AMuLeT: Automated Design-Time Testing of Secure Speculation Countermeasures

Bo Fu*
University of Toronto
Toronto, Canada
fubof@cs.toronto.edu

Leo Tenenbaum*
University of Toronto
Toronto, Canada
leo.tenenbaum@mail.utoronto.ca

David Adler
University of Toronto
Toronto, Canada
d.adler@mail.utoronto.ca

Assaf Klein
Technion - Israel Institute of Technology
Haifa, Israel
assafklein@campus.technion.ac.il

Arpit Gogia
IMDEA Software Institute
Madrid, Spain
arpitgogia@proton.me

Alaa R. Alameldeen
Simon Fraser University
Burnaby, Canada
alaa@sfu.ca

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain
marco.guarnieri@imdea.org

Mark Silberstein
Technion - Israel Institute of Technology
Haifa, Israel
mark@ee.technion.ac.il

Oleksii Oleksenko
Azure Research, Microsoft
Cambridge, United Kingdom
oleksii.oleksenko@microsoft.com

Gururaj Saileshwar
University of Toronto
Toronto, Canada
gururaj@cs.toronto.edu

Abstract

In recent years, several hardware-based countermeasures proposed to mitigate Spectre attacks have been shown to be insecure. To enable the development of effective secure speculation countermeasures, we need *easy-to-use* tools that can automatically test their security guarantees *early-on* in the design phase to facilitate rapid prototyping.

This paper develops AMuLeT, the first tool capable of testing secure speculation countermeasures for speculative leakage early in their design phase in simulators. Our key idea is to leverage model-based relational testing tools that can detect speculative leaks in commercial CPUs, and apply them to micro-architectural simulators to test secure speculation defenses. We identify and overcome several challenges, including designing an expressive yet realistic attacker observer model in a simulator, overcoming the slow simulation speed, and searching the vast micro-architectural state space for potential vulnerabilities. AMuLeT speeds up test throughput by more than 10× compared to a naive design and uses techniques to amplify vulnerabilities to uncover them within a limited test budget. Using AMuLeT, we launch for the first time, a systematic, large-scale testing campaign of four secure speculation countermeasures from 2018 to 2024—InvisiSpec, CleanupSpec, STT, and SpecLFB—and uncover 3 known and 6 unknown bugs and vulnerabilities, within 3 hours of testing. We also show for the first time that the open-source implementation of SpecLFB is insecure.

CCS Concepts: • **Security and privacy → Security in hardware**.

Keywords: Side Channels, Spectre, Defenses, Fuzzing

ACM Reference Format:
Bo Fu, Leo Tenenbaum, David Adler, Assaf Klein, Arpit Gogia, Alaa R. Alameldeen, Marco Guarnieri, Mark Silberstein, Oleksii Oleksenko, and Gururaj Saileshwar. 2025. AMuLeT: Automated Design-Time Testing of Secure Speculation Countermeasures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716247

*Bo and Leo contributed equally as lead authors for this paper

1 Introduction

Spectre attacks [20] exploit speculative execution to access architecturally unreachable information and leak them via μarch side channels. To mitigate these leaks, many hardware countermeasures have been proposed using techniques like *invisible speculation* [19, 30, 37], *undo-based approaches* [29], and *tracking speculative flows* [22, 36, 40, 41]. Many of these countermeasures, however, have been shown to be insecure, often in months. For instance, invisible speculation is vulnerable to *speculative interference attacks* [3], whereas Cleanup-Spec [29] and STT [41] were shown to be insecure in later works [21, 22]. Even defenses proposed by CPU manufacturers have been broken by subsequent attacks

To enable effective secure speculation defenses, we need tools that can automatically test their security. These tools need to satisfy two requirements to be practically adopted:

R1: They need to be applicable *early* in the design phase to facilitate rapid prototyping of countermeasures.
R2: They need to be *easy to use* on existing design artifacts to increase adoption by computer architects.

Currently, computer architects lack tools that meet both requirements. *Formal methods* [9, 33, 34, 38] for reasoning about μarch leaks have limited scalability and require expertise which computer architects often lack (failing **R2**). For instance, Pensieve [38] requires formalizing μarch countermeasures in a dedicated modeling language. *RTL testing tools* [5, 16], while scalable and enabling pre-silicon testing, fail **R1**, as they are inapplicable early in design, when architects prototype features on simulators. A majority of secure speculation countermeasures are prototyped in simulators, making RTL-based tools unsuitable for testing them.

In this work, we address this gap with AMuLeT, the first tool that can test secure speculation countermeasures at design time in μarch simulators. AMuLeT enables Automated μ-architectural Leakage Testing, *i.e.*, the testing of a countermeasure for unexpected speculative leaks to discover vulnerabilities at design time. For practical adoption, we seek to avoid intrusive changes to either the simulator or the countermeasure being tested. For this, AMuLeT adapts *model-based relational testing (MRT) techniques* [25, 26], which found speculative leaks in commercial CPUs, to μarch simulators.

Following the MRT approach, AMuLeT tests the target defense (implemented on a μarch simulator) against a given *leakage contract* [14], an ISA-level model capturing the expected leakage. AMuLeT has two main components: (a) a *model* that maps program executions to *contract traces*, i.e., sequences of ISA-level observations capturing the expected leakage according to the contract, and (b) an *executor* that generates *μarch traces*, which capture the observable side-effects of speculative execution on the simulator. AMuLeT generates random programs executed on both the executor and the model, and compares the expected leaked information (captured by the contract traces) with the actual information leaked by the defense (captured by the μarch traces). Any discrepancy between the two (called a *contract violation*) indicates an unexpected leak, and therefore a potential security vulnerability in the defense.

While we reuse the test and contract trace generation from prior work [25, 26], implementing AMuLeT requires addressing three core challenges (indicated as C1–C3 below).

C1: What CPU state to expose in the μarch trace? The design of the μarch trace is critical in MRT as it captures the observational power of the attacker against which the security guarantees are tested. Simulators allow the definition of extremely expressive μarch traces comprising potentially the entire CPU μarch state. At the same time, μarch trace needs to be grounded in a realistic observer model , so that the detected violations may be exploitable on a real CPU.

In AMuLeT, we implement μarch traces by taking a snapshot of the final cache and TLB states of the test program extracted from the simulator. Our evaluations in § 4.4 show this μarch trace, capturing the observational power of an attacker exploiting memory-system side channels (*explicit channels* [41]), is sufficient to discover exploitable speculative leaks in several secure speculation countermeasures claiming to protect against these side channels. At the same time, it is also easily extensible to other attacker models. In § 4.3, we show that exposing more information in the μarch traces, e.g., branch predictor state or program counter sequence, can also detect *implicit channels* based on branch prediction or resolution, at the cost of reduced testing throughput.

C2: Slow testing speed. Running test cases in a simulator is slow. This severely limits the number of tests that can be executed during a campaign and reduces the chances of finding contract violations. Our key observation is that, counterintuitively, the main bottleneck is the *startup costs* of the simulator rather than the test runtime. This is because current MRT techniques [24–26] generate small test programs (a few tens of instructions) that run quickly (e.g., tens of milliseconds in gem5 [4]), whereas the simulator startup times are nearly two orders of magnitude higher.

To address this, we design a testing harness that runs successive tests without restarting the simulator by overwriting register and memory values between the tests, thereby amortizing the startup costs across multiple tests. Compared to restarting the simulator on each test, AMuLeT improves test throughput by over 10x, as shown in § 3.2.

C3: Low probability of discovering leaks. A key requirement for speculative leaks is contention on μarch resources. To amplify the chances of their occurrence, we test the design with smaller μarch structure sizes (e.g., fewer cache ways, fewer MSHRs), amplifying contention and making leaks easier to discover. With this, AMuLeT uncovers vulnerabilities that could not be detected otherwise, as shown in § 4.5.1.

Evaluation. We use AMuLeT to run the first systematic, large-scale testing campaign against secure speculation countermeasures (all implemented in the gem5 simulator). Our campaign covers an unprotected out-of-order CPU and four secure speculation countermeasures: InvisiSpec [37], CleanupSpec [29], STT [41], and SpecLFB [8]. We detect Spectre-v1 and Spectre-v4 vulnerabilities within minutes in the unmodified (insecure) CPU. In the countermeasures under test, we discover 3 unknown implementation bugs, 3 unknown vulnerabilities in the designs, and confirm 3 known vulnerabilities, all within three hours of testing.

In InvisiSpec, we discover a previously unknown bug in the cache eviction logic, which leaks speculatively accessed addresses via evictions. We also discovered a stronger variant of speculative interference attack [3]. While the prior attack requires a multi-threaded attacker and SMT support, our variant is exploitable by a single-threaded adversary, thereby breaking InvisiSpec's security in a single-threaded setting.

In CleanupSpec, we discovered several new bugs and vulnerabilities. We find leaks of speculatively accessed addresses due to (a) incorrect cleanups of non-speculative load addresses when they match with reordered speculative loads, a previously unknown vulnerability, and (b) a lack of cleanup for speculative stores and speculative requests crossing cache lines, due to implementation bugs. We also re-discovered unXpec [21], a previously known vulnerability.

In STT, AMuLeT automatically flagged a known vulnerability on speculative stores where the TLB is speculatively accessed by tainted stores, as shown previously by DOLMA [22].

