Automated Synthesis of Instruction-Centric Leakage Contracts

Elvira Moreno
IMDEA Software Institute
Universidad Politécnica de Madrid

Tiziano Marinaro
CISPA Helmholtz Center for Information Security
Saarland University

Ryan Williams
Northeastern University

Marco Patrignani
University of Trento

Roberto Guanciale
KTH Royal Institute of Technology

Hamed Nemati
KTH Royal Institute of Technology

Marco Guarnieri
IMDEA Software Institute

1 Introduction

Side-channel attacks exploit variations in processor behavior—such as execution timing or cache access patterns [1,2,7] —to infer sensitive information from otherwise secure software. To defend against such attacks, programmers need to reason about a CPU's microarchitecture. However, the Instruction Set Architecture (ISA)—the traditional abstraction layer between software and hardware—lacks microarchitectural details.

Leakage contracts [3] augment the Instruction Set Architecture (ISA) with a specification of all observable side-channel leaks within a CPU. This enables secure system development since programmers are made aware of the exploitable side-channels that are traditionally obscured at the ISA level. Unfortunately, constructing leakage contracts for modern CPUs is a complex task: it requires extensive reverse engineering, expert knowledge, and significant time investment [4], making it impractical to apply across the diverse landscape of commercially available CPUs.

To address this issue, we propose a *contract synthesis methodology* for automatically synthesizing instruction-centric leakage contracts based on hardware observations extracted from a black-box CPU. For this, we adopt a counterexample-driven synthesis method that refines candidate contracts based on observed hardware behavior. Our methodology ensures that the synthesized contract is *sound*, i.e., it captures *all* leaks exposed during testing. We implement our methodology in a tool called MALCOS (MicroArchitectural Leakage COntract Synthesizer). MALCOS incrementally builds instruction-centric leakage contracts that capture leaks in a given black-box CPU. While MALCOS targets `x86` and `ARM` architectures, the approach is general and can be adapted to other CPUs.

2 Overview

We now present the core aspects of MALCOS with an example.

Example 2.1 (A simple CPU and an attacker)**.** Consider a processor CPU that implements a simple *register file compression* (RFC) optimization [9]. This optimization reduces the physical size of the register file by mapping all logical registers that store the value 0 to the same physical zero register. As pointed out in [9], this optimization can result in timing leaks (due to reducing the pressure on the register file).

Throughout this section we consider a microarchitectural attacker ATK that can precisely observe whenever RFC happens during execution. Hence, ATK can distinguish the program executions associated with the following example consisting of a program p and two initial states σ_1, σ_2:
$p := \texttt{MOV rax,rbx}$ $\quad \sigma_1 := (\texttt{rbx} \mapsto 0)$ $\quad \sigma_2 := (\texttt{rbx} \mapsto 1)$
Here, the program p consists of an instruction assigning to register `rax` the value of register `rbx`, where the value of `rbx` is 0 in initial state σ_1 and a value different from 0 in initial state σ_2. Therefore, RFC happens when executing p from σ_1, but does not happen when executing p from σ_2, which results in different hardware traces for ATK.

However, ATK cannot distinguish the program executions associated with the following example consisting of the same program p and two different initial states σ_3, σ_4:
$p := \texttt{MOV rax,rbx}$ $\quad \sigma_3 := (\texttt{rbx} \mapsto 1)$ $\quad \sigma_4 := (\texttt{rbx} \mapsto 3)$
In this case, the values of register `rbx` are both different from 0, resulting in the same hardware traces.

To capture ISA-level leaks, a leakage contract augments the ISA with a specification of the observable side-channel leaks associated with a given CPU. It maps each (architectural) program execution to a *leakage trace*, i.e., a sequence of (ISA-level) observations exposing potentially leaked information. In MALCOS, a contract is a set of clauses $cl := ex$ `IF` pr, where ex specifies what is added to the leakage trace and a predicate pr modeling when the clause is enabled, i.e., when the observation is added to the trace.

Figure 1 depicts MALCOS's approach for the synthesis of leakage contracts. The approach takes as input a black-box processor CPU and it iteratively constructs a candidate leakage contract *cand* by alternating between two phases: (1) a *leakage testing phase* (`Checker` in Figure 1) where MALCOS attempts at finding new leaks in CPU not captured by *cand* (capturing all leaks discovered so far), and (2) a *con-*

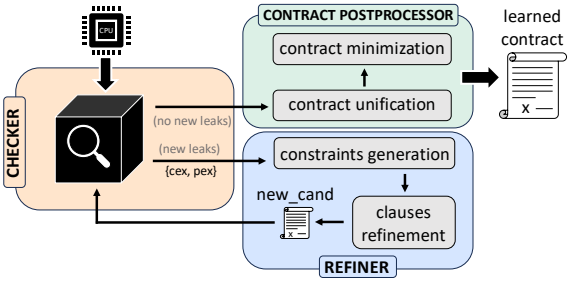

Figure 1: MALCOS contract synthesis process

tract refinement phase (`Refiner` in Figure 1) where MALCOS updates *cand* to account for a newly discovered leak.

