THEME ARTICLE: TOP PICKS FROM THE 2022 COMPUTER ARCHITECTURE CONFERENCES

Revizor: Testing Black-Box CPUs against Speculation Contracts

Oleksii Oleksenko, *Microsoft, Cambridge, CB1 2FB, UK*
Christof Fetzer, *TU Dresden, Dresden, 01187, Germany*
Boris Köpf, *Microsoft, Cambridge, CB1 2FB, UK*
Mark Silberstein, *Technion, Haifa, 3200003, Israel*

Abstract—Speculative execution attacks such as Spectre and Meltdown exploit microarchitectural optimizations to leak information across security domains. These vulnerabilities often stay undetected for years, because we lack the tools for systematic analysis of CPUs to find them. In this article, we introduce such a tool, called Revizor, which automatically detects microarchitectural leakage in black-box CPUs. The key idea is to employ speculation contracts to model the expected information leaks and then to use randomized testing to compare the CPU's leakage against the model and thus detect unexpected leaks. We showcase the effectiveness of this approach on Intel CPUs, where we demonstrate that Revizor is capable of detecting both known and previously-unknown speculative leaks.

The instruction set architecture (ISA) specifies the functional behavior of a CPU but abstracts from its implementation details (microarchitecture). This abstraction enables development of hardware optimizations without requiring changes to the software stack; unfortunately, it also obscures the security impact of these optimizations. Indeed, over the last decade researchers discovered numerous microarchitectural vulnerabilities, including Spectre-style attacks that use microarchitectural state to exfiltrate secret information obtained during transient execution [4]. The problem is expected to get worse as Moore's law subsides and CPU architects apply ever more aggressive optimizations [7].

Yet the search for microarchitectural vulnerabilities in commercial CPUs remains unsystematic. Most of these vulnerabilities (e.g., Spectre [4]) have been discovered in a manual effort, by analysing public documentation, patents, and by laborious experimentation, which is inevitably incomplete.

In this article, we present Revizor, a tool for principled detection of information leaks in black-box CPUs. Its main distinguishing feature is reliance on a leakage model—called *speculation contract*—to precisely describe the information that we expect a given CPU to expose via a given side channel. Revizor uses the leakage model to differentiate between *expected* and *unexpected* information leaks on the CPU under test. To achieve this, Revizor generates random test cases comprised of random programs and their inputs, and compares the information exposed during their execution according to the leakage model, and the one exposed by a real CPU. If the CPU exposes information that the model does not, Revizor reports it as an unexpected leak, which often indicates the presence of a microarchitectural vulnerability.

Such a model-based approach enables systematic testing of black-box CPUs. As Revizor does not assume any specific leakage mechanism, it is capable of detecting various leakage types, even those that have not been discovered before. Importantly, the model remains abstract because Revizor does not directly compare the *behavior* of the model with that of the target CPU, rather it searches for the difference between the *information* they expose. As such, the model needs to contain only the bare minimum microarchitectural details necessary for describing leaks, and this permits us to test complex commercial CPUs without having to know the details of their design.

Digital Object Identifier 10.1109/XXX.0000.0000000

	Observation Clause	Execution Clause
Load	`expose: ADDRESS`	`None`
Store	`expose: ADDRESS`	`None`
Other	`None`	`None`

TABLE 1. Summary of *MEM-SEQ*. The observation clause describes expected leakage through caches, and the execution clause declares that no speculation is expected.

```
1 x ← 10
2 y ← load(100)
3 if(x < 0)
4     y ← load(200)
```

FIGURE 1. A program that exposes only one load address according to *MEM-SEQ*, while accordingly to *MEM-COND* both loads are exposed.

