Automatically Synthesizing Leakage Contracts from Counterexamples

Elvira Moreno-Sanchez*
IMDEA Software Institute
Universidad Politécnica de Madrid
Madrid, Spain

Ryan Williams*
Northeastern University
Boston, USA

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain

Marco Patrignani
University of Trento
Trento, Italy

1 INTRODUCTION

Microarchitectural attacks [2, 3] exploit subtle differences in a program's execution time—due to CPU optimizations—to leak information. Attackers can exploit them to infer secret data from seemingly secure programs. 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 [1] have been recently proposed as a new security abstraction at ISA-level. Such contracts allow specifying at ISA-level the information leaked by a CPU through microarchitectural side-channels; thereby providing a basis for secure programming. However, modern CPUs do not come with dedicated leakage contracts. While deriving such a contract for a specific CPU (and microarchitecture) could be done through an extensive and manual reverse engineering effort, scaling this process to the large number of available commercial CPUs requires automation.

In this work, we propose an automated approach for synthesizing leakage contracts directly from hardware measurements. This approach will allow us to automatically derive leakage contracts for existing commercial CPUs with limited manual effort. We make the following contributions:

- We develop a **domain-specific language** (DSL) for formalizing ISA-level leakage contracts. Our DSL is expressive enough to capture leaks from real-world CPUs. Additionally, contracts specified in our DSL are *executable*. That is, they can be applied to arbitrary x86 programs (and associated inputs) to derive the corresponding *leakage traces* recording all leaks modeled by the contract.
- We develop a **counterexample-based synthesis approach**, which we overview in Section 2, for automatically learning leakage contracts from hardware measurements. Our synthesis approach incrementally learns the leakage contract associated with a CPU by (1) generating random test cases for detecting leaks, (2) executing these test cases on the target CPU to derive hardware measurements, and (3) refining the candidate contract to account for the newly discovered leaks.
- We implemented a prototype of our synthesis approach in a tool called Malcos. The tool uses the Rosette framework [5] as a back-end for synthesis and relies on the Revizor fuzzer [4] to generate test cases and detect leaks in CPUs. We are currently evaluating (1) the expressiveness of our DSL, and (2) the quality (in terms of soundness and precision) of the leakage contracts synthesized by Malcos. So far, we validated Malcos results on selected simple target contracts.

*Both authors contributed equally to this research.

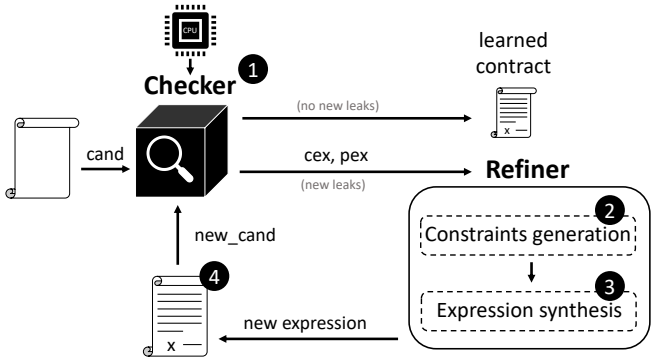

Figure 1: Malcos synthesis approach

2 MALCOS

In this section, we overview Malcos's synthesis approach with an example. Next, we first describe the leakage profile of our target CPU and the overview of our approach.

Target CPU: In our example, we consider a CPU that implements a simple *register file compression* (RFC) optimization [6]. This optimization reduces the physical size of the register file by mapping all logical registers that store the value 0 to the same physical register. These compression schemes, however, often reduce the pressure on the register file thereby resulting in timing leaks, exposing the value of the program counter.

Overview: Figure 1 shows the workflow of the Malcos synthesis approach, which relies on two main components:

(1) The *checker* that, given a candidate contract, tries to discover leaks in the target CPU that are not captured by the contract. That is, the checker looks for a counterexample `cex` to *contract satisfiability* [1]. This counterexample is a sequence of instructions together with a pair of initial states that produce the *same leakage traces* according to the candidate contract, but result in different microarchitectural observations. Concretely, Malcos uses the Revizor black-box CPU fuzzer [4] as a checker, which relies on cache-based side-channels as a source of microarchitectural observations.

(2) The *refiner* that, given a counterexample (describing a newly discovered leak), generates a new expression to be added to the

$$expr_1 := \textit{operand-value 0}\ \texttt{IF}\ [(\textit{operand-type 0} = \texttt{reg})\ \text{and}\ (\textit{operand-access 0} = \texttt{write})]$$

Figure 2: Learned expression

contract that captures the new leak. The refiner instantiates the problem of discovering such an expression as a syntax-driven synthesis task implemented on top of the Rosette solver.

