Power Contracts: Provably Complete Power Leakage Models for Processors

Roderick Bloem
Graz University of Technology

Barbara Gigerl
Graz University of Technology

Marc Gourjon
Hamburg University of Technology
and NXP Semiconductors

Vedad Hadžić
Graz University of Technology

Stefan Mangard
Graz University of Technology

Robert Primas
Graz University of Technology

ABSTRACT

The protection of cryptographic software implementations against power-analysis attacks is critical for applications in embedded systems. A commonly used algorithmic countermeasure against these attacks is masking, a secret-sharing scheme that splits a sensitive computation into computation on multiple random shares. In practice, the security of masking schemes relies on several assumptions that are often violated by microarchitectural side-effects of CPUs. Many past works address this problem by studying these leakage effects and building corresponding leakage models that can then be integrated into a software verification workflow. However, these models have only been derived empirically, putting the otherwise rigorous security statements made with verification in question.

We solve this problem in two steps. First, we introduce a *contract* layer between the (CPU) hardware and the software that allows the specification of microarchitectural side-effects on masked software in an intuitive language. Second, we present a method for proving the correspondence between contracts and CPU netlists to ensure the completeness of the specified leakage models. Then, any further security proofs only need to happen between software and contract, which brings benefits such as reduced verification runtime, improved user experience, and the possibility of working with vendor-supplied contracts of CPUs whose design is not available on netlist-level due to IP restrictions. We apply our approach to the popular RISC-V IBEX core, provide a corresponding formally verified contract, and describe how this contract could be used to verify masked software implementations.

KEYWORDS

Power Side-Channel, Leakage Model, Verification, Contract, Domain-Specific Language, Masking, Probing Security

Author Email: `first.last@iaik.tugraz.at` and `first.last@tuhh.de`

1 INTRODUCTION

Physical side-channel attacks such as power or EM analysis allow attackers within proximity of a device to learn sensitive information like cryptographic keys [31, 45]. One of the most widely used algorithmic countermeasures for protecting a cryptographic implementation against these kinds of attacks is masking [14, 27, 30]. Masking is a secret-sharing technique that splits input and intermediate variables of cryptographic computations into $d \geq t+1$ random shares such that the observation of up to t shares does not reveal any information about their corresponding unmasked value. Masking schemes typically rely on certain assumptions, such as independent computations producing independent side-channel leakage. However, the structure of a CPU architecture can violate these assumptions and introduce additional leakage effects [16, 18, 25, 39, 43]. Such leakage is often also referred to as *order-reducing leakage* because it induces a security loss and thus a gap between formal security assurance and practical resilience. The physical characteristics of gates are relatively well understood and give rise to extended leakage models, which allow constructing *hardware* implementations that reliably mitigate order-reducing leakage [17, 20, 33, 34]. Similarly, when designing masked *software* implementations of cryptographic algorithms, knowing the concrete power side-effects of different instruction types is indispensable. It allows developers to optimize the performance of masked implementations by simplifying the otherwise trial-and-error hardening process [9].

State of the Art. Many works address the problem of characterizing and understanding the leakage behavior of instructions. These can be divided into two categories: works that use empirical methods to determine side-channel leakage, and works that use formal verification approaches to verify side-channel resilience.

On the empirical side, the measurement of a CPU's power consumption combined with a subsequent analysis using statistical methods, is a straightforward approach to determine whether cryptographic software is correctly masked. Any observed leakage effects can be reverse-engineered and taken into account in hardened versions of the respective masked software implementations [1, 23, 24, 36, 38, 43, 48]. Based on these observations, subsequent works derive leakage models that describe the leakage behavior of assembly instructions that are commonly used for masking implementations on CPUs [9, 10, 37]. All existing empirical approaches require high practical effort and cannot guarantee completeness, thereby reducing the confidence in security assessments [5, 39].

On the formal verification side, several works verify the security of masked software implementations under specific masking-related security notions. MaskVerif performs algorithmic software masking verification using generic leakage models [7]. Barthe *et al.* [9] improve upon this by verifying the absence of order-reducing leakage with the tool scVerif which combines user-provided leakage models for assembly instructions with verification routines for the more practice-oriented stateful extensions of the commonly accepted security notions t−NI and t−SNI [7, 8]. However, the security assurance still relies on the completeness of the used leakage models, as it is the case for pure empirical approaches.

An entirely different take on masked software verification was presented with the Coco tool by Gigerl *et al.*, which avoids modeling leakage entirely and instead simulates the execution of masked software directly on the processor netlist [25, 26, 29]. Their approach considers the leakage of every single gate in an extended hardware leakage model and captures a wide range of microarchitectural side-effects. However, their method requires the processor's netlist, which may not be available, e.g., for most Arm-based embedded devices. All other tools operate in fixed software leakage models and are inadequate to protect software against device-specific leakage.

