Automated Template-free Synthesis of Instruction-Centric Leakage Contracts for Black-Box CPUs

***Abstract*—Side-channel attacks pose a significant security threat for modern computing platforms, because they exploit subtle discrepancies in CPU behaviors to leak sensitive information. To model the information leaked through microarchitectural side-channels, recent work proposed *leakage contracts*: an ISA-level security abstraction that provides the foundation for secure CPU programming. Unfortunately, devising a leakage contract for a CPU requires extensive manual effort and thus modern commercial CPUs lack dedicated leakage contracts.**

We present a methodology to extract instruction-centric leakage contracts for major CPU architectures with minimal manual intervention. We implemented this technique in MALCOS, the first template-free tool that automates the synthesis of leakage contracts for black-box CPUs. We evaluate MALCOS on `x86` and `ARM` CPUs and show that the synthesized contracts are precise and sound with respect to all leaks observed during synthesis. Our results demonstrate that learning instruction-centric leakage contracts from black-box CPUs is feasible.

I. Introduction

Side-channel attacks exploit variations in processor behavior—such as execution time [1]–[5] or cache access patterns [6]–[9]—to infer sensitive information from otherwise secure software. To safeguard security-critical software, developers often adopt the Constant Time model [10]. This model assumes that the only leakage sources are control-flow instructions and memory accesses, and secure programs must make both secret-independent to prevent leaks. This assumption, however, is violated in modern CPUs. Microarchitectural optimizations—ranging from arithmetic optimizations for values like 0 and 1 [11] to advanced speculative execution techniques [12, 13]—can introduce new leaks beyond those captured by the Constant Time model. Hence, there is a growing need for techniques that enable developers and system architects to precisely characterize and mitigate such leaks.

Leakage contracts [14] augment the Instruction Set Architecture (ISA) with a specification of all attacker-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. The leakage contract framework [14] ensures that programs for which secret data do not influence any of the leaks specified in the contract are guaranteed to be leak-free at microarchitectural level. Unfortunately, constructing leakage contracts for modern CPUs is a complex task: it requires extensive reverse engineering, expert knowledge, and significant time investment [16], making it impractical to apply across the diverse landscape of commercially available CPUs.

Recently, several techniques have been proposed to synthesize *instruction-centric* contracts [17]–[21], i.e., a class of contracts where leaks are characterized as a function of an instruction's operands. These contracts can capture subtle data-dependent instruction-level optimizations [22]. These techniques reduce the manual effort needed to construct a comprehensive leakage contract for a given CPU by automating the characterization of instruction-level leaks, whereas other classes of leaks can be characterized using prior largely manual approaches [16].

However, existing contract synthesis approaches [17]–[21] suffer from two core limitations. First, they require access to a CPU's design at Register-Transfer Level (RTL) and, so far, have been applied only to small RISC-V cores. This limits their applicability to complex commercial CPUs (e.g., the latest x86 and ARM cores) for which RTL is not publicly available, despite subtle microarchitectural leaks being prevalent in these CPUs. Second, they require a user-provided *contract template* that directly defines the leaks that can be captured by all contracts in the synthesis' search space. For instance, the template might allow capturing leaks that depend on whether "an operand's value is 0." This limits the scope of these tools since they won't be able to capture leaks not explicitly part of the template. For instance, a timing leak introduced by a computation simplification optimization where the multiplication unit short-circuits whenever operands are 0 or 1 cannot be captured by a template that only allows expressing "an operand's value is 0." Although some works [17,20] suggested to manually tailor the template to account for missed leaks, this is time-consuming and reduces automation, and it requires knowledge of the underlying microarchitecture.

To address these issues, we propose a *template-free* methodology to extract instruction-centric leakage contracts from black-box CPUs, which we implement in the MALCOS contract synthesizer. Since MALCOS is black-box, it can directly work with commercial CPUs for which the RTL code is not available. Furthermore, by being template-free, MALCOS overcomes the limitations of template-based approaches and it can synthesize instruction-centric contracts with minimal manual intervention. Next, we overview our contributions.

Contract synthesis methodology (§IV): We propose a methodology for automatically synthesizing instruction-centric leakage contracts based on hardware observations extracted from a black-box CPU. For this, we adopt a counter-example-

driven synthesis method that refines candidate contracts based on observed hardware behavior. Our methodology consists of running the following two steps until reaching a fixed point:

- *Step 1:* Given a candidate contract (initially empty) capturing the leaks observed so far, we use existing relational leakage testing tools [15, 23] to validate the contract against a black-box CPU. These tools generate test programs and run them on the underlying CPU to discover *new microarchitectural leaks* not yet captured by the candidate contract.
- *Step 2:* Newly discovered leaks are synthesized into new ISA-level contract clauses, which are added to the original contract, ensuring that the candidate captures the discovered leak. To ensure that synthesized clauses precisely characterize the leak, our synthesis approach relies on both *counter-examples* (i.e., pair of executions showcasing a leak) and *positive examples* (i.e., pair of executions that are indistinguishable for a microarchitectural attacker).

Our methodology ensures that the synthesized contract is *boundedly-sound*, i.e., it captures *all* leaks exposed during testing. Furthermore, learned contracts have *high precision*, i.e., they do not unnecessarily over-approximate actual leaks.

MALCOS synthesis tool (§V): 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. To generate test cases and detect leaks (step 1), MALCOS leverages any suitable architecture-specific relational leakage testing tool; our implementation uses REVIZOR [23, 24] for `x86` and SCAM-V [15,25] for `ARM`. Contract synthesis (step 2) is performed using the Rosette framework [26].

Differently from existing contract synthesis approaches [17]–[21], MALCOS operates without access to the CPU's RTL design. Thus, it can be applied to off-the-shelf commercial CPUs for which RTL is not available and which lack comprehensive vendor-supplied leakage specifications. MALCOS goes beyond prior work by synthesizing fine-grained contracts instead of simply classifying instructions as safe or unsafe like [19, 21]. Finally, it guarantees that the synthesized contracts capture *all* observed leaks and it does not require user-provided contract templates (unlike [17, 20]).

Evaluation (§VI): To validate our methodology, we evaluated the accuracy of contracts generated by MALCOS on two fronts.

First, using five different contract models (ranging from the simple *constant-time* [27] model to ones capturing advanced security-critical optimizations like *register file compression* [22, 28] and *silent stores suppression* [22]) as ground truth, we used MALCOS to learn leakage contracts. Our results show that MALCOS is able to synthesize contracts that capture the ground-truth leaks and that these contracts are precise and sound based on the quantitative metrics that we have defined. Second, we use MALCOS to learn leakage contracts directly from six `x86` and `ARM` CPUs. Our results show that MALCOS learns contracts that are consistent with manually reverse-engineered models [37] directly from hardware measurements.

Finally, we compare the contracts synthesized by MALCOS with those generated by state-of-the-art *template-based* white-box tools [17]–[21]. The results show that our *template-free* approach eliminates the inflexibility of conventional *template-based* methods, leading to more precise contracts.

Summary of contributions: To summarize, this paper makes the following contributions:

- it presents a black-box method for learning boundedly-sound and precise instruction-centric leakage contracts (§IV);
- it implements this methodology in MALCOS, a template-free leakage-contract extractor for processors (§V);
- it evaluates MALCOS on several architectures and different contract models as ground truth (§VI).

Upon acceptance, we will release MALCOS as open-source with all datasets and scripts to reproduce our experiments.

II. OVERVIEW

We present the core aspects of MALCOS with an example.

A. Capturing Leaks with Leakage Contracts

To write side-channel-free programs for a given CPU and a given microarchitectural attacker, a programmer needs to know which program executions the attacker might distinguish to ensure that no secret data is leaked. The leakage contract framework [14] provides a way of formally characterizing microarchitectural leaks at the ISA level, thereby enabling secure programming. In this framework, a microarchitectural attacker is modelled by mapping each (microarchitectural) execution to a *hardware trace*, i.e., the observations that an attacker can make through microarchitectural side-channels.

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 *contract trace*, i.e., a sequence of (ISA-level) observations exposing potentially leaked information. In MALCOS, a contract is a set of *leakage clauses* $cl := ex$ `IF` pr, where ex specifies what is added to the contract trace and pr is a predicate modeling when the clause is enabled, i.e., when the observation is added to the trace.

A CPU *satisfies* a given contract for a microarchitectural attacker ATK [14], whenever the contract captures all attacker-observable leaks in the CPU. That is, any two executions that result in different hardware traces, i.e., they are distinguishable by ATK, must also result in different contract traces.

B. Synthesizing Leakage Contracts

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`) where MALCOS attempts to find new leaks in CPU not yet captured by $cand$.

2) A *contract refinement phase* (`Refiner`) where MALCOS updates $cand$ to account for a newly discovered leak.

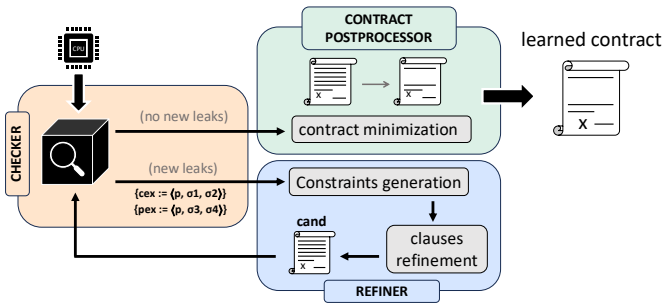


Fig. 1: MALCOS contract synthesis process

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 *counter-example* *cex*, which is a sequence of instructions and a pair of initial states that yield the *same contract trace* under *cand* but different hardware traces, i.e., *cex* is distinguishable by ATK. It also returns *positive examples* *pex*, i.e., test cases indistinguishable for both the contract and the attacker.

We remark that our synthesis methodology is `Checker`-agnostic. To emphasize its flexibility and generality, MALCOS uses two state-of-the-art relational leakage testers: REVIZOR [23] and SCAM-V [15]. These tools differ in methodology (REVIZOR uses fuzzing, SCAM-V symbolic execution), supported leakage sources (REVIZOR handles caches and timing; SCAM-V supports caches), and target architectures (REVIZOR targets `x86`, SCAM-V targets `ARM`).

Step 1 (A cex and a pex for RFC): Consider a processor CPU that implements a simple *register file compression* (RFC) optimization [22]. 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 [22, 28], 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.

Starting from an empty candidate contract ($\texttt{cand} = \emptyset$), 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 $\texttt{cand} = \emptyset$, the `Checker` discovers the test case (p, σ_1, σ_2) as a counter-example (`cex`):

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

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 the same contract traces w.r.t. *cand* but different hardware traces for the attacker ATK.

At the same time, the `Checker` also discovers the test case (p, σ_2, σ_3), where $\sigma_3 := (\texttt{rbx} \mapsto 2)$, as a positive example, since the two executions result in indistinguishable hardware *and* contract traces, i.e., the attacker ATK cannot distinguish them.

