Synthesizing Hardware-Software Leakage Contracts for RISC-V Open-Source Processors

Abstract—**Microarchitectural attacks compromise security by exploiting software-visible artifacts of microarchitectural optimizations such as caches and speculative execution. Defending against such attacks at the software level requires an appropriate abstraction at the instruction set architecture (ISA) level that captures microarchitectural leakage. Hardware-software leakage contracts have recently been proposed as such an abstraction.**

In this paper, we propose a semi-automatic methodology for synthesizing hardware-software leakage contracts for open-source microarchitectures. For a given ISA, it relies on human experts to (a) capture the space of possible contracts in the form of contract templates and (b) to devise a test-case generation strategy to explore a microarchitecture's potential leakage. For a given implementation of an ISA, these two ingredients are then used to automatically synthesize the most precise leakage contract that is satisfied by the microarchitecture.

We have instantiated this methodology for the RISC-V instruction set and applied it to the Ibex and the CVA6, two open-source processors. Our experiments demonstrate the practical applicability of the methodology and uncover subtle and unexpected sources of microarchitectural leakage.

I. Introduction

Microarchitectural attacks [13], [22], [23], [33], [38] compromise the security of programs by exploiting software-visible artifacts of microarchitectural optimizations such as caches and speculative execution. Defending against such attacks at the software level is challenging: instruction set architectures (ISAs), the traditional hardware-software interface, abstract from microarchitectural details and thus do not give any guarantees w.r.t. microarchitectural attacks.

Hardware-software leakage contracts [20], [36] (short: leakage contracts) have recently been proposed as a new security abstraction at the ISA level to fill this gap. Such contracts aim to capture possible microarchitectural side-channel leaks by associating leakage traces, i.e., sequences of leakage observations, with ISA-level executions. For example, a contract could expose the addresses of load and store instructions as leakage observations to capture data cache leaks. Similarly, a contract could expose the operands of division instructions to capture leakage via operand-dependent latencies. Given a contract that faithfully captures the microarchitectural leakage of a processor, it is then possible to program the hardware securely by making sure that all leakage observations are independent of secrets.

However, most of today's processor designs lack any kind of formal specification of their microarchitectural leakage. In this work, we tackle this gap by proposing a semi-automatic methodology for synthesizing leakage contracts from open-source processor designs.

Our methodology consists of four steps:

1) **Definition of Contract Template**. A human expert determines a set of *contract atoms* that capture potential instruction-level leakage observations. For example, a contract atom may expose the value of the register operand of a memory instruction. The set of all contract atoms forms the *contract template*. Any set of contract atoms from the contract template is a candidate *contract*.

2) **Test-Case Generation**. A human expert devises a test-case generation strategy. Each *test case* consists of two ISA-level programs with fixed data inputs. These test cases are used to exercise and analyze a processor's microarchitectural leakage.

3) **Evaluation of Test Cases**. Test cases are automatically evaluated on the target processor design to determine which test cases are *distinguishable*, *i.e.*, lead to distinguishable microarchitectural executions on the processor. Distinguishable test cases expose actual leaks that need to be accounted for at contract level. Additionally, test cases are also automatically evaluated on the contract template to derive the set of *distinguishing atoms*, i.e., those atoms that distinguish the two programs. For instance, an atom exposing the value of register operands for memory instructions will distinguish programs accessing different addresses.

4) **Automatic Contract Synthesis**. Based on the results of the test-case evaluation, a contract, that is, a set of contract atoms, is automatically synthesized from the distinguishing atoms associated with attacker-distinguishable test case. Our approach ensures that the synthesized contract is *satisfied* by the processor design on all test cases, *i.e.*, it captures all leaks exposed by the test cases on the processor. Moreover, it also ensures that the synthesized contract is the *most precise* such contract, *i.e.*, it distinguishes the fewest attacker-indistinguishable test cases.

The proliferation of the open-source RISC-V ISA [10] and its growing ecosystem is promising in this context for two reasons: (1) The simplicity and modularity of the ISA provides a good foundation for the definition of contract atoms. (2) The growing body of open-source RISC-V processor designs of varying complexity provides natural targets for contract synthesis. Leveraging the RISC-V Formal Interface [5], we have instantiated our contract-synthesis methodology for the RISC-V ISA and applied it two open-source processor designs: Ibex [3] and CVA6 [1]. Our experiments reveal subtle previously undocumented cases of microarchitectural leakage and demonstrate the practical applicability of our methodology.

