revizor, and launched a testing campaign with 100'000 programs, each tested with 100 inputs:

```
./cli.py fuzz -c conf.yaml -n 100000 -i 100
```

Already after starting the testing process, we saw signs of speculation: An unusually high number of test cases passed through the filters. (Figure 6 illustrates this observation.)

After about three hours of testing, Revizor detected a violation. At this point, we knew that we have discovered a new speculation type—there has been previously no speculative leak reported within this subset of instructions. However, we did not know the source of leakage.

From Revizor, we received a test case—a program and a sequence of inputs—that witnessed an unexpected information leakage. As the program was generated randomly, it contained many irrelevant instructions, which complicated the investigation. Therefore, we passed the program to Revizor's automatic minimizer, and it returned the following:

```
1 CLD  # instrumentation
2 SUB CL, DL
3 AND RSI, 0b1111111111111 # instrumentation
4 SUB RAX, -1904627778
5 NEG AL
6 CMP EDX, -122
7 AND RBX, 0b1111111111111 # instrumentation
8 SBB byte ptr [R14 + RBX], 111
9 CMP SI, 117
10 AND RDI, 0b1111111111111 # instrumentation
11 ADD RDI, R14 # instrumentation
12 AND RCX, 0xff # instrumentation
13 ADD RCX, 1 # instrumentation
14 REPNE SCASD
```

Next, we tried to find out which specific instruction within this program triggers speculation. For this, we removed one instruction at a time, executed the test case on Revizor, and checked if it still passed the speculation filter. The culprit turned out to be `REPNE SCASD` at line 14.

From there on, we manually reverse-engineered the mechanism behind the speculation of `SCAS`, and it led to the discovery of SCO.

C. Configurations Used in Evaluation

The following is a detailed description of the testing configurations used in §V-B and Figure 5. The names of the ISA subsets are described in Appendix A.

Configuration Spectre V1:

- ISA subsets: `cond`, `nop`, `bit`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`
- Target contract: CT-SEQ
- Spectre V4 patch enabled: True
- Microcode assists permitted: False

The configuration can surface V1 because the instruction set contains conditional branches. In principle, this configuration can also detect V1-Var, yet the probability of detecting V1-Var is much lower compared to V1, thus it has only a minor impact on the measurement results.

The configuration cannot surface V4 because the corresponding patch is enabled; LVI—because microcode assists are not permitted; ZDI—because the subsets do not include divisions; SCO—because the subsets do not include string operations.

Configuration Spectre V4:

- ISA subsets: `nop`, `bit`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`

- Target contract: CT-SEQ
- Spectre V4 patch enabled: False
- Microcode assists permitted: False

The configuration can surface V4 because the instruction set contains memory accesses and the patch is disabled. The configuration cannot surface V1 and V1-Var because the instruction set does not contain branches; LVI—because microcode assists are not permitted; ZDI—because subsets do not include divisions; SCO—because subsets do not include string operations.

Configuration LVI-Null:

- ISA subsets: `nop`, `bit`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`
- Target contract: CT-SEQ
- Spectre V4 patch enabled: True
- Microcode assists permitted: True

The configuration can surface LVI because microcode assists are permitted. The configuration cannot surface V1 and V1-Var because the instruction set does not contain branches; V4—because the corresponding patch is enabled; ZDI—because subsets do not include divisions; SCO—because subsets do not include string operations.

Configuration V1-Var:

- ISA subsets: `nop`, `bit`, `cond`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`
- Target contract: CT-COND
- Spectre V4 patch enabled: True
- Microcode assists permitted: False

The configuration can surface V1-Var because the instruction set contains conditional branches and variable-latency instructions. The configuration cannot surface V4 because the corresponding patch is enabled; LVI—because microcode assists are not permitted; V1 is not reported because the target contract (CT-COND) permits conditional branch misprediction; ZDI—because subsets do not include divisions; SCO—because subsets do not include string operations.

Configuration ZDI:

- ISA subsets: `dmul`, `nop`, `bit`, `cond`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`
- Target contract: CT-SEQ
- Spectre V4 patch enabled: True
- Microcode assists permitted: False

The configuration can surface ZDI because the instruction set contains 64-bit divisions. The configuration cannot surface V1 and V1-Var because the instruction set does not contain branches; V4—because the corresponding patch is enabled; LVI—because microcode assists are not permitted; SCO—because subsets do not include string operations.

Configuration SCO:

- ISA subsets: `strn`, `nop`, `bit`, `cond`, `cmov`, `conv`, `dxfr`, `flag`, `setc`, `logi`
- Target contract: CT-SEQ
- Spectre V4 patch enabled: True
- Microcode assists permitted: False

The configuration can surface SCO because the instruction set contains string operations. The configuration cannot surface V1 and V1-Var because the instruction set does not contain branches; V4—because the corresponding patch is enabled; LVI—because microcode assists are not permitted; ZDI—because subsets do not include divisions.