Work-in-Progress: Synthesis of Leakage Contracts from Examples

Abstract—**Leakage contracts are a new security abstraction modeling—at the ISA level—the information leaked by a CPU through microarchitectural side channels. As show by recent work, contracts provide a foundation for secure programming. However, modern CPUs lack dedicated leakage contracts that accurately capture their specific leakage profiles.**

In this paper, we report on our on-going work towards an automated approach—based on relational testing—to synthesize leakage contracts directly from the underlying hardware. We define a domain-specific language to formalize leakage contracts and develop a counterexample-driven synthesis method to automatically generate contracts that capture all leaks within a given microarchitecture. Our approach is implemented in the MALCOS tool, which can synthesize leakage contracts for CPUs for the `x86`, `ARM`, and `RISC-V` architectures. We evaluate MALCOS performance in terms of the precision and soundness of synthesized contracts.

I. Introduction

Microarchitectural attacks [1], [2] exploit subtle differences in a program's execution time—due to CPU optimizations—to leak information from seemingly secure programs. To defend against such attacks, programmers need to reason about the interactions between software and a CPU's microarchitecture. However, the Instruction Set Architecture (ISA)— the traditional abstraction layer between software and hardware—lacks microarchitectural details.

Leakage contracts [3], [4] 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 report on our on-going work towards 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 (Section III). Following the formalism from [3], our DSL models leakage contracts as a set of *leakage clauses* describing what observations are leaked during program execution. That is, leakage contracts in our DSL map each program execution to a *leakage trace* capturing the leaked information. Importantly, contracts specified in our DSL are *executable*. That is, they can be applied to arbitrary `x86`, `ARM`, and `RISC-V` programs (and associated inputs) to derive the corresponding *leakage traces* recording all leaks modeled by the contract.
- We develop a **counterexample-guided synthesis approach**, which we overview in Section II and detail in Section IV, for automatically learning leakage contracts from hardware measurements. Our synthesis approach incrementally learns the leakage contract associated with a CPU by (1) generating test cases for detecting leaks, (2) executing these test cases on the target CPU to derive hardware measurements capturing the observational power of different cache-based attackers, and (3) refining the candidate contract to account for the newly discovered leaks.

Our approach ensures that the synthesized contract is *satisfied* by the processor design [3] for all explored test cases. That is, the contract captures all leaks detected by the test cases explored during the synthesis process. Additionally, our approach minimizes unnecessary leakage over-approximations by requiring that the synthesized contract cannot distinguish a given set of positive examples, i.e., test cases that produce the same hardware measurements.

- We implement a prototype of our synthesis approach in a tool called MALCOS. The tool uses the Rosette framework [5] as a back-end for synthesis. Moreover, MALCOS relies on the two different testing tools for generating test cases and detecting leaks in CPUs: Revizor [6], [7] (for the `x86` architecture), and Scam-V [8], [9] (for the `ARM` and `RISC-V` architectures).

Our preliminary results (Section V) indicate that MALCOS can successfully synthesize program-specific leakage contracts, showing its ability to identify a sound over-approximation of standard leakage models.

II. Overview

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

Target CPU: In our example, we consider a CPU that implements a simple *register file compression* (RFC) optimization [10]. 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 potentially resulting in timing leaks.

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

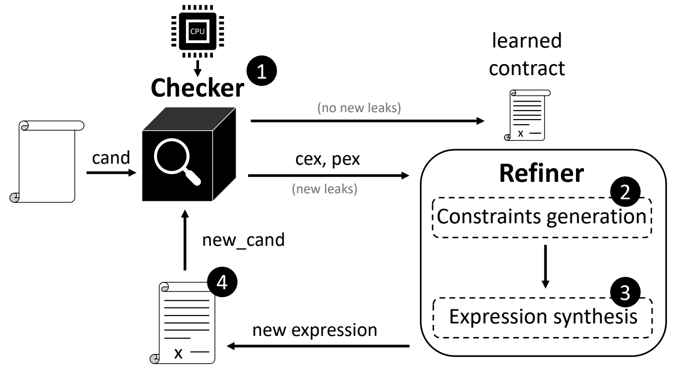

Fig. 1: MALCOS synthesis approach

1) The *checker* that, given a candidate contract, tries to discover leaks in the target CPU that are not captured by the contract. Whenever the checker finds a new leak, it returns a *counterexample*, which is a sequence of instructions together with a pair of initial states that (a) produce the *same leakage traces* according to the candidate contract, but (b) result in different microarchitectural observations, i.e., they are distinguishable by a microarchitectural attacker.
In practice, MALCOS's synthesis approach in parametric in the underlying checker and our implementation uses the Revizor [6] and Scam-V [8] testing tools, which rely 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 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 [5].