To summarize, our main contributions are:

- A methodology for synthesizing hardware-software leakage contracts from open-source processor designs.
- The definition of a concrete contract template for the RISC-V ISA and a corresponding test-case generation strategy.
- The implementation of a contract synthesis toolchain and its application to two open-source RISC-V processor designs.

II. Preliminaries

A. Instruction Set Architectures and RISC-V

Instruction-set architectures (ISAs) define the software-visible interface of a processor. For instance, they define the architectural state, the set of supported instructions, and how these instructions modify the architectural state. We model an ISA as a state machine capturing the execution of programs one instruction at a time. Formally, an ISA is a function $\mathrm{ISA} : \textsc{Arch} \rightarrow \textsc{Arch}$ that maps each architectural state $\sigma \in \textsc{Arch}$ to its successor $\mathrm{ISA}(\sigma)$, obtained by executing the instruction to be executed in σ. To capture an entire execution, we denote by $\mathrm{ISA}^*(\sigma)$ the sequence of states reached from σ by successive application of ISA, i.e., $\mathrm{ISA}^*(\sigma) := [\sigma_0, \sigma_1, \sigma_2, \ldots]$ with $\sigma_0 = \sigma$ and $\sigma_{i+1} = \mathrm{ISA}(\sigma_i)$ for all $i \geq 0$.

B. Microarchitectures

Microarchitectures are concrete implementations of instruction set architectures in processors, which often contain complex performance-enhancing optimizations. They operate at cycle granularity rather than at instruction granularity. Thus, we model them as state machines that define how the processor's state evolves cycle-by-cycle. Formally, a microarchitecture is a function $\textsc{Impl} : \textsc{ImplState} \rightarrow \textsc{ImplState}$ on the set of microarchitectural states $\textsc{ImplState} = \textsc{Arch} \times \mu\textsc{Arch}$ that captures how the microarchitectural state evolves from one cycle to the next. Each microarchitectural state $\sigma \in \textsc{ImplState}$ consists of an architectural part $\sigma_{\mathrm{ISA}} \in \textsc{Arch}$ and a microarchitectural part $\sigma_{\textsc{Impl}} \in \mu\textsc{Arch}$, which models the state of microarchitectural components like caches and branch predictors. Similarly to $\mathrm{ISA}^*(\sigma)$, $\textsc{Impl}^*(\sigma)$ is the sequence of microarchitectural states reached from σ by successive applications of $\textsc{Impl}$.

C. Microarchitectural Attackers

Programs that operate on secret data may leave traces of this secret data in the microarchitectural state. Microarchitectural attacks extract information about secrets from the microarchitectural state by leveraging software-visible side-effects [13], [22], [23], [33], [38], mostly affecting a program's execution time. In this paper, we consider (passive) microarchitectural attackers that can observe and extract information from the microarchitectural state. Formally, we model a microarchitectural attacker as a function $\mu\textsc{Atk} : \textsc{ImplState} \rightarrow \textsc{AtkObs}$ that maps microarchitectural states to attacker observations. Common attacker models studied in the literature, like the one exposing the timing of instruction retirement [32] or the one exposing the final state of caches [15], [38], can be instantiated in our setting. Given an attacker model, two executions $\textsc{Impl}^*(\sigma)$ and $\textsc{Impl}^*(\sigma')$ are *attacker distinguishable* if $\mu\textsc{Atk}(\textsc{Impl}^*(\sigma)) \neq \mu\textsc{Atk}(\textsc{Impl}^*(\sigma'))$, where we lift $\mu\textsc{Atk}$ to sequences by applying it to each element.

D. Hardware-Software Leakage Contracts

The goal of leakage contracts [20] it to capture microarchitectural side-channel leakage at the ISA level to allow reasoning about side-channel security of programs without having to explicitly consider their microarchitectural execution. To this end, contracts associate leakage traces, i.e., sequences of leakage observations, with ISA-level executions.