In SpecLFB, a defense proposed in 2024, AMuLeT discovers a new vulnerability, similar to Spectre-v1, in the open-sourced gem5 implementation. This is due to an undocumented optimization in the implementation that removes protection for the first speculative load in the load-store queue, thereby making it vulnerable to Spectre attacks leaking secret registers with a single speculative load.

Summary of contributions:

1. We introduce AMuLeT, the first tool that can automatically find information leaks in simulated CPUs, and identify weaknesses in the proposed designs of secure speculation countermeasures.
2. We expose the μarch state realistically observable by an attacker, and introduce techniques to amplify the observability of violations in white-box simulators without intrusive changes to the simulator.
3. We identify performance bottlenecks and address them by streamlining the test case execution, improving the testing throughput by an order of magnitude.
4. We launch the first large-scale testing campaigns on several recent secure speculation countermeasures, InvisiSpec, CleanupSpec, STT, and SpecLFB, and automatically find multiple unknown bugs and vulnerabilities in three hours of testing on a commodity server.

We have responsibly disclosed our discoveries to the authors of the countermeasures. AMuLeT is open-sourced at https://github.com/sith-lab/amulet to aid the testing of future countermeasures as they are designed.

2 Background: Testing for speculative leaks

We first discuss leakage contracts [14], which model speculative leaks at the ISA level (§ 2.1), then describe how attackers can be modeled (§ 2.2), and how leakage contracts enable detecting unexpected leaks visible to attackers (§ 2.3). Finally, we overview Revizor [25] (§ 2.4), a testing tool for detecting leaks in CPUs, which we use as a basis for AMuLeT.

2.1 Leakage contracts modeling speculative leaks

Leakage contracts [14] capture the expected μarch leaks at the ISA level. A contract C describes, for any program p and input i, what information might be leaked microarchitecturally when executing p with input i. For this, contract C maps each execution to a *contract trace*, i.e., a sequence of ISA-level observations capturing leaked information. $C(p, i)$ denotes the contract trace for the execution of program p with input i.

Contracts are formalized by annotating ISA instructions with (a) an *observation clause* modeling the information leaked by the instruction, and (b) an *execution clause* modeling if (and how) instructions trigger speculation. Table 1 summarizes the contracts we use in evaluations in § 4:

• The `CT-SEQ` contract models the leakage expected by a CPU with cache side channels and without speculative execution. Its observation clause exposes the addresses of executed load and store instructions as well as the program counter throughout the execution. The execution clause is empty, indicating that no instruction triggers speculation and that the leakage is only on architectural execution paths.

• The `CT-COND` contract models the leakage expected by a CPU with branch prediction. While its observation clause is the same as for `CT-SEQ`, the execution clause specifies that when a conditional branch is executed, the corresponding mis-predicted branch should be explored as well. This captures instructions transiently executed due to branch prediction.

• Finally, the `ARCH-SEQ` contract exposes the program counter, the location of all loads and stores, and the values of all data loaded from memory on architectural program paths. For this, `ARCH-SEQ` extends the observation clause in `CT-SEQ` by additionally exposing the values loaded from memory. As in `CT-SEQ`, the execution clause is empty.

Table 1. Leakage contracts used in this work.

Name	Clauses	
	Leakage	**Execution**
CT-SEQ	PC, LD/ST ADDR	N/A
CT-COND	PC, LD/ST ADDR	Mispredicted Branches
ARCH-SEQ	PC, LD/ST ADDR and values	N/A

2.2 Modeling attackers in speculation-based attacks

In speculation-based attacks, attackers extract information about a victim program by observing changes in the CPU's μarch state through side channels [27, 39]. Like prior works [14, 25], we model attacker observations using μarch traces from victim execution, where each trace captures the μarch changes observed by an attacker. For a program p starting from an input i and μarch context μ (the initial CPU μarch state), $\mu Trace(p, i, \mu)$ denotes the μarch trace for this execution. In § 3, we describe the μarch traces used in AMuLeT.

2.3 Detecting unexpected leakages

A leakage contract captures the *expected leakage* of a CPU under test. Any *unexpected* leak is a contract violation, where the μarch traces leak more information than the contract traces. This is defined precisely as follows:

Definition 2.1 (Contract violation [14])**.** A CPU violates a contract C if there exists a program p, two inputs i, i', and a microarchitectural context μ such that $C(p, i) = C(p, i')$ and $\mu Trace(p, i, \mu) \neq \mu Trace(p, i', \mu)$.

A violation is evidence of an unexpected leak in the CPU as the attacker can distinguish two executions ($\mu Trace(p, i, \mu) \neq \mu Trace(p, i', \mu)$) that should be indistinguishable based on the contract ($C(p, i) = C(p, i')$).

2.4 Model-based relational testing: Revizor

Model-based relational testing (MRT) tools [24, 25] discover unexpected leaks in CPUs by searching for contract violations, and have been successful in finding new vulnerabilities and variants of existing ones [15, 24–26]. Here, we focus on Revizor, the MRT tool we use as a basis for AMuLeT.

Revizor searches for contract violations by (1) generating a random program p and a sequence of random inputs $[i_0, i_1, \ldots]$, (2) collecting the contract and μarch traces for all programs and inputs, and (3) analyzing the traces to identify violations according to Definition 2.1. The testing continues for a fixed number of rounds or until a violation is detected. We now provide further details on the core parts of Revizor:

- *Program generation*: The generator selects a random sequence of assembly instructions from a pool to form a program. It can be configured to constrain the shape of the program's control-flow graph, control the pool of instructions, and configure the instruction frequencies.
- *Input generation*: Each input is a binary file, generated with a (seeded) pseudo-random number generator, that initializes the test program's memory and registers. Inputs can also be mutated, preserving only the parts influencing the contract trace while randomizing others to ensure identical contract traces but potentially different speculative behavior.
- *Collecting contract traces:* Revizor implements an executable version of the contract (called *leakage model*) on top of the Unicorn ISA emulator [1] by (1) adding instrumentation to record observations according to the contract's observation clause, and (2) simulating speculative execution paths as per the execution clause. Revizor collects contract traces by executing the program p with all the inputs $[i_0, i_1, \ldots]$ using the leakage model and recording the observations.
- *Collecting μarch traces:* Revizor implements an *executor* that takes a program p, executes it on the target CPU with each of the inputs $[i_0, i_1, \ldots]$, and measures the μarch trace for each execution. These traces are collected via a side channel attack, like Prime+Probe, where each trace is a set of cache lines evicted by the program.
- *Comparing leakage*: At the end of each testing round, Revizor compares the collected contract and μarch traces to detect violations according to Definition 2.1.

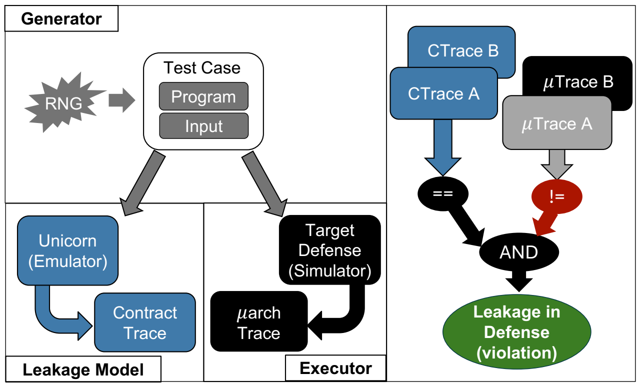

Figure 1. Overview of AMuLeT. We leverage the test-generation and leakage models from prior works [15, 25, 26] and design a new executor in AMuLeT capable of testing target defenses in a μarch simulator.

3 AMuLeT Design

AMuLeT builds on Revizor [25], an MRT tool that finds leaks in silicon CPUs, to enable design-time testing of secure speculation mechanisms in simulators. Next, we provide a high-level overview of AMuLeT, highlight the key challenges in testing designs in a simulator, and explain our solutions.

3.1 Overview of AMuLeT

AMuLeT consists of three main modules, as shown in Figure 1: (1) the *test generator*, which generates random programs and inputs, (2) the *leakage model*, which generates the contract traces, and (3) the *executor*, which generates μarch traces from a simulator implementing the countermeasure under test. Below, we describe each component briefly.

Test Generator. AMuLeT reuses the test generator from Revizor [25] (described in § 2.4) to generate short test programs of up to 5 basic blocks of randomly selected instructions, linked together by jumps in the form of a directed acyclic control flow graph. All memory accesses are forced to access a predefined and initialized memory sandbox. We vary the number of 4KB pages in the sandbox from 1 to 128. Several random inputs are generated for each test program. A combination of a program and input forms a test case.

Leakage Model. AMuLeT reuses the leakage model from Revizor [25] to collect contract traces by executing each test case on the Unicorn [1] CPU emulator. For each of the target defenses, we test them against a contract that matches their security guarantees, following the formal analysis by Guarnieri et al. [14, Section VI]. Specifically, we use the `CT-SEQ` contract for testing InvisiSpec [37] (Futuristic), CleanupSpec [29], and SpecLFB [8], and the `ARCH-SEQ` contract for testing STT [41] (Futuristic).

Executor. The executor generates the μarch traces from a simulator implementing the countermeasure under test. μarch traces model attacker observations for a given countermeasure. When two tests have matching contract traces but different μarch traces, we flag that as a contract violation (cf. Definition 2.1), indicating a speculative leak (and a potential vulnerability). Thus, the design of the μarch trace is critical in determining the types of leaks that can be detected.

