Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions

Jana Hofmann, *Azure Research, Microsoft;* Emanuele Vannacci, *Vrije Universiteit Amsterdam;* Cédric Fournet, Boris Köpf, and Oleksii Oleksenko, *Azure Research, Microsoft*

https://www.usenix.org/conference/usenixsecurity23/presentation/hofmann

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions

Jana Hofmann*[1], Emanuele Vannacci*†[2], Cédric Fournet[1], Boris Köpf[1], and Oleksii Oleksenko[1]

[1]Azure Research, Microsoft
[2]Vrije Universiteit Amsterdam

Abstract

Microarchitectural leakage models provide effective tools to prevent vulnerabilities such as Spectre and Meltdown via secure co-design: For software, they provide a foundation for secure compilation and verification; for hardware, they provide a target specification to test and verify against.

Unfortunately, existing leakage models are severely limited: None of them covers CPU exceptions, which are essential to implement security abstractions such as virtualization and memory protection, and which are the source of critical vulnerabilities such as Meltdown, MDS, and Foreshadow.

In this paper, we provide the first leakage models for CPU exceptions, together with new tools for testing black-box CPUs against them. We run extensive experiments and successively refine these models, until we precisely capture the leakage for a representative subset of exceptions on four different x86 microarchitectures.

In the process, we contradict, refine, and corroborate a large number of findings from prior work, and we uncover three novel transient leaks affecting stores to non-canonical addresses, stores to read-only memory, and divisions by zero.

1 Introduction

Instruction set architectures (ISA) specify the functional behavior of CPUs but not their implementation details. This layer of abstraction leaves room for various microarchitectural optimizations, which often introduce hard-to-detect security vulnerabilities. A prominent example are microarchitectural effects of transient execution, which were exploited by the Spectre [30] and Meltdown [31] attacks.

Microarchitectural Leakage Models. Leakage models address this problem by augmenting the ISA with a specification of all observable side-effects of executing a program [12, 16, 22]. For example, the *constant-time programming model* specifies that an attacker can observe the control flow as well as memory access patterns of the program execution. Similarly to the ISA, leakage models serve two purposes:

- For software, leakage models specify the expected leakage during program execution, and thereby enable principled development of secure software such as cryptographic libraries. As summarized in [12], leakage models have been applied to security testing and verification [16, 22, 35, 37], and to secure compilation [39].
- For hardware, leakage models document the microarchitectural side-effects of instructions without disclosing details of their hardware implementation. They have been applied to validate mechanisms for secure speculation [13, 22, 52], and to test commercial CPUs for unexpected leakage [7, 36–38].

Existing leakage models, however, are incomplete. They do *not* model the possibility that instructions trigger exceptions, due, for instance, to invalid arguments, page faults, or failed permission checks. This is a serious limitation in practice: CPU exceptions are instrumental in many security mechanisms such as virtualization and OS access control, and they are involved in critical vulnerabilities such as Meltdown, MDS, LVI, and Foreshadow. In this paper, we develop and validate the first formal leakage models for CPU exceptions.

Challenges. Coming up with good models for CPU exceptions is challenging due to their diverse microarchitectural behavior: Compared with, e.g., branch speculation [12], transient executions upon exceptions are idiosyncratic and not well documented. The key experimental challenge is to systematically infer and document the leakage behavior of different exceptions and microarchitectures. The key conceptual challenge is to define flexible formal models that concisely capture these findings, especially when inferring the exact microarchitectural behavior is impossible.

Approach. We use an empirical approach to build leakage models for black-box CPUs: We (i) start out with a minimal leakage model; (ii) falsify the model using automatic testing; (iii) (manually) refine the model to account for the obtained counterexamples; and (iv) repeat this process until no further counterexamples are found.

*Equal contribution first authors.
†Work partially done while at Microsoft Research Cambridge.

We develop two techniques to make this approach work: a family of increasingly precise leakage models for exceptions and a tool for testing black-box CPUs against these models. We present both techniques in more detail, and then summarize our evaluation on four different x86 microarchitectures.

Leakage Models for CPU Exceptions. We rely on *leakage contracts* [21], which formalize leakage as an ISA execution semantics supplemented with observation labels. We build models for three increasingly complex behaviors:

1. The CPU executes instructions (including exceptions) and leaks their observation labels in program order.
2. The CPU executes transiently after an exception, for a bounded number of steps, skipping any instruction that depends on the missing result of the faulting instruction.
3. The CPU executes transiently after an exception, and the missing result of the faulting instruction is replaced with a speculative value, which may be leaked by subsequent instructions. We distinguish two variants of this behavior: (a) the exact speculative value is known; or (b) it is unknown, and instead we specify which parts of the state it depends on, such as the operands of the faulting instruction.

Our experiments confirm that these contracts are expressive enough to capture—and distinguish between—a wide range of speculative behaviors.

Testing CPUs against Leakage Models. We rely on *model-based relational testing (MRT)* [35, 37], which generates random programs and compares the leakage they cause on the CPU with the leakage predicted by the model. Our tool extends Revizor [37], an open-source implementation of MRT for x86 CPUs. Extending Revizor involves two challenges:

- *Controlling exceptions.* To raise exceptions deterministically and consistently between the model and the CPU, we create a sandbox environment. This enables us to control the exceptions triggered during execution of test programs, and to handle them with minimal microarchitectural noise.
- *Managing microarchitectural state.* MRT requires pseudo-randomization of the initial microarchitectural state: it has to be deterministic to support robust measurements and diverse to trigger different transient leaks. We develop an initialization algorithm that targets both issues.

While our design is general, our current implementation supports three classes of exceptions: memory errors (including page faults and microcode assists), opcode-based errors (including invalid modes and undefined opcodes), and computational errors (division).

Evaluation on x86 CPUs. We test 4 microarchitectures (Intel Kaby Lake and Coffee Lake, and AMD Zen+ and Zen 3) against 5 leakage models. We investigate 12 variants of exceptions and microcode assists, including those known to cause leakage, e.g., page faults, division errors, and assists triggered by *Accessed* and *Dirty* bits in the page table entry. We run a separate experiment for 24 hours for each combination of CPU, exception variant, and leakage model.

While conducting these experiments, we uncovered undocumented leakage behavior, summarized below.

- On Zen+, we identify a new speculative leak: Division-by-zero can transiently return a result that depends on the result of a *prior* division operation. This hidden state persists across serializing instructions and system calls, but not across privilege boundaries. This finding contradicts previous claims that on AMD processors, division exceptions do not speculatively forward data to dependent instructions [4].
- On Kaby Lake and Coffee Lake, we identify new ways to trigger MDS: When a read-modify-write operation (e.g., an in-memory increment) accesses read-only memory, the faulting instruction triggers MDS. This behavior has so far only been documented for faulting loads; here, the *store* faults.
- On Coffee Lake we identify a new variant of store forwarding: Stores to a non-canonical address can be forwarded to subsequent loads from canonical versions of that address.[1]

We discovered these leaks by analyzing violations of models that formalize claims from the literature. Whether they cause practical vulnerabilities is out of scope of this paper.

Summary of Contributions. We define the first formal leakage models for CPU exceptions and propose an empirical approach to infer the right model using model-based testing. To implement this approach, we develop a tool for testing black-box CPUs against these models. We evaluate this approach on x86 CPUs and uncover three new speculative leaks.

Responsible Disclosure. We reported our observations to AMD and Intel, who acknowledged our findings and investigated their security impact. Intel decided that no new mitigations are required; AMD issued CVE-2023-20588 and plans to publish a security bulletin with mitigation information.

Availability. The source code, experiments, and executable leakage models are available at
`https://github.com/microsoft/sca-fuzzer`

Structure of this Paper. §2 provides background on leakage contracts and MRT; §3 describes our platform for testing CPU exceptions against leakage contracts; §4 defines a baseline contract; §5 §6 and §7 define leakage contracts for out-of-order execution and value speculation; §8 reports our experiments on different x86 microarchitectures; §9 presents applications of our contracts to software development; §10 discusses the architectural and microarchitectural coverage of our approach; §11 reviews related work, and §12 concludes.

2 Background

This section provides background on modeling and testing microarchitectural leakage. We first introduce leakage contracts [22], a modeling framework that captures microarchitectural leaks at the ISA level. We then explain how contracts

[1] A similar behavior was previously reported on AMD [34], albeit with canonical stores being forwarded to non-canonical loads.

can be used to detect unknown vulnerabilities and give an overview of Revizor [37], a tool that implements model-based relational testing for black-box CPUs.

2.1 Modeling Microarchitectural Leakage

As a CPU executes a program, it makes various changes to its microarchitectural state. Some of these changes are observable by an attacker via side channels [45, 51]. We call such observable changes a *hardware trace*. On an abstract level, hardware traces are the result of an unknown function *Measure* operating on a program p, a program input σ, and an initial microarchitectural state μ.

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

The input σ sets the initial contents of the registers and the memory, whereas μ controls, e.g., the initialization of caches and buffers. We keep p, σ, and μ abstract for now and delay their formal definition to Section 4.1.

A program leaks information when its hardware traces depend on its input (which potentially contains secrets) as this enables an attacker to distinguish between different inputs by observing different traces.

To develop software measures against such leakage, we must predict how a program execution affects the hardware traces. For example, it is well known that the control flow and the addresses of memory operations affect shared caches.

Contracts [22] have been proposed as an abstract, ISA-level specification of the expected leakage. As a counterpart for the function *Measure*, the function *Contract* predicts any information the CPU may leak as it executes a program p on input σ. In contrast to *Measure*, however, it abstracts away the microarchitectural details:

$$CTrace = Contract(p, \sigma)$$

We call the observations predicted by a contract a *contract trace*. Contracts are defined by augmenting the ISA with (a) an *execution semantics* that provides an abstraction of how the CPU executes a program, and (b) *observation labels* that describe the information disclosed by each instruction as it is executed. We delay the formal definition of contracts to Section 4.1 as well.

In this work, we use '*CT*' observation labels to model leakage via cache side channels. These labels are based on the constant-time programming paradigm; they expose all control flow decisions (branches, jumps, calls, ...) and the addresses of all memory accesses (loads and stores).

Example 1. *CT-SEQ* [22] is our baseline contract; it models leakage through cache side channels during sequential, in-order execution. Its execution semantics simply follows the architecture, one instruction at a time, without speculation or transient execution.