Our Contribution. We answer to the question of leakage model completeness and establish end-to-end (E2E) security for software executing on a processor. First, we introduce a *contract* between the hardware and the software that defines precise semantics and models side-channel behavior of assembly instructions. We then establish a technique to verify *compliance* of a processor with a contract. A processor is compliant when the leakage of each of its gates and the semantic of instruction is correctly specified in the contract. Put vice-versa, we prove the contract's model of instructions correct and its model of leakages complete. We pave the way for provable E2E security by defining software compliance for threshold probing security notions so that the approach of Barthe *et al.* [9] can be easily mapped to our slightly different language for contracts. We combine hardware compliance and software compliance to prove E2E security: *any* compliant software is secure w.r.t. *all* microarchitectural power side-channel leakage of *any* compliant CPU. Compared to related work, our approach comes with benefits such as more rigorous (practical) security statements, simplified software verification workflows, and the possibility of working with vendor-supplied contracts of CPUs whose design is not available on netlist-level due to IP restrictions. Our contributions also enable the construction of reliable hardened processors, as users can specify the desired leakage model in a contract and modify the CPU implementation to achieve compliance for fixed contracts. We emphasize that the intermediate contract layer enables for the first time portability of secure implementations across processors and improves the separation of secure hardware and software development in general.

1. Contracts. We introduce an intuitive and industry-grade domain-specific language (DSL) called Genoa. Genoa allows specifying Instruction Set Architecture (ISA) semantics and device-specific leakages in contracts. Genoa extends the long-standing Sail language to support leakage specifications. The RISC-V foundation recently picked Sail as the official tool to specify the reference RISC-V ISA and all standard extensions [40, 47]. Models for multiple architectures (e.g., Arm) exist, which can be freely adopted and compiled to software emulators [4]. We reuse existing models as the basis for contracts, augmenting them with leakage specifications and providing an interface for our verification tool. We show that whenever a program is secure with respect to a contract, its concrete execution on a compliant processor is also secure, i.e., no order-reducing leakage can occur. We emphasize that our contracts also support higher-order masking, branching, and secret-dependent memory accesses needed for masked table lookups [15].

2. Hardware Compliance. We present a method to automatically verify the compliance of a processor with a contract. Verification ensures that the leakage of every gate is captured by a leakage specification in the contract and that the contract specifies correct instruction semantics. Our methodology is based on the intuition that if the contract properly models the hardware, then any leakage arising in the hardware can be computed from leakage produced during an execution in the contract. The verification encodes both hardware and contract execution, respectively leakage, as SMT formulas and checks for model gaps using the SMT solver Z3 [6, 19]. If the solver finds no cases where a hardware leakage is not modeled from the contract leakage, we have proven hardware compliance.

3. Case Study. We implement our methods in a tool and apply them to the popular RISC-V IBEX core [32], resulting in a verifiably complete contract for a wide range of instructions that are commonly used for cryptographic implementations. An approach for the verification of masked software implementations from Barthe *et al.* [9] can then easily be adopted to prove the absence of order-reducing leakage of masked software that is executed on a compliant CPU hardware. The contract, which is based on the official RISC-V reference models, is provided as an artifact.

2 SIDE-CHANNEL RESILIENCE

We introduce preliminaries for side-channel security. Hardware circuits and their power side-channel leakage are modeled in Section 2.1. In Section 2.2, we recall the masking countermeasure and the formal notions of provable side-channel resilience.

2.1 Hardware Model and Gate-level Leakage

Processors are digital hardware circuits which can be modeled using labeled directed graphs. For any given circuit (G, W, L), we say that G is the set of gates, $W \subseteq G \times G$ is the set of wires connecting the gates, and $L : G \rightarrow T$ is a labeling defining the type $\tau \in T$ of each gate $g \in G$. The types T depend on the technology that realizes the circuit. In addition to combinatorial gates we only require that the technology contains an input type τ_{in} and a register type τ_{reg}. Input gates only have outgoing wires, and register gates only have one incoming wire. Additionally, every cyclic path in the circuit contains at least one register gate. The state of a circuit is completely defined by the values of its inputs and registers, referred to as *locations*, denoted with $V^h = \{g \in G \mid L(g) \in \{\tau_{\text{in}}, \tau_{\text{reg}}\}\}$. Hardware states are denoted with $\sigma^h \in \mathbb{B}^{|V^h|}$, with optional subscripts. Any location $v^h \in V^h$ just returns the appropriate bit of the state. Any gate g is a function of a state, *i.e.*, $g : \mathbb{B}^{|V^h|} \rightarrow \mathbb{B}$ where gate $g \in G \setminus V$ combines state bits according to its type τ.