These unique properties of Revizor allowed us to run a deep testing campaign on two Intel CPUs, which we chose as the worst-case targets for our method: both are superscalar CPUs with several unpatched microarchitectural vulnerabilities, no detailed descriptions of speculation mechanisms, and no direct control over the microarchitectural state. Despite these complications, Revizor managed to automatically detect contract violations that represent all three of the known types of speculative leakage—speculation of control flow, address prediction, and speculation on hardware exceptions—as well as a new variant of Spectre V1. This result demonstrates the practicality of testing complex real-world CPUs against speculation contracts; it is further reinforced by our follow-up work [5], in which Revizor found two new types of speculative leaks.

Revizor is an active open-source project (https://github.com/microsoft/sca-fuzzer) with a growing interest from hardware and software developers as well as security researchers. It enjoys a steady stream of contributions to extend its speed, precision, and supported hardware platforms.

We next offer a brief overview of speculation contracts, describe Revizor's design, present the testing campaign results, and discuss possible applications.

SPECULATION CONTRACTS

When a CPU executes a program, the execution changes the CPU's microarchitectural state. A co-located attacker can observe some of these changes via a side-channel attack.

	Observation Clause	Execution Clause
Load	`expose: ADDRESS`	`None`
Store	`expose: ADDRESS`	`None`
Cond. Jump	`None`	`transient: if(INVERT_CONDITION){ IP = IP + TARGET}`
Other	`None`	`None`

TABLE 2. Summary of *MEM-COND*. Note that the execution clause describes the speculative behavior of a conditional jump, as the jump takes place (IP is updated) if the condition is false, the opposite of the non-speculative execution.

Speculation contracts [3] (short: contracts) specify all the side-effects observable via a given microarchitectural side channel.

Overview

A speculation contract augments the CPU's ISA as follows:

- For each instruction that may have an observable side-effect, the contract declares an **observation clause**. It describes the data exposed by the instruction.
- For each instruction that may trigger speculative execution, the contract declares an **execution clause**. It describes the effect of speculation, but without specifying the mechanism of speculation.

Example 1. *Consider a CPU that according to the available information does not implement any speculation. Thus, the execution clause for all its instructions is empty, as shown in Table 1. We also know that it has a shared cache, hence we expect loads and stores to have observable side effects, as they change the cache state (e.g., evict cache lines). A co-located attacker can recover the addresses of loads/stores by observing which of the cache sets changed their state via a cache timing side-channel attack (e.g., Prime+Probe). We encode these expectations in the observation clause (Table 1) for loads and stores by specifying that they expose their address. We call this contract MEM-SEQ (memory access leakage with sequential execution).*

When the program in Figure 1 is executed, MEM-SEQ prescribes that the attacker can observe an access to address `100` *due to the load at line 2.*

Example 2. *Assume we find out that the CPU from the previous example implements conditional branch prediction. It means that branches can be mispredicted, and the instructions in the mispredicted flow*

are executed speculatively. We encode this in the contract summarized in Table 2. The execution clause for branches describes the effect of misprediction: conditional branches first take a wrong target, then execute a certain number of instructions, and then roll back and go to the correct target. The observation clause remains the same as in the previous example, implying that the speculatively executed instructions expose their addresses just as the non-speculative ones. The resulting contract describes expected speculative leakage by declaring that any conditional branch can take a wrong target (according to the execution clause) and any memory access after the wrong target can expose its address to the attacker (according to the observation clause). We call this contract MEM-COND (memory access leakage with conditional branch misprediction).

When the program in Figure 1 is executed, MEM-COND prescribes that the attacker can observe two accesses: one to the address `100` *due to the load at line 2, and one to the address* `200` *due to the misprediction at line 3 and the following load at line 4.*

Contract Compliance

Intuitively, a CPU complies with a contract if any program executed on the CPU will leak no more information to an attacker than what the contract specifies.

For example, if a CPU implements only conditional branch prediction, then it complies with *MEM-COND*. However, if it also has indirect branch prediction, it may violate *MEM-COND* because the memory accesses executed after mispredicted indirect branches can leak unexpected information w.r.t. this contract.