Contract refinement phase: The `Refiner` takes the counter-example `cex` (describing a newly-discovered leak) and the positive example `pex` (describing executions that should *not* be distinguished), and generates a new contract clause that captures the new leak. The `Refiner` discovers such a contract clause as a syntax-driven synthesis task implemented on top of the Rosette solver [26].

Positive examples are essential for *precision*. Without them, the `Refiner` may yield an over-approximate clause that exposes more information than needed (e.g., exposing *all* written register values rather than only writes of value 0). The `Refiner` ensures that the generated clause distinguish *as many counter-examples* as possible while distinguishing *as few positive examples* as possible. Finally, the `Refiner` adds the newly generated clause to the original contract to generate a new candidate contract `cand`. This process iterates, i.e., checking and refining to discover leaks not yet captured, until no further leaks are found.

Step 2 (A leakage contract for RFC): The `Refiner` analyzes the counter-example $\texttt{cex} := \langle p, \sigma_1, \sigma_2 \rangle$ and the positive example $\texttt{pex} := \langle p, \sigma_2, \sigma_3 \rangle$ of Step 1 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 those from σ_2 and σ_3. This yields the following contract cl_1:

$$\begin{aligned} cl_1 := \ & \textit{post-operand-value(0)} \ \texttt{IF} \ [(\textit{operand-type(0)} = \texttt{reg}) \\ & \texttt{and} \ (\textit{operand-access(0)} = \texttt{write}) \\ & \texttt{and} \ (\textit{post-operand-value(0)} = \texttt{0})] \end{aligned}$$

This contract exposes on the contract trace the final value of the first operand (index 0) whenever the first operand is a register and it is written with a value 0. Crucially, $\texttt{pex} := \langle p, \sigma_2, \sigma_3 \rangle$, where both states assign non-zero values to `rbx`, so ATK cannot distinguish them, prevents the `Refiner` from generating a coarser clause, yielding a *precise* cl_1 that only fires when RFC actually occurs.

Contract post-processing: 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 (`learned`) contract, further improving its precision. If CPU has no leaks beyond RFC, $\{cl_1\}$ (which exposes all registers written with a value of zero during execution) is returned as final contract.

III. LEAKAGE CONTRACTS

This section formalizes the syntax (§III-A) and the semantics (§III-B) of contracts, as part of a language we dub ICL,

$$
\begin{array}{llll}
\textit{(Bitstrings)} & bs & := & \{0,1\}^n \\
\textit{(Expressions)} & ex & := & bs \mid \ominus ex \mid ex_1 \oplus ex_2 \mid ex[bs_1 : bs_2] \\
 & & \mid & \texttt{REG}(bs) \mid \texttt{OPCODE} \mid \texttt{OP_TYPE}(bs) \\
 & & \mid & \texttt{OP_ACC}(bs) \mid \texttt{OP_VAL}(bs) \mid \\
 & & \mid & \texttt{POST_REG}(bs) \mid \texttt{POST_OP_VAL}(bs) \\
\textit{(Predicates)} & pr & := & \{0,1\} \mid ex_1 = ex_2 \mid ex_1 < ex_2 \\
 & & \mid & \texttt{NOT}\ pr \mid pr_1\ \texttt{AND}\ pr_2 \mid pr_1\ \texttt{OR}\ pr_2 \\
\textit{(Clause)} & cl & := & ex\ \texttt{IF}\ pr \\
\textit{(Contract)} & C & := & \emptyset \mid C, cl
\end{array}
$$

Fig. 2: ICL syntax

and the notion of contract satisfaction [14] (§III-C), which is used to detect counter-examples and positive examples.

A. ICL Syntax

The syntax of ICL is given in Figure 2. In ICL, a *contract* consists of a set of leakage clauses (using the terminology of [14, 23, 28]), where each clause describes a piece of information that might be leaked through side channels by the underlying CPU. Each clause cl consists of a conditional expression ex IF pr where the expression ex indicates the leaked information and the predicate pr expresses under which conditions the leak happens.

The basic type in ICL is bit strings bs, with bitvectors of length one denoting booleans. Expressions ex are constructed by combining bitstrings bs with unary $\ominus ex$ and binary $ex_1 \oplus ex_2$ operators, and the slice operator $ex[bs_1 : bs_2]$, which extracts from ex the substring between indices bs_1 and bs_2. Also, expressions can refer to five pre-defined functions:

- $\texttt{REG}(bs)$ looks up into the (logical) register file and returns the value associated with the register with index bs.
- OPCODE returns the opcode of the instruction.
- $\texttt{OP_TYPE}(bs)$ returns the type, which can be one of $\{\texttt{reg}, \texttt{mem}, \texttt{imm}\}$, of the operand in position bs.
- $\texttt{OP_ACC}(bs)$ returns the access mode, which can be one of $\{\texttt{r}, \texttt{w}, \texttt{rw}\}$, of the operand in position bs.
- $\texttt{OP_VAL}(bs)$ returns the value of the operand in position bs. This value depends on $\texttt{OP_TYPE}(bs)$ so that, if it is a register reg, the operand value will be the value of the register, if it is memory mem, the operand value will be the memory address it points to, and if it is an immediate imm, the operand value will be the value of the immediate.
- $\texttt{POST_REG}(bs)$ and $\texttt{POST_OP_VAL}(bs)$ returns the value of operands and registers after executing the instruction.

Finally, predicates can be boolean values, equalities and inequalities between expressions, or combinations of predicates with the standard logical operators.

B. ICL Semantics

The key element of interest in the semantics of leakage contracts is the *contract trace*, i.e., the sequence of observations obtained by evaluating the clauses in the contract on *every state* in the execution. In order to collect these traces (§III-B3), ICL requires the definition of a system semantics, which captures how the underlying system works (§III-B1) and of a contract semantics, which dictates how the observations that constitute a contract trace are generated (§III-B2).

This separation of semantics is unlike prior work on leakage contracts [14], where system and contract semantics are fused in one. This separation allows ICL to be modular in the underlying system semantics, for example, it can be instantiated with an ISA semantics such as the one of μ-assembly [29] or with a speculative semantics [30].

1) System Semantics: The system semantics $\rightarrow$: $\Sigma \times \Sigma$ maps each state to the next one by executing one instruction in program p (written $p \vdash \sigma \rightarrow \sigma'$), where Σ is the set of all states and $\sigma \in \Sigma$ is a system state. The system semantics is left abstract except for these requirements: First, the set $\Sigma_0 \subseteq \Sigma$ denotes the set of initial states. Second, states $\sigma \in \Sigma$ contain a memory m and a register assignment r that, resp., represent the state of memory and registers. Memories m map addresses (represented as integers) to values. Register assignments r map register identifiers (represented as bitstrings) to their values. We also assume that one of the register identifiers, denoted PC, represents the program counter register.

2) Contract Semantics: The semantics of an ICL contract C is obtained by extending the system semantics with observations $\delta \in Obs$ generated according to C. Formally, each ICL contract C induces a labeled semantics $\rightarrow_C$: $\Sigma \times Obs \times \Sigma$ specified by this inference rule:

$$
\frac{p \vdash \sigma \rightarrow \sigma' \quad \delta = \{[\![e]\!](\sigma, \sigma') \mid e\ \texttt{IF}\ p \in C \wedge [\![pr]\!](\sigma, \sigma') = \top\}}{p \vdash \sigma \xrightarrow{\delta}_C \sigma'}
$$

where $[\![ex]\!](\sigma, \sigma')$ (respectively, $[\![pr]\!](\sigma, \sigma')$) is the result of evaluating expression ex (respectively, predicate pr) in states σ and σ'. Note that $[\![\cdot]\!]$ needs the next-state σ' to evaluate $\texttt{POST_REG}(bs)$ and $\texttt{POST_OP_VAL}(bs)$. In a nutshell, the rule above labels a step of the system semantics ($p \vdash \sigma \rightarrow \sigma'$) with the set δ of observations obtained by evaluating in the current state the expressions of all leakage clauses in C whose predicates are satisfied.

3) Contract Traces: A *contract trace* τ is a sequence of observations obtained by applying the contract semantics $\rightarrow_C$ to an execution $p \vdash \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots$, that is, $\tau = \delta_0 \delta_1 \ldots$ where $p \vdash \sigma_0 \xrightarrow{\delta_0}_C \sigma_1 \xrightarrow{\delta_1}_C \ldots$ In the following, we write $\texttt{CTR}\,(C, p, \sigma)$ to denote the contract trace (under contract C) associated with the maximal execution of program p starting from initial state σ. Furthermore, we say that two initial states σ, σ' are *C-equivalent* for a program p whenever they result in the same contract traces. Formally: $p \vdash \sigma =_C \sigma' \stackrel{\text{def}}{=} \texttt{CTR}\,(C, p, \sigma) = \texttt{CTR}\,(C, p, \sigma')$.

C. Contract Satisfaction

Contract satisfaction precisely characterizes under which conditions a contract captures all leaks in a CPU.

Following [14, 23], 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 the function $\texttt{HTR}_{HW}\,(p, \sigma, Ctx)$, that returns the observations made by the attacker through microarchitectural side-channels on the target CPU HW. The function $\texttt{HTR}_{HW}\,(\cdot)$

Algorithm 1: MALCOS contract synthesis approach

```
1  Procedure ContractSynthesis(target, n, r):
       // Phase 1 - synthesis
2      P ← generateSeedPrograms(n);
3      cand ← [];
4      ctr ← [];
5      CEXs ← [];
6      for p ∈ P do
7          cand, ctr ← getInitialCand(ctr, cand, r);
8          while true do
9              cexs, pexs ← Checker(p, cand, target);
10             CEXs[p] ← CEXs[p] ∪ cexs;
11             if cexs ≠ ∅ then
12                 cand ← Refiner(p, cand, cexs, pexs);
13             else
14                 break
       // Phase 2 - minimization
15     ctr ← append(ctr, cand);
16     ctr ← minimize(ctr, CEXs);
17     return ctr;
```

takes as inputs a program p, the initial state σ processed by the program (i.e., the architectural state including registers and main memory), and the microarchitectural context Ctx in which it executes (i.e., the initial state of microarchitectural components like caches and predictors).

Let Δ be a set of *test cases*, i.e., triples $\langle p, \sigma, \sigma' \rangle$ consisting of a program p and a pair of initial states σ, σ'. A CPU HW *satisfies a leakage contract* C *for a set of test cases* Δ [14] if any two C-equivalent states also result in identical hardware traces in any context, i.e., the executions are indistinguishable by the attacker. Formally:

$$\Delta \vdash \textsc{CtrSat}(C, CPU) \stackrel{\text{def}}{=} \forall \langle p, \sigma, \sigma' \rangle \in \Delta. \forall Ctx. p \vdash \sigma =_C \sigma' \Rightarrow \mathtt{HTR}_{HW}\ (p, \sigma, Ctx) = \mathtt{HTR}_{HW}\ (p, \sigma', Ctx)$$

We say that a CPU *satisfies a contract* C if the CPU satisfies it for all programs and pairs of states. Dually, if a CPU does not satisfy a contract C, there is a *leakage counter-example* $\mathtt{cex} := \langle p, \sigma, \sigma' \rangle$ consisting of a program p and two initial states σ, σ' that produce different hardware traces (for some Ctx) but it generates identical contract traces for C.