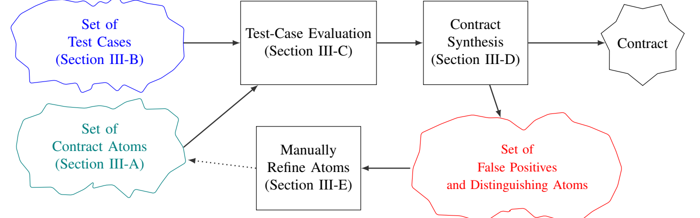

Fig. 1. High-level steps of our contract-synthesis methodology.

A *contract* is a function $\textsc{Ctr} : \textsc{Arch} \rightarrow \textsc{CtrObs}$ that maps architectural states to contract observations. As an example, a contract could expose the addresses of load and store instructions to capture possible leakage via the data cache. Two architectural states σ and σ' are *contract distinguishable* if $\textsc{Ctr}(\mathrm{ISA}^*(\sigma)) \neq \textsc{Ctr}(\mathrm{ISA}^*(\sigma'))$, where we lift $\textsc{Ctr}$ to sequences by applying it to each element of the sequence.

Microarchitecture $\textsc{Impl}$ *satisfies* contract $\textsc{Ctr}$ for instruction-set architecture ISA under attacker model $\mu\textsc{Atk}$ if all contract-indistinguishable executions are also attacker-indistinguishable:

$$\begin{aligned}\forall \sigma, \sigma' \in \textsc{ImplState} : \sigma_{\textsc{Impl}} = \sigma'_{\textsc{Impl}} \implies \\ \textsc{Ctr}(\mathrm{ISA}^*(\sigma_{\mathrm{ISA}})) = \textsc{Ctr}(\mathrm{ISA}^*(\sigma'_{\mathrm{ISA}})) \implies \\ \mu\textsc{Atk}(\textsc{Impl}^*(\sigma)) = \mu\textsc{Atk}(\textsc{Impl}^*(\sigma'))\end{aligned}$$

Note that we require the microarchitectural parts of σ and σ' to initially be the same. Otherwise, the executions could be attacker-distinguishable due to initial differences rather than due to leakage during the execution.

The benefit of contract satisfaction is that it allows reasoning about side-channel security directly at the ISA level: A program that does not leak any secret information according to the contract is also guaranteed not to leak any secret information to attackers on *any* microarchitecture that satisfies the contract [20].

III. Contract-Synthesis Methodology

Our goal is to synthesize contracts directly from a processor's register-transfer level (RTL) design. Figure 1 illustrates the high-level steps of our contract-synthesis methodology outlined in §I. In the following, we describe each step in more detail.

A. Contract Templates

The basic building blocks of contracts are *contract atoms*. Contract atoms capture potential leakage observations at the instruction level. Formally, a contract atom A is a pair (π_A, ϕ_A) where $\pi_A : \textsc{Arch} \rightarrow \mathbb{B}$ determines whether the contract observation is applicable in a particular architectural state, and $\phi_A : \textsc{Arch} \rightarrow O_A$ is the observation function that maps an architectural state to the atom's observation. For example, a contract atom A could expose the divisor operand of a division instruction: π_A would hold whenever the instruction to execute is a division, and ϕ_A would return the value of the divisor.

A *contract template* T is a set of contract atoms. A subset $S \subseteq T$ of the template defines a candidate *contract* $\textsc{Ctr}_S$ as follows: $\textsc{Ctr}_S(\sigma) = \{\phi_A(\sigma) \mid A \in S \wedge \pi_A(\sigma)\}$, *i.e.*, it reveals the observations of all atoms that are applicable in a given state.

B. Test-Case Generation

Our goal is to find a contract that is satisfied by the microarchitecture while being as precise as possible, *i.e.* to find a contract that distinguishes as few executions as possible. Achieving this goal requires (1) to determine which contracts from the template are satisfied by the microarchitecture, and (2) to measure the precision of such contracts.