Contract compliance is defined through the concept of traces. A **contract trace** is a sequence of observations that the attacker can make according to the contract when a CPU executes a program p with an input i. Here we abstractly represent it as a function *Contract* that takes p and i and returns a contract trace *CTrace*:

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

A **hardware trace** is the microarchitectural changes that an attacker *actually* observes when the program p is executed with an input i on the CPU in a given microarchitectural context μ. Accordingly, the process of observing microarchitectural effects can be described as a function *Measure* that returns a hardware trace *HTrace*:

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

A CPU complies with a contract if hardware traces expose the same (or less) information than exposed by contract traces: Formally, contract compliance is defined as a relational property: whenever any two executions of *any* program have the same contract trace (implying the difference between inputs is not exposed), the respective hardware traces should also match.

Definition 1. *A CPU **complies** with a Contract if, for all programs p, all input pairs* (i, i')*, and all initial microarchitectural states* μ*:*

$$Contract(p, i) = Contract(p, i') \implies Measure(p, i, \mu) = Measure(p, i', \mu)$$

In the terminology of information flow properties [8], Definition 1 requires that any program that is non-interferent with respect to a contract must also be non-interferent on the CPU.

Crucially, Definition 1 *does not* compare hardware and contract traces directly, only contract traces to contract traces, and hardware traces to hardware traces. This sidesteps the need to establish a detailed model of the microarchitecture and enables testing of black-box CPUs.

Conversely to Definition 1, a CPU **violates** a contract if there exists a program p, a microarchitectural state μ, and two inputs i, i' that produce the same contract trace but different hardware traces. We call the tuple (p, μ, i, i') a contract **counterexample**. A counterexample indicates a potential microarchitectural vulnerability that was not accounted for by the contract.

REVIZOR

Our proposed tool, Revizor, searches for contract counterexamples by generating random test cases, collecting their contract and hardware traces, and checking if any of them violate the contract according to Definition 1. Revizor's architecture is presented in Figure 2.

Overview

The testing process begins from generation of a test case: a program p and a sequence of inputs $i_0, i_1, \ldots, i_n$. The program is essentially a random sequence of assembly instructions. Each input is a file with random binary contents, used to initialize the program's memory and registers.

Program generator creates the program p by forming a random control-flow graph and then populating it with instructions randomly selected from a predefined

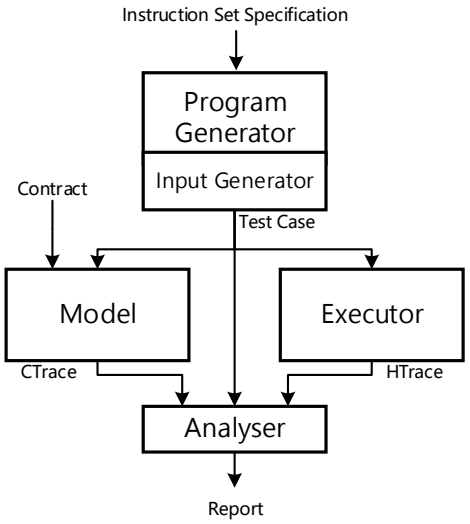

FIGURE 2. Architecture of Revizor.

pool of instructions. The generator can be configured to constrain the shape of the control-flow graph, control the pool of tested instructions, and configure the frequency of certain instructions.

Input generator populates input files with random values. In the initial version of Revizor the generation was completely random, but in our more recent work [5] we implemented an algorithm that uses the contract to improve input generation thus increasing the chance of surfacing contract violations.

Model collects contract traces for the test case. The model is an executable version of the contract implemented with an ISA emulator (Unicorn [6]). The emulator is modified to record observations according to the contract observation clause, and to implement speculation according to its execution clause. The model collects traces by executing the program p with each of the inputs i on this emulator. The resulting contract trace is a list of all recorded observations.

Executor collects hardware traces for the test case by executing the program p with each of the inputs i on the target CPU. The microarchitectural state μ is set indirectly, as it is not directly accessible on black-box CPUs: Each program execution inherently modifies the microarchitectural state, which sets μ for the following executions. Hardware traces are collected by monitoring the microarchitectural changes caused by each execution; specifically, Revizor collects traces by monitoring L1D cache via performance counters, mimicking a powerful attack via a cache timing side-channel.