IV. Synthesizing Leakage Contracts

In this section, we present our methodology for automatically synthesizing instruction-centric leakage contracts.

A. Synthesis Algorithm

Our algorithm (shown in Algorithm 1) automatically constructs a leakage contract from a CPU under test (denoted *target* in this section) in a black-box manner. That is, the `Checker` does not have access to the CPU design, and it only interacts with the CPU by executing *test cases*, each one consisting of a program p and a sequence of initial states, to derive the corresponding hardware traces.

The synthesis algorithm takes three inputs— the CPU under test *target*, the number of test programs n used during the synthesis, and the reset parameter r—and it works in two phases.

Phase 1 spans lines 2–14, and it synthesizes a set of ICL candidate contracts that capture the leaks explored by n test programs. Phase 1 starts by computing a set P of n (randomly generated) test programs (line 2). For each program $p \in P$, the algorithm first computes the starting initial candidate contract *cand* (line 7). This candidate can be the empty contract $\emptyset$, or it can be derived from the contracts generated from prior programs. Every r iterations, the current set of candidates *cand* is added to the accumulated contract *ctr*, and the search is restarted from the empty candidate, thereby allowing MALCOS to recover from mistakes caused by low-quality examples. Next, the algorithm checks whether there are leaks in the underlying CPU under test that are not yet captured by the candidate contract *cand* (line 9, described in §IV-C). For this, the algorithm relies on the `Checker` function, which returns a set of counter-examples *cexs* and a set of positive examples *pexs* for the program p. The former are leaks not yet captured by the contract *cand*, whereas the latter represent executions that are indistinguishable for the microarchitectural attacker (which are used to improve the precision of the synthesized contract). If `Checker` finds at least one counter-example, the algorithm refines the current candidate contract *cand* to account for the newly discovered leaks using the `Refiner` function (line 12; described in §IV-D). If `Checker` cannot find any further counter-examples, the algorithm terminates the testing of the current program, and moves to the next program in P.

Phase 2 spans lines 15–17, and it first consolidates the remaining candidates *cand* into the set of *ctr* contracts, then it derives the final leakage contract for the target CPU by minimizing *ctr* using the `minimize` function (line 16; described in §IV-E). The minimization step allows us to remove unnecessary leakage clauses from the contract, while preserving soundness w.r.t. the explored counter-examples. That is, the final leakage contract still distinguishes all counter-examples in $CEXs$, and it is more precise (i.e., it distinguishes fewer pairs of states with the same hardware trace) than the non-minimized contract. Finally, this phase returns the final leakage contract *ctr* for the target CPU to the user.

B. Choosing the Initial Candidate Contract

To select the initial candidate contract when testing a new program (`getInitialCand` on line 7 of Algorithm 1), MALCOS combines a strategy of *integration* and *resetting* based on a parameter r: the initial contract *cand* for the i-th tested program is obtained by integrating the contracts synthesized for the last $i \bmod r$ programs.This strategy allows the new candidate to account for previously found leaks–through the integration of previous contracts–and, therefore, reduces the time needed to find a candidate that covers all program leaks. At the same time, by resetting the candidate every r programs, it allows MALCOS to recover from mistakes. For instance, it allows resetting the contract when the `Checker` has not provided good enough positive examples for an individual program, which may lead the synthesis to over-approximate the leakage.

C. Finding Leakage Counter-Examples

When learning a new contract, our algorithm relies on a `Checker` that interacts with the CPU to determine whether it satisfies the current candidate contract. The `Checker` function takes as input the current program p, the contract candidate $cand$ and the target CPU $target$, and it checks whether $target$ satisfies the contract $cand$ w.r.t. the contract satisfaction notion from §III-C. Concretely, the `Checker` generates a *test case* for the program p (i.e., a set of initial states for p), derives the corresponding *hardware traces* and *contract traces* from $target$ and $cand$ respectively, and outputs a set of counter-examples and a set of positive examples.

A *counter-example* consists of two $cand$-equivalent states σ, σ' with different hardware traces, i.e., an attacker can distinguish the executions of p starting from σ and from σ' via a microarchitectural side-channel. That is, each counter-example represents a leak not captured by the contract $cand$.

In contrast, a *positive example* consists of two $cand$-equivalent states σ, σ' that also have the same hardware traces, i.e., two states that are indistinguishable for a microarchitectural attacker. Each positive example represents a pair of executions that the final contract should not distinguish. That is, if the final contract distinguishes two executions forming a positive example, the contract is an over-approximation of the desired contract (i.e., it distinguishes more executions than needed).

Note that our synthesis algorithm is not tied to a specific implementation of `Checker`. Rather, it can work on top of any approach for checking contract satisfaction. As we discuss in §V, MALCOS instantiates `Checker` using two state-of-the-art black-box testing tools for contract satisfaction: Revizor [23] for synthesis over the x86 architecture, and Scam-V [15, 25] for synthesis over ARM architectures.

D. Contract Refinement

The `Refiner` function takes as input a candidate $cand$, the test program p, a set of contract counter-examples $cexs$ and as set of positive examples $pexs$ for p and $cand$.

In a nutshell, the function refines the current candidate contract $cand$ to account for newly discovered leaks represented by the counter-examples in $cexs$. This requires finding an ICL contract $cand'$ such that (a) pairs of executions that are $cand$-distinguishable are also $cand'$-distinguishable (i.e., the refined contract is not "forgetting" leaks), and (b) $cand'$ distinguishes at least one counter-example in $cexs$ (i.e., the refined contract indeed captures at least one newly discovered leak).

For this, we cast contract refinement as an optimization syntax-guided synthesis problem. In particular, we refine the contract $cand$ by synthesizing at most m additional leakage clauses $cl_1, \ldots, cl_m$, where m is a parameter of the refinement procedure. Each clause cl_i has to satisfy the following two hard-constraints, where ? denotes a hole to be filled by the synthesis solver according to ICL's grammar:

$$cl_i = (?\ \texttt{IF}\ \texttt{REG}[\mathbf{PC}] = ? \wedge ?) \tag{1}$$

$$\bigvee_{(p,\sigma,\sigma') \in cexs} \texttt{CTR}\,(cl_i, p, \sigma) \neq \texttt{CTR}\,(cl_i, p, \sigma') \tag{2}$$

Hard-constraint 1 ensures that cl_i is an instruction-centric clause, which targets the leak associated with a single instruction in p (captured by the $\texttt{REG}[\mathbf{PC}] = ?$ part of the predicate, allowing us to capture fine-grained leaks that may depend on a specific instruction context). In contrast, hard-constraint 2 ensures that cl_i captures the leaks of at least one of the counter-examples $cexs$ by requiring that cl_i produces distinguishable leakage observations for at least one pair of counter-example states.

Additionally, to ensure the precision of cl_i, we add the following soft-constraint, which the solver should maximize:

$$max\left(\begin{array}{l}\left|\{(p,\sigma,\sigma') \in cexs \mid \texttt{CTR}(cl_i,p,\sigma) \neq \texttt{CTR}(cl_i,p,\sigma')\}\right| \\ +\ \left|\{(p,\sigma,\sigma') \in pexs \mid \texttt{CTR}(cl_i,p,\sigma) = \texttt{CTR}(cl_i,p,\sigma')\}\right|\end{array}\right) \tag{3}$$

The soft-constraint 3 ensures that cl_i distinguishes most counter-examples, i.e., the one capturing more leaks. It also ensures that cl_i will be the "most precise" clause that satisfies constraints 1 and 2, i.e., the one that distinguishes as few positive examples as possible.

The synthesis solver outputs a set of instruction-centric clauses cl_i that are specific for the program p. These clauses are of the form $ex\ \texttt{IF}\ \texttt{REG}[\mathbf{PC}] = val \wedge pr$ where val is the address of one of p's instructions. That is, they refer explicitly to the context of the program p. Then, the `Refiner` generalizes these clauses, allowing them to be applied across programs. Concretely, it replaces each clause $ex\ \texttt{IF}\ \texttt{REG}[\mathbf{PC}] = val \wedge pr$ with an equivalent clause $ex\ \texttt{IF}\ type_{p,\mathbf{PC}=val} \wedge pr$ where $type_{p,\mathbf{PC}=val}$ is a meta-level predicate matching the instruction type of the corresponding instruction at program counter val in p. Thus, $type_{p,\mathbf{PC}=val}$ is defined in ICL as the conjunction of predicates capturing the instruction type characteristics at the specific program counter val (namely $\texttt{OP_TYPE}(bs)$, $\texttt{OP_ACC}(bs)$, and $\texttt{OPCODE}(bs)$). That is, if the clause cl is associated with program counter $addr$ pointing to instruction i, the corresponding generalized clause is obtained by replacing the sub-predicate $\texttt{REG}[\mathbf{PC}] = addr$ with a predicate that is satisfied whenever the program counter points to an instruction of the same type as i (i.e., same op-code and operand types).

Finally, the `Refiner` outputs the new candidate contract $cand'$ as the union of $cand$ and the new clauses $cl_1, \ldots, cl_m$. This refinement satisfies points (a) and (b) above (the former follows from $cand' = cand \cup \{cl_1, \ldots, cl_m\}$ and the latter from the second constraint).

E. Minimization

Phase 2 of our algorithm post-processes the ICL contract to obtain a more precise one. Thus, the `minimize` function removes unnecessary clauses from the contract candidate $cand$. For this, $cand$'s clauses are sorted in terms of their precision (from higher to lower) over the test cases explored during Phase 1, where more precise clauses distinguish less attacker-equivalent pairs of executions. Then, the clauses that do not contribute to distinguishing the collected counter-examples are

iteratively removed, i.e., a clause is removed if all counter-examples are still distinguishable by a contract containing only the remaining clauses. This step allows us to remove "useless" clauses that do not capture actual leaks and only reduce precision. This minimization step significantly contributes to improving the precision of the final contract (see §VI-D).

F. Guarantees

Theorem 1 characterizes the bounded soundness guarantees offered by MALCOS. That is, MALCOS's synthesis approach ensures that synthesized contracts indeed capture all leaks discovered by the `Checker` during synthesis (i.e., returned as counter-examples). That is, the synthesized contract is satisfied by the target CPU w.r.t. the discovered counter-examples.

Theorem 1: Let Δ be the set of counter-examples explored during Algorithm 1 (i.e., $\Delta = \bigcup_p CEXs[p]$ at the end of the algorithm) and C be the synthesized contract w.r.t the target CPU HW. It holds that $\Delta \vdash \textsc{CtrSat}(C, HW)$.