3.2 μarch Trace Design - Challenges and Solutions

As the μarch traces model the attacker observations, they play a critical role in determining the speed and efficacy of testing for leaks. Below, we highlight the key challenges in designing the μarch trace in AMuLeT and then our solutions.

C1. Determining μArch States Exposed in μArch Trace
A simulator provides white-box access to the *entire* CPU's μarch state, which can be potentially exposed via the μarch trace. However, exposing information that is too detailed may reduce the testing throughput. Moreover, although more detailed μarch traces can result in more contract violations (cf. Definition 2.1), not all discovered violations may lead to exploitable leaks for software-based attackers.

Consider the following options for the μarch trace, which expose different μarch information. The first uses a snapshot of the cache and TLB state at each test case's end (i.e., L1D-cache and D-TLB tags). This models a realistic software-based attacker inferring the cache or TLB state by performing memory accesses and checking for cache or TLB hits/misses (an *explicit channel* as per the STT taxonomy [41]). The second option uses a snapshot of the branch-predictor (BP) state (consisting of local/global history tables, branch target buffer, etc.) at the end of the test case. This models a more sophisticated attacker that can infer information about *how* program branches have been executed from the secret-dependent BP state (this is an example of an *implicit channel based on prediction* in STT's taxonomy).

Finally, the third option models an attacker physically probing the hardware, i.e., monitoring μarch state transitions throughout the execution of the test case. For example, monitoring the sequence of program counter values (PC) and branches, or monitoring each transaction on the L1-Cache bus exposing the sequence of addresses of each load/store. This provides much more precise and exhaustive information. For instance, differences in the sequence of PCs can detect *implicit channels* based on branch resolution (as per STT's taxonomy), whereas a secret-dependent reordering of loads may induce a difference in the cache replacement state. However, not all of these transient differences may be exploitable in practice; some of these secret-dependent orderings may not result in changes observable by a realistic attacker (e.g., reordered loads map to different cache sets).

In § 4.3, we evaluate testing campaigns using each of these three μarch trace formats and show that AMuLeT can effectively detect different kinds of violations regardless of the trace format. By default, in AMuLeT, we use state-based μarch traces to model the most realistic attacker. We use the data cache and TLB snapshots (first option) for the μarch traces, given that such memory-system-based side channels are exploited in the majority of speculation-based attacks and most defenses [8, 29, 37, 41] that we seek to test with AMuLeT protect against at least these side channels.

In AMuLeT, we use the μarch trace comprising a snapshot of the final TLB and caches state, which models a realistic attacker observing memory-system side channels.

Our evaluation in § 4.4 shows that such a μarch trace provides sufficient information to discover exploitable violations across a wide variety of defenses. We remark, however, that any other μarch state observable to an attacker (e.g., second or third option) can also be used as part of the μarch trace.

C2. Determining Initial State for μArch Traces. Prior works [25, 26] collect μarch traces on real CPUs using cache side channels like Flush+Reload [39] or Prime+Probe [27]. While we directly extract the final state of the cache from the simulator and do not need to infer it via a side channel, this still leaves the question: What should be the initial state of the caches before starting a test case?

The most intuitive approach is to start deterministically from a clean cache state, thus eliminating noise.

We observe that the best results are obtained when initializing the L1 cache by filling it up with addresses from outside the memory sandbox of the test case. This translates into 64 x 8 addresses for an 8-way, 32KB L1 cache.

Starting from a fully occupied cache set ensures we not only detect leakages due to speculative cache line installs (addresses installed by the program), but also due to replacements (evicted addresses). In our evaluation (§ 4.2), we show that initializing the cache state in this way increases the number of detected violations compared to the naive approach.

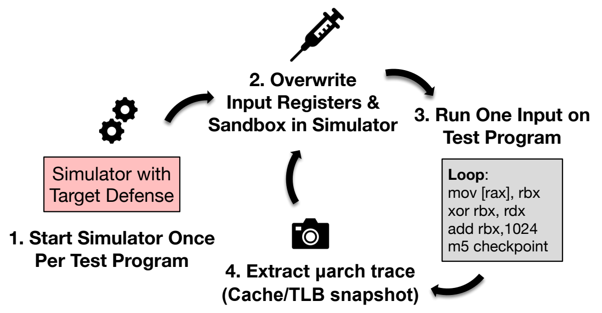
2. Overwrite Input Registers & Sandbox in Simulator
3. Run One Input on Test Program
Simulator with Target Defense
Loop:
mov [rax], rbx
xor rbx, rdx
add rbx,1024
m5 checkpoint
1. Start Simulator Once Per Test Program
4. Extract µarch trace (Cache/TLB snapshot)

Figure 2. Design of μarch trace extraction in AMuLeT-Opt.

C3. Alleviating Slowdown Due to Slow Simulator Startup. While MRT tools for testing silicon CPUs face a performance bottleneck due to test and contract trace generation [26], the bottleneck for testing countermeasures in simulators is the simulation runtime, i.e., the time to execute the test and extract the μarch trace. The naive approach of generating a μarch trace for each test program and input involves packaging the test case (program and input) in a binary, running the binary on the simulator, and extracting the μarch trace at the end of the simulation. Unfortunately, we observe that this approach is not very performant due to the use of short test cases (approximately 50 instructions per test case)[1].

Characterizing AMuLeT-Naive. We analyze the execution time for a single test program in Table 2. We run tests on the default Out-of-Order CPU (non-secure) in gem5, which we run in SE mode. We see that 97% of the time is spent in gem5 as expected; however, 96.1% of the time is spent in gem5 start-up, and only 0.9% is spent simulating the instructions. This is because gem5 takes a few seconds to initialize, but just tens of milliseconds to run a test case of ~50 instructions. Thus, the startup time is the primary bottleneck.

Table 2. Breakdown of time per test program, using a Naive and Optimized μarch trace extraction in AMuLeT (with 140 inputs/program), in a campaign running 30 test programs.

Component	Naive	Opt
gem5 startup	156 s (96.1%)	0.2 s (1.6%)
gem5 simulate	1.4 s (0.9%)	11 s (88.5%)
μTrace extraction	0.9 s (0.5%)	0.6 s (4.6%)
Test generation	0.5 s (0.3%)	0.3 s (2.5%)
CTrace extraction	0.1 s (0.1%)	0.1 s (0.6%)
Others	3.4 s (2.1%)	0.3 s (2.2%)
Total	159 s (100%)	12 s (100%)

AMuLeT-Opt. We adopt an optimized μarch trace extraction method in AMuLeT (Opt in Table 2). Rather than restarting the simulation for each program input, we continue to execute test cases for different inputs of the same program by directly overwriting the register and memory values in the simulated binary in the gem5 process without restarting the simulator. As shown in Figure 2, we start the simulation with a binary that loops over the test program instructions. After each successive input, we overwrite the input register values in the binary and continue the simulation. At the end of each iteration, we extract the corresponding μarch trace.

[1]Short test cases are beneficial to ensure that side effects of speculation are observable in the final cache state obtained at the end of the test case.

Unlike AMuLeT-Naive, which restarts the simulator for each input, AMuLeT-Opt only restarts it for each test program, so the startup cost is amortized across all inputs.

Benefits of AMuLeT-Opt. As shown in the *Opt* column of Table 2, the gem5 startup time is significantly reduced (consuming 2% of the time), and the bottleneck is now the time for simulating instructions (consuming 89% of the time). Note that our simulation time per program increases, since we need 10x more instructions to reset the cache state per test input with addresses from outside the sandbox (64×8 instructions for a 8-way, 32KB L1D-cache). We considered adding a special custom instruction to reset the cache to further reduce the time per test but ruled out this idea to avoid intrusive changes to the design that may potentially change its behavior. The time per test program for AMuLeT-Opt is just 12 seconds, 13x lower compared to AMuLeT-Naive, which takes 2.7 minutes per test program with all its inputs.

AMuLeT-Opt has an additional benefit that it preserves the μarch state of the predictors (like branch predictor or memory dependence predictor) between test cases. This is beneficial for finding leaks as it results in a wider variety of predictions in successive inputs. However, this also means that a violation might be due to differences in the initial μarch context. Following prior work [25], we *validate* a violation by re-running the violating inputs with the other test case's μarch starting context and check if the violation persists. We provide detailed comparisons of the speed of testing and efficacy in finding violations between Naive and Opt in § 4.2.

3.3 Analyzing Violations

Violations are detected by AMuLeT when there is a difference in the μarch traces (final D-cache or TLB state) for two inputs to a program with the same contract trace (cf. Definition 2.1). On detecting a violation, AMuLeT outputs the program and the pair of inputs causing the violation with their μarch traces. Our violation analysis workflow, shown in Figure 3, has two steps: (a) root cause analysis of a given violation and (b) identifying unique violations by filtering out similar ones.

(a) Root Cause Analysis. Like any fuzzing approach, identifying the root cause of a violation in AMuLeT is a manual process. This is because fuzzing typically automates the testing process, whereas root causing detected vulnerabilities is orthogonal and often dependent on the vulnerability itself.

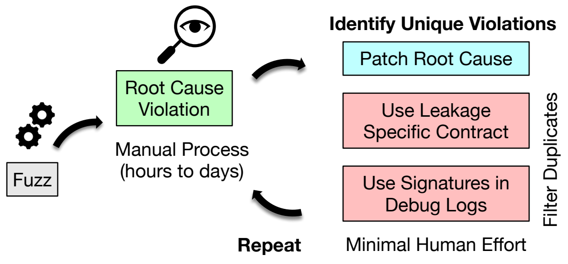
Fuzz
Root Cause Violation
Manual Process (hours to days)
Identify Unique Violations
Patch Root Cause
Use Leakage Specific Contract
Use Signatures in Debug Logs
Filter Duplicates
Minimal Human Effort
Repeat

Figure 3. Analyzing violations discovered with AMuLeT.