We now explain how MALCOS can learn the contract associated with our target CPU when starting from an empty candidate contract, i.e., $\texttt{cand} = \emptyset$. To simplify the overview, we consider a simple microarchitectural attacker that can observe whether RFC happens. First, MALCOS runs the *checker* to identify leaks that are not captured by the candidate (❶). For this, the checker generates multiple 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:

$$p := \texttt{MOV rax,rbx} \quad \sigma_1 := (\texttt{rbx} \mapsto 0) \quad \sigma_2 := (\texttt{rbx} \mapsto 1)$$

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 state σ_1 and a value different from 0 in 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 microarchitectural observations for the attacker. That is, the test case $\texttt{cex} := \langle p, \sigma_1, \sigma_2 \rangle$ is a counterexample for the candidate contract $\texttt{cand} = \emptyset$.

expr₁ := *operand-value 0* `IF` [(*operand-type 0* = `reg`) and (*operand-access 0* = `write`)]

Fig. 2: Learned expression

Next, MALCOS uses the refiner to analyze the counterexample $\texttt{cex} := \langle p, \sigma_1, \sigma_2 \rangle$ and to generate a DSL expression capturing the leak. The refiner starts by simulating the architectural execution of the counterexample 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 distinguishes the counterexample `cex` (❸), i.e., for which the executions of p starting from σ_1 and σ_2 lead to different expr_1 values. For instance, MALCOS might learn the expression expr_1 in 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, 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 $\texttt{cex} := \langle p, \sigma_1, \sigma_2 \rangle$, the checker can discover positive examples like the following one:

$$p := \texttt{MOV rax,rbx} \quad \sigma_3 := (\texttt{rbx} \mapsto 1) \quad \sigma_4 := (\texttt{rbx} \mapsto 2)$$

This positive example consists of the same program p described above, and a new pair of states (σ_3, σ_4) where the values of register `rbx` are both different from 0, thus resulting in indistinguishable executions for our RFC attacker. That is, the test case $\texttt{pex} := \langle p, \sigma_3, \sigma_4 \rangle$ is a positive example for the candidate $\texttt{cand} = \emptyset$.

expr₁ := *TRUE* `IF` [(*operand-type 0* = `reg`) and (*operand-access 0* = `write`) and (*operand-value 0* = `0`)]

Fig. 3: More precise contract for register file compression

Our synthesis approach now aims at synthesizing a DSL expression that (1) distinguishes as many counterexample $\texttt{cex}_1, \ldots, \texttt{cex}_n$ while (2) distinguishing as few positive examples $\texttt{pex}_1$, ..., $\texttt{pex}_m$ as possible.

Using positive examples MALCOS can learn the expression $expr_2$ (see Figure 3), which exposes when the first operand is a register and it is written with a value `0`.

The expression $expr_2$ is more precise than $expr_1$, while still capturing RFC leaks because it only exposes whenever `0` is written to a register rather than exposing all register values throughout the execution.

Program-dependent leakage: Another case of overapproximation is when leakage of an instruction depends on its context: i.e., the state affecting the surrounding instructions. For example, in a simple microarchitecture the leakage of register compression may be measurable only if the subsequent instruction does not take a jump. In fact, taking a jump can flush the pipeline, which allows the previous instruction to complete before the subsequent instruction is executed, independently if register compression has occurred.

In these cases, a leakage is connected to an instruction in a specific location of the program. To account for these leaks, MALCOS also supports synthesizing *program-specific leakage contracts*, i.e., contracts that describe the leaks associated with a fixed program. For this, (1) we restrict checkers to detect leaks only w.r.t. a given program (i.e., test cases now consist *only* of a pair of inputs), and (2) we restrict the refiner to synthesize leakage expressions that, in addition to describe the leak, also refer to a specific program counter in the program.

```
[1] mov x9, x2 obs (x2 = 0) if cond is F
[2] b.cond 4
[3] add x1, x2, x3
[4] mov x6, x3 obs (x3 = 0)
```

Fig. 4: Case 2: the first mov leaks information only if the condition of the following branch is false; the second mov always leaks

We illustrate this with a `ARM` program in Figure 4, for which the synthesized program-specific contract is depicted in red next to the corresponding instructions. The program contains two `mov` instructions, the first of which is followed by a conditional branch instruction: the program jumps to `4` if cond holds. The first `mov` instruction only leaks the compression of `x2` if the condition of the subsequent branch does not hold, since taking the jump flushes the pipeline and allows the first `mov` to complete before the second `mov` is executed. The second `mov` is not followed by a jump, and therefore it always leaks the compression of `x3`.