V. Implementation

This section presents our contract synthesis tool. To show the generality of our approach, we implemented two backends to support `x86` and `ARM` architectures. We refer to MALCOS with each backend as MALCOS-X86 and MALCOS-ARM. Both follow the workflow outlined in Algorithm 1, but they differ in: the contract satisfaction tool (§V-A), the way contracts are refined (§V-B), and contract post-processing (§V-C). We remark that the differences between the two backends ultimately stem from the different contract satisfaction tools used.

A. Code Examples and Contract Satisfaction

The `Checker` part of MALCOS generates test programs (function `generateSeedPrograms` in Algorithm 1) and extract counter- and positive examples for a given contract candidate. MALCOS implements the `Checker` using state-of-the-art relational contract testing tools: REVIZOR [23] for `x86` and SCAM-V [15] for `ARM`. Both tools take as input a contract and try to discover contract violations (i.e., counter-examples) in the CPU under test by generating random programs and inputs, executing them on the actual CPU and at contract-level, and comparing collected traces. Off-the-shelf, both tools only output counter-examples, so we modified them to also extract positive examples. Each call to the `Checker` produces a single counter-example in MALCOS-X86 but yields multiple counter-examples in MALCOS-ARM.

MALCOS-X86*:* For `x86`, we rely on REVIZOR, a black-box hardware fuzzer that uses differential testing to compare pairs of hardware traces with pairs of contract traces to detect contract violations, and we extend it to support ICL contracts.

Given that REVIZOR uses randomly-generated programs and inputs during the testing process, we use two optimizations implemented in REVIZOR—program minimization and data minimization—to tame randomness and improve the precision of the synthesized contracts. *Program minimization* minimizes test programs that result in violations. For this, whenever REVIZOR finds a counter-example for a program p, REVIZOR iteratively removes instructions from p so long as it can still find a counter-example in the smaller program p'. *Data minimization*, instead, identifies which parts of the initial states lead to a violation. For this, whenever REVIZOR finds two architectural states σ and σ' causing a violation, it iteratively modifies σ' by copying part of the state from σ (by copying registers and memory at a byte-granularity) until the violation persists. MALCOS-X86 applies these minimization steps automatically whenever the `Checker` finds a counter-example, and the minimized counter-example is then used for synthesis.

MALCOS-ARM*:* For `ARM`, we use SCAM-V as `Checker`, which combines techniques from program verification and fuzzing to perform relational testing to validate the candidate contracts. For a generated test program p, SCAM-V uses *symbolic execution* to synthesize a relation that identifies which states are observationally equivalent according to the contract being validated. Next, SCAM-V generates an instance of this relation in terms of two input states. Finally, similar to REVIZOR, SCAM-V runs the generated program with two inputs on hardware to collect hardware traces.

B. Contract Refinement

To refine the candidate contract (`Refiner` in Algorithm 1), we implement the instruction-centric clause synthesis process described in §IV-D using Rosette [26], a solver-aided language extending Racket. We formalized ICL's syntax and semantics in Rosette, and we collect all state sequences from counter-examples $cexs$ and positive examples $pexs$ to instantiate the synthesis constraints 1–3. Rosette uses the Z3 SMT solver [31] to synthesize ICL clauses that satisfy these constraints.

Note that MALCOS-X86 synthesizes one clause per instruction in the program under test p during refinement, whereas MALCOS-ARM synthesizes a single clause for the entire program. This is due to the program minimization step in MALCOS-X86 (§V-A), which pinpoints *all* instructions involved in a leak, enabling clause synthesis for each instruction. In contrast, MALCOS-ARM does not minimize the program, which may include irrelevant instructions. By synthesizing a single clause, we allow the solver to directly identify leak sources without unnecessary approximations from clauses tied to unrelated instructions.

C. Contract Post-Processing

Both MALCOS-X86 and MALCOS-ARM minimize the final ICL contract into a more precise one using the minimization step from §IV-E. Thus, both toolchains sort clauses by the number of positive examples they (unnecessarily) distinguish, ordering from those that distinguish less (i.e., more precise clauses) to those that distinguish more. This is evaluated using the notion of C-equivalence (§III-B3), where we say a clause cl_2 is at least as precise as cl_1 if $\forall(p, \sigma, \sigma').\, p \vdash \sigma =_{cl_2} \sigma'$ then $p \vdash \sigma =_{cl_1} \sigma'$. This implication allows us to filter out redundant or less precise clauses. The key difference between the toolchains resides on the domain in which this condition is verified, which ultimately stems from the different checkers used. MALCOS-ARM uses an SMT solver to formally verify the

formula above (i.e., the universal quantification ranges over all possible pairs of states in the program), whereas MALCOS-X86 relies on testing the condition on the set of known counter-examples ($\forall(p, \sigma, \sigma') \in CEXs$).

VI. Evaluation

In our evaluation, we address the following questions:

RQ1: How good are MALCOS' contracts?
RQ2: What is the impact of contract minimization?
RQ3: Can MALCOS synthesize contracts from hardware?
RQ4: How do MALCOS-learned contracts compare to *template-based* approaches?

We begin by introducing the metrics used to evaluate contract quality in **RQ1**, **RQ2** and **RQ4** (§VI-A), then we describe the target leakage contracts used as ground truth in **RQ1**-**RQ2** (§VI-B). Finally, we answer **RQ1**-**RQ4** in §VI-C–VI-F.

A. Metrics

Leakage contracts act as binary classifiers over pairs of executions, i.e., a contract indicates whether an observer can distinguish two executions. To evaluate the quality of the learned contracts, we use two standard metrics for binary classification: precision and soundness. These metrics are defined w.r.t. a *validation set* V consisting of test cases (p, σ, σ') where p is a program and σ, σ' are initial states.

Precision measures how precisely the learned contract reflects leaks in the target system. Following [17,20], precision is defined as: $\frac{TP}{TP+FP}$, where TP is the number of true positives, i.e., test cases in V that are distinguishable using both contract and hardware traces, and FP is the number of false positives, i.e., test cases in V that are distinguishable only by the contract traces but not by the hardware traces.

Soundness measures the correctness of the learned contract. We define it as $\frac{TP}{TP+FN}$, where FN is the number of false negatives, i.e., test cases in V that are distinguishable by hardware traces but not by the contract traces.

B. Contract Models

To answer **RQ1**-**RQ2**, we use MALCOS to synthesize leakage contracts against a set of fixed ground-truth contract models, i.e., MALCOS uses these models as synthesis target.[1] These models, described next, capture representative microarchitectural behaviors and relevant leakage scenarios.

- **Constant-time (ct)** models the constant-time observer commonly used when reasoning about side-channels in cryptographic implementations [27]. **ct** exposes the value of the program counter and the addresses of load and store operations throughout the execution.
- **Tag Index (TagIdx)** differs from **ct** in that it exposes only the tag and set index of memory accesses (instead of the whole memory address like in **ct**). **TagIdx** is often used to reason about the security of cryptographic implementations against cache-based side-channel attacks [32].
- **Register file compression (RFC)** models the leaks induced by the compression optimization described in §II.
- **Silent Store (SilStore)** differs from **RFC** in that it produces a leak when a value 0 is written to memory (instead of a register like **RFC**). **SilStore** models leaks from store operations that do not alter memory contents [22], and its effects have been observed on several Intel CPUs [33].
- **Multiplication simplification (mul)** models leaks caused by simplification over multiplications [22]. Specifically, **mul** exposes if a multiplication's operands are 0 or 1.

[1] MALCOS-X86 and MALCOS-ARM have been extended to support a leakage contract C as synthesis target. For this, the hardware trace associated with program p and state σ is computed by simulating p using an ISA simulator and, for each instruction, all clauses in C are evaluated w.r.t. the current architectural state to compute the corresponding leakage observations. Note that, computing the hardware trace when using a given contract as synthesis target involves no hardware measurements, and the hardware trace consists only of the leakage observations computed by the ISA simulator.

C. *RQ1: How Good Are* MALCOS' *Contracts?*

RQ1 evaluates the contracts generated by Algorithm 1.

Experimental setup: We use MALCOS-X86 and MALCOS-ARM to synthesize contracts against the models from §VI-B.

For MALCOS-X86, for each target contract, we randomly generate 10000 programs and 100 random states per program, resetting the contract after every 500 programs (i.e., $R = 500$), and we use MALCOS-X86 to synthesize the contracts (using 100 positive examples during synthesis). We repeat the experiment outlined above 10 times, with different initial randomness seeds. We then measure MALCOS runtime and the precision/soundness of the learned contracts against a validation set of 50000 new random programs and 100 new random states per program (i.e., 100^2 pairs of traces), and report the average of both metrics.

For MALCOS-ARM, for each target contract, we randomly generate 100 programs, with 20 positive examples (cached at line 9 in Algorithm 1 and reused among the while loop iterations of the program under test) and 10 counter-examples, resetting the contract after every program (i.e., $R = 1$). We then measure MALCOS runtime and the precision/soundness of each synthesized contract against a validation set consisting of 100 new programs, each with 100 pairs of states (50 target-distinguishable pairs and 50 target-indistinguishable pairs).

For MALCOS-X86 and MALCOS-ARM, we report the number of clauses in the synthesized contracts, i.e., the final number of clauses after the postprocessing phase in Algorithm 1.

Results: Table I shows our results. Both toolchains successfully learn contracts that capture all target leaks, i.e., soundness is always 100%. Both tools resolved the over-approximations during the minimization process, achieving 100% precision in almost all cases (see Table II for comparison).

We manually inspected the case where MALCOS-X86 could not achieve 100% precision (**TagIdx**, 79%), and we observed conservative over-approximations that justify not reaching 100% precision, confirming that the lower precision is due to MALCOS-X86 exposing more address bits than are actually needed in the final contract. These over-approximations results from low-quality examples, whose impact can be mitigated

Contract	MALCOS-X86 with Ctr. minimization			MALCOS-ARM with Ctr.minimization		
	avg. P/S	avg. Time	avg. # Clauses	P/S	Time	# Clauses
ct	1.0/1.0	11h 57m	6	1.0/1.0	5h 40m	2
TagIdx	0.79/1.0	18h 57m	14.4	1.0/1.0	13h 42m	2
RFC	1.0/1.0	10h 32m	9.8	1.0/1.0	12h 51m	6
mul	1.0/1.0	24h 21m	15.7	1.0/1.0	22h 34m	4
SilStore	1.0/1.0	1h 23m	1	1.0/1.0	5h 29m	1

TABLE I: MALCOS results: Precision (P), Soundness (S), and Number of Clauses (# Clauses).