We analyze violations in AMuLeT that manifest as differences in the D-Cache and TLB-based μarch traces, by first identifying the load instruction responsible for the differing addresses in the μarch traces. We then trace back along the program data flow to find the mis-speculated instruction dependent on varying inputs (the source of the leak). This identifies the entire mis-speculated instruction sequence causing the violation. We do this using a script that parses gem5's debug logs (containing information like load/store addresses, branch prediction, etc.) and provides a side-by-side comparison of memory accesses under the two violating inputs and highlights differences. It also displays squashes, which helps locate the cause of the mis-speculation. Once the instruction sequence causing the violation is thus identified, we examine the gem5 debug logs for this sequence to pinpoint the code in the defense causing the leakage. For violations discovered in this work, root cause analysis took a few hours to few days, based on the complexity of a violation.

(b) Identifying Unique Violations. Once we root cause a given violation, we avoid re-discovering similar violations in two ways: (1) fixing the root cause when possible, (2) filtering similar violations using contracts that expose this leakage, or by inspecting the debug logs. When fixing the vulnerability in gem5 took less than 10 lines of code, we wrote a patch and re-ran the violations to see which got resolved—this worked for violations UV1 and UV3 in InvisiSpec and CleanupSpec (§§ 4.5 and 4.6) with minimal effort.

If the vulnerability was fundamental to a defense and hard to patch, we used a contract exposing this leakage in the contract trace to filter these violations, as in prior works [25, 26]—this approach worked for KV1 and UV6 in InvisiSpec and SpecLFB (§§ 4.5 and 4.7) and required few lines of code in AMuLeT. When characterizing leaks at the contract level was not straightforward, we identified a unique signature for a violation based on patterns in the μarch traces or simulator debug logs and used regex-based scripts to filter similar violations. For example, InvisiSpec violation (UV2; § 4.5) shows MSHR-related stalls in debug logs due to speculative interference, while CleanupSpec violation (UV4; § 4.6) shows load requests crossing cache line boundaries in debug logs. This approach, which is akin to defining a leakage-specific contract, isolates violations with similar signatures, and worked for the remaining violations in InvisiSpec and CleanupSpec (§§ 4.5 and 4.6). We continue steps (a) and (b) until all violations are root caused. When left with a handful of violations in a defense (e.g., less than 10), we manually inspect the debug logs to verify that all of them have the same signature—this approach worked for STT (§ 4.8). Overall, identifying unique violations takes minimal manual effort.

3.4 Amplifying Leakages in Simulators

So far, AMuLeT directly tests any CPU design or simulator-based countermeasure for leaks without any modifications, as if it were a black-box. However, programs that induce leakages can be fundamentally hard to find with random testing, and exhaustively traversing the vast μarch state space of a design to check for leakage is impractical.

To efficiently uncover such leaks, we leverage the fact that observing speculative leakage has two requirements: (1) a speculative "access" instruction that reads the data to be leaked, and a (2) speculative "transmitter" instruction, that leaks the data via contention on a μarch resource (i.e., a covert channel) and impacts the μarch trace. To make leaks easier to observe, we increase the chance of contention on μarch structures by configuring the target design with smaller μarch structure(s) (e.g., smaller L1-Caches, fewer cache ways, fewer MSHRs). This approach makes covert channels, where contention is unlikely to occur in short test cases (e.g., Prime+Probe, MSHR contention), more likely, thus increasing the chance of detecting speculative leaks.

Note that we do not modify the defense itself or test infeasible CPU configurations. We use valid configurations of these structures with reduced sizes to increase the probability of short test cases inducing contention. This will make any speculative leaks that exist in the design more observable.

In evaluations, we show that AMuLeT can already discover leaks in countermeasures while testing with default configurations, as we show in § 4.4; but it can discover more interesting leaks with amplification, as we show in § 4.5.1.

3.5 Implementation

We integrate AMuLeT with the gem5 simulator [4]. Our test and contract trace generation build on Revizor's implementation [25]. We add μarch trace extraction to the gem5 code of each defense, and run gem5 in Syscall Emulation (SE) mode.

For the μarch trace, we use the final state of the L1D-cache and D-TLB extracted from the simulator at the end of a test case. We reset the L1D-cache state after each test case by filling it with addresses from pages outside the memory sandbox, which also evicts the TLB entries (for InvisiSpec and STT), or by invalidating the caches directly using a simulator hook if this is supported by the specific simulator version (for SpecLFB and CleanupSpec). For InvisiSpec, CleanupSpec, and SpecLFB, we use a sandbox with 1 physical page (as the TLB is not protected), whereas for STT we use a sandbox with 128 pages since we seek to test it also for TLB leaks.

4 Evaluation

Here, we answer the following research questions:

RQ1 Can AMuLeT detect known leaks in non-secure CPUs? How fast is AMuLeT-Opt compared to AMuLeT-Naive?
RQ2 How does the choice of the μarch trace format affect the number and kinds of violations found?
RQ3 Can AMuLeT detect known and unknown leaks in defenses? How effective is our leakage amplification?
RQ4 Can AMuLeT detect more interesting and more fundamental vulnerabilities, as we fix the simpler ones?

We answer RQ1 and RQ2 by testing the Out-of-Order gem5 CPU with AMuLeT (§ 4.2–4.3), and RQ3 and RQ4 by testing InvisiSpec, CleanupSpec, STT, and SpecLFB (§ 4.4–4.8).

4.1 Evaluation Methodology

We use AMuLeT to test the baseline Out-of-Order CPU in gem5 (O3CPU) and four different countermeasures: InvisiSpec [37], CleanupSpec [29], STT [41], and SpecLFB [8]. For the baseline, we test the insecure O3CPU in the gem5 code-base from InvisiSpec. For each countermeasure, we test its publicly available implementation, run with the default configuration flags. For InvisiSpec and STT, we run them in their Futuristic mode, as it provides the strongest security.

We run our testing campaigns on an AMD EPYC 128-Core CPU. For comparisons between AMuLeT-Naive and -Opt (§ 4.2), due to the slower speed of AMuLeT-Naive, we run shorter test campaign of 16 parallel instances of AMuLeT, with each instance executing 100 test programs and 140 inputs per program (224k test cases in total). For testing InvisiSpec, CleanupSpec, STT, and SpecLFB (§ 4.4), we run 100 parallel instances, each executing 200 test programs, and 140 inputs per program (2.8M test cases in total).

On discovering a violation, we analyze it to identify its root cause following the process in § 3.3.

4.2 Testing Baseline Out-of-Order CPU

To answer RQ1, we test the insecure baseline O3CPU against two contracts: `CT-SEQ`, which allows cache-based leaks on sequential execution, and `CT-COND`, which additionally allows leaks on mispredicted branches. We run campaigns with AMuLeT-Naive and AMuLeT-Opt using the default trace format (L1D-cache and D-TLB). Table 3 shows the execution time, number of violating test cases detected, and detection time per violation, averaged over 16 parallel runs.

Detected Violations. AMuLeT-Naive detected violations against `CT-SEQ`. These were due to branch mispredictions, namely instances of Spectre-v1, where a value was leaked speculatively on a mispredicted conditional jump instruction.

AMuLeT-Opt found violations of both `CT-SEQ` and `CT-COND`. The latter were instances of Spectre-v4, where a store is speculatively bypassed by a younger load to the same address, which reads the value in memory from before the store and leaks it by encoding it in a subsequent load's address.

Table 3. Results of testing the baseline Out-of-Order CPU.

Metric	Contract	Naive	Opt	Ratio
Time (minutes)	CT-SEQ	289	25	11.7x
	CT-COND	289	33	8.7x
Number of violations	CT-SEQ	5.8	9.9	1.7x
	CT-COND	0	0.1	N/A
Detection time (minutes)	CT-SEQ	49.8	2.5	19.9x
	CT-COND	N/A	330	N/A

Execution Time. The test campaign with AMuLeT-Naive took 289 minutes (13 test cases per second). AMuLeT-Opt improved on this, taking up to 33 minutes per campaign (114 test cases per second). This speedup is due to the amortized startup cost of the simulator. AMuLeT-Opt sometimes requires additional validation of violations, which explains the difference in time for `CT-SEQ` and `CT-COND`. However, the reduction in startup cost significantly outweighs the extra validation cost, leading to a net speedup of 9–11x.

Detection Time. AMuLeT-Naive detects `CT-SEQ` violations in less than 1 hour on average, demonstrating that even a basic AMuLeT implementation is useful in detecting leaks. AMuLeT-Opt detects a `CT-SEQ` violation in 2.5 minutes (20x faster) and a `CT-COND` violation in 6.5 hours. It takes longer to discover Spectre-v4 (`CT-COND` violation) compared to Spectre-v1 (`CT-SEQ` violation) as the probability that a load and store address match and mispredict is low in random tests.

AMuLeT-Opt finds more violations compared to AMuLeT-Naive with the same number of tests because it initializes the cache with full sets, which results in violations through speculative installs and also via evictions. AMuLeT-Opt also preserves the branch predictor and memory-dependence predictor state between inputs, which allows a broader set of predictions and increases the chances of a violation.