Example 2. Leaks may also occur during *transient execution*, i.e., the speculative execution of instructions that are never retired [30]. We call them *speculative leaks*. A leakage contract that captures Spectre V1 is obtained by choosing an execution semantics that executes mispredicted branches (and thus collects transient *CT* observations labels) for a certain number of steps before continuing with the correct branch.

To experimentally validate contracts, or uncover new leaks, we compare the leakage predicted by contract traces with the leakage observed in hardware traces on a CPU under test.

Definition 1 (Contract Violation [22]). A CPU violates a contract if there exists a program p, a pair of inputs (σ, σ'), and a microarchitectural state μ such that $Contract(p, \sigma) = Contract(p, \sigma')$ and $Measure(p, \sigma, \mu) \neq Measure(p, \sigma', \mu)$.

We call the tuple $(p, \sigma, \sigma', \mu)$ from Definition 1 a *counterexample*: The contract predicts that an attacker cannot distinguish between inputs σ and σ', whereas the tuple witnesses that this is not true for some μ. A counterexample is an unexpected leak, which may or may not be a practical vulnerability. Making this distinction requires a manual security analysis, which is out of scope of this approach.

2.2 Testing against Leakage Models

Model-based tools [7, 35, 37] use contracts to systematically search for unexpected leaks in CPUs. In this paper, we base our work on one such tool, namely Revizor [37].

Revizor searches for contract counterexamples by generating random test cases. A *test case* consists of a random program p and randomly generated inputs $\sigma_0, \sigma_1, \ldots, \sigma_n$. The program is generated by creating a random control-flow graph and filling it with instructions sampled from a predefined pool of instructions; the inputs are generated by filling the memory and the CPU registers with values generated by a pseudo-random number generator. For each test case, Revizor collects both the contract trace and the hardware trace:

- Contract traces are collected by the *contract model*, an executable version of the *Contract* function. The contract model is implemented with an ISA emulator (Unicorn [40], based on QEMU), which is modified to follow the execution semantics of the contract and to record its observation labels.
- Hardware traces are collected by the *executor*, which implements the *Measure* function by executing program p on input σ on the target CPU. As it is not possible to directly set the microarchitectural state of a black-box CPU, the executor instead performs warm-up computation to initialize it to some deterministic microarchitectural state μ. The executor obtains hardware traces by monitoring the microarchitectural changes caused by each execution via a cache side-channel attack.

After collecting traces, Revizor checks for a contract violation according to Definition 1 (a program and two inputs for which the contract traces agree but the hardware traces differ) and reports them to the user.

3 Tooling for Testing Exception Leakage

We extend Revizor with a sandbox for deterministic triggering and handling of exceptions, adapt the test-case generation, add hooks for building leakage models for exceptions, and develop techniques for a stable measurement environment. These changes amount to ~2600 new lines of code.

Naming Convention. There are various names for the events that cause a CPU to redirect its execution to the exception handler, such as exceptions, traps, and faults. In this paper, we refer to all of them as exceptions.

Scope. We focus on so-called synchronous exceptions, which are directly caused by the executing thread. They are commonly used as a mechanism for enforcing security invariants, which implies that unexpected leaks might expose sensitive data. Our goal is to model the leakage of the CPU due to transient execution in the time between the triggering of the exception and the execution of the exception handling code. The leakage due to the code of the exception handler itself is out of scope (and presumably covered by contracts that are not specific to exceptions).

3.1 Sandbox for Exceptions

Model-based relational testing requires exceptions to be raised consistently by the leakage model and by the CPU under test. However, the default exception handling mechanisms in both the model and the executor do not meet this requirement: The model will terminate upon an exception, while the executor will jump to the OS-provided exception handler. To avoid such inconsistencies, we create a sandbox environment for testing exceptions that is *uniform* and *controlled*. Uniform means that exceptions are thrown and handled identically on the model and the executor. Controlled means that we can determine which exceptions will be triggered during the tests, despite the fact that test cases are randomly generated. In the following, we explain how we achieve these goals.

3.1.1 Exception Handlers

As we consider the exception handling code out of scope, we make all handlers empty; they simply terminate the test case.

- On the executor side, we overwrite the OS-provided Interrupt Descriptor Table (IDT) for the duration of the tests. For each tested exception, we set the corresponding IDT entry to the exit address of the test case. Hence, upon an exception, the CPU pushes the program counter (PC) onto the stack, reads the corresponding handler's address from the IDT, and then jumps to the executor, which collects the resulting hardware traces. Such direct management of the IDT is possible because Revizor's executor runs in kernel mode.
- On the model side, we add a hook function executed after the model throws an exception. To mimic the CPU pushing the PC on the stack and dereferencing the IDT entry for this exception, the function records the PC and the exception ID in the contract trace, then it terminates the test case.

Handling Assists. In this paper, we also test speculative leaks upon microcode assists, which are effectively exceptions handled by firmware. Our implementation supports assists triggered by the *Accessed* and *Dirty* bits of the page table. Their firmware handler sets the corresponding bit and re-executes the faulting instruction. As we cannot modify handlers in the firmware, we include their effect in the model: We clear the permissions on the page that triggered the fault (see §3.1.2) and re-execute the instruction.

3.1.2 Exception Triggers

We next discuss how to deterministically trigger the three classes of exceptions we consider in this paper.

To trigger *memory-based errors* (e.g., page faults), we manage the permissions on the memory sandbox. The working memory of a test case consists of two pages initialized with the values from input σ. The test case is instrumented such that all memory accesses are forced into these pages. The properties of the second page are configurable to create a controlled environment for triggering memory exceptions while still permitting a broad range of interactions between faulting and other instructions. We implement this design as follows:

- In the executor, we create an interface to modify the page table entry (PTE) of the second page (similar to the design of Transynther [33]). For example, if we want to test writes to read-only memory, we would clear the `RW` (read-write) bit in the page table. With this configuration, any randomly generated test case will experience a page fault whenever it attempts to store a value in the second page.
- In the model, we emulate the page table configuration by making the second page either inaccessible or read-only. This allows us to trigger the faults uniformly with the executor, without having to represent the page table structure within σ.

To trigger *opcode-based errors*, we simply add the corresponding opcodes/instructions to the list of instructions used by the generator to create test cases. For *computational errors*, we rely entirely on the fact that inputs σ are random, hence some of the computations inherently experience errors (e.g., divisions by zero).

3.2 Test Case Generation

Next, we explain our adaptation of the program generation and input generation of Revizor (described in §2.2). We refer to §8.1 for the specific configurations used in each experiment.

- Since we focus on implicit control flows due to exceptions, the control-flow-graph generator is configured to produce straight-line code.
- The pool of instructions always includes the subset of x86-64 ISA supported by Revizor excluding control-flow instructions and conditional moves. This pool is extended with

instructions that are specific to each of the exceptions we test (see Appendix A for a complete list).

- The input generator is adapted to fill the two pages of our sandbox and set their PTE (see §3.1).
- To test exceptions caused by access attempts to non-canonical addresses (i.e., virtual addresses whose bits 63 to the most significant bit are *not* all zeros or all ones), we inject instructions flipping random higher-order bits of addresses used in memory accesses.

3.3 Hook for Leakage Models

From the software perspective, a transient information leak upon an exception happens *after* the exception was thrown and *before* the exception is handled. Therefore, to build a contract for such a leak, we need a hook function executed between these events.

The emulator used by Revizor to implement contracts, Unicorn [40], only provides hooks into exception handling. This is too late; an exception may have already corrupted the emulator's state. To avoid this, we need to intercept the state before the instruction faults, not after.

We achieve this by checkpointing the emulator's state before every instruction. If an instruction faults, we roll back to the last checkpoint and execute the hook that implements the contract's transient semantics. Afterwards, we either call the exception handler or, in case of a microcode assist, re-execute the instruction without the fault (see §3.1.1).

3.4 Initializing the Microarchitectural State

Model-based relational testing requires pseudo-randomization of the microarchitectural state μ: The state has to be deterministic to check traces based on Definition 1, yet it also has to be diverse to trigger different transient leaks.

Revizor's current mechanism for pseudo-randomizing μ (§2.1) has limitations for testing exceptions: (1) it primes (and hence clears) the cache before execution (this prevents leaks such as Foreshadow [8], which is triggered by page faults on L1-cached addresses); (2) it introduces nondeterminism in the hardware trace when transient instructions access memory and return a value used to access memory again. We address these issues with the following algorithm for collecting hardware traces.

1. Invalidate caches (`wbinv`) and flush buffers (`verw`).
2. Load input σ into sandbox memory.
3. Set PTE as described in §3.1.2 and invalidate TLB.
4. Prime $m < 64$ L1D cache sets as a part of Prime+Probe.
5. Stabilize the CPU pipeline by executing a long sequence of memory fences (`mfence`).
6. Execute the test case p.
7. Probe the m cache sets and return the hardware trace.

This algorithm reduces nondeterminism by isolating the measurements from the side effects of previously-executed code (steps 1–5) and by making the cache state dependent on the input σ (steps 1–2).

A broader diversity of microarchitectural states is achieved by steps 4–5: Priming only m (here: $m = 60$) out of 64 L1D cache sets allows the data to remain cached in $64 - m$ sets (we call this mode *Partial Prime+Probe*). This enables four kinds of memory accesses to appear in a test case: no exception + L1D miss, no exception + L1D hit, exception + L1D miss, and exception + L1D hit.

4 A Baseline Model for Exceptions

As a baseline, we extend the *CT-SEQ* contract defined by [22] to support exceptions.

4.1 Baseline Model without Exceptions

Syntax. We formally describe programs with a toy ISA language μASM based on [21] and adapted to our needs.

$$
\begin{array}{lll}
\textbf{Basic Types} & & \\
\textit{(Registers)} & x & \in \textit{Regs} \\
\textit{(Values)} & n, \ell & \in \textit{Vals} = \mathbb{N} \cup \{\bot\} \\
\textbf{Syntax} & & \\
\textit{(Expressions)} & e & ::= n \mid x \mid \ominus e \mid e_1 \otimes e_2 \mid \textbf{ite}(e_1, e_2, e_3) \\
\textit{(Instructions)} & i & ::= x \leftarrow e \mid \textbf{load}\ x, e \mid \textbf{store}\ x, e \\
 & & \quad \mid \textbf{spbarr} \mid \textbf{return} \mid \textbf{invalid} \\
\textit{(Programs)} & p & ::= i \mid p_1; p_2
\end{array}
$$

Expressions e are built from register variables x, constants n, labels ℓ, as well as unary and binary operators and conditionals. Instructions i comprise assignments, loads, stores, a speculation barrier, returns, and an invalid instruction. Programs are lists of instructions. Compared to [21], our language contains a return instruction to return from an exception handler. Additionally, the invalid instruction always causes an exception (to model, e.g., invalid opcodes). As our focus is on analyzing the leakage caused by exceptions, the language does not feature branching instructions.