Ideally, one would like to formally verify contract satisfaction. However, contract verification techniques [36] do not yet scale to processors of the complexity of e.g. the CVA6. Thus, we resort to a testing-based approach and evaluate contract satisfaction on systematically-generated test cases. Formally, a test case is a pair (σ, σ') of microarchitectural states such that $\sigma_{\text{IMPL}} = \sigma'_{\text{IMPL}}$. A test case is *attacker distinguishable* if $\mu\text{ATK}(\text{IMPL}^*(\sigma)) \neq \mu\text{ATK}(\text{IMPL}^*(\sigma'))$. A test case is *atom distinguishable* for an atom A if $\text{CTR}_{\{A\}}(\text{ISA}^*(\sigma_{\text{ISA}})) \neq \text{CTR}_{\{A\}}(\text{ISA}^*(\sigma'_{\text{ISA}}))$. By construction, a test case is contract distinguishable for contract CTR_S if it is atom distinguishable for at least one atom in S. Thus, checking atom distinguishability for all atoms is sufficient to characterize contract distinguishability for the entire contract template.

Test cases also allow to measure the precision of a contract. For this, we adopt the standard notion of precision used in the evaluation of binary classifiers: $\textit{Precision} = \frac{TP}{TP+FP}$, where TP is the number of true positives, i.e., test cases that are contract distinguishable *and* attacker distinguishable, and FP is the number of false positives, i.e., test cases that are contract distinguishable *but* not attacker distinguishable. Higher precision contracts are desirable as they rule out fewer programs at the contract level that could actually be executed securely on the processor.

Our methodology requires a human expert to devise a test-case generation strategy used to generate the set of test cases *TC*.

C. Test-Case Evaluation

Given a set of test cases *TC*, we need to determine for each test case (1) whether it is attacker distinguishable, and (2) which contract atoms distinguish it. Attacker distinguishability can be determined via simulation of the microarchitecture with a suitable attacker model. Similarly, atom distinguishability can be determined by evaluating all contract atoms in parallel on top of a simulation of the instruction set architecture.

The test-case evaluation phase has two outputs: (1) The set of attacker-distinguishable test cases $\textit{Dist} \subseteq \textit{TC}$. (2) For each test case t, the set of distinguishing atoms $\textit{distinguishing}(t) \subseteq T$.

D. Contract Synthesis

We now show how to use integer linear programming (ILP) to synthesize a contract from the template that distinguishes all attacker-distinguishable test cases *TC* and maximizes precision.

The ILP uses a boolean variable s_A for each atom A in the contract template *T* to encode whether the atom is selected to be part of the synthesized contract or not. Further, the ILP uses a boolean variable c_t for each attacker-indistinguishable test case $t \in \textit{Indist} = \textit{TC} \setminus \textit{Dist}$, which, using constraints detailed below, is forced to be 1 for test cases that are contract distinguishable and thus are false positives.

Maximizing precision is equivalent to minimizing the number of false positives, as the number of true positives is the same for all correct contracts. Thus, the objective function of the ILP is $\min \sum_{t \in \textit{Indist}} c_t$. To ensure that only correct contracts are considered, we introduce the following constraint for each attacker-distinguishable test case $t \in \textit{Dist}$: $\sum_{A \in \textit{distinguishing}(t)} s_A \geq 1$, *i.e.*, at least one atom that distinguishes t must be selected for the contract. For each attacker-indistinguishable test case $t \in \textit{Indist}$ and each contract atom $A \in \textit{distinguishing}(t)$ we further introduce the constraint: $s_A \leq c_t$. This ensures that c_t can only be 0 for test cases that are not contract distinguishable.

From the solution to the ILP we extract the synthesized contract via the variables s_A and the false-positive test cases via the variables c_t.

E. Refinement of the Contract Template

In addition to returning a contract that maximizes precision, our implementation also returns a ranking of the contract atoms according to the number of false positives caused by their inclusion in the contract and the corresponding test cases. Inspecting these test cases allows a human expert to identify contract atoms that should be refined to obtain a more precise contract.

IV. Instantiating the Methodology for RISC-V

We instantiated the above methodology for the RISC-V instruction set, more precisely its `I` and `M` subsets.

A. RISC-V Contract Template

Instructions of the same type usually show similar leakage behavior, e.g., if one division operation leaks the divisor, then other division operations likely also leak the divisor. Based on this intuition, we identified several potential leakage sources and added a contract atom for each instruction type and each applicable leakage source to the contract template. For such an atom, π_A determines whether the current instruction has the given type and ϕ_A extracts the corresponding leakage from the architectural state. For example, if the division leaks the divisor, which is stored in the register `RS2`, the contract atom would be (`DIV`, `REG_RS2`) meaning that for every division, the value of register `RS2` is leaked.