Contract	MALCOS-X86 without Ctr. minimization			MALCOS-ARM without Ctr.minimization		
	avg. P/S	avg. Time	avg. # Clauses	P/S	Time	# Clauses
ct	0.70/1.0	10h 31m	288.2	0.51/1.0	5h 21m	24
TagIdx	0.14/1.0	14h 14m	388.9	0.50/1.0	12h 44m	204
RFC	0.30/1.0	8h 40m	368.8	0.50/1.0	11h 36m	271
mul	0.20/1.0	14h 58m	255.5	0.50/1.0	18h 20m	404
SilStore	0.13/1.0	1h 19m	32.2	0.58/1.0	5h 9m	86

TABLE II: Precision (P), Soundness (S), and Number of Clauses (# Clauses) without contract minimization.

by reducing the reset parameter or increasing the number of positive examples as shown in Appendix D. For example, MALCOS-X86 with more frequent resetting ($R = 100$) achieves a precision of 96% for **TagIdx**.

We also manually inspected all synthesized contracts with 100% precision (for MALCOS-X86 and MALCOS-ARM) and confirmed their equivalence to the ground truth. In particular, for MALCOS-ARM, we used SCAM-V's SMT-backend to verify that synthesized contracts and target ones are equivalent.

D. RQ2: What is the Impact of Contract Minimization?

RQ2 evaluates the impact of contract minimization (line 16 in Algorithm 1) on the quality of the synthesized contracts.

Experimental setup: To assess the impact of contract minimization, we repeat the experiments of §VI-C while disabling the minimization phase for MALCOS-X86 and MALCOS-ARM. We measure the average MALCOS runtime, precision/soundness of the synthesized contracts against the same validation sets as in §VI-C, and the number of clauses in the synthesized contracts (without minimization).

Results: Table II reports the results of our experiment. Comparing the results in Table II with those in Table I (where the synthesized contracts have been minimized) shows that, without minimization, MALCOS synthesizes contracts that are significanlty less precise, even though soundness stays at 100%. That is, minimization indeed is useful and improves the precision of synthesized contracts.

These results can be understood more clearly when comparing the number of clauses in the synthesized contracts with and without minimization (cf. columns *# Clauses* in Tables I and II). Without minimization, the final contracts contain many more clauses that might unnecessarily introduce over-approximations (thereby resulting in less precise contracts) without contributing to soundness.

Finally, as expected, removing the minimization step reduces the total execution time (since MALCOS can directly terminate at line 15 of Algorithm 1). This impact is more noticeable in more complex contracts, such as **TagIdx** and **mul**, whereas the impact is very limited or absent for simpler contracts like **ct** and **SilStore**.

*E. **RQ3:** Can MALCOS Synthesize Contracts From Hardware?*

RQ3 focuses on whether MALCOS can be used to learn contracts from x86 (§VI-E1) and ARM (§VI-E2) CPUs.

1) x86: Using MALCOS, we synthesize contracts for 13 subsets of the x86 ISA. Altogether, these subsets result in a complete base x86-64 user-level instruction set, excluding the instructions currently not compatible with REVIZOR. For these subsets, we re-use REVIZOR's initial configuration from [24] where (a) all subsets include the same 8 base arithmetic/logic instructions (including their versions with memory operands), needed by REVIZOR to work properly, and (b) each subset extends the base instructions with unique subset-specific instructions. More details are available in Appendix B.

Experimental setup: We target four different Intel CPUs: i5-6500 (Skylake), i5-1335U (Raptor Lake), Ultra5-225U (Arrow Lake), and Ultra7-258V (Lunar Lake). For each ISA subset, we use MALCOS-X86 to synthesize the contract capturing the leaks associated with the subset's instructions. Each subset was tested with 100000 randomly generated programs, and each program executed with 100 inputs, restarting from an empty candidate every 500 programs (i.e., $R = 500$), and we used up to 100 positive examples for synthesis. To account for speculatively executed instructions, we configure REVIZOR to explore at ISA-level also mispredicted branches[2] (to account for branch speculation) and to disable store-bypass speculation (by enabling the `ssbd` patch [34]). To derive the hardware traces, we configure REVIZOR to use two different executor modes: `TSC` [35], where traces are obtained by measuring the test cases' execution time with `RDTSCP`, and `P+P` [35], where hardware traces contain the cache sets that were accessed during the test case execution. These modes correspond to different microarchitectural attackers: a timing-based attacker and an access-based cache attacker one [9].

Results: MALCOS-X86 performed in total 260 million test executions for each of the CPUs, over the 13 subsets of the x86 ISA and the two executor modes. The execution took 19 hours and 45 minutes per CPU on average.[3] MALCOS-X86 spent an average of 1 hours and 35 minutes per CPU for contract refinement across all subsets, and 6 minutes on average for contract minimization. Table III summarizes the results of the synthesis campaign, and it highlights the types of synthesized clauses for each subset. We manually inspected the synthesized contracts and highlight the following findings:

[2]For this, we enabled the `cond` execution clause in Revizor, i.e., using the always-mispredict speculative semantics from [29] as system semantics $\rightarrow$ for MALCOS (§III-B).

[3]During testing, MALCOS's throughput is aligned with prior evaluations of REVIZOR [24].

• MALCOS-X86 synthesized clauses capturing the same leakage sources per subset across the different CPUs and executor modes.

• On average, the amount of attacker-indistinguishable pairs of states explored during the testing campaign with respect to the synthesized contracts is only 10.67%.

• For all subsets, MALCOS-X86 synthesized clauses exposing the addresses of memory accesses, which capture leaks through the data cache. Even for subsets like `nop`, the base instructions might still result in memory operations. E.g., MALCOS-X86 synthesized the following clause that exposes the address of a memory load/store:

$$cl_{mem} \coloneqq \textit{OP_VAL(0)}\ \texttt{IF}\ [(\textit{OP_TYPE(0)}\ \texttt{=}\ \texttt{mem})\ \texttt{and}\ (\textit{OP_ACC(0)}\ \texttt{=}\ \texttt{r/w})]$$

• For the `cond` subset (the only one including control-flow statements), MALCOS-X86 synthesized clauses exposing the program counter, thereby exposing control-flow leaks. For instance, MALCOS-X86 synthesized the cluase cl_{pc} below that exposes the program counter after the execution of a control-flow statement:

$$cl_{pc} \coloneqq \textit{POST_REG(}\textbf{\textit{PC}}\textit{)}\ \texttt{IF}\ (\textit{OPCODE}\ \texttt{=}\ \texttt{jmp})$$

• For the `dmul` subset, MALCOS-X86 synthesizes clauses exposing information about the divisor operand. This is consistent with prior findings showing that division operations are not constant-time [36]. For instance, MALCOS-X86 synthesized the clause cl_{div} exposing the dividend whenever the divisor is zero:

$$cl_{div} \coloneqq \textit{OP_VAL(1)}\ \texttt{IF}\ [(\textit{OPCODE}\ \texttt{=}\ \texttt{div})\ \texttt{and}\ (\textit{OP_VAL(0)}\ \texttt{=}\ \texttt{0})]$$

• For the *strn* subset (which includes `rep` instructions), MALCOS-X86 synthesized clauses exposing the `rep` counter which determines how many times an instruction is repeated. E.g., consider this clause synthesized by MALCOS-X86:

$$cl_{repe} \coloneqq \textit{OP_VAL(1)}\ \texttt{IF}\ [(\textit{OPCODE}\ \texttt{=}\ \texttt{repe stosd})\ \texttt{and}\ (\textit{POST_OP_VAL(2)}\ \texttt{=}\ \texttt{0})]$$

Here, cl_{repe} exposes the destination pointer when the `rep` counter is zero, i.e., in the final iteration.

• For all target CPUs, we validated MALCOS-X86's results against the instruction latencies from *uops.info* [37], a collection of state-of-the-art reverse-engineered latency models for x86 CPUs. The leakage sources identified by MALCOS-X86 are consistent with the *uops.info*'s latency profiles, i.e., the leaks found by MALCOS-X86 correspond to all instructions with variable latency in [37].

2) ARM: Using MALCOS, we synthesize a contract associated with ARM memory instructions. We analyzed 30 relevant load/store instructions, including word, (signed) byte, (signed) halfword, unscaled offsets, and register pairs (see Appendix C for the list of analyzed instructions).

	Skylake	Raptor Lake	Arrow Lake	Lunar Lake
`cond`: Conditional branches	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`strn`: String operations	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`dmul`: Division and mult.	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`flag`: Operations on flags	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`lock`: Atomics w/ LOCK	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`atom`: Atomics w/o LOCK	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`dxfr`: Data transf. (load/store)	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`setc`: Conditional byte set	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`nop`: NOP instructions	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`logi`: Logical operations	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`conv`: Data type conversion	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`cmov`: Conditional moves	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲
`bit`: Bit test and bit scan	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲	♦ ■ ● ▲

TABLE III: Summary of x86-64 synthesis campaign across 13 ISA subsets and 4 Intel microarchitectures. Solid symbols indicate that MALCOS synthesized clauses capturing the corresponding leakage source (♦: memory address, ■: control flow, ●: repetition counter, ▲: division by zero); grey symbols indicate that MALCOS did not synthesize such clauses.

Experimental setup: We used MALCOS-ARM to synthesize contracts against a Cortex-A72 (on Raspberry Pi4) and Cortex-A76 (on Raspberry Pi5) CPUs. To derive hardware traces, SCAM-V monitors the L1 cache state immediately after each test case execution, in an access-driven attack fashion [9]. It uses a *platform module* that runs in ARM TrustZone to configure page-table attributes to control memory cacheability, clear the caches before execution, insert memory barriers around the test case code, and read the cache state using privileged debug instructions. Each experiment was repeated 10 times, and the final cache state was checked for discrepancies.

MALCOS-ARM tested each instruction individually. For each contract refinement iteration, we used up to 10 counter-examples and 20 positive examples for synthesis.

Results: The synthesis campaign took ≈5.5 hours for Cortex-A72 and ≈21.2 hours for Cortex-A76. On average across the two campaigns, 4 minutes were spent on contract refinement and 9 minutes on minimization, with the remaining time was spent generating counter- and positive examples. The runtime difference arises because Cortex-A72 tests ran concurrently on five boards, whereas Cortex-A76 tests ran on a single board.

MALCOS-ARM produced two minimized contracts, one per CPU, each consisting of the same six clauses. Two clauses expose the cache tag (all bits above bit 12) and index (bits 6-12) associated with individual memory loads and stores.

$$cl_{mem} \coloneqq \textit{(OP_VAL(1)[64:6])}\ \texttt{IF}\ (\textit{OPCODE}\ \texttt{=}\ \texttt{ld*/st*})$$

MALCOS-ARM successfully captures the leakage of accessed cache lines. The least significant 6 bits represent the address offset and can be ignored when identifying the accessed line.

The remaining four clauses are associated with `LDP`, `LDPSW`, and `STP`, which access two 32-bit words or two 64-bit doublewords from memory at the same time. Similarly to cl_{mem}, these clauses again expose the tag and index bits of the accessed memory addresses.