The execution of a circuit happens in clock cycles. For a state σ^h_j, we denote the next state as σ^h_{j+1}. The registers of the next state have values reflecting the values of their inputs in the previous cycle, *i.e.*, $g(\sigma^h_{j+1}) := g'(\sigma^h_j)$ with $(g', g) \in W$, whereas the next state of circuit inputs is determined by the environment.

We now proceed to define the power side-channel leakage that is exposed to an adversary. The root cause of power side-channels is that CMOS logic draws power, or emits electromagnetic radiation, mainly if a transistor switches its state. Thus, CMOS gates have a data-dependent power consumption. The leakage behavior of CMOS gates themselves is relatively well understood and can be modeled by a few simple leakage effects which allow an (idealized) *probing adversary* to observe the (intermediate) values of gates and wires without any loss due to measurement noise [5, 20, 30]. The seminal work of Ishai *et al.* [30] introduced *value leakage* which allows an idealized adversary to observe the value of any wire connected to a gate at the beginning or end of a cycle, *i.e.*, its stable signal. The value leakage λ_g exposed by the gate is its value $\lambda_g(\sigma_j^h) = g(\sigma_j^h)$ in the state σ_j^h. Besides value leakage, additional leakage effects are also observable in hardware. We define our extended probing model in close relation to the robust probing model of Faust *et al.* [20]. *Transition leakage* refers to the phenomeneon that the power consumption of CMOS gates depends on the charges (state) of the gate before computation. As such transition leakage allows observing whether the value of a gate changed during a clock cycle but also whether the value changed from zero to one or vice-versa. Formally, our idealized adversary is able to observe the initial value and the resulting value of each gate. The observable gate leakage is then the concatenation of the old and new gate values, *i.e.*, $\lambda_g(\sigma_{j-1}^h, \sigma_j^h) = g(\sigma_{j-1}^h)||g(\sigma_j^h)$. This sufficiently captures any real-world transition leakage function computable from the old and new gate values. In addition to these main phenomena, there are *glitches* caused by propagation delay in the temporary logic states of combinatorial circuits within one clock cycle (and thus rather ephemeral) [35] and *couplings* caused by inductive coupling of adjacent wires [17]. These can be modeled by defining $\lambda_g(\sigma_{j-1}^h, \sigma_j^h)$ for non-register gates as the concatenation of all possible values the gate g could take on due to these effects. We use $\mathcal{L}_{0,m}^h$ to denote the observable *gate-level* leakage throughout the execution starting in state σ_0^h and ending in state σ_m^h. While the techniques described in the following apply to all effects, for the purpose of this paper, we focus on value leakage and transition leakage. Hence, we define $\mathcal{L}_{0,m}^h = \{\lambda_g^h(\sigma_{j-1}^h, \sigma_j^h) \mid g \in G, 1 \leq j < m\}$.

2.2 Provable Security and Simulatability

Applying masking to a cryptographic algorithm requires to replace the primitive operations (e.g., logical conjunction, exclusive or, addition) by masked computations, often called *gadgets*, which compute the same operation securely on shares [28, 30, 41, 46]. The challenge in the design and implementation of gadgets is to maintain the security of the secret-sharing: it must remain information theoretically impossible to learn the secrets or intermediate values by observing up to t leakages caused by the circuit. In sufficiently noisy environments this leads to an exponential gain of security in the order of t [13, 44]. However, many works do not take the full gate-level leakages into account, resulting in nominally secure implementations which are exploitable at an lower-than-advertised security order. Especially for masked software, the resulting gap in the security assurance allows to break the implementation by observing, e.g., transition leakage of the processor executing the program [5, 33, 42]. The focus of this work is to reduce the gap by enabling software security assessments to include the *complete* set of gate-level leakages.

We give an overview of the notation associated with masking, and formalize gadgets and their security. Masking heavily relies on random variables. We write the names of random variables in lowercase, e.g., x_i, and use lowercase boldface names for sets of variables, e.g., $\boldsymbol{x} = \{x_0, \ldots, x_n\}$. Each random variable x_i, respectively set $\boldsymbol{x}$, is associated with a probability distribution $\Pr[x_i]$, respectively $\Pr[\boldsymbol{x}]$. Each secret x_i is encoded (masked) using $d \geq t+1$ shares and we write $\overline{x_i} = \{x_i^0, \ldots, x_i^d\}$ for the shares which encode x_i, where x_i^j denotes for $0 \leq i < n$ and $0 \leq j < d$ the jth share of the ith secret. The set of all shares is denoted by $\overline{\boldsymbol{x}} = \{\overline{\boldsymbol{x}}_0, \ldots, \overline{\boldsymbol{x}}_{n-1}\}$.