We first defined a base template capturing the architectural state that directly influences the execution of an instruction:

- **Instruction leakages (`IL`)** expose values from an instruction's encoding: The operation `OP`, the destination and source registers `RD`, `RS1`, and `RS2`, and the immediate value `IMM`.
- **Register leakages (`RL`)** expose the values of registers: The values of the source registers `REG_RS1` and `REG_RS2` before execution and the final value of the destination register `REG_RD`.
- **Memory leakages (`ML`)** expose the memory addresses and memory contents accessed by an instruction: `MEM_R_ADDR` and `MEM_W_ADDR` expose the accessed addresses, whereas `MEM_R_DATA` and `MEM_W_DATA` expose the accessed content.

During evaluation, we noticed that the above categories are sufficient but that the precision of the contract can be improved by adding the following contract atoms:

- **Alignment leakages (`AL`)** expose the alignment of a memory access: `IS_WORD_ALIGNED` exposes whether the last two

bits of the memory address are 00 and IS_HALF_ALIGNED exposes whether the last two bits of the address are not 11.

- **Branch leakages (BL)** expose whether a branch is taken or not: This can be BRANCH_TAKEN for taken branches and NEW_PC to indicate the target of the branch. The latter one also applies to unconditional jumps.
- **Data-dependency leakages (DL)** expose data dependencies in the program: RAW_RS1_n, RAW_RS2_n, RAW_RD_n, and WAW_n, respectively, indicate Read-After-Write dependencies on register RS1, RS2, and RD and Write-After-Write dependencies within a distance of n instructions.

We remark that not all atoms are applicable to every instruction type, e.g. some instructions do not have an immediate value, which cannot be leaked. With a maximum distance of $n = 4$, the above template results in a total of 762 atoms.

B. Test-Case Generation

The set of test cases used for the simulation has a large effect on the synthesized contract, as the synthesis algorithm cannot know about leakages that are not exposed by the set of test cases. Thus, ideally the test-case generation algorithm should consider potential microarchitectural implementations and their potential leakage. Additionally, in order to obtain a concise contract from the given contract template, the set of contract atoms should be considered when generating the test cases.

The test-case generation method we use is simple but has shown effective on the tested targets. Note that we only generate the architectural state and fix the initial microarchitectural part of the microarchitectural state to allow reusing test cases for different microarchitectures. Each test case consists of two programs which aim to be differentiable by one specific contract atom.

Each program consists of three parts, of which only the second part differs among the two programs in the test case: (1) First, we initialize every architectural register to a randomly selected value. (2) Next, we try to trigger the leakage of the contract atom we want to test for. For this, we generate a random instance of the given instruction type. We then derive two sequences of instructions that could be differentiated by the given atom, e.g., if we want to test whether the immediate value leaks, we alter the immediate value or if we want to test whether the memory contents that are read leak, we insert an earlier instruction in both programs writing different values to the same address. (3) Finally, we append randomly selected instructions that aim to surface the leakage and to make sure all instructions from (2) are executed completely.

C. Attacker Model

We consider an attacker that can observe the timing of instruction retirements at cycle granularity. Our implementation of this attacker model relies on the RISC-V Formal Interface (RVFI) [5], a standard interface for RISC-V processors exposing information about the execution of programs, including the cycles at which instructions retire. RVFI simplifies the interaction with the microarchitecture and allows reusing most of our implementation for any processor implementing it.

By instantiating two instances of the respective core and simulating the execution of both programs of a test case in parallel, our implementation determines attacker distinguishability by comparing the cycles at which instructions retire in the two programs. Other attacker models could be implemented either using the RVFI or by exposing the required signals directly from the design.

D. Identifying Distinguishing Atoms

To determine the distinguishing atoms for a test case, we need to evaluate all contract atoms on top of an architectural simulation of both programs of a test case.

However, as long as the microarchitecture correctly implements the ISA, the sequence of architectural states can also be extracted from the microarchitectural states upon instruction retirements [36]. Thus our implementation piggy backs on the simulation of the microarchitecture described in §IV-C and it uses the RVFI to extract the relevant parts of the architectural states whenever an instruction retires. The simulation produces a VCD waveform which is then analyzed to evaluate all contract atoms and thus to determine the distinguishing atoms of a test case. Adaptation to other cores that support the RVFI should be straightforward.