	MALCOS-X86	LEASYN [17]	RTL2μPATH [18]	VELOCT [19]	MALCOS-ARM	LEASYN [17]	RTL2μPATH [18]	VELOCT [19]
Ibex-small	1.0 / 24.2 / 1.0	1.0 / 22.3 / 0.5	0.99 / 16.9 / 0.4	0.99 / 13.9 / 0.1	1.0 / 14 / 1.0	1.0 / 4 / 0.8	1.0 / 4 / 0.8	1.0 / 4 / 0.7
Ibex-mult-div	1.0 / 30.2 / 1.5	1.0 / 19.8 / 0.6	0.68 / 17.6 / 0.6	0.68 / 15.4 / 0.2	1.0 / 20 / 1.3	1.0 / 7 / 1.0	0.85 / 7 / 1.0	0.85 / 7 / 0.9
Ibex-optimizations	0.89 / 51.7 / 2.0	0.57 / 20.3 / 0.8	0.37 / 19.7 / 0.5	0.32 / 15.8 / 0.4	0.92 / 35 / 4.8	0.56 / 17 / 3.1	0.52 / 13 / 2.3	0.52 / 14 / 2.6
Ibex-slice	0.73 / 100.5 / 2.1	0.41 / 29.6 / 0.4	0.36 / 20.3 / 0.3	0.18 / 15.8 / 0.2	0.84 / 129 / 6.4	0.58 / 5 / 1.1	0.56 / 5 / 0.9	0.56 / 5 / 0.9

TABLE IV: Comparison of precision, number of clauses, and (relative) execution time with templates from previous works

3) Summary: These results confirm that MALCOS can synthesize contracts from black-box CPUs. Since MALCOS relies on black-box learning, the quality of generated contracts is inherently tied to the quality of the `Checker`'s test cases, and comprehensively identifying all leaks may require more tests. For instance, Cortex-A53 can leak the cache line offset accessed by memory loads [15], but we did not observe this behavior in our synthesis campaign. Improving existing `Checkers` is, however, beyond the scope of our work.

F. RQ5: How do MALCOS*-learned Contracts Compare to Template-based Approaches?*

Previous work [17, 18, 20, 21] have developed contract synthesis tools based on templates and white-box methods that require (1) access to the processor's RTL design and (2) a set of user-provided clause templates defining the synthesis search space. Our work overcomes these dependencies by introducing the first black-box synthesis approach for leakage contracts that is *template-free*. Since no existing tool operates under the same black-box, *template-free* conditions as MALCOS, we cannot compare against them on the same hardware targets (note that [17, 18, 20, 21] target small open-source RISC-V cores whereas MALCOS focuses on x86 and ARM commercial CPUs). Instead, we design a controlled evaluation that measures the *precision* of contracts synthesized by MALCOS against the same search space as each competing technique.

Experimental setup: To enable a fair comparison, we define a set of ground-truth models as synthesis targets. This allows us to do the comparison without requiring RTL access.

We define four ground-truth models inspired by the leakage profile of the RISC-V **Ibex** core [38] (which is the CPU targeted by the competing *template-based* tools [17,18,20,21]), and of some optimizations covering representative microarchitectural behaviors and relevant leakage scenarios (like those in §VI-B): (1) **Ibex-small**: the default "small" configuration of **Ibex** with constant-time multiplication (three cycles) and without caches (which leaks information about the outcome of branch instructions and accessed memory addresses). (2) **Ibex-mult-div**: **Ibex-small** extended with a non-constant–time multiplication unit [39] whose execution time depends on the operand values and it additionally leaks whether divisions and multiplications by zero happen. (3) **Ibex-optimizations**: **Ibex-mult-div** extended with the leaks captured by the contract models defined in §VI-B. (4) **Ibex-slice**: **Ibex-mult-div** extended with memory leaks that expose only the tag and set index of memory accesses (**TagIdx** from §VI-B), and arithmetic leaks whose execution time depends on the sign (e.g., divisions) or size (e.g., multiplications) of the operands. Note that **Ibex** is a RISC-V core, whose ISA is not supported by MALCOS. For this reason, we instantiate the leakage profiles above for the x86 and ARM ISA (rather than RISC-V).

We compare MALCOS against LEASYN [17] (which extends [20]), RTL2μPATH [18], and VELOCT [19] (which extends CONJUNCT [21]), the three most recent and relevant *template-based* white-box synthesis tools for leakage contracts. To allow the comparison, we emulate their behavior by restricting the synthesis grammar to the clause templates supported by each technique. We also run MALCOS in its original unrestricted grammar mode as an additional point of comparison. To have a fair comparison among the tools and MALCOS, we run the evaluation in MALCOS-X86 and MALCOS-ARM with the experimental setup as in §VI-C (to account for the impact of different `Checkers`). We then measure the precision of the synthesized contracts. For each leakage profile and tool, we also measure the number of clauses in the final contract and the relative execution time w.r.t. MALCOS's execution time on **Ibex-small**.

Results: Table IV summarizes the results. For the simplest **Ibex-small** model, all templates are sufficient to synthesize precise contracts. Yet, when considering more complex leakage profiles, restrictive templates cannot precisely express the leaks as part of a contract. Notably, VELOCT (which only supports a template exposing all operands of an instruction) and RTL2μPATH (which only supports a template exposing a subset of all operands of an instruction) fail in precisely capturing the leaks of **Ibex-mult-div** since they cannot precisely capture the specific values of the operands, e.g., *OP_VAL(0)*=`0`. LEASYN supports more complex templates that have been hand-crafted to capture the leaks in existing **Ibex** CPUs [20] and, thus, its template can still precisely capture the leaks in **Ibex-mult-div**. As soon as the templates become too restrictive, precision drops significantly. MALCOS can precisely capture the leaks in **Ibex-small** and **Ibex-mult-div**. Moreover, due to its template-free nature, it always results in more precise contracts than those associated with the templates supported by existing tools in complex targets like **Ibex-optimizations** and **Ibex-slice**. Also, MALCOS generates contracts with more clauses since template-freeness permits generating multiple more precise clauses. In contrast, other techniques operate over a more limited search space and converge earlier to a sound (but less precise) contract with fewer clauses due leakage over-approximations. As a result, MALCOS generally incurs higher execution time, while template-based techniques are correspondingly faster due to earlier convergence.

VII. DISCUSSION

Limitations: MALCOS is limited by several design choices. First, it focuses on instruction-centric contracts, synthesizing

only leak clauses (what is leaked) according to the terminology of [14, 23, 28]. MALCOS also cannot represent stateful clauses that capture leaks across multiple instructions (e.g., those from computation reuse or operand packing [22, 28]). Moreover, MALCOS explores only user-mode code, as our `Checkers` do not support privileged instructions, excluding kernel-level leaks (e.g., page table accesses). Finally, MALCOS is limited to leaks that are exercised and detected by the `Checker` (which is the only component interacting with the target CPU). That is, leaks not exercised during testing or not detectable by the `Checker` will not by captured by MALCOS.

Black-box synthesis: MALCOS demonstrates that contract synthesis for off-the-shelf CPUs is feasible without RTL access. RTL approaches [17]–[21] learn more exhaustive contracts but require access to the CPU's design and manual effort, limiting their applicability. MALCOS guarantees *bounded* soundness (all detected leaks are captured), using test cases to approximate precision. This makes it practical for commercial CPUs lacking RTL or vendor-supplied specifications.

False negatives: The synthesized contracts capture only leaks exercised by the test cases and detected by the `Checker`. Leaks outside these conditions result in *false negatives*.[4]

False positives: Despite minimization (§IV), contracts may still over-approximate leaks, resulting in *false positives*. This occurs if the `Postprocessor` cannot remove all over-approximations, or if some leaks (e.g., only a hash of a register leaks) are inherently hard to infer precisely from test cases.

Threats to validity of the evaluation: To cover a large space of test cases and to reduce biases introduced by manually selected test-cases, we decided to construct the validation sets using the random program generators included in the two relational testing tools supported by MALCOS, that is, REVIZOR and SCAM-V. In particular, for MALCOS-X86, on average, more than 70% of the trace pairs are distinguishable by the target, whereas for MALCOS-ARM, the validation set (generated using the SMT-backend of SCAM-V) is constructed so that 50% of the trace pairs are distinguishable. This ensures that the generated validation sets exercise both leaking and non-leaking behaviors, and, thus, result in informative metrics.

Other side channels: MALCOS can generalize to other observational models (e.g., power [40] or encrypted memory [**?**]) by integrating domain-specific checkers. For instance, power leaks can be modeled via ISA-level observational equivalence [40], requiring a checker that can test for power leaks.

Leakage contracts and secure programming: Contracts provide the foundations for leak-free software. As shown in [14], ensuring that contract traces are not dependent on program secrets is sufficient to guarantee the absence of microarchitectural secret-dependent leaks for any CPU satisfying the contract. For a clause ex `IF` pr, one needs to ensure that (a) the predicate pr is always secret-independent and that (b) whenever pr holds, the expression ex is also secret-independent.

[4] In §VI, "false negatives" are cases distinguishable by hardware but not by contract traces, so 100% soundness covers all *detected* leaks, not all leaks.

VIII. RELATED WORK

Leakage Contracts: Several works aim to formalize microarchitectural features to bridge the hardware-software abstraction gap [41]. Some introduce operational semantics for speculative and out-of-order execution [29,30,42]–[47], at varying abstraction levels. Others capture microarchitectural side-effects using axiomatic semantics [48]–[50]. Our work builds directly on the leakage contract framework [14], which provides a formal basis for relating contracts to concrete leaks in CPUs.

Synthesis of Contracts: The closest work to MALCOS includes RTL2MμPATH [18], which verifies contracts directly from RTL, and Mohr et al. [20] and LEASYN [17], who build contracts from manually-specified clauses and RTL tests. CONJUNCT [21] and VeloCT [19] synthesize coarse-grained contracts for timing attacks, classifying instructions as "safe" or "unsafe" from RTL. We remark that all these tools are template-based, that is, they require user-provided clause templates directly constraining the synthesis search space. In contrast, MALCOS automatically learns precise, fine-grained contracts for black-box CPUs without templates or RTL access, using counterexample-guided synthesis.

Counter-Example-Guided Synthesis: Different works have applied counter-example-Guided Synthesis (CEGIS) [51] to various tasks, including those related with the hardware-software interface. For instance, Heule et al. [52] and Liu et al. [53] apply CEGIS to automatically synthesize semantics and language semantics. In contrast, PipeSynth [54] uses formal synthesis to generate axioms for microarchitectural memory consistency. Finally, SYNTHCT [55] synthesizes translations of leaky instructions into leak-free code. MALCOS follows at high-level the standard CEGIS approach applying example-guided synthesis to leakage contracts expanding the space of examples to pairs of traces instead of single examples.

Automatic Reasoning about Side Channels: Various frameworks automate side-channel analysis, including the fuzzers used in this work, REVIZOR [23, 24] and SCAM-V [15, 25]. Other tools include Osiris [56] (timing-based fuzzing using ISA specs), CheckMate [57] (automated exploit/test generation for hardware), and Xenon [58] (formal verification of constant-time hardware). There are also many other approaches that leverage symbolic execution, fuzzing, or formal analysis for side-channel vulnerability detection [45, 56, 59]–[63].