States σ range over tuples $\langle m, a \rangle$ of a memory $m : \mathbb{N} \to \textit{Vals}$ and a register assignment $a : \textit{Regs} \to \textit{Vals}$. We write $\sigma(x)$ instead of $a(x)$ and $\sigma(n)$ for $m(n)$. When executing program p, the program counter **pc** points to the current instruction, to which we refer with $p(\sigma(\textbf{pc}))$. We write $\sigma[x \mapsto v]$ to assign value v to register x in state σ.

Contracts. A contract defines the execution of a single instruction by a relation $\sigma \xrightarrow{\tau} \sigma'$ that transforms the architectural state σ to σ' and produces an observation label τ.

CT-SEQ is a contract that describes an in-order (sequential) semantics with constant-time observation labels. This means that the contract records in the observation label every memory access and every control-flow change. As an example,

we recall below their rule for the load instruction, where $[\![e]\!]_\sigma$ denotes the value of expression e in state σ (the definition is given in Appendix C).

$$\textsc{Load}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{load}\ x, e \quad x \neq \mathbf{pc} \quad n = [\![e]\!]_\sigma}{\sigma \xrightarrow[\text{ct}]{\texttt{load}\ n}{}^{\text{seq}} \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1, x \mapsto \sigma(n)]}$$

The state is updated by assigning the value at address n to register x and setting **pc** to the next instruction. The observation label `load` n states that the attacker can observe the address. We refer to [22] for a detailed description of the contract.

A *run* of program p on input σ_0 is the longest sequence $\sigma_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} \sigma_n$, where each τ_i is a label such as `load` n. The function $Contract(p, \sigma_0)$ returns the trace $\tau_1, \ldots, \tau_n$.

4.2 Baseline Model with Exceptions

We extend *CT-SEQ* to reason about exception triggers and handlers. To do so, we first formalize exception conditions.

Exception Conditions. The ISA defines the conditions upon which each instruction can trigger an exception. When these conditions are satisfied, the CPU directs execution to corresponding error handling code. We model the conditions under which exceptions are triggered as a function $EC(\sigma, p)$ that takes as input the architectural state σ and the program p. If $p(\sigma(\mathbf{pc}))$ faults, $EC(\sigma, p)$ returns the location ℓ of the corresponding handler; otherwise it returns $\perp$.

Example 3. To model page faults, we use a function *mapped* that indicates whether an address is mapped in the virtual address space. Let ℓ be the address of the page fault handler. For $p(\sigma(\mathbf{pc})) = \mathbf{load}\ x, e$, we set

$$EC(\sigma, p) = \begin{cases} \perp & \text{if } mapped([\![e]\!]_\sigma) \\ \ell & \text{otherwise} \end{cases}$$

Contracts for Exceptions. When an exception is triggered, the control flow is diverted to the exception handler, whose code may execute a **return** instruction to return to the faulting instruction. We extend *CT-SEQ* with two new rules for exceptions (presented in Figure 1): one that formalizes faulting instructions and one for the return instruction. This is enabled by augmenting the state σ with a return stack r. Faithful to the constant-time paradigm, the new rules expose all control flow changes. The rest of the contract (given in Appendix C) is as in [22], except for the return stack in the state, and for an additional condition $EC(\sigma, p) = \perp$ in the other rules modelling normal (unexceptional) execution. In the following sections, σ always refers to the triple $\langle m, a, r \rangle$.

Implementation. As *CT-SEQ* does not model transient execution, its implementation simply redirects the control flow to the exception handler upon an exception. The handlers our models define emulate a minimal CPU handler (see §3.1.1).

$$\textsc{Exception}\quad \frac{EC(\sigma, p) = \ell}{\langle \sigma, r \rangle \xrightarrow[\text{ct}]{\texttt{exc, pc}\ \ell}{}^{\text{seq}} \langle \sigma[\mathbf{pc} \mapsto \ell], \sigma(\mathbf{pc}) \cdot r \rangle}$$

$$\textsc{Return}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{return} \quad EC(\sigma, p) = \perp}{\langle \sigma, \ell \cdot r \rangle \xrightarrow[\text{ct}]{\texttt{pc}\ \ell}{}^{\text{seq}} \langle \sigma[\mathbf{pc} \mapsto \ell], r \rangle}$$

Figure 1: Rules for exceptions in *CT-SEQ*. Upon an exception, the program jumps to the handler at location ℓ and pushes the location of the faulting instruction to the return stack r. The **return** instruction resumes from the location at the top of the stack. Both rules leak the new program counter ℓ together with an exception label (`exc`) in the case of EXCEPTION.

5 Model for Transient Execution

The baseline leakage model of §4 assumes that the CPU executes instructions (including exceptions) in program order. Instead, modern CPUs execute data-independent instructions in parallel and out of program order. In particular, when an instruction faults, a number of instructions are executed transiently and leave microarchitectural traces [31, 34] before the CPU finally handles the exception and flushes the pipeline. This section presents the *CT-DH* contract, which accounts for the delayed handling of exceptions.

Modeling Goals. *CT-DH* captures the microarchitectural traces of transiently executing instructions that follow the faulting instruction. The key assumption of this contract is that instructions that depend on the result of the faulting instruction are *not* transiently executed.

Design. *CT-DH* behaves as *CT-SEQ* as long as no exception occurs. Upon an exception, *CT-DH* snapshots the current architectural state and transiently continues the execution of independent instructions. After w steps, the contract restores the saved architectural state and jumps to the exception handler. Hence, the contract discards the architectural effect of the transient execution while recording their microarchitectural effects in the observation trace. The transient window parameter w is bounded in practice by the size of the reorder buffer on an out-of-order CPU.

We implement this idea by lifting the contract semantics to operate on the top element of a *stack* of architectural states (similar to the branch speculation contract in [22]). We use this stack to push snapshots of the architectural state when an exception happens, and to pop snapshots after w transient execution steps. To skip instructions that depend on the result of the faulting instruction, *CT-DH* maintains the a D of registers whose values are unavailable. This set is initialized with the destination register of the faulting instruction. We then add to D any register that would be assigned by a skipped

$$\textsc{Step}\quad \frac{p(\sigma(\mathbf{pc}))=i \quad sregs(i)\cap D=\emptyset \quad EC(\sigma,p)=\bot \quad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}}\, \sigma'}{\langle \sigma,\omega+1,D\rangle\cdot s \xrightarrow{\tau}{}^{\text{dh}}_{\text{ct}}\langle \sigma',\omega,D\setminus dregs(i)\rangle\cdot s}$$

$$\textsc{Skip}\quad \frac{p(\sigma(\mathbf{pc}))=i \quad sregs(i)\cap D\neq\emptyset}{\langle \sigma,\omega+1,D\rangle\cdot s \rightsquigarrow^{\text{dh}}_{\text{ct}}\langle \sigma[\mathbf{pc}\mapsto\sigma(\mathbf{pc})+1],\omega,dregs(i)\cup D\rangle\cdot s}$$

$$\textsc{Exception}\quad \frac{p(\sigma(\mathbf{pc}))=i \quad EC(\sigma,p)=\ell \quad \sigma \xrightarrow{\text{exc, pc }\ell}{}^{\text{seq}}_{\text{ct}}\, \sigma'}{\langle \sigma,\infty,\emptyset\rangle \xrightarrow{\text{exc}}{}^{\text{dh}}_{\text{ct}}\langle \sigma[\mathbf{pc}\mapsto\sigma(\mathbf{pc})+1],w,dregs(i)\rangle\cdot\langle \sigma',\infty,\emptyset\rangle}$$

$$\textsc{Rollback}\quad \frac{}{\langle \sigma,0,D\rangle\cdot\langle \sigma',\omega',D'\rangle\cdot s \xrightarrow{\text{pc }\sigma'(\mathbf{pc})}{}^{\text{dh}}_{\text{ct}}\langle \sigma',\omega',D'\rangle\cdot s}$$

$$\textsc{TransientException}\quad \frac{p(\sigma(\mathbf{pc}))=i \quad sregs(i)\cap D=\emptyset \quad EC(\sigma,p)=\ell \quad \omega\neq\infty}{\langle \sigma,\omega+1,D\rangle\cdot s \xrightarrow{\text{exc}}{}^{\text{dh}}_{\text{ct}}\langle \sigma[\mathbf{pc}\mapsto\sigma(\mathbf{pc})+1],\omega,dregs(i)\cup D\rangle\cdot s}$$

Figure 2: *CT-DH* contract. Parameter ω (initially ∞) counts the remaining steps in the transient window; D (initially $\emptyset$) is the set of dependencies. $dregs(i)$ is the set of registers i re-assigns ($\{x\}$ for $i\in\{(\textbf{load}\ x,e),(x\leftarrow e)\}$ and $\emptyset$ otherwise). $sregs(i)$ is the set of registers on which i depends (all registers occurring in i except the registers in $dregs(i)$).

instruction, and remove from D any register assigned in a non-skipped instruction.

Formalization. The formal contract is given in Figure 2. If an instruction does not depend on registers in D and if no exception occurs, it is executed as in *CT-SEQ* (rule STEP), possibly removing the destination register from D. If the instruction depends on a register in D, it is skipped (rule SKIP). In case of an exception (rule EXCEPTION), the program is transiently executed for w steps, after which we continue with the exception handler (represented by σ'). The ROLLBACK rule is triggered after w transient steps; it reverts the architectural effects of the transient execution. Finally, TRANSIENTEXCEPTION describes exceptions during transient execution, which we model to not induce a jump to the exception handler. This translates the fact that exceptions trigger a machine clear resulting a pipeline flush, i.e., no instruction from the handler of the second exception will be executed. This behavior is explicitly documented by Intel [28] and has been observed for AMD machines as well [41].

Implementation. We implement *CT-DH* by extending the *CT-SEQ* implementation with snapshotting and dependency tracking. For the snapshotting, we use existing techniques in Revizor that realize speculation after conditional jumps [37]. For the dependency tracking, we augment the state with a dependency set that we update as described in Figure 2.