V. Experimental Evaluation

We evaluate our methodology on two open-source RISC-V cores, Ibex [3] and CVA6 [1], both in a configuration implementing RV32IMC. Our methodology allowed us not only to obtain precise contracts for these cores, but also to analyze the impact of the test-case generation method and the granularity of the contract template.

A. Experimental Setup

To enable the simulation of test cases we first convert the processor sources from SystemVerilog to standard Verilog using sv2v [6] for Ibex and Yosys [8] with the Yosys-systemverilog plugin [7] for CVA6. and then compile and execute the two cores embedded in a testbench using Icarus Verilog [4]. In both cases, some manual adjustments to the processors were needed to eliminate unsupported SystemVerilog constructs.

Both cores support the RISC-V Formal Interface, and we use it to extract the attacker observations and to determine the distinguishing atoms as described in §IV-C and §IV-D. The contract synthesis algorithm is implemented in Java using Google's OR-Tools [2] to solve the integer linear program.

B. Ibex Core

We first synthesized a contract for the Ibex core using the base contract template described in Section IV-A. To this end, we used 100,000 test cases for synthesis and evaluated the performance of the obtained contract using an independent set of 2,000,000 test cases. The results allowed us to refine the contract template (as described in Section IV-A) by adding alignment leakages (AL), branch leakages (BL), and data-dependency leakages (DL).

Figure 2 shows the precision of the obtained contracts for different contract templates in terms of the number of test cases used for synthesis (from 0 to 100,000). Some leakage sources are only discovered after a while, e.g., the first data-dependency leakages are discovered after about 10,000 test cases, which

explains the drop in precision for templates that do not include `DL` at that point. We observe that the refined templates lead to an increase in precision as they allow for more fine-grained leakage contracts. Data-dependency leakages (`DL`), in particular, enable significant precision improvements.

Figure 3 shows the sensitivity of the synthesized contracts for the full contract template depending on the number of test cases used for synthesis. We adopt the standard notion of sensitivity used in the evaluation of binary classifiers: $\mathit{Sensitivity} = \frac{TP}{TP+FN}$, where FN is the number of false negatives. Initially, sensitivity increases rapidly, as additional test cases frequently reveal new sources of leakage. After the first 15,000 test cases, however, the curve flattens. The final contract has a sensitivity of 99.93%.

Finally, we synthesized a contract, summarized in Table I, using the larger set of 2,000,000 test cases. In total, the synthesized contract includes 82 atoms. ✓ indicates that all instructions in this category show leakages in the respective category (as introduced in Section IV-A), ✗ indicates that no leakages were found, ◦ indicates that there are some leakages in this category, but not all instructions have these leakages, and - indicates that the atom does not apply to this category. The distance n in the `DL` category is always 1.

Next, we overview some of our findings: (1) Our experiments and the synthesized contracts show that the Ibex core leaks whether memory accesses are aligned or not. Indeed, the documentation of the Ibex core confirms that requests on the memory interface are always aligned to a word boundary, which is compatible with our observation, but the impact on information leakage had not been established before. (2) We also discovered that the timing of branch instructions depend on whether the branch is taken or not taken even if the branch target is the same in both cases, e.g., `BEQ r1 r2 4` jumps to the next instruction independently of `r1` and `r2`.

C. CVA6 Core

We similarly analyzed the CVA6 core. A summary of the contract obtained using 500,000 test cases is shown in Table II. In total the synthesized contract consists of 77 atoms.

The CVA6 core uses a more complex memory interface that does not expose anything about a specific memory access in the analyzed setup. We remark that, even though the CVA6 features a simple branch predictor, the contract template originally composed for the Ibex, was sufficient to capture the CVA6's leakage.

As the CVA6 core has a deeper and more complex pipeline than the Ibex, data and control dependencies can have a more pronounced effect in the CVA6 core, which is reflected in the synthesized contract: While the data dependencies all have a distance of $n = 1$ as forwarding is effective here, we observe distances of up to $n = 4$ due to control dependencies upon branch instructions.