IX. CONCLUSION

A precise ISA-level leakage contract is critical for secure programming. We introduced a method to extract such contracts from black-box CPUs and validated it with MALCOS on `x86` and `ARM` CPUs. Our results show that MALCOS can synthesize precise and boundedly sound contracts that capture all exercised instruction-level microarchitectural leaks. Furthermore, the results show that our *template-free* approach eliminates the inflexibility of conventional methods, leading to more precise contracts than *template-based* approaches.

References

[1] D. A. Osvik, A. Shamir, and E. 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*, ser. CT-RSA'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 1–20. [Online]. Available: https://doi.org/10.1007/11605805_1

[2] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1967–1984. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[3] Y. Yarom and K. Falkner, "Flush+Reload: a high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014, pp. 719–732.

[4] C. Percival, "Cache missing for fun and profit," in *In Proc. of BSDCan 2005*, 2005.

[5] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[6] O. Acıiçmez and Ç. K. Koç, "Trace-driven Cache Attacks on AES (Short Paper)," in *Proceedings of the 8th International Conference on Information and Communications Security*, ser. ICICS. Springer-Verlag, 2006, pp. 112–121.

[7] M. Neve and J.-P. Seifert, "Advances on access-driven cache attacks on AES," in *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, ser. SAC'06. Springer-Verlag, 2007, pp. 147–162.

[8] Y. Tsunoo, T. Saito, T. Suzaki, and M. Shigeri, "Cryptanalysis of DES implemented on computers with cache," in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES'03, LNCS. Springer, 2003, pp. 62–76.

[9] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 490–505.

[10] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, ser. Lecture Notes in Computer Science, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. Springer, 2006, pp. 207–228. [Online]. Available: https://doi.org/10.1007/11745853_14

[11] E. Atoofian and A. Baniasadi, "Improving energy-efficiency by by-passing trivial computations," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 7–pp.

[12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[13] 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

[14] 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.

[15] 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

[16] J. Hofmann, E. Vannacci, C. Fournet, B. Köpf, and O. Oleksenko, "Speculation at fault: Modeling and testing microarchitectural leakage of CPU exceptions," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7143–7160.

[17] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Synthesis of sound and precise leakage contracts for open-source risc-v processors," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 111–125.

[18] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2mμpath: Multi-μpath synthesis with applications to hardware security verification," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 507–524.

[19] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-houdini: Scalable invariant learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 603–618.

[20] G. Mohr, M. Guarnieri, and J. Reineke, "Synthesizing hardware-software leakage contracts for RISC-V open-source processors," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2024, Valencia, Spain, March 25-27, 2024*. IEEE, 2024, pp. 1–6. [Online]. Available: https://doi.org/10.23919/DATE58400.2024.10546681

[21] S. Dinesh, M. Parthasarathy, and C. W. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 3735–3753. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00180

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] 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.

[27] 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.

[28] G. Barthe, M. Böhme, S. Cauligi, C. Chuengsatiansup, D. Genkin, M. Guarnieri, D. M. Romero, P. Schwabe, D. Wu, and Y. Yarom, "Testing side-channel security of cryptographic implementations against future microarchitectures," in *Proceedings of the 31st ACM Conference on Computer and Communications Security*, ser. CCS 2024. ACM, 2024.

[29] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[30] X. Fabian, M. Guarnieri, and M. Patrignani, "Automatic detection of speculative execution combinations," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 965–978. [Online]. Available: https://doi.org/10.1145/3548606.3560555

[31] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[32] 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.

[33] T. Downs, "Intel's "zero-optimization" of performance counters," https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html, May 2020, accessed: 2025-04-05. [Online]. Available: https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html

[34] Intel, "Speculative store bypass / cve-2018-3639 / intel-sa-00115," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html.

[35] Microsoft, "Configuration file," 2024, accessed: 2025-06-16. [Online]. Available: https://microsoft.github.io/sca-fuzzer/user/config/#executor-configuration

[36] R. L. Schröder, S. Gast, and Q. Guo, "Divide and surrender: Exploiting variable division instruction timing in {HQC} key recovery attacks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6669–6686.

[37] "uops.info – latency, throughput, and port usage information for instructions on recent x86 microarchitectures." [Online]. Available: https://uops.info/table.html

[38] "Ibex: An embedded 32 bit RISC-V CPU core." [Online]. Available: https://github.com/lowRISC/ibex

[39] "Ibex RISC-V core – multiplier/divider block." [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/instruction_decode_execute.html#mult-div

[40] R. Bloem, B. Gigerl, M. Gourjon, V. Hadzic, S. Mangard, and R. Primas, "Power contracts: Provably complete power leakage models for processors," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 381–395. [Online]. Available: https://doi.org/10.1145/3548606.3560600

[41] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 666–680. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833707

[42] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *Proceedings of the 28th ACM Conference on Computer and Communications Security*, ser. CCS 2021. ACM, 2021.

[43] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 913–926.

[44] R. Guanciale, M. Balliu, and M. Dam, "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1853–1869. [Online]. Available: https://doi.org/10.1145/3372297.3417246

[45] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–30, 2021.

[46] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[47] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS 2021-Network and Distributed Systems Security*, 2021.

[48] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic hardware-software contracts for security," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 72–86.

[49] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely, "The code that never ran: Modeling attacks on speculative evaluation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1238–1255.

[50] H. P. de León and J. Kinder, "Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 235–248. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833774

[51] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 1–8.

[52] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: automatically learning the x86-64 instruction set," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 237–250.

[53] J. Liu, C. Murphy, A. Grover, K. J. Johnson, T. Reps, and L. D'Antoni, "Synthesizing formal semantics from executable interpreters," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 362–388, 2024.

[54] C. Norman, A. Godbole, and Y. A. Manerkar, "Pipesynth: Automated synthesis of microarchitectural axioms for memory consistency," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 513–527.

[55] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "Synthct: Towards portable constant-time code," in *NDSS*, 2022.

[56] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow, "Osiris: Automated discovery of microarchitectural side channels," *CoRR*, vol. abs/2106.03470, 2021. [Online]. Available: https://arxiv.org/abs/2106.03470

[57] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 947–960. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00081

[58] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "Solver-aided constant-time hardware verification," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 429–444.

[59] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 1427–1444. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

[60] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[61] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–31, 2020.

[62] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural modeling for security evaluation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[63] 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.

APPENDIX A
D. USE OF GENERATIVE AI AND LLMS

This paper has been refined for grammar, spelling, and minor style improvements with Grammarly.

APPENDIX B
X86 INSTRUCTION SUBSETS USED IN §VI-E1

The synthesis campaign in §VI-E1 targets 13 subsets of the x86-64 ISA. These subsets have been derived from a prior Revizor testing campaign [24]. Altogether, these subsets result in a complete base x86-64 (user-level) instruction set, excluding those instructions that are not (yet) compatible with REVIZOR. The subsets do not cover: (a) system instructions, (b) instruction incorrectly emulated by Unicorn, (c) control-flow instructions not supported by Revizor (e.g., indirect jumps), and (d) ISA extensions such as AVX or x87 that require more complex test case generation algorithms to avoid failures, which are also not compatible with REVIZOR.

Each of the tested subsets consisted of several basic arithmetic instructions (including their versions with memory operands) and of several instructions that are unique to the given subset. The exact instructions are as follows:

- `cond` (conditional branches): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `J*`, `LOOP*`, `JMP` (unconditional direct jump).
- `strn` (string operations): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `CLC`, `CLD`, `CMC`, `LAHF`, `LOCK`, `REPE`, `REPNE`, `SAHF`, `SCASB`, `SCASD`, `SCASW`, `STC`, `STD`.
- `dmul` (division and multiplication): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `DIV`, `IMUL`, `MUL`.
- `flag` (operations on flags): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `CLC`, `CLD`, `CMC`, `LAHF`, `SAHF`, `STC`, `STD`.
- `lock` (atomics with `LOCK` prefix): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `LOCK ADC`, `LOCK ADD`, `LOCK CMP`, `LOCK DEC`, `LOCK INC`, `LOCK NEG`, `LOCK SBB`, `LOCK SUB`, `LOCK BSF`, `LOCK BSR`, `LOCK BT`, `LOCK BTC`, `LOCK BTR`, `LOCK BTS`, `LOCK NOT`, `LOCK OR`, `LOCK TEST`, `LOCK XOR`.
- `atom` (atomics without `LOCK` prefix): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `CMPXCHG`, `XADD`, `LOCK CMPXCHG`, `LOCK XADD`.
- `dxfr` (data transfer): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `BSWAP`, `MOV`, `MOVSX`, `MOVZX`, `XCHG`.
- `setc` (conditional byte set): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `SET*`.
- `nop` (`NOP` instructions): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `NOP`.
- `logi` (logical operations): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `AND`, `NOT`, `OR`, `TEST`, `XOR`.
- `conv` (data type conversion): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `CBW`, `CDQ`, `CWD`, `CWDE`.
- `cmov` (conditional moves): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `CMOV*`.
- `bit` (bit test and bit scan): `ADC`, `ADD`, `CMP`, `DEC`, `INC`, `NEG`, `SBB`, `SUB`, `BSF`, `BSR`, `BT`, `BTC`, `BTR`, `BTS`.

APPENDIX C
ARM INSTRUCTION SET USED IN §VI-E2

The synthesis campaign from §VI-E2 targets a subset of the memory instructions in the ARM ISA. These instructions cover various addressing modes and data sizes, including standard, unprivileged, and unaligned variants. The selected subset ensures representative coverage of typical memory access patterns relevant to our analysis. Next, we list all memory instructions covered by our campaign:

- `LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`, `LDRSW`
  Load instructions for various data sizes (byte, halfword, signed byte/halfword, word).
- `STR`, `STRB`, `STRH`
  Store instructions for byte, halfword, and word sizes.
- `LDUR`, `LDURB`, `LDURH`, `LDURSB`, `LDURSH`, `LDURSW`
  Load instructions with unscaled immediate offset addressing.
- `STUR`, `STURB`, `STURH`
  Store variants corresponding to `LDUR` instructions.
- `LDP`, `LDPSW`, `STP`
  Load/store pair instructions for loading/storing two registers simultaneously.
- `LDTR`, `LDTRB`, `LDTRH`, `LDTRSB`, `LDTRSH`, `LDTRSW`
  Load instructions with unprivileged access supporting various data sizes and signed variants.
- `STTR`, `STTRB`, `STTRH`
  Store instruction variants with unprivileged access.

APPENDIX D
MALCOS EVALUATION OF PARAMETERS

In this section, we evaluated the impact of different parameters of MALCOS using MALCOS-X86. In particular, we evaluated the impact of: number of positive examples (§D-A), contract resetting parameter R (§D-B), synthesis depth (§D-C), and minimization options (§D-D).