As AMuLeT-Opt performs significantly better, we only use this version next and we refer to it simply as AMuLeT.

4.3 Evaluating different μarch trace formats

To answer RQ2 and assess the trade-off between precision, performance, and percentage of violating test cases detected by different μarch traces, we ran testing campaigns on the baseline Out-of-Order CPU with four types of μarch traces exposing different fine-grained μarch information:

- **Baseline (L1D+TLB):** The μarch trace consists of the final addresses in the L1D cache and D-TLB.
- **BP State:** The μarch trace consists of the final state of the branch predictor (BP) and the end of the test case.
- **Memory access order:** The μarch trace consists of the ordered list of all memory accesses (PCs and addresses).
- **Branch prediction order:** The μarch trace consists of the ordered list of branch PCs and their predicted targets.

We measure the *fraction of total violations* as the percentage of violations detected by a trace format divided by the

Table 4. Results of testing InvisiSpec, CleanupSpec, STT, SpecLFB, and the baseline Out-of-Order CPU with AMuLeT-Opt. The campaigns consisted of 100 parallel instances of AMuLeT, each executing 200 programs, each with 140 inputs.

Defense	Contract	Detected Violation?	Avg. Detection Time (sec.)	Number of Unique Violations	Testing Throughput (test cases/sec.)	Campaign Execution Time
Baseline	CT-SEQ	YES	1.7	2	752	1 hr 2 min
InvisiSpec	CT-SEQ	YES	2.1	1	630	1 hr 14 min
CleanupSpec	CT-SEQ	YES	1.1	3	2592	18 min
SpecLFB	CT-SEQ	YES	1.6	1	2595	18 min
STT	ARCH-SEQ	YES	10371	1	34	23 hr 3 min

total violations detected by any trace format. For each of the formats, we also report the fraction of their violations also detected by baseline trace format. Table 5 shows the results.

Table 5. Results of testing the baseline O3CPU with different μarch trace formats, across 100 parallel instances of AMuLeT each executing 200 programs, each with 140 inputs, compared to baseline trace (L1D Cache and TLB).

Trace format	Throughput (test cases / sec.)	Fraction of total violations	Violations covered by baseline trace
Baseline (L1D+TLB)	580	79.8%	100%
BP state	27	6.9%	70.8%
Memory access order	67	91.9%	80.9%
Branch prediction order	302	2.6%	77.8%

The BP state, branch prediction order, and memory access order μarch traces have lower throughput than the baseline. This is because these μarch traces suffer from additional *validations*. Recall from § 3.2 that violations may be observed due to difference in the initial μarch context rather than inputs, and AMuLeT confirms or rejects a violation by re-running the inputs with the same initial μarch state. These traces are more impacted by varying initial μarch states, causing more validations and lowering the test throughput.

The baseline μarch trace consisting of the final L1D cache and D-TLB state detects almost 80% of the total violating test cases without significantly slowing down the fuzzer. Although the memory access order trace detects a higher fraction (92%) of the violations, it is an order of magnitude slower in throughput due to extra validations, and not all of its violations may be directly exploitable by a realistic attacker. While we confirm that the BP state or branch prediction order traces detect instances of implicit channels based on branch predictions, a majority (>70%) of these violations are also detected by the baseline μarch trace, since differences in the speculative control flow may also manifest as differences in data-flow and accessed addresses.

The baseline μarch trace (L1D cache and D-TLB state) provides the best trade-off between speed and coverage. Since most defenses [8, 29, 37, 41] protect against leaks through the cache, we use the baseline μarch trace (L1D cache and D-TLB state) in the rest of the evaluation. This is sufficient to detect leakages in all defenses we test.

4.4 Testing Defenses with AMuLeT

To answer RQ3 and RQ4, we test four secure speculation mechanisms, InvisiSpec, CleanupSpec, STT, and SpecLFB. We also test the baseline CPU as a comparison point. We test InvisiSpec, CleanupSpec, and SpecLFB against `CT-SEQ` as these defenses claim to protect against speculative memory-system side channels [8, 29, 37]. We test STT with `ARCH-SEQ`, which captures STT's non-interference guarantee [14, 41]. Table 4 shows the results of these campaigns.

Types of Violations. For the baseline non-secure CPU, we find similar violations as previously discussed in § 4.2. We also find contract violations for InvisiSpec, CleanupSpec, STT, and SpecLFB and analyze their root causes in § 4.5–§ 4.8. Surprisingly, a majority of these are new leakages that differ from previously known vulnerabilities [3, 21, 22].

Performance. The violations in InvisiSpec, CleanupSpec, and SpecLFB were discovered in less than 3 seconds on average. This is comparable to the baseline CPU, whereas on STT, it takes longer (3 hours on average). CleanupSpec and SpecFLB tests are faster than InvisiSpec (2590+ vs 630 test cases/second) as they start from a clean cache state (which only requires flushes to the sandbox addresses), whereas InvisiSpec requires filling the cache with addresses that conflict with the sandbox, requiring more instructions.

STT tests are much slower (34 test cases/second) as its simulation on Gem5 is 17x slower than InvisiSpec, due to its higher complexity. The overall testing performance for the Baseline is higher compared to § 4.2 (752 vs 114 test cases/second for the baseline) because we run 100 parallel instances of AMuLeT in this campaign instead of 16.

4.5 Vulnerabilities Found in InvisiSpec

InvisiSpec's design [37] claims that speculative loads are invisible to the caches; when loads become safe, an *expose* operation makes the load visible to the caches (installing and evicting cache lines). Below, we describe the leaks we discovered in InvisiSpec, differentiating between Known Vulnerabilities (KV) and Unknown Vulnerabilities (UV).

UV1. Speculative L1 D-Cache Evictions. All violations in InvisiSpec in Table 4 were due to a previously unknown vulnerability caused by an implementation bug. Figure 4 shows one such test that caused a violation. Figure 4a (Line 11) shows a mis-speculated load, whose address depends on an input (rbx). Based on the input, the speculative load's address differs, as shown in Figure 4b. We observe that this speculative address is leaked based on an address evicted from the cache. In Input A, the speculatively accessed address is 0x3a00, which evicts the valid address 0x13a00 (initially in the L1D-cache) and is thus absent in the final state. Instead, in Input B, the accessed address 0x3100 evicts 0x13100, which is absent from the μarch trace.

```
 1  .bb_main.2:
 2  OR byte ptr [R14 + RDX], AL
 3  LOOPNE .bb_main.3
 4  JMP .bb_main.exit
 5
 6  .bb_main.3: # misspeculated
 7  AND BL, 34
 8  AND RAX, 0b111111111111
 9  CMOVNBE SI, word ptr [R14 + RAX]
10  AND RBX, 0b111111111111
11  XOR qword ptr [R14 + RBX], RDI
```

(a) Test Asm Causing Violation

L1D-cache Tags in μarch trace

```
Input A
... 0x130c0 0x13100 ... 0x139c0 ...
Input B
... 0x130c0 ... 0x139c0 0x13a00 ...
```

Speculative Load in Program

```
Input A
Load Line-11, Addr: 0x3a10
Input B
Load Line-11, Addr: 0x3110
```

(b) Two inputs to Asm that evict different addresses

Figure 4. Example of Test Asm Causing Violation in InvisiSpec due to Speculative L1D-cache Evictions.

Listing 1. InvisiSpec's Speculative L1-Cache Eviction Bug

```
 1      // Cache Miss
 2      if (L1Dcache.cacheAvail(in_msg.LineAddress)) {
 3          // Space Available, Send Request to L2
 4          trigger(mandatory_request_type_to_event
 5          (in_msg.Type), in_msg.LineAddress, ...);
 6
 7      } else { // No Space Available
 8          // L1 Eviction (Even For Speculative Requests)
 9          trigger(Event:L1_Replacement,
10          L1Dcache.cacheProbe(in_msg.LineAddress), ...);
11      }
```

Listing 2. Patch for InvisiSpec Speculative Eviction Issue

```
    if (L1Dcache.cacheAvail(in_msg.LineAddress)
        || (in_msg.Type == RubyRequestType:SPEC_LD)) {
        // Space Available or Spec Load, Send Request to L2
        trigger(mandatory_request_type_to_event
        (in_msg.Type), in_msg.LineAddress, ...);

    } else { // L1 Eviction Only for Non-Speculative Requests
        trigger(Event:L1_Replacement, ...
```

Figure 5. Bug in InvisiSpec's Gem5 implementation discovered by AMuLeT that leaks addresses of speculative loads via L1D-cache evictions and breaks its security guarantees.

Root Cause: While speculative loads in InvisiSpec are supposed to be invisible to the cache hierarchy, we observe that its Gem5 implementation does not match this. As shown in the code snippet from InvisiSpec in Listing 1 (line 9), on a speculative load causing a L1D-cache miss, if a cache set is fully occupied, we observe that a speculative load initiates an L1D-cache replacement regardless of whether it is safe or not to do so. So, a mis-speculated load might evict a conflicting address from the L1D-cache, thus leaking its address and breaking InvisiSpec's security guarantees.

Fix: This issue is an implementation bug and can be easily patched. As shown in Listing 2, we modify the implementation so that L1-replacements are only executed on non-speculative loads, i.e., when the load is safe and ready to be exposed to the cache hierarchy. This fixes the leakage, as campaigns after the patch in § 4.5.1 found no violations.