D. Computation Time

Table III compares the execution time of the contract synthesis algorithm for the two cores. The simulation of a test case on the CVA6 core is much slower than on the Ibex core, which is

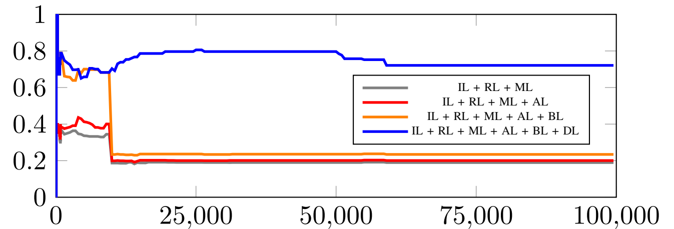

Fig. 2. Precision of contracts (y-axis) w.r.t. 2,000,000 test cases for different contract templates starting from the base contract (`IL+RL+ML`) depending on the number of test case (x-axis) used for contract synthesis.

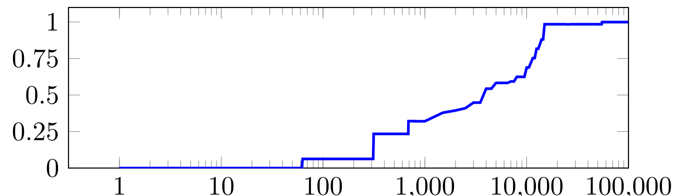

Fig. 3. Sensitivity of contracts (y-axis) w.r.t. 2,000,000 test cases using the full contract template (`IL+RL+ML+AL+BL+DL`) depending on the number of test cases (x-axis) used for contract synthesis. Note the logarithmic x-axis.

due to the much higher complexity of the design and due to the translation into standard Verilog using Yosys. However, as can be seen, the contract synthesis algorithm is still able to synthesize a contract for the CVA6 core in a reasonable amount of time.

VI. Related Work

Leakage contracts: Leakage contracts [20], which our methodology builds on, are an abstraction for capturing timing leaks at ISA level. The leaks captured in a contract are formally connected with the actual timing leaks in a hardware implementation. There are several tools for reasoning about contracts: for verifying contract satisfaction against RTL processor designs [11], [36], for detecting contract violations for black-box CPUs [12], [27]–[29], and for verifying program security w.r.t. a given contract [16], [19]. Contracts synthesized using our methodology may serve as inputs to these tools.

TABLE I
Synthesized contract for the Ibex processor.

	IL	RL	ML	AL	BL	DL
Arithmetic instructions	◦	◦	-	-	-	◦
Division, Remainder	✗	◦	-	-	-	◦
Multiplication	◦	✗	-	-	-	✓
Loads	◦	✗	✗	✓	-	✗
Stores	◦	✗	✗	✗	-	✗
Branch instructions	◦	✗	-	-	✓	✗

TABLE II
Synthesized contract for the CVA6 processor.

	IL	RL	ML	AL	BL	DL
Arithmetic instructions	◦	◦	-	-	-	◦
Division, Remainder	◦	◦	-	-	-	◦
Multiplication	✗	◦	-	-	-	◦
Loads	◦	✗	✗	✗	-	◦
Stores	✗	◦	✗	✗	-	✗
Branch instructions	✗	✗	-	-	✓	◦

TABLE III
PERFORMANCE MEASUREMENTS FOR CONTRACT SYNTHESIS USING 100,000 TEST CASES ON AN AMD RYZEN THREADRIPPER PRO 5995WX WITH 512GB OF RAM USING UP TO 128 THREADS.

	Ibex	CVA6
Compilation of the testbench	7.2 s	2123 s
Simulation of a single test case	0.2 s	88 s
Extraction of distinguishing atoms	175 ms	75 ms
Computation of the contract	15.6 s	16 s
Overall computation time	5.3 min	1175 min

Modeling timing leaks: There are multiple formal approaches for studying timing leaks. Most of them capture leaks at program level: from simple models associated with "constant-time programming" [9], [26] to more complex ones capturing leaks of transient instructions [14], [16], [19], [30], [34]. Other approaches, instead, capture leaks on simplified processor models [18], [24], [35]. Our work enables synthesizing program-level models from actual processor implementations, while previous approaches rely on manually written models that have no formal connection to particular concrete processors.