A gadget operates on input tuple $(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$ returning tuple $(\overline{\boldsymbol{y}}, \boldsymbol{o}, \boldsymbol{\mathcal{L}})$, each consisting of random variables. The input shares $\overline{\boldsymbol{x}}$ are a set of t-wise independent encodings $\overline{x_i}$, each encoding a secret variable $\boldsymbol{x}_i$. $\boldsymbol{r}$ represents a set of independent and uniformly random variables, $\boldsymbol{p}$ are public inputs independent of secrets. The output of a gadget consist of output shares $\overline{\boldsymbol{y}}$, public outputs $\boldsymbol{o}$, and observable leakage $\boldsymbol{\mathcal{L}}$. Each individual output is a random variable y_i^j (respectively o_i) and computed as a function of the gadget's inputs, *i.e.*, $y_i^j = f_i^j(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$. During its execution a gadget produces observable leakage $\boldsymbol{\mathcal{L}} = \{\lambda_0(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}), \ldots, \lambda_m(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})\}$, which an attacker can observe, e.g., through power measurements. The attacker's goal is to learn information about the unshared secret inputs $\boldsymbol{x}$.

Threshold non-interference (t−NI) and strong threshold non-interference (t−SNI) are two prominent security notions for proving the security of gadgets against idealized adversaries [7, 8]. These have been extended in [9] into Stateful t−(S)NI to incorporate that physical execution involves state and public in- and outputs. Security of gadgets in these notions is shown by proving that the observations an attacker makes can be *simulated* without knowing the secret values, thereby proving that no information can be gained from t observations. In the following, we formalize what it means to simulate random variables, and restate t−NI and t−SNI.

Definition 1 (Simulation Procedure). *Let $\boldsymbol{c}$ and $\boldsymbol{h}$ be sets of possibly related random variables and $\boldsymbol{r}$ be a set of independent and uniformly distributed variables. The simulation procedure $S : Dom(\boldsymbol{c} \cup \boldsymbol{r}) \rightarrow Dom(\boldsymbol{h})$ (*simulator *for short) samples the random variables $\boldsymbol{r}$ to simulate the distribution of $\boldsymbol{h}$ from $\boldsymbol{c}$. We say that simulator S simulates $\boldsymbol{h}$ from $\boldsymbol{c}$ and $\boldsymbol{r}$ if $Pr[S(\boldsymbol{c}, \boldsymbol{r})] = Pr[\boldsymbol{h}]$.*

Importantly, the variables $\boldsymbol{c}$ and $\boldsymbol{h}$ are not necessarily independent, meaning $\Pr[\boldsymbol{h} \mid \boldsymbol{c}]$ could be different from $\Pr[\boldsymbol{h}]$, *i.e.*, their distributions are somehow related. This is central in the definitions of Stateful t−NI and t−SNI, however, we introduce a non-probabilistic way of *modeling* (instead of simulating) the outcome of a computation from a related but different value.

Definition 2 (Modeling Function). *Let $f_H : H \rightarrow V$ and $f_C : C \rightarrow U$ be deterministic functions. We say that a deterministic function $f_S : U \rightarrow V$ is a* modeling function *which models f_H from f_C under deterministic $\Psi : H \times C \rightarrow \mathbb{B}$ whenever*

$$\forall h \in H, c \in C : \Psi(h, c) \Rightarrow f_S \circ f_C(c) = f_H(h). \tag{1}$$

Definition 2 is strong: whenever modeling function f_S models f_H then it also simulates it, captured by Lemma 1.

Lemma 1 (Modeling Functions Simulate). *Let* $\boldsymbol{h}$ *and* $\boldsymbol{c}$ *be sets of possibly dependent random variables with deterministic function* f_H *computing a set of dependent random variables* $\boldsymbol{v}$ *and function* f_C *computing dependent random variables* $\boldsymbol{u}$*. If function* f_S *models* f_H *from* f_C *whenever* $\Psi(\boldsymbol{h}, \boldsymbol{c})$ *then it also simulates* $\boldsymbol{v}$*:*

$$Pr\left[f_S \circ f_C\left(\boldsymbol{c}\right) \mid \Psi(\boldsymbol{h}, \boldsymbol{c}) = \top\right] = Pr\left[\boldsymbol{h} \mid \Psi(\boldsymbol{h}, \boldsymbol{c}) = \top\right]. \qquad (2)$$

Stateful t−(S)NI requires a probabilistic simulator to simulate observations on leakage or outputs shares independently of secrets and a function modeling public outputs from public inputs.