III. CONTRACTS DSL

In this section, we formalise our domain-specific language for expressing leakage contracts.

A. Syntax

(Bit string)	*bs*	:=	$\{0,1\}^n$
(Identifiers)	*id*	:=	$REG[bs]$
		|	$operand\_type[bs]$
		|	$operand\_access[bs]$
		|	$operand\_value[bs]$
(Expressions)	*ex*	:=	$bs \mid id \mid \ominus ex \mid ex_1 \oplus ex_2$
		|	$ex[ex_1 : ex_2]$
(Predicates)	*pr*	:=	$\{0,1\} \mid ex_1 = ex_2 \mid ex_1 < ex_2$
		|	`not` $pr \mid pr_1$ `and` pr_2
		|	pr_1 `or` pr_2
(Clause)	*cl*	:=	*ex* `IF` *pr*
(Contract)	*ctr*	:=	$\emptyset \mid \{cl_1, cl_2, ..., cl_n\}$

Fig. 5: The MALCOS contract language

Figure 5 formally defines the basic types and syntax of our contract DSL. The basic types of our DSL are bit strings (*bs*), with bitvectors of length one representing booleans. The DSL syntax consists of identifiers (*id*), expressions (*ex*), predicates (*pred*), observations (*obs*) and contracts (*ctr*). Identifiers are used to extract information of the architectural state from the registers (*REG*), the value of an operand of an instruction (*operand_value*), such as addresses of a load, the type of an operand (*operand_type*), or the access type (*operand_access*). An expression (*ex*) can include identifiers (*id*), bit strings (*bs*), and standard operations over bit strings. A predicate (*pr*) states conditions that must be satisfied in order to produce the observations. Predicates can be Boolean values or expressions involving logical operations between bit strings or predicates such as equality, negation, conjunction and disjunction. Clauses (*cl*) are formed by combining conditional expression based on predicates (*ex IF pr*), and contracts are expressed as possibly empty sets of clauses.

In the following, we say that a contract is *program-specific* if every predicate constraints the program counter to be a constant. A program-specific contract allows to express that an instruction leaks only in a specific context, for example, it may specify that only one specific load of a given program leaks the accessed address.

B. Semantics

A contract in our DSL maps program executions to *leakage traces*, i.e., sequence of observations obtained by evaluating the clauses in the contract on *every state* in the program execution.

We now formalize this concept. Let $\rightarrow_p$ be a program semantics mapping each architectural state σ to the next state σ'. Each contract C induces a labelled semantics $\rightarrow_C$ that extends architectural executions with labels capturing the values of each contract clause in C. Concretely, the contract semantics is defined by the following rule:

$$\frac{\sigma \rightarrow_p \sigma' \quad \delta = \{[\![cl]\!](\sigma) \mid cl \in C\}}{\sigma \xrightarrow{\delta}_{p,C} \sigma'}$$

where $[\![cl]\!](\sigma)$ denotes the observation obtained by evaluating clause cl in state σ. Examples of observations include, the address of memory `loads` or of the `pc` at certain instructions.

A *contract trace* τ is a sequence of observations (o) obtained by applying the contract semantics $\rightarrow_{p,C}$ to an execution, that is, $\tau = o_i \circ \cdots \circ o_{j-1}$ in which $\sigma_i \xrightarrow{o_i}_{p,C} \ldots \xrightarrow{o_{j-1}}_{p,C} \sigma_j$. In the following we use $Contract(C, p, \sigma)$ to represent the trace of the contract C associated with the maximal execution of the program p starting from the input σ.

Given a contract C, two traces are *C-equivalent* if they agree on all contract observations. Since the ISA semantics is deterministic, two input states are *C-equivalent* if their maximal traces are *C-equivalent.*

C. Contract satisfaction

We now conclude by instantiating the notion of contract satisfaction [3] for our framework.

For this, we first introduce the notion of hardware traces, which capture the observational power of the (microarchitectural) attacker. We represent the hardware trace as the output of a function:

$$HTrace = Attack(p, \sigma, Ctx)$$

that returns the observations made by an attacker on all available microarchitectural side-channels. The function *HTrace* takes three input parameters: (1) the victim program (p); (2) the input state σ processed by the victim's program (i.e., the architectural state including registers and main memory); (3) the microarchitectural context Ctx in which it executes.