We now explain how MALCOS can learn the contract associated with our target CPU when starting from an empty candidate contract, i.e., $\texttt{cand} = \emptyset$. For simplicity, we consider a simple microarchitectural attacker that can observe whenever RFC happens. First, MALCOS runs the *checker* to identify leaks that are not captured by the candidate (❶). For this, the checker generates multiple random test cases (each one consisting of a program and a pair of initial states), executes them on our target CPU, and performs microarchitectural observations to detect potential leaks. Given that $\texttt{cand} = \emptyset$ while the target CPU leaks through RFC, the checker discovers the following counterexample describing the RFC leak:

$$p := \text{MOVE}\ x, y, \quad s_1 := (y \mapsto 0) \quad s_2 := (y \mapsto n \neq 0)$$

Here, the program p consists of an instruction assigning to register x the value of register y, where the value of y is 0 in state s_1 and any value different from 0 in state s_2. Therefore, RFC happens when executing p from s_1, but does not happen when executing p from s_2, which results in different microarchitectural observations for the attacker. That is, the test case $\texttt{cex} := \langle p, s_1, s_2 \rangle$ is a counterexample for the candidate contract $\texttt{cand} = \emptyset$.

Next, MALCOS uses the refiner to analyze the counterexample $\texttt{cex} := \langle p, s_1, s_2 \rangle$ and generate a DSL expression capturing the leak. The refiner starts by simulating the architectural execution of the counterexample (using the Unicorn CPU simulator) and collects information about all architectural states (e.g., register values, operand information for the executed instructions, the program counter) explored during execution. The refiner uses all this information to generate the symbolic constraints for the synthesis problem (❷). Then, the refiner uses the Rosette solver to identify a new DSL expression $expr_1$ that distinguish the counterexample cex (❸), i.e., for which the executions of p starting from s_1 and s_2 lead to different $expr_1$ values. For instance, MALCOS might learn the expression $expr_1$ (Figure 2), which distinguishes the counterexample. The expression exposes the value of the first operand whenever the first operand is a register and it is the target of a register write.

The refiner adds the newly discovered expression $expr_1$ to the candidate contract cand. MALCOS works in a iterative fashion by performing a new round of checking and refinement to discover further leaks ignored by the new candidate contract cand (❹). Given that the contract from Figure 2 is sufficient to capture the RFC leaks (since it exposes all values written to registers during execution), MALCOS terminates and outputs the learned contract in Figure 2.

Fixing over-approximations: MALCOS iteratively refines the candidate contract from counterexamples. This, however, can lead to contracts that over-approximate leaks in the target CPU, i.e., the candidate contract might expose more information than needed to capture the actual leaks.

For instance, consider the learned contract in Figure 2, which exposes the value of written registers during program execution. While this distinguishes any two executions leaking through RFC,

$$expr_2 := \textit{PC}\ \texttt{IF}\ [(\textit{operand-type 0} = \texttt{reg})\ \text{and}\ (\textit{operand-access 0} = \texttt{write})\ \text{and}\ (\textit{operand-value 0} = \emptyset)]$$

Figure 3: Learned expression using positive examples

it would also distinguish executions where no RFC happens, e.g., where no register takes the value 0. To mitigate these overapproximations, we extend our synthesis approach to account for positive examples, that is, test cases that are indistinguishable for both the contract and the microarchitectural attacker. For instance, given the initial candidate contract, $\texttt{cand} = \emptyset$, apart from the counterexample $\texttt{cex} := \langle p, s_1, s_2 \rangle$, the checker can discover positive examples like the following one:

$$p := \text{MOVE}\ x, y, \quad s_3 := (y \mapsto n \neq 0) \quad s_4 := (y \mapsto m \neq 0)$$

This positive example consists of the same program p described above, and a new pair of states (s_3, s_4) where the values n and m are any value other than 0, thus resulting in no different microarchitectural observations for the attacker. That is, the test case $\texttt{pex} := \langle p, s_3, s_4 \rangle$ is a positive example for the candidate, $\texttt{cand} = \emptyset$.

Our synthesis approach now aims at synthesizing a DSL expression that (1) distinguishes the counterexample cex while (2) distinguishing as few positive examples $\texttt{pex}_1, \ldots, \texttt{pex}_n$ as possible. Using positive examples MALCOS can learn the expression $expr_2$ (see Figure 3), which exposes the program counter, **pc** whenever the first operand is a register and it is written with a value $\emptyset$.

The expression $expr_2$ is more precise than $expr_1$, while still capturing RFC leaks because it only exposes whenever $\emptyset$ is written to a register rather than exposing all register values.

3 PRELIMINARY EVALUATION

To evaluate the impact of positive examples, we performed a preliminary evaluation using a simulated CPU that leaks the value of the program counter throughout the execution (associated with the contract $expr := PC\ \texttt{IF}\ True$) and measured precision and soundness, where precision measures how many of the leaks captured by the synthesised contracts are true positives, whereas soundness measures the number of false negatives.

In our experiments, we ran MALCOS to synthesize contracts with and without positive examples. To validate the precision and soundness of the synthesized contract, we further randomly generated 50 programs with 15 inputs each, which we used as validation set. For these validation test cases, we then collected contract and microarchitectural traces for (1) the simulated CPU, and (2) the synthesized contracts. The results show that, without positive examples, the average precision of the contracts is 30%, while with positive examples it increases to 100%. In contrast, soundness is stable at an average of 100% with and without positive examples.

REFERENCES

[1] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. 2021. Hardware-software contracts for secure speculation. In *S&P*. IEEE.
[2] P. Kocher et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020).
[3] M. Lipp et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020).
[4] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *ASPLOS*.
[5] E. Torlak and R. Bodik. 2013. Growing solver-aided languages with Rosette. In *Onward!*
[6] J. R. S. Vicarte et al. 2021. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *ISCA*. IEEE.