Definition 3 (Stateful t−(S)NI [7–9]). *Gadget* $\mathsf{G}\left(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}\right) = \left(\overline{\boldsymbol{y}}, \boldsymbol{o}, \boldsymbol{\mathcal{L}}\right)$ *is Stateful* t−(S)NI *iff for every set* $\mathbf{e} \subseteq \boldsymbol{\mathcal{L}} \cup \overline{\boldsymbol{y}}$*, with* $|\mathbf{e}| \leq t$*, there exists a subset of input shares* $\mathbf{s} \subseteq \overline{\boldsymbol{x}}$*, with* $\forall i : |\mathbf{s} \cap \overline{\boldsymbol{x}_i}| \leq t' \leq t$*, a set of uniformly random variables* $\boldsymbol{r}'$*, a modeling function* $F : Dom(\boldsymbol{p}) \longrightarrow Dom(\boldsymbol{o})$ *modeling public outputs* $\boldsymbol{o}$ *from public inputs* $\boldsymbol{p}$ *and a simulator* $S : Dom(\mathbf{s}, \boldsymbol{r}', \boldsymbol{p}) \longrightarrow Dom(\mathbf{e})$ *simulating observations* $\mathbf{e}$ *from a subset of shares* $\mathbf{s}$*, random* $\boldsymbol{r}'$ *and public inputs* $\boldsymbol{p}$ *such that* $Pr\left[S\left(\mathbf{s}, \boldsymbol{r}', \boldsymbol{p}\right), F\left(\boldsymbol{p}\right)\right] = Pr\left[\mathbf{e}, \boldsymbol{o}\right]$ *and* $\boldsymbol{o} = F\left(\boldsymbol{p}\right)$*. For* t−NI $t' = |\mathbf{e}|$ *while the stricter* t−SNI *notion requires* $t' = |\mathbf{e} \backslash \overline{\boldsymbol{y}}|$.

The tool scVerif allows proving software gadgets secure under these notions for custom definitions of the observable leakage behavior $\boldsymbol{\mathcal{L}}$ [9]. However, to prove security with respect to gate-level leakage, the provided model of $\boldsymbol{\mathcal{L}}$ must capture absolutely all gate-level leakages of the CPU executing a gadget. In case a gate-level leakage is modeled incorrectly, the tool could assert security although the gadget can be broken with less than t observations at the gate-level. As analyzed by Balasch *et al.* [5], such leakage may halve the security order, *i.e.*, $t' = \frac{t}{2}$. Even worse, Gigerl *et al.* report protection losses that scale with the number of processor pipeline stages [26]. Our work mitigates such losses by verifying that all gate-level leakages are modeled.

3 HARDWARE-SOFTWARE CONTRACTS

In Sections 3.1 and 3.2, we describe how to build a contract which completely captures the leakage exposed by every single gate of a processor. We introduce our DSL called Genoa and demonstrate it by describing our contract for IBEX. In Section 3.3 we define how to verify the security of masked software against the model specified in a contract.

We then turn towards the question of model completeness: In Section 3.4 we define *compliance*, a property which connects gate-level leakage of a processor to the leakage model specified in a contract. Finally, we prove E2E security by showing that if a processor's hardware complies with a contract, the contract models all gate-level leakages. This allows to reduce security assessments to the model specified in the contract and allows to show that whenever a specific masked software implementation is secure in the contract, then, the security order of the software is not reduced when executed on real, compliant hardware. In Section 4 we derive an automated verification tool to check compliance w.r.t. gate-level transition leakage of realistic processors.

3.1 Expressing Contracts in Genoa

A contract defines the instruction semantics and exposed side-channel information of a processor from the perspective of a software developer, *i.e.*, which data is leaked via power side-channels

Listing 1: Contract model of state defined in Genoa.

```
// adopted from RISCV Sail Model, see license in Listing 6
register PC : bits(32)
register nextPC : bits(32)
register x1 : bits(32) ...
// shadow registers
register rf_pA : bits(32) // register file read port A
register rf_pB : bits(32) // register file read port B
register mem_last_addr : bits(32) // address of last access
register mem_last_read : bits(32) // value of last access
```

when an instruction is executed in conjunction with the semantic of the instruction. Contracts must specify correct instruction semantics to be able to express accurate data leakage. Besides the instruction perspective contracts allow to execute and thereby model entire programs. In practice, a contract is a user-supplied text file containing specifications of instructions written in Genoa, which is an extension of the industry-grade Sail DSL [4]. In a nutshell, Sail is designed to model the semantic of instructions of arbitrary ISAs and we add the ability to specify leakage.