Analyzer checks if any of the collected traces violate Definition 1. For this, it groups the inputs that produce the same contract trace into *equivalence classes*, and checks if all hardware traces in the same class match. If there is at least one pair of different traces, these constitute a counterexample to the contract, and Revizor reports the unexpected leakage to the user.

```
1 ADD ax, 0          // add 0 to ax
2 CMP rax, 10        // compare rax with 10
3 JE .l1             // jump if equal
4 SUB eax, 4253
5 MOV rax, [rbx]     // load from address in rbx
6 CMOVNBE ebx, eax
7 JMP .l2            // unconditional jump
8 .l1:
9 MOV rax, [rax]     // load from address in rax
10 .l2:              // end of the program
```

FIGURE 3. A program that surfaces a violation of *MEM-SEQ*.

Example 3. *Consider a round of a testing campaign where an x86 CPU with branch prediction is tested against MEM-SEQ. The round begins by generating a random program*[1] *shown in Figure 3, as follows: Revizor creates a random control-flow graph with three nodes; connects the nodes by placing jumps (lines 3, 7); and adds random instructions from a pool of x86 instructions (lines 1–2, 4–6, 9). Revizor complements the program with a sequence of random inputs:*

i_1=`{rax=40,rbx=5 }`, i_2=`{rax=2,rbx=20}`
i_3=`{rax=10,rbx=70}`, i_4=`{rax=10,rbx=40}`

Revizor executes the test case on the MEM-SEQ model, which collects the load addresses, producing the following traces:

$ctrace_1$=`{load *5 }`, $ctrace_2$=`{load *20}`
$ctrace_3$=`{load *10}`, $ctrace_4$=`{load *10}`

Note that $ctrace_1$ *and* $ctrace_2$ *contain addresses of the load at line 5, while* $ctrace_3$ *and* $ctrace_4$ *record line 9 because the jump at line 3 takes a different branch when* `rax==10`.

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

$htrace_1$=`{ECL *5}`,
$htrace_2$=`{ECL *20}`
$htrace_3$=`{ECL *70, ECL *10}`,
$htrace_4$=`{ECL *40, ECL *10}`

In the first two traces, the branch at line 3 is not taken, and the loads from addresses `5` *and* `20` *evict*

[1]The test case is simplified for clarity. In practice, the program contains more instructions, and the inputs assign values to multiple registers and to several pages of memory.

the corresponding cache lines. In addition, the first two executions train the branch predictor, so the next two executions experience mispredictions. For input i_3, the CPU executes a speculative load from address `70` *(line 5) and a non-speculative load from address* `10` *(line 9). For i_4, the CPU speculatively loads from address* `40` *(line 5) and non-speculatively loads from address* `10` *(line 9). If the cache line size is 64 bytes, i_3 and i_4 evict different cache lines, which results in different hardware traces.*

Accordingly, the last two inputs form a counterexample: $ctrace_3 = ctrace_4$ *and* $htrace_3 \neq htrace_4$. *Revizor detects it and reports to the user.*

We next describe how we solve some of the challenges that appear when testing black-box CPUs.

Sandbox for Collecting Traces

Our model-based testing approach requires that test cases are executed in an identical way by the model and the executor. Namely, the test cases have to follow the same (architectural) control-flow path on both the model and the target, and their (non-speculative) memory accesses have to use the same addresses. Otherwise, the discrepancies between the model and the executor could cause violations of Definition 1, leading to false positives.

We avoid such false positives by creating identical *sandboxes* in the model and the executor:

- The generator forms a directed acyclic graph (DAG) as a basis for creating programs, which ensures that the program executes a predictable sequence of instructions (e.g., it does not jump to uninitialized memory), and that the model and the executor follow the same control path.
- The generator instruments all loads and stores to confine them within a predefined memory region.
- Instructions that could fault (e.g., divisions) are instrumented to prevent the faults.
- The program p is loaded into the memory regions that have identical addresses within the executor and the model.