Discussion. The contract *CT-DH* defines a *deterministic* semantics. This may seem at odds with its use to describe the leakage of out-of-order execution, which is typically modeled using nondeterminism. The reason why a deterministic definition is adequate for this contract is that our experimental hardware traces also abstract from potential nondeterminism due to reorderings; they collect the cache sets that have been accessed, but not the ordering of their access.

6 Model for Value Speculation

The leakage model presented in §5 describes a CPU that transiently executes all instructions that do not depend on the destination register of the faulting instruction. Vulnerabilities such as Meltdown [8, 31] or MDS [11, 42, 46] demonstrate that sometimes CPUs *do* transiently execute instruction despite their dependencies on the faulting instruction. They rely instead on various kinds of speculative values, including stale values, constants, or values drawn from different of CPU buffers. We call these behaviors *value speculation* (whether or not they involve an explicit value predictor). This section defines *CT-VS*, a first leakage model for value speculation.

Modeling Goals. *CT-VS* assumes that the faulting instruction produces a transient result used by the following instructions, and that we know how this transient result is produced. An example is null-injection (LVI-NULL) [9], where a faulting load transiently returns zero, a common dummy value used, e.g., as a Meltdown patch [25].

Design. *CT-VS* behaves similarly to *CT-DH* and also uses the same snapshotting mechanism. There are two key differences:

1. It assigns to the destination register of the faulty instruction the value given by a new function $TV(\cdot)$, which may depend on the type of exception and the architecture.
2. Instead of skipping instructions that depend on the faulting instruction, it executes them in a transient state that depends on the value given by $TV(\cdot)$.

Example 4. To model an architecture vulnerable to LVI-NULL, we define $TV(\cdot)$ as a function that returns 0 for page faults triggered by load instructions

$$TV(\sigma,p,\ell)=0 \quad \text{if } p(\sigma(\mathbf{pc}))=\textbf{load}\ x,e$$

Here, $TV(\cdot)$ takes the state σ, program p, and exception handler location ℓ, and it returns the value to assign to the desti-

nation register (here x) of the faulting instruction. Passing the handler location enables $TV(\cdot)$ to return different values for instructions that may fault with different exceptions.

Formalization. As for *CT-DH*, we keep track of the speculation window, and we distinguish between exceptions thrown during normal execution and transient execution. If an exception occurs on an instruction that would update a register (**load** x,e or $x \leftarrow e$), we now assign x the value defined by $TV(\cdot)$. For the load instruction, the resulting rule for a (non-transient) exception is the following.

$$\textsc{Exception}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{load}\ x,e \quad EC(\sigma,p) = \ell \quad \sigma \xrightarrow{\texttt{exc, pc}\ \ell}{}^{\text{seq}}_{\text{ct}}\ \sigma' \quad TV(\sigma,p,\ell) = n}{\langle \sigma, \infty \rangle \xrightarrow{\texttt{exc}}{}^{\text{vs}}_{\text{ct}} \langle \sigma[x \mapsto n, \mathbf{pc} \mapsto \sigma(\mathbf{pc})+1], w \rangle \cdot \langle \sigma', \infty \rangle}$$

The TRANSIENTEXCEPTION rule is adapted similarly. The *CT-VS* contract also has rules ROLLBACK and STEP analogous to the *CT-DH* contract (but without the dependency set D). Similar rules can be defined for faulting instructions other than memory accesses.

Implementation. The snapshotting mechanisms for transient execution is implemented as in the *CT-DH* contract. To enable different implementations of $TV(\cdot)$, we define a different *CT-VS* contract class for each exception, which we discuss in §8.

7 Model for Unknown Value Speculation

The *CT-VS* contract assumes that we can define a function that predicts the transient values assigned by faulting instructions. For many exceptions, this transient value is unknown and may thus a priori contain *any* secret from the architectural state. In this section, we describe how to obtain meaningful contracts in the presence of unknown transient values—as long as we can identify what these values depend on.

Modeling Goals. Our goal is to express how dependencies of transient values on varying components of the architectural state change the possible microarchitectural leakage of delayed exception handling.

We use the code snippet displayed in Figure 3 as a motivating example to illustrate how information about these dependencies can be used for characterizing leakage of CPUs. For this, assume that the dividend a is a public constant, and that division by zero assigns an unknown transient value to register x. We distinguish between two cases:

1. If the transient value in x depends only on the (non-secret) operands of the division, the memory access in line 2 does not leak secret information.
2. If the transient value in x depends on additional, unknown components of the architectural state, the memory access in line 2 may leak their secret information.

In both cases, the memory access in line 3 leaks information from an unknown memory location.

```
1 x ← a div 0  # assigns unknown value to x
2 load y,x     # leaks x
3 load z,y     # leaks value from memory
```

Figure 3: Program demonstrating leakage through unknown value speculation. The first instruction attempts to divide the value stored in a by 0, which causes an exception and transiently assigns an unknown value to x. The next instructions are executed transiently. The first leaks the value of x and the second that of the location x points to.

In this section, we develop a contract that can bound the potential leakage of unknown speculation mechanisms and can thus distinguish between these two cases. This contract also gives insights for the placement of speculation barriers.

Design. The core idea behind the *CT-VS-Unknown* contract is to keep a record of all the values on which a transient value depends and to delay their exposure until some derivative of that transient value appears in a contract observation.[2]

We implement this idea using a form of forward taint tracking, where the taint consists of the set of all values a register may depend on. When a fault occurs, we initialize the taint of destination register. Depending on this initialization, we obtain contracts of varying permissiveness. For this paper, we consider two variants:

1. The *CT-VS-Ops* contract initializes the taint with the value of all source operands of the faulting instruction. For example, for a faulting division $x \leftarrow a$ **div** b, these are the values of a and b. A violation of *CT-VS-Ops* indicates that the CPU injects data beyond the operands into the destination.
2. The contract *CT-VS-All* initializes the taint with the entire architectural state (technically, we use its hash). It is the most permissive contract considered in this paper; it can only be violated by leaking data that is *not* contained in the current architectural state, e.g., some value from a different thread.

Formalization. The formalization of the dependency tracking algorithm is given in Appendix D. Here, we summarize the key features:

- We taint registers and addresses with a set $\{d_1, d_2, \ldots\}$ of dependencies. Each dependency d consists of an instruction i, another location l (l can be a register or a memory address), and the value v that l had when executing i.
- When a fault occurs, the taint of the destination register is initialized either with the source operands (*CT-VS-Ops*) or the hash of the architectural state (*CT-VS-All*).
- If a tainted register is used as a source operand, we propagate its taint to the destination register. To correctly collect all dependencies, we also add the values of the other (non-tainted) source operands to the taint of the destination. For

[2] An alternative would be to immediately expose the dependencies as observations, akin to the treatment of operands of variable-latency instructions in constant-time leakage models [2]. However, this coarser approach precludes software defenses such as speculation barriers.

example, if x is tainted with $t(x)$ and we execute $z \leftarrow x + y$, then z now depends on all values in $t(x)$ and on the value of y.

- If we store the content of a tainted register x to address n, then the taint of x propagates to address n. If we load from n, the taint propagates back to the destination register. We decided to taint memory locations because, even though speculative stores will not retire, they can be forwarded to loads from the same address.
- If we access a memory location represented by expression e, and we cannot evaluate e because it contains a tainted register (whose value we don't know), then we taint the destination with the full architectural state, represented by $hash(\sigma)$. This models that we cannot give any guarantee about the content of the destination. The taints in expression e are exposed as observation.
- If a register or address is the destination in an instruction without tainted source operands, we remove its taint.

Implementation. We implement *CT-VS-Ops* and *CT-VS-All* for all exceptions considered in §8. For *CT-VS-Ops*, we define that the operand of a load is the accessed address, not the value stored at that address. We also use the hash of the program input instead of the hash of the entire architectural state. From an information flow perspective, these two are equivalent under the assumption that the attacker knows the program and that the microarchitectural state is reset between two executions of the program.

8 Evaluation

In our evaluation, we test a diverse range of exceptions on x86 CPUs against a series of increasingly permissive contracts. The experiments target two goals: to evaluate our tooling for testing exceptions and to showcase our approach of creating valid leakage models by incremental refinement.

Testing Targets. We test the following machines:

- `Int1`: Intel Core i7-7700 (Kaby Lake), ucode `0xf0`
- `Int2`: Intel Xeon E-2288G (CoffeeLake), ucode `0xf0`
- `AMD1`: AMD Ryzen 5 2600X (Zen+), ucode `0x800820d`
- `AMD2`: AMD EPYC 7543P (Zen 3), ucode `0xa001143`

Methodology. We test the exceptions described in §8.1 on each of the testing targets with the tool described in §3. For each exception, we start with the least permissive contract (*CT-SEQ*) and test the exception for 24 hours or until the tool detects a violation. Upon detection, we manually inspect the counterexample to determine the root cause. Then, we repeat the experiment with a more permissive contract that allows the detected counterexample. We repeat this process until we find a contract for which our tool does find a violation.

Configuration. The program generator is configured to produce programs with 32 instructions and 8 memory accesses from the instruction pool in Appendix A. The executor is in Partial Prime+Probe mode (§3.4). Contract-driven input generation and the speculation filter are enabled (see [38]).

As our models (so far) can handle only one type of exception at a time, we make the following adjustments: To prevent Spectre, we configure the generator to produce straight-line code. To prevent LVI-Null from triggering divisions by zero, we exclude divisions when testing `Int2` and `Int1`. To prevent Spectre V4, we enable microcode patches.

On False Positives. We call any situation in which two program executions produce matching contract traces but mismatching hardware traces a *positive*. This is a *false positive* when the mismatch is *not* due to differences in program inputs (and hence does not constitute an information leak). A false positive can be caused by a difference in the initial microarchitectural state (due to imperfect initialization), or by external noise corrupting a hardware trace. Revizor applies several heuristics to eliminate false positives [37]. We observed no false positives in our testing campaigns, i.e., we could identify a genuine information leak for all detected violations.

8.1 Tested Exceptions

We test three classes of exceptions, which we describe next.

Memory Errors. Memory errors are essential mechanisms for providing security. They facilitate isolation between virtual machines and memory protection in the operating system, e.g., page faults are crucial to maintaining access control. In line with previous work [33], we consider microcode assists equivalent to exceptions since they enable Meltdown-like leaks [9, 11, 42, 46]. We test the following memory errors.