Genoa extends Sail by a dedicated `leak` statement to express that specific values are observable through a side-channel. For example, a statement of the form `leak(val1, val2)` indicates that the processor may leak any combination of the source operands, *i.e.*, any value that can be computed using these operands. Users are free to specify more fine-grained leakage using concrete functions, e.g., the Hamming-Distance. Barthe *et al.* applied this concept in [9] to a custom DSL but require users to develop complex models from scratch, involving considerable effort. As we will show, modeling leakage in a contract becomes as easy as adding few `leak` statements to one of the many existing Sail models for RISC-V, Arm, etc. (Genoa supports all Sail models), providing an interface to our tool and applying it to check for modeling gaps (more on this in Section 5) Our contract for IBEX is shown in Listings 1 to 6.

The Sail manual [2] and the work of Armstrong *et al.* [4] provide in-depth explanations of the syntax, we give a brief overview. In Listing 1 we define the architectural state of a processor, consisting of 32-bit registers which are declared as global variables. Additional *shadow registers* are introduced to model leakage which arises from microarchitectural state in hardware. For example, `rf_pA` is used to remember the value last read from the register file but is not used in the specification of instruction semantics. Its value is maintained in the model and later on leaked in `leak` statements to model leakage of instructions accessing the register file since such leakage involves the value of the register read last [43]. Every contract must specify a step function defining how a single instruction is executed. For IBEX, `step_ibex` shown in Listing 2 decodes the machine code instruction (`encdec`) provided as parameter `op` and returns whether the instruction executes (`execute`) successfully. Both `encdec` and `execute` are `scattered` into multiple clauses which describe the decoding, respectively execution, for a few instructions loosely belonging to a category (see Listing 6). Each category is represented by a datatype `ast`, e.g., `RTYPE` for instructions operating on three ISA registers, represented by three indices for destination and two operands, as well as another datatype `rop` for different operations. Listing 3 shows the model of `RTYPE` instructions; `encdec` maps between instruction bits and `ast` representations using conditional

Listing 2: Model of instruction-step χ defined in Genoa.

```
// adopted from RISCV Sail Model, see license in Listing 6
function step_ibex (op : bits(32)) -> bool = {
  nextPC = PC + 4;
  let instruction = encdec(op);
  let ret = execute(instruction);
  tick_pc();
  match ret {
    RETIRE_SUCCESS => return true,
    RETIRE_FAIL => return false}}
```

Listing 3: Contract model of R-type instructions in Genoa.

```
1  // adopted from RISCV Sail Model, see license in Listing 6
2  type regidx = bits(5) // index of register 0b00001 = x1
3  enum rop = {RISCV_ADD, RISCV_SUB, RISCV_SLL, RISCV_SLT,
       ↪ RISCV_SLTU, RISCV_XOR, RISCV_SRL, RISCV_SRA,
       ↪ RISCV_OR, RISCV_AND}
4  union clause ast = RTYPE : (regidx, regidx, regidx, rop),
5  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
6    <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
7    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] ==
         ↪ bitzero)
8  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLT)
9    <-> 0b0000000 @ rs2 @ rs1 @ 0b010 @ rd @ 0b0110011
10   if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] ==
         ↪ bitzero)
11 ...
12 function clause execute (RTYPE(rs2, rs1, rd, op)) = {
13   let rs1_val = X(rs1);
14   let rs2_val = X(rs2);
15   common_leakage(rs1_val, rs2_val);
16   let result : bits(32) = match op {
17     RISCV_ADD => rs1_val + rs2_val,
18     RISCV_SLT => EXTZ(bool_to_bits(rs1_val <_s rs2_val)),
19     ... };
20   overwrite_leakage(rd, result);
21   X(rd) = result;
22   return RETIRE_SUCCESS}
```

pattern matching. In line 6 rs1 represents the index bits of the first source register. The instruction semantic and leakage is specified in execute, X(rs1) returns the value of the register addressed by rs1. Leakage which is common across multiple instruction categories is exposed with a call to function common_leakage (we defer the descriptions to Section 5.2, Listing 4). The semantics of the different operations (add, signed less than, etc.) is defined in the match statement. Additional leakage is specified by overwrite_leakage (Listing 6) before writing the result to the destination register. In summary, Genoa allows designers to quickly construct and adjust contracts, while the human-readable specification supports the systematic development of side-channel protected software.

3.2 Contract Formalization

In the previous section we explained how to express contracts in the Genoa DSL. We now describe Genoa's profound formal semantics which is the basis for security verification and compliance checking.