When MALCOS cannot find new leaks, it simplifies the contract *cand* (`Postprocessor` in Figure 1), which is returned to the user. Next, we provide further details on each phase.

Leakage testing phase. The `Checker` takes a candidate contract *cand* and a CPU (treated as a black-box), and tries to discover leaks not yet captured by *cand*. Intuitively, the `Checker` executes programs and observes leaks w.r.t. a given attacker ATK. When it finds a new leak, it returns a *counterexample cex*, which is a sequence of instructions and a pair of initial states that yield the *same leakage trace* under *cand* but different hardware traces, i.e., distinguishable by ATK. It also returns *positive examples pex*, i.e., test cases indistinguishable for both the contract and the attacker.

Contract refinement phase. The *`Refiner`* takes a counterexample `cex` describing a newly-discovered leak, and it generates a new contract clause that captures the new leak, i.e., the new clause must distinguish the counterexample. Additionally, the *`Refiner`* can take positive examples `pex`, i.e., test cases that are indistinguishable for both the contract and the microarchitectural attacker, to guide the generation of the contract clause by informing it about which executions should *not* be distinguished by the synthesized clause. This clause is then added to the original contract to generate a new candidate contract `new_cand`. The `Refiner` discovers such a contract clause as a syntax-driven synthesis task implemented on top of the Rosette solver [8].

Example 2.2. Consider the processor CPU and attacker ATK from Example 2.1. Starting from an empty candidate contract (`cand` $= \emptyset$), the *`Checker`* attempts to discover a leak. For this, the `Checker` generates test cases, each one consisting of a program and a pair of initial states, executes them on the target CPU, and computes the hardware traces to detect potential leaks. Given that `cand` $= \emptyset$, the `Checker` discovers the test case (p, σ_1, σ_2) as a counterexample, and the test case (p, σ_3, σ_4) as a positive example.

The `Refiner` analyses the counterexample and the positive example to generate a clause capturing the leak. It uses the Rosette solver to identify a new clause *cl* of the form *ex* `IF` *pr* that distinguishes executions of p from σ_1 and σ_2 but not distinguishes executions of p from σ_3 and σ_4. This yields cl_1 bellow, which exposes when the first operand is a register and it is written with a value `0`.

$cl_1 :=$ *post-operand-value 0* `IF` [(*operand-type 0* = `reg`)
`and` (*operand-access 0* = `write`)
`and` (*post-operand-value 0* = `0`)]

Finally, the `Refiner` adds the newly-discovered cl_1 to the candidate contract `cand`. This process iterates, i.e., checking and refining to discover leaks not yet captured, until no further leaks are found. Afterwards, MALCOS invokes the `Postprocessor` to minimize the contract by unifying different clauses and by reducing the number of unnecessary clauses to simplify the final contract.

x86-64 Subset	Mem. addr.	Ctrl. flow	`rep` counter	Div. by 0
`cond`: Conditional branches	✓	✓	×	×
`strn`: String operations	✓	×	✓	×
`dmul`: Division and mult.	✓	×	×	✓
`logi`: Logical operations	✓	×	×	×
`cmov`: Conditional moves	✓	×	×	×

Table 1: Selection of the x86-64 synthesis campaign. ✓ means that MALCOS synthesized a clause capturing leaks associated with the corresponding leakage source; the absence of clauses associated with the leakage source is marked with ×.

3 Preliminary Evaluation

In the evaluation we investigate whether MALCOS can learn contracts from actual hardware by using it to synthesize contracts for the `x86` and `ARM` architectures.

x86. We target an Intel i5-6500 CPU, and we synthesized contracts for 13 subsets of the `x86` ISA using REVIZOR [6] as `Checker`. For each ISA subset, we run the synthesis loop for 24 hours. Table 1 shows a selection of the results of our campaign. We highlight the following: (1) for all subsets, MALCOS synthesized clauses capturing leaks through the data cache; (2) for the `cond` subset (the only one including control-flow statements), MALCOS synthesized clauses exposing control-flow leaks; (3) for the `div` subset, MALCOS synthesizes clauses exposing the divisor operand. (4) and for the `strn` subset, MALCOS synthesize clauses exposing the rep counter which determines how many times an instruction is repeated. Overall, the contracts are aligned with the community's understanding of instruction-level leaks in the analyzed CPU.

ARM. We target a Raspberry Pi4 with a Cortex-A72 CPU, and we synthesized contracts associated with ARM memory instructions using SCAM-V [5] as `Checker`. The synthesized clauses (after postprocessing) expose the tag and index bits associated with individual memory loads and stores, in addition to clauses associated with the `LDP`, `LDPSW`, and `STP` instructions, which access two 32-bit words or two 64-bit doublewords from memory at the same time.

References

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

[2] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1967–1984. USENIX Association, August 2020.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[9] 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)*, pages 347–360. IEEE, 2021.