- Page Not Present (`#PF`): *Present* bit is 0 for the second page in the sandbox (see §3.1).
- Write to Read-Only Page (`#PF`): *Read/Write* bit is 0.
- SMAP Fault (`#PF`): *User* bit is 1, and SMAP is enabled.
- Page Accessed (microcode assist): *Accessed* bit is 0.
- Page Dirty (microcode assist): *Dirty* bit is 0.
- Non-canonical Access (`#GP`): Memory accesses to addresses in non-canonical form are inserted in the test case at generation time (see §3.2)

Computational Errors. This class includes the exceptions caused by impossible or wrong computations. Speculation on such errors can introduce unexpected values in the program data flow. Here we focus on divisions and test the following:

- Division by Zero (`#DE`): Divisions are instrumented to prevent overflows but not divisions by zero.
- Division Overflow (`#DE`): Divisions are instrumented to prevent divisions by zero but not overflows.

In both cases, some of the randomly generated test cases throw `#DE`. Even though this does not happen in all test cases, our tool quickly discards the test cases without exceptions due to the speculation filter introduced in [38].

Opcode-based Errors. This class includes the exceptions triggered by undefined, invalid, and debug instructions. These exceptions are critical for security because they prevent unexpected events from happening in the system. For example,

#UD (undefined opcode exception) is thrown when non-VM code attempts to execute a VM management instruction (e.g., VMCALL). We test the following faults (details in Appendix A):

- Invalid Opcode (#UD): The UD2 instruction and the 32-bit instruction opcodes (invalid in 64-bit mode) are added to the instruction pool; the first to execute throws #UD.
- Incompatible CPU Mode (#UD): Virtualization instructions are added to the instruction pool. On Intel, this corresponds to the VTX extension [26] and on AMD to the SVM extension [5]. Since our tests run in a non-virtualized CPU mode, the first such instruction to execute throws #UD.
- Breakpoint: INT1 and INT3 are added to the instruction pool to trigger Debug (#DB) and Breakpoint exceptions (#BP).

8.2 Testing Results

Our results are summarized in Table 1. We first describe the new speculative leaks we discovered and then proceed to explain the model refinement process.

8.2.1 Discovered Speculative Leaks

We discovered three novel leaks, i.e., leaks that were not reported previously in the literature or even contradict previous reports. Our approach cannot automatically answer whether these leaks can be exploited in practice.

Divider State Sampling (DSS). While testing divisions by zero, we discovered unexpected cross-instruction leakage. We observed that the data processed by non-faulting divisions impacts the speculative values returned by subsequent divisions by zero. We suspect that the vulnerable CPUs return a stale state of the divide unit (although we are not able to confirm). We managed to trigger this behavior only on AMD1. Consider the following instruction sequence:

```
1 # rax, rdx, or rbx contain a secret
2 DIV rbx  # rbx != 0, non-faulting division
3 ...
4 MOV rcx, 0  # trigger division by zero
5 DIV rcx
6 MOV rdi, [array_base + rdx] # leak remainder
```

At line 2, a division is performed on secret data. The instruction at line 5 attempts to divide the content of rdx:rax by the value in rcx; the division faults and returns a speculative value. This value is dependent on the operands of the division at line 2. Line 6 exposes the value through the cache state.

Notably, the divisions at line 2 and 5 do not have to be close. We verified that the leak persists even if we replace line 3 with a long sequence of serializing instructions or even a system call. However, we were unable to reproduce the leak across privilege levels: When executing the faulting division (i.e., line 5) in user mode, we verified that it does *not* leak data from previous divisions executed in kernel mode.

Read-Modify-Write Speculation. While testing page faults and microcode assists triggered by writes to read-only memory, we discovered a new way of triggering MDS, LVI, and Foreshadow that affects read-modify-write instructions. These leaks have been previously reported for faulting loads but not for stores. Consider the following instruction sequence:

```
1 # rax = read-only address
2 XADD [rax], rbx  # fault; rbx = speculative value
3 MOV rcx, [array_base + rbx]  # leak into cache
```

Line 2 exchanges the content of [rax] with rbx and stores their sum into [rax]; as rax points to read-only memory, the store faults. However, the first load is also treated as if it experienced a fault—even though loading is permitted—and it returns a speculative value. The value is assigned to rbx, and then leaked in line 3. On Int1, we verified that the speculative value depends on previously accessed data as in Foreshadow [8] and MDS [11, 42, 46]. On Int2, we observe zero injection (i.e., the load returns zero) as in LVI-Null [9].

Non-canonical Store Forwarding. While testing non-canonical memory accesses, we discovered that values stored to a non-canonical address can be forwarded to subsequent loads from the canonical version of that address. We observed this behavior on Int1 and Int2. Consider the following instruction sequence:

```
1 # rax = non-canonical address
2 # rbx = canonical address with matching bits [0:47]
3 MOV [rax], secret  # faulting store
4 MOV rcx, [rbx]  # forwarding; rcx = secret
5 MOV rcx, [array_base + rcx]  # leak into cache
```

In line 3, the store triggers an exception. The secret value is forwarded to line 4 and assigned to rcx. Line 5 reveals the value by modifying the cache state.

Note that similar leaks were previously reported: Canonical stores being forwarded to non-canonical (faulting) loads was reported on both AMD [34] for Intel [11] CPUs. Data forwarding when writing to and reading from a non-canonical address was reported on Intel [11].

Exception Chaining. Anecdotally, we found interesting cases where an exception caused a "chain reaction". Consider a division with a memory operand pointing to a non-present page: When it is executed on Int2 or Int1, the corresponding load may speculatively return zero. This zero is then used as a divisor, it causes a division-by-zero exception, which also triggers speculation, and it yields another speculative value.

8.2.2 Model Refinement for Memory Errors

Our results are given in Table 1. Due to considerable differences between the tested CPUs, we describe the refinement process for each of them separately.

AMD. To model memory errors on AMD1 and AMD2, we first attempt to use *CT-SEQ*. We find violations of this model for

Fault	Variant	*CT-SEQ*				*CT-DH*				*CT-VS*				*CT-VS-Ops*				*CT-VS-All*			
		Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2
#PF	invalid	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓NI	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	read-only	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓NI	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	SMAP	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓NI	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
#GP	non-canonical	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓CI	✓CI	✗	✗	✓	✓	✗	✗	✓	✓
#BR	MPX	✗	✗	n/a	n/a	✓	✓	n/a	n/a	✓	✓	n/a	n/a	✓	✓	n/a	n/a	✓	✓	n/a	n/a
uCode assist	A-bit	✗	✗	✓	✓	✗	✗	✓	✓	✗	✓NI	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	D-bit	✗	✗	✓	✓	✗	✗	✓	✓	✗	✓NI	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
#DE	div by zero	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓
	div overflow	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
#UD	undef. opcode	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	invalid mode	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#DB + #BP	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Summary of the testing results. Here, ✗- contract violation; ✓- contract compliance; ✓- contract compliance through a weaker contract; ✓NI - compliance *CT-VS*-NI, a variant of *CT-VS* that returns zero for speculative loads; ✓CI - compliance *CT-VS*-CI, a variant of *CT-VS* that executes a speculative canonical access upon a non-canonical one; *n/a* - "not applicable" (Memory Protection Extension (MPX) is not supported by AMD CPUs).

page faults and non-canonical accesses, but not for microcode assists. The violations contain memory access instructions that occur after the faults (in program order) and are executed before the exception is caught. This behavior corresponds to delayed exception handling (§5).[1]

We refine our model to account for delayed exception handling mechanisms with *CT-DH*. Our experiments do not find violations of the contract for page faults and assists; non-canonical accesses, however, violate *CT-DH*. Our investigation reveals that the violations are caused by the speculative behavior previously described by Musaev et al. [34].

Finally, to model non-canonical accesses, we test them against *CT-VS-CI*, a *CT-VS* variant with loads returning values from the canonical version of the faulting address. The testing campaign does not find violations of this contract.

Intel CoffeeLake. We start with *CT-SEQ* and find violations similar to AMD. We next move to *CT-DH*. MPX exceptions do not produce violations (in line with previous research [10]), but the tool finds violations for all other exceptions. For load-based exceptions, they appear to be caused by zero injection in faulting loads (LVI-Null [9]). For store-based exceptions, we find that a speculative value gets forwarded, which leads to the discovery of *Read-Modify-Write Speculation*, see §8.2.1.

We next attempt to model zero injection with *CT-VS-NI*, a variant of *CT-VS*, where faulting loads speculatively return zero. This time, the contract is violated only by non-canonical accesses. Our investigation of the violation leads to the discovery of *Non-canonical Store Forwarding*, see §8.2.1.

Intel Kaby Lake. We start with *CT-SEQ* and find violations similar to AMD. We then test *CT-DH* and find violations too; They appear to be caused by MDS and Foreshadow.

[1]The same behavior does not cause violations with assists, however, because assists do not terminate execution of test cases, hence this information is exposed by non-speculative execution.

We next implement a number of versions of *CT-VS*, each corresponding to a hypothesis about the value returned by faulting loads, but we continue to observe violations.

Therefore, we move on to testing the CPU against the *CT-VS-Ops* contract. It successfully filters out violations caused by value forwarding from the L1D cache (i.e., Foreshadow). However, we continue to discover violations since, in MDS, the speculative value depends not only on the address operand but also on previously stored or loaded values. Thus, we resort to an even more permissive contract, *CT-VS-All*. This time, we do not find violations on page faults and assists, but non-canonical accesses still cause violations. Our investigation reveals that they are caused by *Non-canonical Store Forwarding* (§8.2.1). The tainting algorithm of *CT-VS-All* only taints the destination address of the non-canonical store, it does not taint the canonical version of the address. If the tainted address is not accessed afterwards, the taint is never exposed, which results in the described leak.

8.2.3 Model Refinement for Computational Errors

We again begin with *CT-SEQ* and find violations similar to those from memory errors. We then test against *CT-DH* and find violations as well. Our investigation reveals that both types of division errors return speculative values.

Based on previous work [10], we expect the speculative value to be zero, and we implement the corresponding version of *CT-VS*. We test against it, but surprisingly, we find violations, and the counterexamples reveal that the speculative value is often non-zero. We attempt to reverse-engineer the speculative value, but we fail to find a pattern.

Using *CT-VS-Ops*, we test the hypothesis that the speculative value depends entirely on the division operands. This contract is satisfied on most targets, except for `AMD1`. An investigation of the violation leads to the discovery of *Divider*

State Sampling, see §8.2.1. Specifically, the violation is triggered because DSS introduces an information flow between division instructions, which is not allowed by *CT-VS-Ops*.