KV1. Speculative Instruction Fetches. In previous campaigns, when we included the L1I-cache in the μarch trace, we detected violations where L1I-cache state differed between two inputs. This is because, based on an input, the execution time can vary due to differences in speculative hits and misses, causing differences in instruction fetch behavior that speculatively brings additional lines into the L1I-cache. InvisiSpec [37] acknowledges that it does not protect the L1I-cache, making this a known vulnerability. This shows AMuLeT's ability to detect unprotected threat vectors.

Table 6. Results of testing InvisiSpec (Patched) with smaller μarch structures (fewer L1D-cache ways and MSHRs)

InvisiSpec Configuration	Time	Violation
Patched, 8-way L1D, 256 MSHRs	52 min	✗
Patched, 2-way L1D, 256 MSHRs	20 min	✗
Patched, 2-way L1D, 2 MSHRs	26 min	✓

4.5.1 Amplifying the Vulnerability in InvisiSpec.

We add the bug-fix in Listing 2 and continue our testing campaigns of InvisiSpec (Patched), running 100 parallel runs of 200 tests. As shown in Table 6, we do not observe further violations after patching with the default, 8-way L1D-cache. To answer RQ3 and see if we can amplify the vulnerability, we test with smaller μarch structures that experience higher contention. We run campaigns with 2-way L1D-cache (down from 8 ways) and with D-cache Miss-Status-Handling Registers (MSHRs) reduced to 2 (down from the default 256).

Table 6 shows that reducing the L1D-cache from 8 to 2 ways speeds up fuzzing campaigns by 2.6×. This is because initializing a smaller L1D-cache requires fewer instructions per test case. But this does not result in new violations. However, reducing MSHRs from 256 to 2 reveals new violations. These leaks stem from a previously unknown variant of the *speculative-interference attack* [3]. Unlike prior attacks, which requires a multi-threaded attacker and SMT, our new variant (**UV2**) allows attacker observations from the same core, breaking InvisiSpec's secuity in a single-threaded setting.

```
1  .bb_main.1:
2  AND DL, 7
3  AND RSI, 0b111111111111
4  CMOVL EAX, dword ptr [R14 + RSI]
5  ...
6  OR qword ptr [R14 + RCX], RAX
7  JNO .bb_main.2
8  JMP .bb_main.exit
9
10 .bb_main.2: # misspeculated
11 AND RCX, 0b111111111111
12 LOCK AND dword ptr [R14 + RCX],
        EDI
13 AND RAX, 0b111111111111
14 OR EDI, dword ptr [R14 + RAX]
```

(a) Test Asm Causing Violation

L1D-cache Tags in μarch trace

Input A
... 0x38c0 0x3f40 ...
Input B
... 0x38c0 0x3e80 0x3f40 ...

Speculative Load in Program

Input A
PC: 0x402152, Addr: 0x3500
Input B
PC: 0x402152, Addr: 0x3140

(b) Two inputs to Asm that speculatively access different addresses

Figure 6. Test Asm Causing Violation in InvisiSpec due to Speculative Interference because of MSHR contention.

UV2. Same-Core Speculative Interference Attack. In the example generated by AMuLeT, shown in Figure 6a, we have a speculative load, **SL** (Line 14 in Figure 6a), which shares the MSHRs with another non-speculative load, **NSL** (Line 6). Based on whether **SL** has a cache hit/miss, it causes contention on the MSHRs, subsequently delaying the execution of **NSL**. As shown in Table 7, Input A has an **SL** to address 0x3500, which misses the L2 cache, so an MSHR is occupied by this speculative request; this stalls a later Expose operation of the **NSL** to 0x3e80. Since InvisiSpec must perform an Expose operation to move lines in its speculative buffer to the cache, and 0x3e80 is unable to issue its Expose request before the test case ends, it is never brought into cache. Conversely, Input B has a **SL** to 0x3140, which hits in the L2 cache; so the occupied MSHR is freed up quickly, allowing the Expose of the **NSL** to 0x3e80 to issue and be installed in the L1 cache before the test case ends. By accessing 0x3e80 after the test ends, an adversary can observe whether the access was fast or slow, dependent on input A or B.

Root cause: The delayed **NSL** Expose is in fact observable by an adversary that subsequently accesses *any* address. As the cache-controller queues are in-order, the stall due to the Expose at the head of the queue causes *all* subsequent requests (e.g., a cache hit to `0xbeef`) to be stalled; whereas in the case where the **NSL** is not delayed, the request (e.g., a cache hit to `0xbeef`) is serviced faster. We also discovered variants of this leakage where MSHR contention is due to a miss in InvisiSpec's speculative buffer (instead of L2 cache).

Fix: This vulnerability is a fundamental design issue. A redesign that addresses the original speculative interference attacks, such as GhostMinion [2], which ensures that younger loads cannot influence the execution time of older loads, will also address the variants we discovered.

Table 7. Explanation of MSHR interference violation in InvisiSpec. The column MSHRs show the addresses in the L1-MSHRs. Blue and Green are speculative loads (SL), while Red and Black are non-speculative loads (NSL). Input A induces speculative MSHR interference, but Input B does not.

Input A		Input B	
Operation	**MSHRs**	**Operation**	**MSHRs**
SpecLoad 0x3500	3500	SpecLoad 0x3140	3140
–	3500	L2 hit 0x3140	
Replace 0x13e80	3500, 13e80	Replace 0x13e80	13e80
Expose 0x3e80– **stall!**	3500, 13e80	Expose 0x3e80	13e80, 3e80
m5exit	3500, 13e80	m5exit	13e80, 3e80
Load 0xbeef– **slow!**	3500, 13e80	Load 0xbeef– **fast!**	13e80, 3e80

4.6 Vulnerabilities Found in CleanupSpec

CleanupSpec [29] allows speculative loads to modify cache state and cleans up these state changes on a mis-speculation. Below, we describe the leaks we discovered in CleanupSpec and classify them as per their root cause in Table 8. We identify 2 new bugs causing insufficient cleaning and 1 new vulnerability causing excessive cleaning.

Table 8. Types of CleanupSpec violations found from 100 parallel runs of 200 test programs, 140 inputs, with the unmodified CleanupSpec (Original) and after a fix for speculative-stores not being cleaned up (Patched).

Violation Type	Original	Patched
Speculative Store Not Cleaned	✓	✗
Split Requests Not Cleaned	✓	✓
Too Much Cleaning	✓	✓

UV3. Speculative Store Not Cleaned Bug. The first violation in CleanupSpec was due to incorrect cleanup for speculative stores. As shown in Listing 3, cleaning speculative cache state requires tracking cache hit/miss metadata. While this tracking exists in `readCallback()` for speculative loads, it is missing in `writeCallback()` for speculative stores. Consequently, addresses of speculative stores are not cleaned on a mis-speculation, causing leakage. We discovered this when Spectre-v1-like tests with speculative stores caused L1D-cache differences, and the debug log showed incorrect metadata, as shown in Figure 7. We patched this by fixing `writeCallback()` to update cleanup metadata on speculative stores. After the fix (**Patched** in Table 8), these violations no longer occur, resolving the bug.

UV4. Split Requests Not Cleaned Bug. We discovered a subsequent violation due to a CleanupSpec bug related to cleanup of requests crossing cache line boundaries. When a load or store accesses data across cache lines (e.g., a 4-byte store to an address two bytes from the cacheline boundary), they spawn split-requests in gem5, i.e. multiple memory-system requests. We see that CleanupSpec does not clean

Listing (3) CleanupSpec bug - speculative store not cleaned

```
// Logic in ReadCallback()
if (L1Hit)
    pkt->setL1Hit();
...
// Metadata for Cleaning
if (!pkt->isL1Hit())
    DO_CLEANUP(...);
```

Debug Log on Speculative Store

```
WriteCallback() for Spec Store
L1Hit: 0, L2Hit: 0, L2Miss: 0
```

Listing (4) CleanupSpec Bug: no cleanup for split request

```
if (loadQueue[load_idx]->
    isSplitReq()){

    ++lsqSquashedLoadsSplitReq;

    // TODO: Cleanup for
          SplitReq
}
```

Figure 7. Implementation bugs in CleanupSpec's gem5 implementation discovered by AMuLeT's testing campaigns.

speculative split requests, as shown in Listing 4 from CleanupSpec code. We detected this with test cases like Spectre-v1 where speculative loads crossed cacheline boundaries.

UV5. Too Much Cleaning Vulnerability. Finally, we discover a new vulnerability in CleanupSpec due to an incorrect cleanup of non-speculative loads when non-speculative loads get reordered with transient loads to the same cache line.

Table 9 shows the operations in a test causing such a violation. Input A has a younger, speculative load **SL** (`PC = 0x40114d`) to the same address as an older non-speculative load **NSL** (`PC = 0x40113c`). For input B, the **SL** address differs from the **NSL** address. In both cases, the **SL** has a cache miss, installs the address, and is cleaned and evicted on a mis-speculation. However, for Input A, cleaning also removes any trace of the **NSL** to that address. In contrast, for Input B, as the **NSL** address differs from the **SL**, it remains in the cache after mis-speculation. Thus, these two inputs have different μarch traces resulting in leakage.

Root cause and Fix: This is the first discovered speculative interference vulnerability [3] on CleanupSpec, where interaction between transient and non-speculative loads corrupts cleaning metadata. A potential mitigation can identify the reordering of non-speculative and transient loads to the same addresses at commit time, and set a `noClean` flag for the younger speculative load(s). We leave this for future work.