Informally, a CPU *satisfies a contract C for a program p* if for any two program's inputs that are C-equivalent, they produce indistinguishable hardware traces in any context, i.e., the executions are indistinguishable by an adversary operating at the microarchitectural level:

$$\forall \sigma, \sigma', Ctx.\ Contract(p, \sigma) = Contract(p, \sigma')$$
$$\implies Attack(p, \sigma, Ctx) = Attack(p, \sigma', Ctx)$$

We say that a CPU *satisfies a contract C*, if the CPU satisfies it for every program.

We now introduce some auxiliary terminology that we will use later in Section IV. A *test case* $\langle p, \sigma_1, \sigma_2 \rangle$ for our contract satisfaction analysis consists of a program p and a pair of input states σ_1, σ_2. When analyzing contract satisfaction for a single program (where p is fixed), we often refer—for conciseness—to a test case simply as a pair of input states $\langle \sigma_1, \sigma_2 \rangle$.

If a CPU does not satisfy a contract, there is a *leakage counterexample* `cex` $:= \langle p, \sigma_1, \sigma_2 \rangle$. That is, there exists a test case of a program p and two initial states σ_1, σ_2 that produce different hardware traces while generating C-equivalent contract traces.

Algorithm 1: MALCOS learning algorithm

```
Procedure ContractSynthesize($cand$):
    while true do
        $pex, cex \leftarrow$ Checker($cand$);
        if $cex$ then
            $expr \leftarrow$ Refiner($pex, cex$);
            $cand := expr$;
        else
            return $cand$;
        end
    end
    Procedure Refiner($pex, cex$):
        $constr \leftarrow$ ConstrGenerator($pex, cex$);
        $expr \leftarrow$ ExprGenerator($constr$);
        if $expr$ then
            return $expr$;
        else
            return failed;
        end
    end
end
```

IV. Synthesis Approach

We outline the contract synthesis algorithm followed by MALCOS in Algorithm 1. It relies on two main procedures: the `Checker`, to find counterexamples describing leaks, and the `Refiner`, to refine the current candidate contract to account for newly discovered leaks. We start by discussing the algorithm and then explain the `Checker` (Section IV-A) and `Refiner` (Section IV-B) procedures.

MALCOS takes as input a candidate contract, *cand*, which may be initially empty (i.e., it does not expose anything). The algorithm iterates over all the programs generated by the `Checker` and attempts to give an answer to whether the target processor satisfies *cand*; if not, it provides a proof of unsatisfiability, *cex*, (potentially, a list of counter-examples that showcase the unsatisfiability), and a list of positive-examples *pex*, i.e., examples that satisfy the current contract. Then, the `Refiner` attempts to refine *cand* with one or more counterexamples *cex* and positive examples *pex*, generating constraints and searching for an expression, *expr*, accordingto the MALCOS's DSL. If successful, MALCOS adds *expr* to *cand*, which is then updated and evaluated again. When *cand* satisfies the target properties, i.e., the `checker` is not able no find more *cex*, the process the procedure halts and return the learned *cand*.

A. `Checker Modules`

MALCOS is built on top of two microarchitectural fuzzers, REVIZOR [6] (for `x86` architecture) and SCAM-V [8], [9] (for `ARM` and `RISC-V` architectures), which we use as our `Checker` modules.

The `Checker` is responsible to find leaks in the target CPU (*target*) that are not yet captured by the candidate (*cand*). That is, in the terminology of Section III, the checker discovers *leakage counterexamples* for the candidate contract *cand*. To this end, the `Checker` generates a set of random programs and pairs of inputs, where the inputs are a set of values that initialize the CPU architecture (registers, flags, and memory). The `Checker` then runs rounds of fuzzing.

On REVIZOR this fuzzing rounds consist on: (1) executing the Unicorn CPU emulation system to obtain the contract traces (o); (2) executing the target CPU to collect the hardware traces (τ); (3) check for contract leaks. The fuzzing rounds end when the `Checker` finds a leakage counterexample $\texttt{cex} := \langle p, \sigma_1, \sigma_2 \rangle$, which as already mentioned consist of a program p and two initial states σ_1, σ_2 that produce different hardware traces while generating C-equivalent contract traces.