Finally, we model DSS with *CT-VS-All*, and we do not observe any violations anymore.

8.2.4 Model Refinement for Opcode-based Errors

We begin with *CT-SEQ* and, for the first time, we do not find violations for any of the exceptions. This either means that the faulting opcodes are treated as serializing events—and indeed the AMD security advisory suggests that for `INT3` [3]—or that the exceptions are detected at the early stages of the pipeline, which makes the speculation window too short to cause observable changes to the microarchitectural state. This finding shows that at least some of the exceptions exhibit a strictly sequential behavior and can thus be used without concerns regarding speculative attacks. It also confirms the results of the manual investigation of invalid opcode reported by Canella et al. [10].

8.3 Detection Time and Performance

Lastly, we provide a summary of the detection time and the testing speed (complete results are given in Appendix B).

The number of test cases needed to find a violation varied significantly between the tested contracts and exceptions, ranging from less than 10 and up to 80k test cases. We observed the fastest detection when testing against *CT-SEQ*, with most of the violations detected after just a few test cases (seconds of testing). Such fast detection is caused by the fact that any speculative memory access in a test case leads to a violation. Detection of both *CT-DH* and *CT-VS* violations required several hundred test cases for most targets (at most nine thousand), which corresponds to less than an hour of testing. Detection of *CT-VS-Unknown* violations required the most test cases as the contract exposes a large amount of information, making it difficult to find a test case that exhibits leakage not exposed by the contract. The violations were detected within several thousand of test cases (at most 80k), corresponding to several hours of testing (at most 11 hours).

The testing speed also varied between different targets and contracts. In most cases, the speed was in the range of 2k–20k test cases per hour. The speed tended to be higher when testing non-speculating exceptions against simple contracts: The highest testing speed was 54k test cases per hour when testing `#DB` and `#BP` exceptions against *CT-SEQ* on the high-end CPU `Int2`. The speed tended to be slower for more complicated contracts (e.g., *CT-VS-All*) and for configurations that triggered a lot of speculation (e.g., `#PF`), as both of these factors increased the modeling complexity and made contract trace collection into a performance bottleneck. We observed the lowest speed of 1k test cases per hour when testing `#PF` against *CT-VS-Unknown* on a desktop CPU `Int1`.

9 Programming against Exception Contracts

In this section, we discuss the software-facing consequences of our exception contracts. We show how contracts help to define, test, and patch information leakage of a program.

Contract-based Noninterference. The security of a program w.r.t. information leaks is typically defined as a noninterference property. In the context of side channels, a program is noninterferent if its secrets do not influence the hardware traces observable by an attacker. As contracts overapproximate these observations, we can check noninterference by checking if the contract traces depend on the secrets.

To define noninterference, the program's input is split in a part containing secret data (H for high-security) and a part containing public data (L for low-security). Formally, this is described with a function $\pi\colon \mathbb{N} \to \{L,H\}$ that assigns labels L and H to each memory location of the initial state σ. We say that two program inputs are *low-equivalent* ($\sigma \simeq_L \sigma'$) if and only if they agree on the content of their low-labeled memory.

Accordingly, for a CPU that has been shown to satisfy a contract $Contract(\cdot)$, the program is noninterferent if changing the content of high-labeled memory does not influence the resulting contract traces:

Definition 2 (Noninterference wrt a Contract (NI) [22]). A program p is *noninterferent with respect to a contract* $Contract(\cdot)$ and a policy π if for all pairs of inputs (σ,σ') with $\sigma \simeq_L \sigma'$, we have $Contract(p,\sigma) = Contract(p,\sigma')$.

Testing Security of Programs. Executable leakage contracts can be used to check if a program p satisfies Definition 2. To do so, one executes $Contract(p,\cdot)$ on different pairs of low-equivalent states and checks if their contract traces agree.

These tests can be done independently of the target CPU, or even on multiple CPUs simultaneously, as long as they satisfy the same contract. Therefore, contracts enable us to decouple the testing of CPUs for information leakage primitives (which is the topic of this paper) from checking whether these primitives could expose program secrets. Modern high-performance CPUs can be costly and lengthy to test. However, testing needs to be done only once per model (with local retests after microcode patches). Program testing should be done after every update, but it is much cheaper due to the contract's intentional minimalism and white-box nature.

Enforcing Security of Programs. There is a growing body of work on enforcing NI or related properties in programs via compilation [39], code transformation [47], and using program analysis techniques that can detect violations of noninterference in source or binary code [13, 22]. While an exploration of how to extend each of these approaches to exception contracts is out of scope, we follow a proposal in [21] and discuss how programs may achieve NI based on our contracts:

- The *sequential contract* (*CT-SEQ*) leads to a generalization of the constant-time programming model, which achieves

NI by not branching on secrets and not accessing memory in a secret-dependent way. *CT-SEQ* adds that the conditions that trigger exceptions should not depend on a secret.

• To guarantee NI for programs based on the *delayed exception handling contract* (*CT-DH*), instructions located after an exception (in program order) should be prepended with a speculation barrier if they may expose a secret. Alternatively, they may artificially be made dependent, in spirit of Speculative Load Hardening [14].

• The *value speculation contracts* (*CT-VS*) imply that instructions following an exception may process unexpected values. This leaves the programmer only with speculation barriers to achieve NI; as for the LVI mitigations proposed by Intel [27]. However, with a contract that predicts the exact speculative value, program analysis tools could be developed to optimize such patches.

• The *unknown value speculation contracts* (*CT-VS-Ops* and *CT-VS-All*) describes the MDS world, where faulty instructions return an unknown value. For MLPDS-vulnerable CPUs, this translates into an attacker being able to leak values loaded in previous instructions. In such cases, speculation barriers have to be inserted before all instructions that operate on values that depend on the faulting instruction. There is a fine difference, however, between *CT-VS-Ops* and *CT-VS-All* in this context. In *CT-VS-All*, the speculative value might contain any (secret) value from the architecture. To prevent leakage, every instruction that triggers such an exception must have a speculation barrier. With *CT-VS-Ops*, however, the value is known to be derived from the faulting instruction's operands. Unless the operands contain secrets, placing a barrier only after loads based on speculative values is sufficient.

10 Coverage and Limitations

We distinguish between two kinds of coverage: microarchitectural coverage, which accounts for the explored features of the CPU implementation, and architectural coverage, which accounts for the explored ISA features.

Microarchitectural Coverage. As we do not have access to CPU internals in black-box testing, microarchitectural coverage is impossible to measure. We believe that our evaluation covered a large portion of the microarchitecture: Each experiment ran for 24 hours; most violations were detected within one hour of testing, and the longest took 11 hours (see §8.3).

Architectural Coverage. The x86-64 architecture supports 256 exceptions types, each mapping to a unique entry in the IDT [26]. Therefore, we did not aim for completeness, and focused on demonstrating our approach's effectiveness on 7 different exceptions. They represent a diverse set of behaviors and include those exceptions that are most notorious regarding speculative information leaks.

Table 2 summarizes the supported exceptions and lists the missing Revizor features for those not supported so far. The

Name	IDT Vector	Exception	Support / Missing Requirement
#DE	0	Division Error	✓
#DB	1	Debug	✓
#BP	3	Breakpoint	✓
#BR	5	Bound Range	✓
#UD	6	Undefined Opcode	✓
#GP	13	General Protection	✓
#PF	14	Page Fault	✓
#OF	4	Overflow	32-bit Mode
-	10–12	Segment Exceptions	32-bit Mode
#AC	17	Alignment Check	User Mode
#VE	20	Virtualization Exception	Guest VM Mode
#NM	7	Device Not Available	FP or SIMD
#MF	16	x87 Exception	FP
#XM	19	SIMD Exception	SIMD
#CP	21	Control Protection	CFI
-	2	NMI Interrupt	Asynch. Exception
-	32-255	User Defined Interrupts	Asynch. Exception
#DF	8	Double Fault	Exception Handler
#MC	18	Machine Check	HW failure
-	9	Deprecated	-
-	15, 22–32	Reserved	-

Table 2: Supported exceptions and the missing features required to support the remaining exceptions.

missing features can be divided into the following groups:

• *Unsupported Execution Mode:* Exceptions #OF, #TS, #NP, #SS are possible only in the 32-bit legacy mode, which is not supported by Revizor. Similarly, #AC can be triggered only in user mode and #VE only if the test case is executed within a VM. Adding support for these execution modes would require considerable changes to Revizor's executor and is left to future work.

• *Unsupported Instruction Type:* Exceptions #NM, #MF, #XM, and #CP require support for floating-point, SIMD, and CFI instructions, respectively. So far, there does not exist a leakage model or any testing support for these instructions (even without considering exceptions). Hence, we did not include them in our study.

• *Unsupported Triggers:* Some exceptions require special setups to be triggered. #DF can be triggered only within an exception handler, which in our setup is not part of the test case. #MC is a hardware fault and cannot be triggered by our software-only setup.

• *Asynchronous Exceptions:* We did not target asynchronous exceptions (software and hardware interrupts) for two reasons. (1) Software interrupts (generated by the INT instruction) do not trigger transient execution according to the Intel manuals [26]; this is consistent with out investigation of #DB and #BP in §8. Hardware interrupts can trigger transient execution [41], but no such vulnerability has been reported so far. (2) While our contracts can be extended to model asynchronous exceptions, testing them would be challenging: The CPU may handle interrupts at unpredictable points; this nondeterminism would have to be resolved by mapping the

observed hardware trace to the right contract trace.

Tested Microcode Assists. We also tested microcode assists in this paper, namely those related to *Access* and *Dirty* page table bits. There are several other types of assists mentioned in the Intel documentation [28], including those caused by x87 instructions, transitions between SSE and AVX code, AVX store instructions, and Intel SGX [15]. We did not cover them as Revizor lacks support for the corresponding instructions and execution modes. Besides, assists are not part of the ISA, which is why we cannot be more specific about coverage.

Reproduced Leaks. Our tool was able to detect all known speculative leaks for the exceptions we support, as shown in §8. Namely, it reproduced the leaks underlying Foreshadow [8], MDS [11,42,46], LVI-Null [9], Spectre V1.2 [29], the SMAP version of Meltdown [10], zero result for division errors [10], delayed MPX exception handling [10], and transient handling of non-canonical accesses [34]. All of the above leaks were reproduced through pure random testing, without explicitly searching for them. Naturally, we did not reproduce leaks in exceptions that are not covered by Revizor (e.g., LazyFP [43] requires floating-point operations, and the original version of Meltdown [31] requires a page fault triggered from user space).