Detecting leaks through testing: Revizor [28], [29] and Scam-V [12], [27] search for contract violations for black-box CPUs. However, they can only be applied post-silicon and require physical access to a CPU. Other approaches [17], [25], [37] aim to detect leaks by analyzing hardware measurements without relying on leakage contracts but, again, apply only post-silicon. Finally, SpecDoctor [21] and SigFuzz [31] focus on detecting microarchitectural timing side channels in RTL designs in the pre-silicon phase. In contrast to our work they do not aim to characterize leakage at ISA level.

VII. CONCLUSION

ISA-level models of the information leaked microarchitecturally by processors are a critical component for the development of systems resistant to microarchitectural attacks. In this paper, we showed how to semi-automatically synthesize such models, in the form of hardware-software leakage contracts, directly from RTL processor designs. This allowed us to derive ISA-level descriptions of leakage for two open-source RISC-V cores, Ibex and CVA6, which so far lacked a formal specification of their microarchitectural leakage properties.

We will open source our toolchain and make the leakage contracts for the Ibex and the CVA6 available to the community.

REFERENCES

[1] "CVA6: A 6-stage RISC-V CPU core capable of booting linux." [Online]. Available: https://github.com/openhwgroup/cva6

[2] "Google's Operations Research tools." [Online]. Available: https://github.com/google/or-tools

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

[4] "The icarus verilog compilation system." [Online]. Available: https://github.com/steveicarus/iverilog

[5] "RISC-V Formal Verification Framework." [Online]. Available: https://github.com/SymbioticEDA/riscv-formal

[6] "sv2v: SystemVerilog to Verilog." [Online]. Available: https://github.com/zachjs/sv2v

[7] "systemverilog-plugin – SystemVerilog support for Yosys." [Online]. Available: https://github.com/chipsalliance/systemverilog-plugin

[8] "Yosys Open SYnthesis Suite." [Online]. Available: https://github.com/YosysHQ/yosys

[9] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security*, 2016.

[10] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[11] R. Bloem, B. Gigerl, M. Gourjon, V. Hadzic, S. Mangard, and R. Primas, "Power contracts: Provably complete power leakage models for processors," in *CCS*, 2022.

[12] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *MICRO-54*. ACM, 2021.

[13] J. V. Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[14] 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 *PLDI*, 2020.

[15] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 4:1–4:32, 2015.

[16] X. Fabian, M. Patrignani, and M. Guarnieri, "Automatic detection of speculative execution combinations," in *CCS*, 2022.

[17] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *NDSS*, 2020.

[18] R. Guanciale, M. Balliu, and M. Dam, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *CCS*, 2020.

[19] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," in *IEEE S&P*, 2020.

[20] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *IEEE S&P*, 2021.

[21] J. Hur, S. Song, S. Kim, and B. Lee, "SpecDoctor: Differential fuzz testing to find transient execution vulnerabilities," in *CCS*, 2022.

[22] 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 *IEEE S&P*, 2019.

[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[24] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019.

[25] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis background superscalar memory architecture," in *USENIX Security*, 2020.

[26] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC*, 2005, pp. 156–168.

[27] H. Nemati *et al.*, "Validation of Abstract Side-Channel Models for Computer Architectures," in *CAV*, 2020.

[28] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box CPUs against speculation contracts," in *ASPLOS*, 2022.

[29] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing," in *IEEE S&P*, 2023.

[30] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *CCS*, 2021.

[31] C. Rajapaksha, L. Delshadtehrani, M. Egele, and A. Joshi, "SIGFuzz: A framework for discovering microarchitectural timing side channels," in *DATE*, 2023.

[32] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *CHES*, 2003.

[33] S. van Schaik *et al.*, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.

[34] M. Vassena *et al.*, "Automatically eliminating speculative leaks from cryptographic code with Blade," in *POPL*, 2021.

[35] J. R. S. Vicarte *et al.*, "Opening Pandora's box: A systematic study of new ways microarchitecture can leak private data," in *ISCA*, 2021.

[36] Z. Wang, G. Mohr, K. v. Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *CCS*, 2023.

[37] D. Weber *et al.*, "Osiris: Automated discovery of microarchitectural side channels," in *USENIX Security*, 2021.

[38] Y. Yarom and K. Falkner, "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *USENIX Security*, 2014.