SCAM-V, on the other hand, combines techniques from program verification and fuzzing to perform relational testing to validate the candidate contracts. The most glaring difference between the two approaches is that for a generated test program p, SCAM-V uses *symbolic execution* to synthesize a relation that for p identifies states that are observationally equivalent according to the contract being validated. Next, it generates an instance of this relation in terms of two input states, and finally, SCAM-V runs the generated program with two inputs on hardware and compares the measurements on the side channel of the processor. The generated inputs satisfy the synthesized relation, implying (if the contract is valid) that the side-channel data on hardware cannot be distinguished either. If we can distinguish the two runs on hardware, we have a counterexample that invalidates the contract.

B. `Refiner`

The `Refiner` is the main module of MALCOS and it handles contract synthesis. The `Refiner` (see Algorithm 1) takes as input (1) a candidate contract $cand$ describing the current leakage model capturing all leaks found by the `Checker` so far, (2) a list pex of positive examples that satisfy the current contract candidate $cand$ and (3) a list cex of counterexamples to the current candidate $cand$ found by the `Checker`.

At every invokation, the `Refiner` refines the candidate contract, *cand*, to account for the discovered leak (the counterexamples in *cex*). This requires finding a contract, $cand'$, such that: (i) program executions that are $cand$*-equivalent* are also $cand'$*-equivalent*, and (ii) no counterexample in cex is also a counterexample for $cand'$.

As a basis for the synthesis task, the `Refiner` turns each test case $\langle p, \sigma, \sigma' \rangle$ in cex and pex into a pair of *runs*, where each run is the sequence of architectural states σ_i explored when executing the program p starting from σ (respectively σ'). Concretely, for each explored state σ_i, the run contains the current state of the registers, the *opcode* of the executed instruction as well as information about this instruction's operands, such as their address, type or access type. For instance, consider the following instruction: **and** rax, rbx. Figure 6 shows the architectural state information corresponding to the state *just before* executing the instruction.

(Regs)	$:= (v_{\texttt{RAX}} \mid v_{\texttt{RBX}} \mid v_{\texttt{RCX}} \mid v_{\texttt{RDI}} \mid v_{\texttt{RDX}} \mid v_{\texttt{RSI}} \mid v_{\texttt{PC}})$
(Opcode)	$:=$ `AND`
(Operand 0)	$:=$ (`RAX` | `REG` | `READ_WRITE`)
(Operand 1)	$:=$ (`RBX` | `REG` | `READ`)

Fig. 6: Architectural state information associated with the instruction **and** rax, rbx. Above, v_{regId} denotes the value associated with register $regId$.

These *runs* are the basis from which the `Refiner` generates the new contract. MALCOS uses Rosette [5], a solver-aided language that extends Racket, to (1) formalize the semantics of our DSL, (2) generate the synthesis constraints associated with the runs from cex and pex, and (3) synthesize the new leakage clause that distinguishes all counterexamples in cex while distinguishing as few positive examples in pex as possible. First, the `Refiner` generates constraints which it uses for synthesizing expressions. These constraints help in the identification of discrepancies between the states associated to the *runs*. Then, the `Refiner` uses the SMT solver Z3 to synthesize a new leakage clause that describes the leakage from the *counterexample* into MALCOS contract language.

Note that, for each contract, *cand*, and *counterexample*, $\langle \text{p}, \sigma_1, \sigma_2 \rangle$, there may be different valid refinements. For example, if p contains a memory access and a multiplication where the accessed address and the multiplication operands differ between σ_1 and σ_2, *cand* could be refined by adding one of two observation clauses: one exposing the accessed memory address and the other exposing the multiplication operands. This refinement stage is meant to produce a *stronger* contract at each iteration. The strongest contract is the one that satisfies the following relation: let ctr_1, ctr_2 be two contracts, we say that ctr_1 is stronger than ctr_2, written as $ctr_1 \sqsupseteq ctr_2$, if and only if ctr_1 cannot expose more than ctr_2.

In our current approach, we perform refinement by adding the synthesized leakage clause cl to the current candidate contract $cand$. As a result, the new candidate contract $cand \cup cl$ immediately satsfies the requirements (i) and (ii) mentioned at the beginning of this section.

V. PRELIMINARY EVALUATION

In this section, we report on our preliminary evaluation of MALCOS's ability to automatically learn leakage contracts for `x86` and `ARM` architectures.

Next, we first introduce the common metrics we use to evaluate the quality of learned contracts (Section V-A). Then, we describe the target leakage contracts used as ground truth in our experiments (Section V-B). We conclude by describing our preliminary results for the `x86` (Section V-C) and `ARM` (Section V-D) architectures.

A. Metrics

In the absence of established standard measures, in this section we define the *precision* and *soundness* metrics to assess the quality of the learned contracts.