Table 9. Test cases showing "Too Much Cleaning" vulnerability in CleanupSpec: sequence of operations

Input A				Input B			
Cycle	**PC**	**Type**	**Addr.**	**Cycle**	**PC**	**Type**	**Addr.**
1060	0x4011bb	Load	0x1100	1060	0x4011bb	Load	0x1100
1150	0x40114d	SpecLd	0x10c0	1150	0x40114d	SpecLd	0x1080
1423	0x40113c	Load	0x10c0	1423	0x40113c	Load	0x10c0
1429	0x40114d	Undo	0x10c0	1580	0x40114d	Undo	0x1080
μarch trace		0x1100		**μarch trace**		0x1100 0x10c0	

KV2. UnXpec [21] Vulnerability. In campaigns where we included the L1I-cache state in the μarch trace, we detected violations similar to the unXpec vulnerability [21]. In these violations, the inputs had different L1I-cache states at the end. The root cause was a difference in cleanup operations causing varying execution times and the instruction fetch speculatively installing extra lines in the L1I-cache.

Table 10 shows the operations in one such test case. Input A has a speculative load to `0x1180` that is an L1 hit, requiring no cleanup. Whereas, for Input B, the speculative load to `0x1840` is an L1 miss, that requires a cleanup on mis-speculation. As cleanup is on the critical path of execution, this increases the execution time for Input B compared to Input A. This is like the UnXpec [21] vulnerability that uses the difference in cleaning operation times to leak information.

As the execution is slower for Input B, the fetch unit speculatively fetches addresses beyond the end of the test, installing these in the L1I-cache. This causes a violation compared with Input A, which does not exhibit such behavior.

Table 10. CleanupSpec vulnerability to UnXpec [21]

Input A				Input B			
Cycle	**PC**	**Type**	**Addr.**	**Cycle**	**PC**	**Type**	**Addr.**
1054	0x40119b	Load	0x1180	1054	0x40119b	Load	0x1180
1056	0x4011b7	SpecLd	0x1180	1056	0x4011b7	SpecLd	0x1840
1207	0x4011a6	Load	0x1940	1207	0x4011a6	Load	0x1940
				1213	0x4011b7	Undo	0x1840
1219	0x401190	Store	0x1940	1240	0x401190	Store	0x1940

4.7 Vulnerabilities found in SpecLFB

SpecLFB [8] adds security checks to the cache Line-Fill Buffer (LFB) to prevent leaks via speculative cache misses. It blocks speculative cache misses from being installed into the cache until they are *safe* (like Delay-On-Miss [30]). Based on this, we tested SpecLFB's gem5 implementation against the `CT-SEQ` contract, the most intuitive contract based on the security guarantees described in the SpecLFB paper [8], and surprisingly found it to be insecure with respect to this contract.

UV6. First Speculative Load Not Protected in SpecLFB. The violating tests discovered by AMuLeT on SpecLFB are all similar to Spectre-v1, where the secret is in a register, as shown in Figure 8 (b). We discovered that the root cause of these violations is an undocumented optimization in SpecLFB's implementation that marks speculative loads incorrectly as *safe* if they are the first speculative load in the load-store queue. SpecLFB identifies *unsafe* speculative loads to be delayed by checking a flag, `isUnsafe`, for each load. The `isUnsafe` flag depends on whether the load is speculative either due to branch speculation (`isCUSL`) or memory dependence speculation (`isMUSL`), *and* it also depends on an `isReallyUnsafe` flag that is cleared if the load is the first speculative load, as shown in Figure 8 (a,c). Thus, Spectre variants with a single speculative load still leak information as they are not blocked from installing into the cache. We validate that these violations disappear when we expose this leakage of register values in the contract (`ARCH-SEQ`).

(a) Root Cause of SpecLFB Leak

```
if (isPrevNoUnsafe()):
    // no prior unsafe loads
    clearReallyUnsafe();
else if (!isReallyUnsafe()):
    // prior unsafe loads exist
    setReallyUnsafe();

// only unsafe loads stalled
if (!isUnsafe()):
    clearStall();
```

(b) Violation Test Asm

```
# RBX is secret
CMP RAX, 0 # non-zero RAX
JNE .l1
# RAX == 0, misprediction
  MOV RAX, ptr[R14 + RBX]
  JMP .l2
.l1:
# RAX == 0
  MOV RAX, ptr[R14 + 64]
.l2: ..
```

(c) isUnsafe(): uses isReallyUnsafe (cleared for first speculative load)

```
bool isUnsafe():
  return !isSquashed() && ( (isCUSL() || isMUSL()) && isReallyUnsafe());
```

Figure 8. Vulnerability in SpecLFB discovered by AMuLeT. The root cause is an undocumented optimization in SpecLFB code that clears protections for the first speculative load.

4.8 Vulnerabilities Found in STT

STT [41] uses taints to keep track of registers holding data accessed speculatively from memory or its derivatives, and it blocks the execution of instructions that leak tainted data via side channels. Below, we describe the violation **(KV3)** we discovered in STT, which was due to tainted speculative stores incorrectly executing and accessing the TLB.

KV3. Speculative Store Leaking via TLB. The violations in STT manifested as a difference in the TLB state in the μarch trace. The root cause was a speculative store with tainted address incorrectly installing an entry in the D-TLB and leaking its address. This leak was previously known [22].

Figure 9 shows an example violation. In Figure 9(a), a mispredicted conditional jump (JS) in Line 1 causes CMOVP to speculatively load data in Line 6, which is encoded in the store address in Line 8. Figure 9(b) shows two inputs (A and B) leading to a violation. In both, speculatively loaded data is encoded in the store address (0x9c78 in A and 0xdcb8 in B). While these do not appear in the cache, their TLB entries appear in the μarch trace, leaking the speculatively accessed data. The root cause is an implementation bug that allows tainted speculative stores to be executed; blocking their TLB access, as proposed by DOLMA [22], would fix this leak.

```
1  JS .bb_main.1
2  JMP .bb_main.4
3  .bb_main.1: # mispredicted
4
5  AND RCX, 0b1111111111111111111
6  CMOVP AX, word ptr [R14 + RCX]
7  AND RAX, 0b1111111111111111111
8  MOV dword ptr [R14 + RAX], EBX
```

(a) Snippet of Test Asm

TLB Entries in μarch trace

Input A
0x6000 0x9000 0x33000 0x4b000
Input B
0x6000 0xd000 0x33000 0x4b000

Speculative Acesses in Program

Input A
LD: Line-6, Addr: 0x4b238
ST: Line-8, Addr: 0x9c78
Input B
LD: Line-6, Addr: 0x4b238
ST: Line-8, Addr: 0xdcb8

(b) Two inputs causing a violation

Figure 9. Example of Test Asm Causing Violation in STT due to speculative store installing TLB entry.

5 Discussion

5.1 Portability to Other Defenses, Simulators, ISAs

Due to AMuLeT's modular structure consisting of the (1) test case *generator*, (2) the *leakage model* that provides contract trace, and (3) the *executor* that provides the μarch traces, it is quite easy to port AMuLeT to different defenses simply by changing the executor. Table 11 shows the lines of code (LoC) added or modified for each defense we tested. A majority of the LoC added/modified in the Gem5 simulator are for the test orchestration (452-532) and inter-process communication with AMuLeT (270-353), which are isolated and largely independent of the defense or simulator. The LoC for trace extraction (226-319) added to different μarch components in the simulator is minimal. The majority of these LoC can be copied from one defense to another with minor changes.

Table 11. Lines of Code (LOC) added/changed to different defenses to enable testing with AMuLeT.

Defense	Test Harness	Socket-Based Communication	Trace Extraction	Total LoC
InvisiSpec	532	353	445	1330
CleanupSpec	452	270	226	948
STT	486	276	296	1058
SpecLFB	465	338	319	1122

AMuLeT is easily portable to any CPU simulator, as long as it models CPU speculation (so it exhibits speculation-based vulnerabilities), and allows examining, extracting and resetting the μarch state (cache, branch predictor states, etc.). This covers most common μarch simulators (e.g., Gem5 [4], Marss-x86 [28], Champsim [12]).

Porting AMuLeT to other ISAs (ARM, RISC-V) is also feasible, but it requires a test case generator that support the new ISA semantics and requires support for accurate emulation of the new ISA in Unicorn [1] used by the leakage model. For example, while porting AMuLeT to Ghostminion [2] (built on the ARM ISA), we discovered bugs in Unicorn's ARM hooks that hindered our testing. Future work can address this and extend AMuLeT to other ISAs.

5.2 Detecting Additional Leaks with AMuLeT

Our campaigns using AMuLeT primarily test for leaks through caches and TLB, because most countermeasures [8, 29, 37] protect against cache-based leakage. However, by including additional μarch state in the μarch traces (e.g., branch-predictor state, execution order of loads or branches), AMuLeT can detect a wider range of speculative leaks, as shown in Table 5. Additionally, the state of other μarch predictors, like cache way-predictors, value predictors, or prefetchers, can be added to the μarch trace to discover new vulnerabilities [7, 31]. Thus, AMuLeT can be used to detect leaks from new micro-architectural features to be added in to a processor well before it is commercialized.

5.3 Limitations of AMuLeT

Limits of Randomized Testing. AMuLeT, like any randomized testing tool, can only demonstrate insecurity, but cannot prove a defense's security.[2] This makes it complementary to, not a replacement for, formal verification techniques, which can prove security but often lack scalability. For instance, Pensieve [38] requires manual modeling of defenses in domain-specific languages and several hours to verify security for programs up to nine cycles. In contrast, AMuLeT finds vulnerabilities in seconds and is easily portable to new defenses, making it a cost-effective first step to identify insecurity (e.g., how we show SpecLFB [8]'s implementation is insecure) before attempting complex formal verification.