Exception Handlers. The exception handler within Revizor's executor is always located at the same address in memory. In principle, however, some CPUs could use a branch predictor-like module to predict the address of handler. This would lead to speculative leaks similar to Spectre V2. Similarly, we only used an empty exception handler. Even though so far, there has not been any speculative leak that would target exception handlers specifically, it would still be preferable to test program executions with handlers.

11 Related Work

We discuss related work that models and tests for leakage via microarchitectural side channels (related attack papers are discussed above).

We begin with the two existing tools for model-based black-box testing, which are most closely related to our work. We then discuss template-based and white-box approaches. We finally discuss models and tools for microarchitectural side channels that do not involve transient execution.

Black-box Modelling and CPU Testing. Two tools implement model-based black-box testing for CPUs: Revizor [37] (described in §2) and Scam-V [7, 35]. These tools rely on random testing to probe the CPU for microarchitectural leaks and compare the observed leakage to the leakage predicted by a formal (ISA-based) leakage model. Revizor supports x86, while Scam-V supports Arm. Unlike Revizor, Scam-V leverages symbolic execution to automatically generate inputs that yield the same formal leakage. Both tools focus on programs with a range of instructions for memory access and explicit control flows (e.g., conditional branches) to detect Spectre-like leakage. They do not support exceptions.

Our work is the first to develop leakage models for exceptions and the implicit control flows they induce. It is thus also the first to extend a model-based tool to systematically test CPUs against them.

Template-based CPU Testing. Another class of tools takes as input code templates known to trigger transient execution attacks to discover attack variants. Transynther [33] is a tool that focuses on MDS (and thus also targets leakage caused by exceptions). It uses combinations of known MDS building blocks and microcode assists to detect new MDS leaks. In order to verify the leakage, the observed values are compared to those used to fill the microarchitectural buffer in a previous phase. SpeechMiner [50] focuses on Meltdown; it quantifies the exploitability of different vulnerabilities by measuring the outcome of race conditions like data fetching latency on a given CPU.

Our model-based testing approach is less restricted in the program generation and can thus uncover new leaks. In contrast, Transynther and SpeechMiner can only detect variants that follow a predefined template. As empirical evidence, although both papers tested read-only faults in their evaluation, none detected read-modify-write speculation (see §8.2.1).

White-box CPU Testing. Other tools analyze open-source hardware designs for pre-silicon detection of speculative leaks. Since the microarchitecture is known, more general validation techniques apply. Unlike black-box approaches, they are not applicable to commercial off-the-shelf CPUs. CheckMate [44] is based on a microarchitectural specification in terms of happens-before relations. IntroSpectre [18] relies on an RTL description of the CPU to detect Meltdown-type leaks; it randomly combines templates and detects the leakage of a known secret nonce inserted into a security domain.

Testing Other Microarchitectural Side Channels. Other tools explore leaks on black-box CPUs that are not caused by transient execution. ABSynthe [19] synthesizes contention-based side-channel attacks by inferring interactions between several x86 instructions. Osiris [49] is a tool that searches for new side channels by observing timing differences in randomly generated code snippets. Plumber [24] is a recent approach based on abstractions of code snippets that are known to cause leakage. These tools are complementary to our work; they focus on the extraction of microarchitectural traces, whereas we focus on the speculation that leaks information into microarchitectural state.

Speculative Leakage Models. A number of leakage models have been proposed to detect instances of Spectre in software, such as InSpectre [20], Spectector [21], SpecuSym [23], and KLEESpectre [48]; see the SoK by Cauligi et al. [12] for an extensive overview. Most of these models target Spectre V1, and none of them models exceptions. Moreover, the correctness of these models has not been tested on real CPUs,

and there are instances where a key assumption of the model has been disproven in practice (e.g., KLEESpectre assumes that store-based Spectre V1 is impossible, which was shown incorrect in [44] and [37]).

Other Leakage Models. Besides speculative execution, leakage models have been extensively used for detecting classical microarchitectural side channels (e.g., CacheAudit [17], Casym [6]), timing channels (e.g., Jasmin [1]), and physical channels (e.g., Miracle [32]). These models, however, do not consider speculative execution and thus cannot be used to detect leaks caused by speculation upon exceptions.

12 Conclusion

In this paper, we have provided a family of formal leakage models for CPU exceptions together with tool support for testing black-box CPUs against them. We have run extensive experiments to iteratively refine these leakage models to faithfully capture the leakage of four different x86 microarchitectures. In the process, we have uncovered three novel transient leaks and have contradicted, refined, and corroborated a large number of findings from prior work.

Acknowledgments. We thank the shepherd and the anonymous reviewers for their comments. This work was supported by the NWO through project "Intersect".

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium*, 2016.

[3] AMD. Software techniques for managing speculation on AMD processors. https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf, 2018.

[4] AMD. Speculation behavior in AMD micro-architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf, 2019.

[5] AMD. *AMD64 Architecture Programmer's Manual*. 2023.

[6] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[7] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. Validation of side-channel models via observation refinement. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[8] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, 2018.

[9] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium*, 2019.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[12] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022.

[13] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[14] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6), 2010.

[15] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016.

[16] Hernán Ponce de León and Johannes Kinder. Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks. In *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1), 2015.

[18] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. INTROSPECTRE: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[19] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[20] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[21] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[22] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *42nd IEEE Symposium on Security and Privacy (S&P)*, 2021.

[23] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. In *ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[24] Ahmad Ibrahim, Hamed Nemati, Till Schlüter, Nils Ole Tippenhauer, and Christian Rossow. Microarchitectural leakage templates and their application to cache-based side channels. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[25] Intel. Speculative Execution Side Channel Mitigations - Revision 3.0. https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf, 2018.

[26] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2019.

[27] Intel. Load Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/load-value-injection.html, 2020.

[28] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2023.

[29] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *CoRR*, abs/1807.03757, 2018.

[30] 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. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, (S&P)*, 2019.

[31] 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. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, 2018.

[32] Ben Marshall, Dan Page, and James Webb. Miracle: Micro-architectural leakage evaluation. *Cryptology ePrint Archive*, 2021.

[33] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium*, 2020.

[34] Saidgani Musaev and Christof Fetzer. Transient execution of non-canonical accesses. *CoRR*, abs/2108.10771, 2021.

[35] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *32nd International Conference on Computer-Aided Verification (CAV)*, 2020.

[36] Hamed Nemati, Roberto Guanciale, Pablo Buiras, and Andreas Lindner. Speculative leakage in ARM cortex-a53. *CoRR*, abs/2007.06865, 2020.

[37] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: testing black-box cpus against speculation contracts. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[38] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (S&P)*, 2023.

[39] Marco Patrignani and Marco Guarnieri. Exorcising spectres with secure compilers. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[40] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: Next generation CPU emulator framework. In *BlackHat USA*, 2015.

[41] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium*, 2021.

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[43] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[44] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated synthesis of hardware exploits and security litmus tests. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[45] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010.

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[47] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. In *48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2021.

[48] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(3):1–31, 2020.

[49] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium*, 2021.

[50] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities. In *27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

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

[52] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

A Instruction Set Tested in Evaluation

Our evaluation (§8) used the following instruction pool: `(LOCK) ADD/ADC`, `(LOCK) SUB/SBB`, `(LOCK) INC`, `(LOCK) DEC`, `(LOCK) NEG`, `(I)DIV`, `(I)MUL`, `(LOCK) AND`, `(LOCK) NOT`, `(LOCK) OR`, `(LOCK) XOR`, `BSF`, `BSR`, `(LOCK) BT`, `(LOCK) BTC`, `(LOCK) BTR`, `(LOCK) BTS`, `BSWAP`, `MOV`, `MOVSX`, `MOVZX`, `XCHG`, `CMP`, `TEST`, `CBW`, `CDQ`, `CWD`, `CWDE`, `CLC`, `CLD`, `CMC`, `LAHF`, `SAHF`, `STC`, `STD`, `LEA`, `NOP`, `SET*`. Some experiments added instructions to trigger exceptions:

- `#DB+#BP: INT1, INT3`.
- `#UD` (undef. opcode): `UD2`, `DAA/DAS`, `AAA/AAS`, `PUSHA/POPA`, `BOUND`, `CALLF/JMPF`, `LES`, `AAM/AAD`, `PUSH/POP` (32-bit-only opcodes).
- `#UD` (inv. mode), Intel: `INVEPT(PID)`, `VMXOFF`, `VMLAUNCH(RESUME/CALL/CLEAR/READ/WRITE/PTRLD/PTRST)`.
- `#UD` (inv. mode), AMD: `VMRUN`, `VMLOAD`, `VMSAVE`, `CLGI`, `VMMCALL`, `INVLPGA`.

B Performance Details

Table 3 details the testing campaign presented in §8.2.

Fault	Variant	*CT-SEQ*				*CT-DH*				*CT-VS*				*CT-VS-Ops*				*CT-VS-All*			
		Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2	Int1	Int2	AMD1	AMD2
#PF	invalid	✗ 1	✗ 2	✗ 1	✗ 1	✗ 120	✗ 49	✓ 30k	✓ 182k	✗ 4.4k	✓ 25k	-	-	✗ 1.3k	-	-	-	✓ 26k	-	-	-
	read-only	✗ 3	✗ 1	✗ 1	✗ 3	✗ 37	✗ 289	✓ 37k	✓ 21k	✗ 416	✓ 52k	-	-	✗ 5.3k	-	-	-	✓ 29k	-	-	-
	SMAP	✗ 1	✗ 1	✗ 3	✗ 1	✗ 3	✗ 65	✓ 32k	✓ 187k	✗ 930	✓ 25k	-	-	✗ 246	-	-	-	✓ 27k	-	-	-
#GP	non-canonical	✗ 1	✗ 2	✗ 3	✗ 2	✗ 155	✗ 46	✗ 30	✗ 137	✗ 229	✗ 30	✓ 70k	✓ 60k	✗ 49k	✗ 8.8k	-	-	✗ 35k	✗ 12k	-	-
#BR	MPX	✗ 49	✗ 172	-	-	✓ 225k	✓ 714k	-	-	-	-	-	-	-	-	-	-	-	-	-	-
uCode assist	A-bit	✗ 3	✗ 1	✓ 188k	✓ 45k	✗ 7	✗ 3	-	-	✗ 295	✓ 44k	-	-	✗ 572	-	-	-	✓ 116k	-	-	-
	D-bit	✗ 1	✗ 3	✓ 51k	✓ 33k	✗ 7	✗ 25	-	-	✗ 1k	✓ 45k	-	-	✗ 2k	-	-	-	✓ 115k	-	-	-
#DE	div by zero	✗ 82	✗ 25	✗ 23	✗ 12	✗ 9k	✗ 4k	✗ 785	✗ 312	✗ 37k	✗ 386	✗ 2.3k	✗ 1.7k	✓ 127k	✓ 168k	✗ 80k	✓ 112k	-	-	✓ 168k	-
	div overflow	✗ 3	✗ 18	✗ 1	✗ 1	✗ 4	✗ 91	✗ 13	✗ 66	✗ 102	✗ 1.4k	✗ 1.8k	✗ 29	✓ 31k	✓ 73k	✓ 72k	✓ 43k	-	-	-	-
#UD	undef. opcode	✓ 254k	✓ 727k	✓ 580k	✓ 298k	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	invalid mode	✓ 90k	✓ 96k	✓ 114k	✓ 214k	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
#DB + #BP	-	✓ 332k	✓ 1.3M	✓ 649k	✓ 325k	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 3: Additional details for Table 1. This table shows the number of rounds executed within 24 hours (performance) for tests without a violation (✓) and the number of rounds executed before the violation (detection time) for tests with a violation (✗).