The *precision* metric measures the accuracy of learned contracts, i.e., how precisely the learned contract reflects leakages of the target model: $\mathit{Precision} = \frac{Dist}{Dist+MDist}$, where *Dist* is the number of test cases that are both target and learned models distinguishable; and *MDist* is the number of test cases that are distinguishable for the learned model but **not** for the target.

On the other hand, the *soundness* metric measures the correctness of the learned contract: $\mathit{Soundness} = \frac{Indist}{Indist+TDist}$, where *Indist* is the number of test cases that are both target and learned models indistinguishable; and *TDist* is the number of test cases that are distinguishable for the target but **not** for the learned model.

B. Contract Models

We now describe the leakage contracts used as ground truth in our experiments.

Program counter contract: The **pc** contract leaks the value of the program counter throughout the execution [11].

Constant-time contract: The **ct** contract models the constant-time observer [12], which is commonly used when reasoning about side channels in cryptographic algorithms. It exposes the value of the program counter and the addresses of load and store operations throughout the execution.

Tag Index contract: The **TagIdx** contract differs from **ct** in that it only exposes the tag and set index associated with each memory access (instead of the whole memory address like in **ct**). This model is often used to check the security of programs when memory is organized in blocks that are transferred atomically between the different memory layers [13].

Register file compression: The **rfc** contract models the leaks induced by the register file compression optimization described in Section II.

Multiplication simplification: The **mul** contract models timing leaks associated with common computation simplification over multiplications [10]. In particular, **mul** exposes whenever the operands of a multiplication are either 0 or 1.

C. Preliminary Results for the `x86` *architecture*

In this experiment, we use MALCOS (with Revizor [6] as a checker) to synthesize program-specific leakage models for the `x86` architecture. For this, we first extended Revizor to synthesize leakage models directly from ground-truth implementations of the five contracts from Section V-B.

For each of the contracts from Section V-B, we randomly generate 10 programs and we use MALCOS to learn a program-specific contract with (1) varying number of inputs (from 25 to 250) and (2) with or without positive examples (4 in total). We measure the soundness and precision of each learned contract against a (disjoint) validation set of 10'000 inputs. Figure 7 and Figure 8 report the average precision and soundness achieved by the contracts learned by MALCOS against the corresponding ground-truth.

The results in Figure 7 indicate that, the use of positive examples increases the precision of the learned contracts. In contrast, Figure 8 indicates that the soundness of the learned contract grows up to 1 at the increase of the number of inputs increases the soundness.

D. Preliminary Results for the `ARM` *architecture*

In our evaluation for `ARM` architecture (with Scam-V [8] as a checker), we assessed the effects of the amount of counterexamples and positive examples that were provided as input. We constructed a dataset of 40 different programs consisting of basic instructions such as branch, memory and arithmetic operations.

By varying the number of positive (`pex`) and negative (`cex`) examples, we measured the precision and soundness of the synthesis focusing on the **ct** and **TagIdx** ground-truth contracts with a validation set of 100 examples (50 positive and 50 negative). The findings, which are shown in Figure 9 and Figure 10, demonstrate that increasing the number of examples improves the precision in almost all cases.

VI. CONCLUSION

We presented an overview of our on-going work towards an automated approach, which we implemented in the MALCOS synthesis tool, for synthesizing leakage contracts directly from hardware measurements. We see three concrete next steps as future work: (1) performing an extensive performance evaluation to assess the precision and soundness of the contracts learned by MALCOS even when considering unrestricted contracts (rather than only program-specific ones), (2) using MALCOS to perform an in-depth security analysis of `x86`, `ARM`, and `RISC-V` CPUs, and (3) extending our approach to support more complex leakage contracts, e.g., those supporting transient leaks.

REFERENCES

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[2] 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 *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. [Online]. Available: https://doi.org/10.1109/SP.2019.00002

[3] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1868–1883.

[4] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *Proceedings of the 30th ACM Conference on Computer and Communications Security*, ser. CCS 2023. ACM, 2023.

[5] E. Torlak and R. 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*, 2013, pp. 135–152.

[6] O. Oleksenko, C. Fetzer, B. Köpf, and M. 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*, 2022, pp. 226–239.