Limited Search Space. AMuLeT, like prior black-box fuzzing works [15, 25, 26], is limited in the search space it can test within the vast μarch state space and design space possible for a processor. AMuLeT focuses its search by reducing the sizes of μarch structures (e.g., reducing MSHRs and cache ways) to prioritize high-contention states where leaks are more likely. This prioritization increases the likelihood of finding exploitable speculative leaks without altering the simulator or defenses under test. However, the extensive state space remains a limitation for comprehensive leak discovery, and future work may explore adaptive strategies for coverage-guided fuzzing or focused test generation to improve AMuLeT's detection capabilities.

6 Related Work

Post-silicon fuzzing for microarchitectural leaks. *Covert Shotgun* [11] and *ABSynthe* [13] automatically test CPUs for covert channels emanating from μarch contention. *Osiris* [35] uses fuzzing to discover new side channels. *Transynther* [23] tests CPUs for new variants of MDS attacks by mutating attack templates. *Scam-V* [24] uses model-based relational testing and random test generation to discover undocumented cache-based leaks in ARM CPUs. In contrast, AMuLeT focuses on *pre*-silicon testing of speculation countermeasures early at design time, much before they are deployed in CPUs.

RTL Fuzzing. Several works propose design-time testing of CPU designs at Register-Transfer Level (RTL). *WhisperFuzz* [5] detects timing side channels in RTL using coverage-driven fuzzing, whereas *SpecDoctor* [16] detects transient execution vulnerabilities in RTL using attack templates. Neither work detects a wide variety of speculative leaks like AMuLeT: WhisperFuzz can only test for constant-time execution, and it is inapplicable to test most secure speculation countermeasures [8, 29, 37, 41]; SpecDoctor tests only one type of leakage, where a secret in memory is exposed in speculation, and it lacks an extensible leakage model like AMuLeT. Other tools [17, 18, 32] that test CPU designs at RTL using golden reference models only detect functional bugs. In contrast to RTL-based tools, AMuLeT tests secure speculation countermeasures at the earliest stage of μarch design, when features are being prototyped in simulators. This allows computer architects to detect and address potential leaks in μarch countermeasures early on.

Formal Verification. These approaches reason about and prove the absence of leaks in hardware designs.

Micro-architectural Tools. Checkmate [33] uses μarch happens before graphs to generate security litmus tests. *Pensieve* [38] uses bounded model checking to reason about the security of countermeasures [2, 37] expressed as high-level μarch models. Both approaches require advanced modeling capabilities and considerable human effort to formalize the models in custom DSLs, which makes them impractical for design-phase testing. Their scalability is also limited: Checkmate has only been applied to simple in-order CPU models and attacks with up to 7 instructions, while Pensieve only analyzes programs up to 9 simulation cycles in a few hours. In contrast, AMuLeT can work with complex out-of-order CPU models in μarch simulators, without artificial limits on simulation time or attack programs. It also requires minimal integration effort, only requiring the addition of trace modules before testing can begin with new defenses.

RTL-Based Tools. UPEC [10] proves the absence of leaks caused by transient instructions in RTL designs, whereas LeaVe [34] proves whether an RTL design satisfies a leakage contract. These tools require implementations of defenses in RTL to test them. In contrast, AMuLeT can test defenses early on in μarch simulators, even before the RTL is generated.

7 Conclusion

AMuLeT is the first tool to enable the testing of countermeasures for speculative leaks in μarch simulators. Designed to achieve high testing speed and efficacy, AMuLeT enables large-scale campaigns on four countermeasures, revealing 3 known and 6 unknown leakages in them, including for the first time, a vulnerability in the implementation of the recently proposed SpecLFB. With AMuLeT, we enable designers of future defenses to test their counter-measure at design time, reducing the risk of insecure deployments.

Acknowledgments

This work is supported by the Spanish Ministry of Science and Innovation under TED2021-132464B-I00 PRODIGY; the Spanish Ministry of Science and Innovation under the Ramón y Cajal grant RYC2021-032614-I; the Spanish Ministry of Science and Innovation under PID2022-142290OB-I00 ESPADA; the Natural Sciences and Engineering Research Council of Canada (NSERC) under RGPIN-2023-04796; the Natural Sciences and Engineering Research Council of Canada (NSERC) under RGPIN-2021-03729; an NSERC Undergraduate Student Research Award (USRA); an SFU Faculty Recruitment Grant; and gifts from Intel Corporation and Ampere Computing.

[2] "Testing shows the presence, not the absence of bugs." - Djikstra [6]

A Artifact Appendix

A.1 Abstract

This appendix describes artifacts accompanying the AMuLeT paper, which introduces a tool for automated, design-time testing of secure speculation countermeasures. The artifact consist of two parts: (1) the AMuLeT framework, our tool customized for testing defenses in simulators, and (2) extensions to the Gem5 implementation of defenses that we test with AMuLeT. We include instructions for running campaigns with AMuLeT on speculative execution countermeasures such as InvisiSpec, CleanupSpec, STT, and SpecLFB, and reproducing the key results in the paper.

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

- **Algorithm:** Implements AMuLeT's fuzzing techniques.
- **Compilation:** Automatically through dockerfiles.
- **Run-time environment:** Requires Docker. We use linux environments inside Docker to run Amulet and gem5.
- **Hardware:** ~100 cores, 128GB RAM to run parallel test campaigns.
- **Metrics:** Test Throughput, Avg. Detection Time.
- **Output:** Table 5 (results of test campaigns).
- **Experiments:** Instructions to run campaigns are provided in the README.
- **How much disk space required (approximately)?:** 30GB
- **How much time is needed to complete experiments (approximately)?:** 80 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow automation framework used?:** Docker
- **Archived?:** https://doi.org/10.5281/zenodo.14847073

A.3 Description

A.3.1 How to access.

- **AMuLeT's git repository**: https://github.com/sith-lab/amulet. You can clone this directly from Github.
- **Defenses' git repository**: https://github.com/sith-lab/amulet-gem5. This contains the defenses we test using AMuLeT. This is cloned directly during the runs.

A.3.2 Hardware dependencies.

- x86-based Server CPU
- At least 128GB RAM, 100 cores to run parallel tests

A.3.3 Software dependencies.

- **Docker**: We run all our tests inside docker containers.
- Each of our defenses have different software dependencies. So we provide Dockerfiles which automatically set up the required software dependencies.

A.4 Installation

1. Clone the GitHub repository:

```
$ git clone https://github.com/sith-lab/amulet.git
```

2. Run the artifact:

```
$ cd amulet ; ./run_artifact.sh ;
```

This will run the campaigns for each of our defenses in **Table 5**, run InvisiSpec with reduced cache configurations to produce **Table 6**, and run Baseline (no defense) with different trace formats for **Table 4**.

A.5 Evaluation and expected results

You should be able to reproduce Tables 4, 5, and 6, and discover violations in each defense. The runtimes and average detection time can vary based on differences in hardware capabilities and the exact campaign configuration run.

A.6 Experiment customization

To reduce run times, one can decrease the number of test programs (default 200) as fewer test programs shorten run time. Fewer parallel campaigns (default 100) can be used on systems with fewer cores and memory.

References

[1] Unicorn - The Ultimate CPU emulator. https://www.unicorn-engine.org/, 2022. (Accessed: May 1, 2024).

[2] Sam Ainsworth. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 592–606, 2021.

[3] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1046–1060, 2021.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. In *USENIX Security 2024*, 2024.

[6] Buxton and Randell. Software engineering techniques, 1969.

[7] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*, 2024.

[8] Xiaoyu Cheng, Fei Tong, Hongyu Wang, Zhe Zhou, Fang Jiang, and Yuxing Mao. Speclfb: Eliminating cache side channels in speculative executions. In *USENIX Security*, 2024.

[9] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[10] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Transactions on Computers*, 72(1):222–235, 2022.

[11] Anders Fogh. Covert Shotgun. https://cyber.wtf/2016/09/27/covert-shotgun/, 2016. (Accessed: December 1, 2018).

[12] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition, 2022.

[13] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[14] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883. IEEE, 2021.

[15] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7143–7160, 2023.

[16] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, 2022.

[17] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[18] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[19] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the Design Automation Conference (DAC)*, 2019.

[20] 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 *IEEE Security and Privacy (SP)*, 2019.

[21] Mengming Li, Chenlu Miao, Yilong Yang, and Kai Bu. unxpec: Breaking undo-based safe speculation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 98–112. IEEE, 2022.

[22] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1397–1414, 2021.

[23] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

[24] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 225–248. Springer, 2020.

[25] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: testing black-box cpus against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.

[26] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, S&P 2023. IEEE, 2023.

[27] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA'06*, 2006.

[28] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: a full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 1050–1055, New York, NY, USA, 2011. Association for Computing Machinery.

[29] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An" undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 73–86, 2019.

[30] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 723–735. ACM, 2019.

[31] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[32] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. 2024.

[33] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960. IEEE, 2018.

[34] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2128–2142, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432, 2021.

[36] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 572–586, 2019.

[37] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[38] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural modeling for security evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[39] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[40] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 707–720. IEEE, 2020.

[41] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019.