A. Impact of Positive Examples

To address leakage over-approximations, MALCOS uses positive examples to guide the `Refiner` about which executions should not be distinguished by the synthesized clause.

Experimental setup: For each target ground-truth contract, we randomly generate 100 programs and 100 random states per program, and we use MALCOS-X86 to synthesize the corresponding contracts for different numbers of positive examples, ranging from 0 (no positive examples) to 150. Afterwards, we measure the precision of the synthesized contracts and the average execution time of clause refinement step (i.e., the average execution time of a single call to the `Refiner`).

Results: Figure 3 shows the impact of the number of positive examples on the precision of the synthesized contract (plot on the left) and on the clause refinement time (plot on the right). Note that both plots report the average metrics computed across all 100 programs. We highlight that increasing the number of positive examples increases both the precision of the synthesized contracts as well as the refinement time (since more positive examples result in more complex synthesis constraints). We remark, however, that the precision starts to plateau at 100 positive examples, whereas the refinement time keeps increasing. For this reason, we decided to use 100 positive examples in the evaluation in §VI as it is the best trade-off between achieved precision and refinement time.

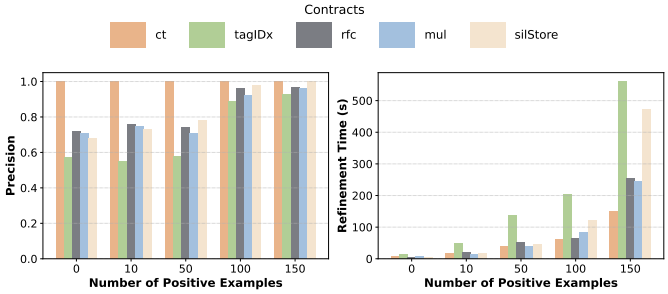


Fig. 3: MALCOS-X86: Impact of positive examples on precision and refinement time

MALCOS-X86							
Contract	N. Programs Before Restart	N. Clauses Before Unification	N. Clauses After Unification	N. Clauses After Minim.	avg. Contract Minim. Time	avg. Total Refinement Time	avg. P/S
ct	100	2826.8	1093.9	6	15h 34min 45s	26h 36min 47s	1.0/1.0
	500	365.8	288.2	6	54min 17s	5h 40min 47s	1.0/1.0
	1000	170.7	154.8	6	14min 40s	3h 4min 44s	1.0/1.0
	5000	87.3	35.8	5	1min 20s	27min 25s	1.0/1.0
	10000	19.6	19.4	4.2	30s	13min 58s	0.99/1.0
TagIdx	100	3005.4	1285.7	8.4	6h 38min 51s	54h 34min 35s	0.96/1.0
	500	407.3	388.9	14.4	1h 18min 46s	12h 49min 47s	0.79/1.0
	1000	243.9	240	17.1	14min 18s	7h 14min 26s	0.35/1.0
	5000	156.2	57	11.3	1min 9s	1h 47min 25s	0.10/0.99
	10000	30.3	30.3	7.3	31s	56min 22s	0.10/0.99
RFC	100	3117.3	1126.7	9.8	14h 11min 14s	24h 16min 53s	1.0/1.0
	500	496.3	368.8	9.8	1h 5min 2s	6h 7min 23s	1.0/1.0
	1000	245.2	209.1	11	21min 4s	3h 28min 17	0.88/1.0
	5000	102	56.1	11.5	2min 28s	36min 58s	0.59/1.0
	10000	31.1	31.1	7.2	51s	18min 59s	0.48/1.0
mul	100	3788.1	1422.4	16	19h 55min 59s	57h 38min 43s	1.0/1.0
	500	485.4	255.5	15.7	3h 17min 4s	13h 29min 35s	1.0/1.0
	1000	319.3	285	16.6	44min 3s	7h 28min 29s	0.99/1.0
	5000	122.9	73.8	13.2	3min 44s	1h 45min 19s	0.41/1.0
	10000	40.9	40.9	10.4	1min 17s	1h 3min 4s	0.41/1.0
SilStore	100	304.4	105.6	1	14min 27s	3h 48min 40s	1.0/1.0
	500	47.2	32.2	1	2min 3s	1h 13min 48s	1.0/1.0
	1000	22.3	18.3	1.5	38s	33min 37s	0.83/1.0
	5000	12.2	3.9	1.7	7s	4min 21s	0.49/1.0
	10000	2.7	2.2	2.2	3s	1min 45s	0.43/1.0

TABLE V: The table shows the impact on the final contract size (in clauses), minimization time, total refinement time, and average precision and soundness metrics given different values of the resetting parameter R (i.e., after how many tested programs we reset the contract candidate to $\emptyset$).

B. Impact of Contract Resetting

Next, we study the impact of the contract resetting parameter R (which indicates after how many programs is the candidate contract reset as indicated in §IV-B) on the contract synthesized by MALCOS-X86.

Experimental setup: We use MALCOS-X86 to learn contracts for all the ground truth contracts as targets. For each ground-truth contract, we use 10000 programs with 100 states each, and 100 positive examples. We ran MALCOS-X86 with different values for the resetting parameter R ranging from 100 (so, resetting after every 100 programs) to 10000 (so, no resets during synthesis). We repeat the experiment outlined above 10 times, with different initial randomness seeds. Each time, we measure the number of clauses before and after post-processing, the total execution time for the contract minimization and clause refinement (i.e., the sum of all time spent in the `Refiner`), and the average precision and soundness of the synthesized contracts against a validation set of 50000 programs with 100 inputs each.

Results: Table V reports the results of this experiment. We highlight the following insights:

(1) A lower value of R (i.e., more frequent restarts from an empty candidate contract) result in more precise and sound contracts. Intuitively, this happens because restarts allow MALCOS-X86 to "recover" from over-approximated clauses synthesized in previous steps.

(2) More frequent restarts result in a higher execution time. This is due to the fact that synthesis from an empty contract often requires exploring more counter-examples (and, therefore, performing more calls to the synthesizer), thereby increasing MALCOS-X86's execution time.

In the evaluation in §VI, we decided to set R to 500 as it offers a good trade-off between total execution time and contract precision.

C. Impact of Synthesis Depth

The synthesis depth limits the complexity of the ICL clauses explored by the synthesizer, where synthesis depth *n* indicates that the solver will consider only expressions whose syntax tree has depth *n*. A small depth value restricts the ability of the solver to find complex solutions by limiting the search space to "simple" expressions. Conversely, an excessive depth exponentially increases the search space and the synthesis time.

Experimental setup: For each target ground-truth contract, we randomly generate 100 programs and 100 random states per program, and we use MALCOS-X86 to synthesize the corresponding contracts for different synthesis depth values, ranging from 1 to 4. Afterwards, we measure the precision of the synthesized contracts and the average execution time of clause refinement step (i.e., the average execution time of a single call to the `Refiner`).

Results: Figure 4 shows the impact of synthesis depth on the (average) precision of the synthesized contract (plot on the left) and on the (average) clause refinement time (plot on the right). Note that both plots report the average metrics computed across all 100 programs. We highlight that increasing the synthesis depth improves both the precision of synthesized contracts and the refinement time. However, precision starts to plateau at depth 3, whereas refinement time keeps increasing (due to a larger search space). In particular, for synthesis depth 4, the solver reaches a time-out (set at 600 seconds) in 89 of the 100 cases of the **TagIdx** contract. For this reason, we decided to use synthesis depth 3 in the evaluation in §VI as it is the best trade-off between precision and refinement time.

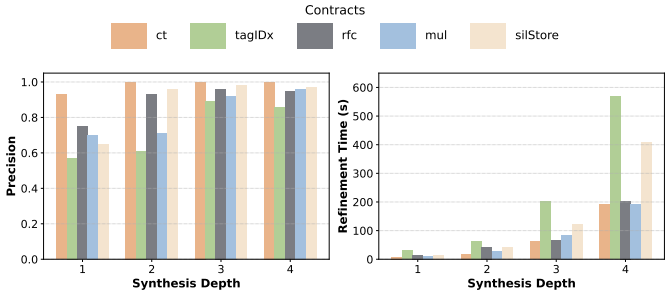


Fig. 4: MALCOS-X86: Impact of synthesis depth on precision and refinement time

D. Impact of Minimization

Next, we study the impact of the three kinds of minimization optimizations used by MALCOS-X86—program/data minimization (from Revizor, §V-A) and contract minimization (§V-C).

Experimental setup: We use MALCOS-X86 to learn a contract for all the ground truth contracts as target and using 10000 programs with 100 states each and resetting after 500 programs. We ran MALCOS-X86 using four minimization configurations:

(1) *All Minimization*: all minimization optimizations are enabled;
(2) *Ctr. Minimization*: only contract minimization is enabled;
(3) *TC Minimization*: only program/data minimization is enabled;
(4) *No Minimization*: no minimization optimizations are enabled.

We repeat all the experiments outlined above 10 times, with different initial randomness seeds. For each configuration, we measured the total execution time of MALCOS-X86 as well as the average precision and soundness of the synthesized contracts against a validation set of 50000 programs with 100 inputs each.

Results: Table VI reports our experiment results. We highlight the following findings:

1) Enabling all minimization optimizations increases precision by an average of 70% with respect to the *No Minimization* cases.
2) The execution time is mostly affected by the use of program/data minimization. Not using program/data minimization results in more complex constraints (and a larger search space) for the solver, which again results in longer execution times (see, for instance, how configurations *All Minimization* and *Prg. Minimization* are faster than configurations *Ctr. Minimization* and *No Minimization*).
3) Contract minimization is the optimization that contributes the most to improving precision, as it eliminates overapproximations in the final contracts.

Based on these results, in the evaluation in §VI we always enable all three optimizations.

	MALCOS-X86							
Contract	All Minimization		Ctr. Minimization		TC Minimization		No Minimization	
	avg. P/S	avg. Time	avg. P/S	avg. Time	avg. P/S	avg. Time	avg. P/S	avg. Time
ct	1.0/1.0	11h 57min	1.0/1.0	29h 17min	0.70/1.0	10h 31min	0.68/1.0	25h 30min
TagIdx	0.79/1.0	18h 57min	0.30/1.0	33h 48min	0.14/1.0	14h 14min	0.06/1.0	33h 7min
RFC	1.0/1.0	10h 32min	0.92/1.0	28h 11min	0.30/1.0	8h 40min	0.27/1.0	24h 41min
mul	1.0/1.0	24h 21min	0.91/1.0	51h 5min	0.20/1.0	14h 58min	0.19/1.0	47h 40min
SilStore	1.0/1.0	1h 23min	1.0/1.0	10h 15min	0.13/1.0	1h 19min	0.07/1.0	10h 7min

TABLE VI: Precision (P), Soundness (S) and average of total time for different minimization configurations on contracts.