Handling Microarchitectural States

Our approach requires pseudorandomization of the microarchitectural state μ: on one hand, it has to be deterministic to check traces according to Definition 1; on the other hand, it has to be diverse to trigger different speculative leaks. As black-box CPUs normally do not provide mechanisms for direct manipulation of the microarchitectural state, we have to do it indirectly.

We strive to make the microarchitectural state deterministic by creating an isolated measurement environment: The executor runs as a kernel module, on a single core, with hyperthreading, prefetching, and interrupts disabled. The executor also monitors System Management Interrupts to discard those measurements corrupted by an interrupt. Before executing an input, the executor flushes caches, invalidates TLB, flushes microarchitectural buffers with `VERW`, and executes a sequence of memory fences to reset the CPU pipeline.

We make the state diverse through the execution of the test case itself: The executor runs each of the inputs in a sequence, without interruptions, which means that the executions of $[i_0...i_{n-1}]$ indirectly set μ_n for i_n. For example, every time a test case executes a branch, it trains the branch predictor for the next inputs.

Naturally, these techniques are imperfect and they do not provide us with full control of the microarchitectural state. However, they have proven themselves sufficient for detecting speculative information leaks with little to no false positives.

TESTING CAMPAIGN

Experimental Setup

We test two CPUs with several x86-64 ISA subsets. The experiments are summarized in Table 3.

Rows 1 and 2 are the CPUs under test. We test Intel Core i7-6700 (Skylake) and Intel Core i7-9700 (Coffee Lake). For Skylake, we run experiments with Spectre V4 [2] microcode patch enabled and disabled.

Row 3 is the pool of instructions used to generate test cases: `AR` is in-register arithmetic and logic operations; `MEM` is loads/stores and in-memory variants of `AR`; `VAR` is variable-latency operations (divisions). `CB` is conditional branches.

We also create a setup for testing information leakage upon microcode assists.[2] To this end, we modify the executor's sandbox to clear the "Accessed" page table bit in one of the sandbox pages such that the first store to/load from this page triggers an assist. This mode is enabled for Targets 7 and 8.

We test each of the targets against the following contracts: *MEM-SEQ* exposes leakage only during sequential execution (see Example 1); *MEM-COND* additionally exposes leakage after mispredicted branches (see Example 2); *MEM-BPAS* exposes leakage during

[2]A microcode assist is a hardware exception that is transparently handled by an internal microcode routine to execute a complex operation, such as setting a page table bit.

	Target 1	Target 2	Target 3	Target 4	Target 5	Target 6	Target 7	Target 8
CPU	Skylake						Skylake	CoffeeLake
V4 patch	off			on			on	
Instruction Set	AR	AR+MEM	AR+MEM+VAR	AR+MEM+VAR	AR+MEM+CB	AR+MEM+CB+VAR	AR+MEM	
Microcode Assists	Disabled						Enabled	

TABLE 3. Description of the experimental setups.

	Target 1	Target 2	Target 3	Target 4	Target 5	Target 6	Target 7	Target 8
MEM-SEQ	✓	×(V4)	×(V4)	✓	×(V1)	×(V1)	×(MDS)	×(LVI-Null)
MEM-COND	✓	×(V4)	×(V4)	✓	✓	×(V1-var)	×(MDS)	×(LVI-Null)
MEM-BPAS	✓	✓	×(V4-var)	✓	×(V1)	×(V1)	×(MDS)	×(LVI-Null)
MEM-COND-BPAS	✓	✓	×(V4-var)	✓	✓	×(V1-var)	×(MDS)	×(LVI-Null)

TABLE 4. Summary of the testing results. Here, ×- Revizor detected a violation; ✓- Revizor detected no violations within 24h of testing; ✓- the target is compliant through a weaker contract. In parenthesis are Spectre-type vulnerabilities revealed by the detected violations.

sequential execution and due to speculative store bypass (i.e., Spectre V4); and *MEM-COND-BPAS* combines all three contracts together.