C Semantics of *CT-SEQ* with Exceptions

The full *CT-SEQ* contract with exceptions is presented in Figure 4. It is largely similar to *CT-SEQ* without exceptions [22] and builds on the plain semantics of μASM [21]. The evaluation of expressions (written $[\![e]\!]_\sigma$) is straightforward. All rules except the EXCEPTION rule assume that no exception occurs during the execution of the instruction ($EC(\sigma,p)=\bot$). LOAD, RETURN and EXCEPTION are described in §4. Rule BARRIER ignores the speculation barrier instruction, which has no effect in sequential execution. ASSIGN updates a register x with the value of an expression e. TERMINATE models the case that the end of the program is reached. STORE moves the value stored in register x to memory location $n=[\![e]\!]_\sigma$ and, similarly to the LOAD instruction, exposes n.

D Definition of *CT-VS-Unknown*

In this section, we provide further formal details of the *CT-VS-Unknown* contract.

Representation of Taints. A taint is a set of triples (i,l,v), where i denotes a program location, l is either a register, a flag, or a memory location, and v is the value stored at l before the execution of instruction i. We use *Taints* to denote the type of taints, which are sets over $\mathbb{N}\times(Regs\cup\mathbb{N})\times Vals$.

Evaluating Tainted Expressions. We define a few functions that facilitate the taint propagation in the formal semantics. Let $t : Regs\cup\mathbb{N}\to Taints$ be a function that maps registers and memory locations to taints. For $l\in Regs\cup\mathbb{N}$, we define $t(l)=\emptyset$ if l is not tainted. For an expression e, we use the predicate $tainted(e)$ if e contains a tainted register. Function $taint(e,t,i)$ generates the taint for an expression e with respect to taints t and instruction i. It aggregates the values of the registers in e. If a register is tainted, it uses the taint instead:

$$
\begin{aligned}
taint(n,t,i) &:= \emptyset \\
taint(x,t,i) &:= \{(i,x,[\![x]\!]_\sigma)\} \qquad \text{if } t(x)=\emptyset \\
taint(x,t,i) &:= T \qquad \text{if } t(x)=T\neq\emptyset \\
taint(\ominus e,t,i) &:= taint(e,t,i) \\
taint(e_1\otimes e_2,t,i) &:= taint(e_1,t,i)\cup taint(e_2,t,i) \\
taint(\textbf{ite}(e_1,e_2,e_3),t,i) &:= taint(e_1,t,i)\cup taint(e_2,t,i) \\
&\quad\cup taint(e_3,t,i)
\end{aligned}
$$

Operational Semantics. In Figure 5, we present the main rules of the *CT-VS-Ops* instance of the contract. We focus on the rules that formalize memory accesses, the initialization of taints, and their exposure as observation labels. Rules that do not interact with the memory are defined as in *CT-VS*, just with the additional taint propagation we describe in §7.

Transient execution is handled similarly to *CT-DH* using a speculation window w, after which the transient state is rolled back. The state consists of the architectural state σ, the mapping t of current taints, and a counter ω.

The EXCEPTION rule describes how to generate the taint upon an exception. Here, we consider a fault occurring during a non-transient assignment instruction; the definition is similar for other instructions and nested exceptions. The taint T is generated according to the definition of $taint(\cdot)$ by collecting the values of all registers in e.

For loads and stores, we need to distinguish two cases, depending on whether e contains a tainted register. If it does not, then we propagate the taint (if it exists) from the source operand to the destination (rules LOAD and STORE): Loads propagate the taint of the accessed address to the register, and stores propagate the taint from the register to the address. If the taint T is empty, the rules also implicitly remove a previous taint from the destination. If e is tainted (TAINTEDLOAD and TAINTEDSTORE), then taint T is exposed. Loads additionally taint x with $hash(\sigma)$, indicating that the register might depend on an arbitrary memory location. For stores, we do not know the address we write to (as e is tainted), so we taint the entire memory with $hash(\sigma)$, indicating that we cannot give any guarantees about the memory contents anymore.

Expression Evaluation

$[\![n]\!]_\sigma = n \quad [\![x]\!]_\sigma = \sigma(x) \quad [\![\ominus e]\!]_\sigma = \ominus[\![e]\!]_\sigma \quad [\![e_1 \otimes e_2]\!]_\sigma = [\![e_1]\!]_\sigma \otimes [\![e_2]\!]_\sigma \quad [\![\textbf{ite}(e_1,e_2,e_3)]\!]_\sigma = \text{if } [\![e_1]\!]_\sigma \text{ then } [\![e_2]\!]_\sigma \text{ else } [\![e_3]\!]_\sigma$

***CT-SEQ* Contract**

BARRIER
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \qquad EC(\sigma,p) = \bot}{\langle \sigma, r\rangle \xrightarrow{\quad}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto a(\mathbf{pc})+1], r\rangle}$$

ASSIGN
$$\frac{p(\sigma(\mathbf{pc})) = x \leftarrow e \qquad x \neq \mathbf{pc} \qquad EC(\sigma,p) = \bot}{\langle \sigma, r\rangle \xrightarrow{\quad}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc})+1, x \mapsto [\![e]\!]_\sigma], r\rangle}$$

TERMINATE
$$\frac{p(\sigma(\mathbf{pc})) = \bot \qquad EC(\sigma,p) = \bot}{\langle \sigma, r\rangle \xrightarrow{\quad}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \bot], r\rangle}$$

RETURN
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{return} \qquad EC(\sigma,p) = \bot}{\langle \sigma, \ell \cdot r\rangle \xrightarrow{\texttt{pc}\ \ell}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \ell], r\rangle}$$

EXCEPTION
$$\frac{EC(\sigma,p) = \ell}{\langle \sigma, r\rangle \xrightarrow{\texttt{exc, pc}\ \ell}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \ell], \sigma(\mathbf{pc}) \cdot r\rangle}$$

LOAD
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{load}\ x,e \qquad x \neq \mathbf{pc} \qquad n = [\![e]\!]_\sigma \qquad EC(\sigma,p) = \bot}{\langle \sigma, r\rangle \xrightarrow{\texttt{load}\ n}{}^{\text{seq}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc})+1, x \mapsto \sigma(n)], r\rangle}$$

STORE
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{store}\ x,e \qquad n = [\![e]\!]_\sigma \qquad EC(\sigma,p) = \bot}{\langle \sigma, r\rangle \xrightarrow{\texttt{store}\ n}{}^{\text{seq}}_{\text{ct}} \langle \sigma[n \mapsto \sigma(x)], \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc})+1], r\rangle}$$

Figure 4: *CT-SEQ* contract for μASM programs with exceptions.

EXCEPTION
$$\frac{p(\sigma(\mathbf{pc})) = x \leftarrow e \qquad EC(\sigma,p) \neq \bot \qquad \sigma \xrightarrow{\texttt{exc, pc}\ \ell}{}^{\text{seq}}_{\text{ct}} \sigma' \qquad taint(e,\emptyset,\sigma(\mathbf{pc})) = T}{\langle \sigma, t, \infty\rangle \xrightarrow{\tau}{}^{\text{vs-ops}}_{\text{ct}} \langle \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc})+1], t[x \mapsto T], w\rangle \cdot \langle \sigma', t, \infty\rangle}$$

LOAD
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{load}\ x,e \qquad EC(\sigma,p) = \bot \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma' \qquad \neg tainted(e) \qquad t([\![e]\!]_\sigma) = T}{\langle \sigma, t, \omega+1\rangle \xrightarrow{\tau}{}^{\text{vs-ops}}_{\text{ct}} \langle \sigma', t[x \mapsto T], \omega\rangle \cdot s}$$

TAINTEDLOAD
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{load}\ x,e \qquad EC(\sigma,p) = \bot \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma' \qquad tainted(e) \qquad taint(e,t,\sigma(\mathbf{pc})) = T}{\langle \sigma, t, \omega+1\rangle \xrightarrow{T}{}^{\text{vs-ops}}_{\text{ct}} \langle \sigma', t[x \mapsto hash(\sigma)], \omega\rangle \cdot s}$$

STORE
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{store}\ x,e \qquad EC(\sigma,p) = \bot \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma' \qquad \neg tainted(e) \qquad [\![e]\!]_\sigma = n \qquad t(x) = T}{\langle \sigma, t, \omega+1\rangle \xrightarrow{\tau}{}^{\text{vs-ops}}_{\text{ct}} \langle \sigma', t[n \mapsto T], \omega\rangle \cdot s}$$

TAINTEDSTORE
$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{store}\ x,e \qquad EC(\sigma,p) = \bot \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}} \sigma' \qquad tainted(e) \qquad taint(e,t,\sigma(\mathbf{pc})) = T}{\langle \sigma, t, \omega+1\rangle \xrightarrow{T}{}^{\text{vs-ops}}_{\text{ct}} \langle \sigma', t[1 \mapsto hash(\sigma), 2 \mapsto hash(\sigma), \ldots], \omega\rangle \cdot s}$$

Figure 5: Excerpt from the *CT-VS-Ops* contract.