The small-steps semantics of Genoa are defined as a reduction:

$$(\delta, s, \mathcal{L}) \mapsto (\delta', s', \mathcal{L}') \,. \tag{3}$$

δ is the context containing the definition of functions and the values of local and global variables, s is a sequence of statements and $\mathcal{L}$ is the leakage exposed during execution. After the execution of one Genoa statement (not to be confused with an instruction) δ' is the resulting context, s' are the remaining statements and $\mathcal{L}' \supseteq \mathcal{L}$ is the resulting leakage. Leakage cannot be erased. All statements except leak do not add leakage and their transformation rules stay the same as in Sail [3]. A leak statement appends its operands $v_1, \ldots, v_n$ to the execution leakage ($\cdot ; \cdot$ is a sequence of statements).

$$(\delta, \text{leak}\,(v_1, \ldots, v_n)\,; s, \mathcal{L}) \mapsto \left(\delta, s, \mathcal{L} \cup \left\{v_1(\sigma^c)||\ldots||v_n(\sigma^c)\right\}\right) . \tag{4}$$

A leak statement may expose multiple values, which allows abstracting away from particular assumptions such as Hamming-Distance leakage, as processors are allowed to leak any combination of the values exposed by a leak. While Genoa does not feature a construct to sample random values, sampling can be mimicked by reading from a dedicated state region containing randomness.

The behavior of a program is defined by user-supplied executions semantics which are specified in the contract. The contract specification written in Genoa thus defines the context δ for small-step execution and, as for hardware, the contract state $\sigma^c \in \mathbb{B}^{|V^c|}$ denotes the values of variables $v^c \in V^c$, further on referred to as locations. Based on these definitions, we can now define the semantics for the execution of an entire instruction, denoted by the step function χ, starting in state σ_i^c and returning the state σ_{i+1}^c and a set of side-channel leakages $\mathcal{L}_i^c$ of executing the i^{th} instruction:

$$\chi\left(\sigma_i^c\right) = \left(\sigma_{i+1}^c, \mathcal{L}_i^c\right) \tag{5}$$

The instruction to be executed is determined by the state σ_i^c itself, e.g., by the value of the program counter. The execution of an instruction corresponds to the many-steps evaluation of the instruction-steps function χ using the small-steps semantics described before. χ is part of the contract (for IBEX step_ibex) and supplied by the user; to simplify our tool state σ_i^c is implicitly passed while the instruction to be executed is passed explicitly. A step can either fail or succeed, indicated by a Boolean flag, the criteria for failing the execution is governed by user-defined assumptions. For IBEX these prohibit illegal instructions, accesses of non-existent registers or unaligned memory accesses. In the following, we depict execution of an entire instruction in the contract with:

$$\sigma_i^c \xrightarrow{\mathcal{L}_i^c} \sigma_{i+1}^c \tag{6}$$

Finally, we define the execution of programs in a contract. Contract $[\![\cdot]\!]^c$ models the execution of program P starting in initial state σ_0^c and resulting in state σ_n^c while producing the accumulated observable side-channel information $\mathcal{L}_{0,n}^c = \bigcup_{i=0}^{n-1} \mathcal{L}_i^c$, *i.e.*,

$$[\![\mathrm{P}]\!]^c\left(\sigma_0^c\right) = \left(\sigma_n^c, \mathcal{L}_{0,n}^c\right) . \tag{7}$$

3.3 Software Security

In this section we link security of abstract gadgets to the execution in a contract or on hardware, *i.e.*, we define security w.r.t. the leakages specified in a contract or gate-level leakage of a processor.

Gadgets have as inputs and outputs either shares, random or public values, which are linked to the definition of Stateful t–(S)NI (Definition 3). However, when the implementation of a gadget is executed within a contract or hardware then the gadget's inputs are located in the state σ with an implementation-specific placement, e.g., shares could be in registers or memory. We introduce policies π to translate between the structured in- and outputs of a gadget and the states in a contract, respectively hardware. Input policy $\pi_{\text{in}} : (\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}) \leftrightarrow \sigma_0$ constructs a state given values of variables for input shares, random and public variables but also the converse; extracting the values of gadget inputs given a state. In practice, such a policy is an annotation which defines where shares, random and public (initial) values are located w.r.t. locations of the state. Similar, output policy $\pi_{\text{out}} : (\overline{\boldsymbol{y}}, \boldsymbol{o}) \leftrightarrow \sigma_n$ maps between the values of public outputs and output shares of a gadget and state σ_n resulting from an execution. Since Stateful t−(S)NI is defined for random variables let $\boldsymbol{\sigma}$ denote the random variable for states. Using policies we can link Stateful t−(S)NI security of gadgets to the execution of their concrete implementation P within big-steps semantics $[\![\cdot]\!]$ representing either the contract or hardware:

Definition 4 (t−(S)NI of $[\![\mathsf{P}]\!]$ under π_{in}, π_{out}). *An implementation* P *of gadget* G $(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$ *is* t−(S)NI *secure w.r.t. semantic* $[\![\cdot]\!]$ *and placement policies* π_{in}, π_{out} *if the gadget* $\mathsf{G}'(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}) = (\overline{\boldsymbol{y}}, \boldsymbol{o}, \boldsymbol{\mathcal{L}}_{0,n})$ *is* t−(S)NI *according to Definition 3, where the inputs of the gadget correspond to the starting states* $\boldsymbol{\sigma}_0 = \pi_{in}(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$ *while leakages and gadget outputs correspond to the random variables* $\pi_{out}(\overline{\boldsymbol{y}}, \boldsymbol{o}) = \boldsymbol{\sigma}_n$ *resulting from execution* $[\![\mathsf{P}]\!](\boldsymbol{\sigma}_0) = (\boldsymbol{\sigma}_n, \boldsymbol{\mathcal{L}}_{0,n})$.