Results

We test each of the target-contract combinations for 24 hours or until a violation is found. Upon a violation, we manually investigate the counterexample to determine its cause. The results are summarized in Table 4.

Target 1: As a baseline, we test the most narrow instruction set AR containing only arithmetic operations). We expect the target to comply with the most restrictive contract (*MEM-SEQ*). The experiments confirm it: Revizor detects no violations within 24 hours of testing. Since other contracts expose strictly more information, the target inherently complies with them too. This experiment shows that Revizor successfully mitigates measurement noise, producing no false violations.

Target 2: We next repeat the experiment with a broader set of instructions, AR+MEM. This time, Revizor detects violations of *MEM-SEQ* and *MEM-COND* within about two hours of testing. Upon inspection, we identify those violations as representative of Spectre V4. Revizor does not detect violations of *MEM-BPAS* and *MEM-COND-BPAS*, which is expected as they encode (and thus permit) leakage caused by V4.

Target 3: We further augment the instruction set with divisions (the only variable-latency instructions in the base x86) to AR+MEM+VAR, and Revizor finds violations of *MEM-BPAS* and *MEM-COND-BPAS*, although this time it takes over 20 hours to detect them. Upon inspection, the counterexamples reveal a novel variant of Spectre V4 that leaks the timing of division (*not* permitted to be exposed according to the contract). We describe this variant in detail in our original ASPLOS'22 paper [10].

Target 4: We change the experiment described in Target 3 by enabling a V4 patch on Skylake. Our experiments do not surface contract violations, showing that the V4 patch is effective, also against the novel V4 variant.

Target 5: When augmenting AR+MEM with conditional branches to AR+MEM+CB, Revizor detects violations of *MEM-SEQ* and *MEM-BPAS* within several minutes of testing. Upon inspection, these are representative of Spectre V1. Revizor detects no violations of *MEM-COND* and *MEM-COND-BPAS*, which is expected as the contracts permit leakage caused by branch misprediction.

Target 6: When further augmenting the instruction set with variable-latency instructions to AR+MEM+CB+VAR, Revizor detects violations of *MEM-COND* and *MEM-COND-BPAS*. Similar to Target 3, the violations represent novel variants of Spectre V1.

Target 7: We next perform experiments with microcode assists enabled in the executor. To test the assists in isolation, we test AR+MEM, and we enable V4 patch to avoid violations caused by V4. Revizor now detects violations of all contracts within about 10 minutes. We identify them as representative of MDS [1].

Target 8: We repeat the experiment in Target 7, but now on Coffee Lake, which has a hardware MDS patch. Revizor detects violations on it as well, also within 10 minutes. We identify them as caused by LVI-Null [9].

In summary, Revizor could automatically generate gadgets that represent all three of the known types of speculative leakage: speculation of control flow, address prediction, and speculation on hardware exceptions. The analysis is robust and did not produce false positives, demonstrating the practicality of testing complex real-world CPUs against speculation contracts.

LOOKING AHEAD

The discovery of Spectre, Meltdown, Foreshadow, and similar microarchitectural vulnerabilities has made it painfully clear that aggressive optimizations for runtime and power consumption in today's CPUs may have unforeseen security implications. One of the reasons why these vulnerabilities slipped between the cracks despite rigorous quality control, is that unlike routine measurements of performance and correctness properties, there has been no generic way to express and test the microarchitectural security properties of CPUs, until now.

With Revizor, we change this situation: we provide the first method and tool that can test modern fabricated black-box CPUs against precise specifications of their microarchitectural security, aka speculation contracts, via automated, comprehensive and objective assessment of their compliance. The version of Revizor described in the paper demonstrates the feasibility of this approach, by *automatically* identifying an important class of known transient vulnerabilities, as well as previously unknown variants, without the previously-unavoidable laborious educated guesswork.