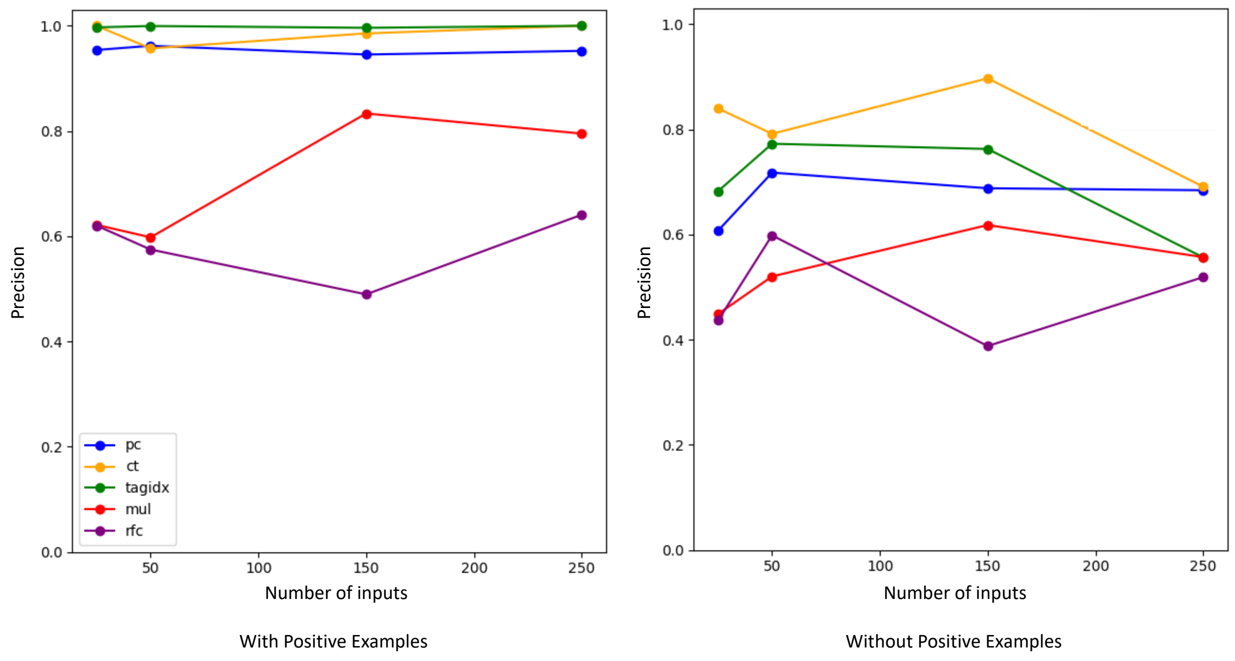

Fig. 7: Average of precision per inputs

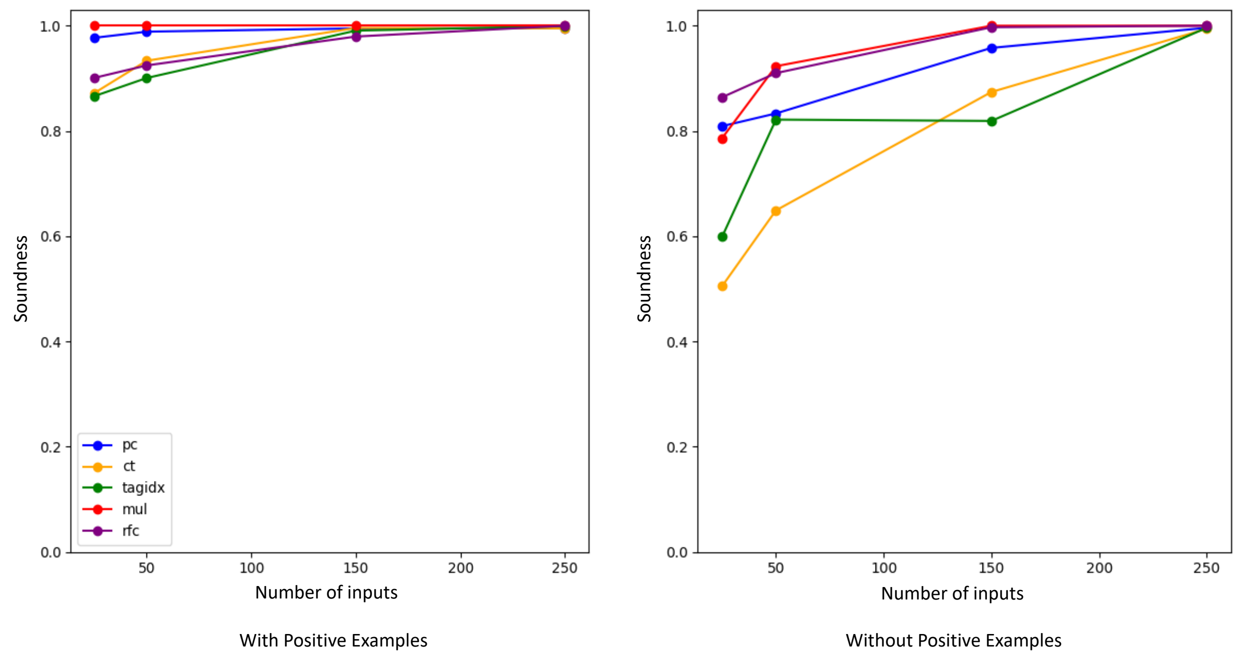

Fig. 8: Average of soundness per inputs

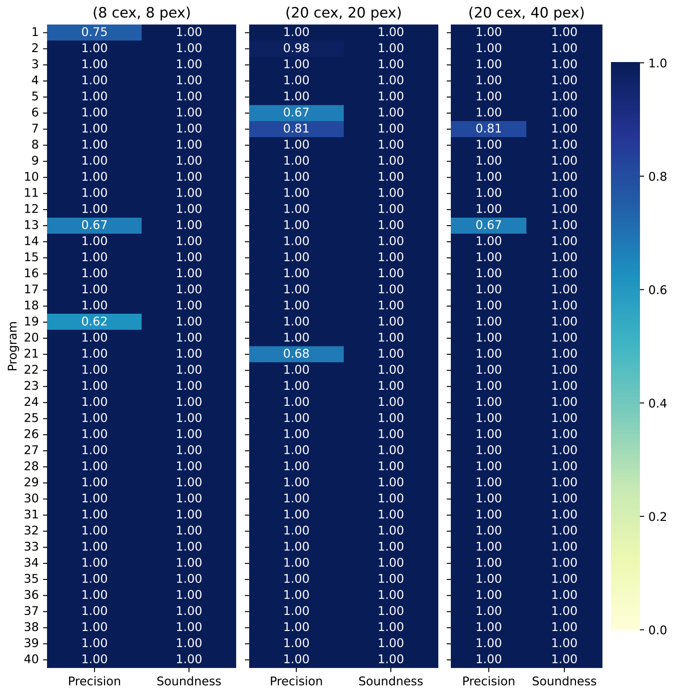

Fig. 9: Results per program for the CT contract.

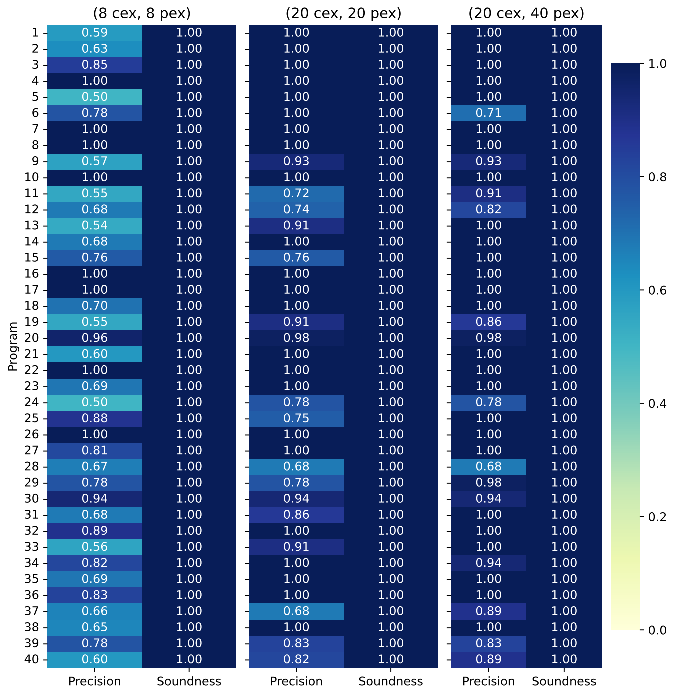

Fig. 10: Results per program for the TagIdx contract.

[7] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing," *arXiv preprint arXiv:2301.07642*, 2023.

[8] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. 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. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_12

[9] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Greece, October 18-22*, 2021. [Online]. Available: https://doi.org/10.1145/3466752.3480130

[10] 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 *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 347–360.

[11] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Information Security and Cryptology-ICISC 2005: 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers 8*. Springer, 2006, pp. 156–168.

[12] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying {Constant-Time} implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.

[13] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on information and system security (TISSEC)*, vol. 18, no. 1, pp. 1–32, 2015.