Crucially, while information leakage is a non-functional property, contracts allow us to specify it in functional terms via speculation and observation clauses. This makes the approach accessible to hardware architects without background in formal methods, and thus increases the chances of its adoption by the industry. We hope that Revizor and its derivatives will eventually become a part of routine correctness tests throughout the CPU design and development process.

Today, we build contracts ourselves. We start with a contract that encodes our understanding of the official and unofficial public documentation (often partial or inaccurate), and we gradually refine the contract by testing it with Revizor and modifying the contract according to the counterexamples that Revizor finds.

In the future, we envision that CPU vendors themselves will augment ISA specifications with speculation contracts, much like the way memory consistency models have become an inherent part of the CPU spec. Alternatively, if a vendor will not provide a contract for a CPU (e.g., for older CPU models), the community of the CPU users could build the contract on their own, following our refinement-based approach.

Tools such as Revizor will then enable customers and certification bodies to independently validate that CPUs comply with their stated leakage properties. This presents an opportunity for hardware vendors to give evidence of the security properties of their microarchitecture—without having to disclose the low-level details of their IP—and provides an incentive to architect systems with microarchitectural security as a first-class citizen.

REFERENCES

1. C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. "Fallout: Leaking data on Meltdown-resistant CPUs," in *Proc. of the 2019 Conf. on Computer and Communications Security*, 2019, pp. 769-784.
2. Google Project Zero. Speculative Execution, Variant 4: Speculative Store Bypass. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528
3. M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. "Hardware-software contracts for secure speculation," in *Proc. 2021 IEEE Symp. on Security and Privacy*, 2021, pp. 1868-1883.
4. P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution," in *Proc. 2019 IEEE Symp. on Security and Privacy*, 2019, pp. 1-19.
5. O. Oleksenko, M. Guarnieri, B. Köpf, M. Silberstein. "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing," in *Proc. 44th IEEE Symp. on Security and Privacy*, submitted for publication.
6. Unicorn Engine. [Online]. Available: https://github.com/unicorn-engine/unicorn
7. J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher. "Opening pandora's box: A systematic study of new ways microarchitecture can leak private data," in *Proc. 48th Ann. Int. Symp. on Computer Architecture*, 2021, pp. 347-360.
8. A. Sabelfeld and A. C. Myers. "Language-based information-flow security," *IEEE Journal on selected areas in communications.*, vol. 21, no. 1, pp. 5–19, 2003.
9. J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, F. Piessens, and K. Leuven. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection,", in *Proc. 2020 IEEE Symp. on Security and Privacy*, 2020.
10. O. Oleksenko, C. Fetzer, B. Köpf, M. Silberstein. "Revizor: Testing Black-Box CPUs against Speculation Contracts," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2022.

Oleksii Oleksenko is a Researcher at Microsoft, Cambridge, UK. His current research interest is confidential computing, with a focus on microarchitectural leaks. He received his Ph.D. degree from the Technical University of Dresden (TU Dresden). Contact him at t-oleksenkoo@microsoft.com.

Christof Fetzer is head of the Systems Engineering Chair in the Computer Science Department at the Technische Universität Dresden, Germany. His research focuses on simplifying confidential computing while providing strong protection of data, code, and secrets. He holds a Ph.D. degree from the University of California, San Diego. Contact him at christof.fetzer@tu-dresden.de.

Boris Köpf is a Principal Researcher at Microsoft, Cambridge, UK. His research focuses on techniques for tracking information flow in microarchitecture and machine learning systems. Boris holds a Ph.D. from ETH Zurich. Contact him at boris.koepf@microsoft.com.

Mark Silberstein is an Associate Professor at the Electrical and Computer Engineering Department, Technion – Israel Institute of Technology. His research interests are computer systems at large, with the emphasize on accelerator architectures, Operating Systems, Computer Networks and systems security. He holds a Ph.D. in Computer Science from the Technion. Contact him at mark@ee.technion.ac.il.