The actual verification of security for software implementations follows the same principles as outlined by Barthe *et al.* [9], which also describes representation of policies. However, the dependent type system of Genoa enables new approaches to verification of masked conversion functions and arithmetic masking in rings with prime moduli for security orders $t > 1$. We leave the development of verification tools to dedicated future work.

3.4 Hardware Compliance With a Contract

We now turn towards the question of model completeness and define *compliance with a contract*, a formal property expressing that the results and leakages from execution on a CPU can be modeled by a contract according to Definition 2. This property is verified in Section 4 and ensures, as we prove in Section 3.5, that any implementation that is Stateful t−(S)NI secure in a contract mitigates any order-reducing leakage caused by the gate-level leakage of a processor.

A program P executed in initial hardware state σ_0^h leads to leakages $\boldsymbol{\mathcal{L}}_{0,m}^h$ and final state σ_m^h when executed on processor $[\![\cdot]\!]^h$:

$$[\![\mathsf{P}]\!]^h\left(\sigma_0^h\right) = \left(\sigma_m^h, \boldsymbol{\mathcal{L}}_{0,m}^h\right) \tag{8}$$

In contrast to contracts the execution proceeds in clock-cycles instead of instruction-steps, *i.e.*, one step in hardware corresponds to one clock cycle as defined in Section 2.1:

$$\sigma_j^h \xrightarrow{\mathcal{L}_j^h} \sigma_{j+1}^h. \tag{9}$$

Compliance expresses the property that all leakage and all outputs of hardware execution $[\![\cdot]\!]^h$ can be modeled (according to Definition 2) from execution in a contract $[\![\cdot]\!]^c$ as long as the starting states are similar, *i.e.*, execute the same program under equivalent inputs, depicted in Figure 1.

$$\begin{tikzcd}
{[\![\mathsf{P}]\!]^c\left(\sigma_0^c\right):} & \sigma_0^c \arrow[r, "\mathcal{L}_0^c"] \arrow[d, equal, "\mathcal{M}"] & \sigma_1^c \arrow[r, "\mathcal{L}_1^c"] & \dots \arrow[r, "\mathcal{L}_{n-1}^c"] & \sigma_n^c \arrow[d, equal, "\mathcal{M}"] \\
{[\![\mathsf{P}]\!]^h\left(\sigma_0^h\right):} & \sigma_0^h \arrow[r, "\mathcal{L}_0^h"] & \sigma_1^h \arrow[r, "\mathcal{L}_1^h"] & \dots \arrow[r, "\mathcal{L}_{m-1}^h"] & \sigma_m^h
\end{tikzcd}$$

Figure 1: Compliance for a full program execution

In Definition 5 we introduce a Boolean relation between hardware state σ^h and contract state σ^c expressing that the values contained at a specific location v^h in the hardware can be modeled from a location v^c in the contract, e.g., register `x1` models its counterpart in hardware. Which contract locations model some hardware location is defined in the simulation mapping $\mathcal{M}$ provided by users alongside every contract and checked by our tool. The mapping specifies for all registers in the hardware (including finite state machines, decode stages, etc.) a location in the contract modeling the hardware location. To ease notation assume there are contract locations $v_0^c, v_1^c \in V^c$ which are constant zero, respectively one, and are later on used to express constraints on hardware execution.

Definition 5 (Similar states: $\sigma^h \simeq_{\mathcal{M}} \sigma^c$). *Two states* σ^h *and* σ^c, *with respective locations* V^h *and* V^c *are* similar *under simulation mapping* $\mathcal{M} \subseteq V^h \times V^c$, *written as* $\sigma^h \simeq_{\mathcal{M}} \sigma^c$, *if and only if*

$$\bigwedge_{(v^h, v^c) \in \mathcal{M}} v^h(\sigma^h) = v^c(\sigma^c) \tag{10}$$

Simulation mapping $\mathcal{M}$ *is said to be* complete *if for all hardware locations* V^h *a mapping is defined.*

We now give the definition of compliance, ensuring that semantic and leakage of execution of hardware is correctly modeled:

Definition 6 (Compliance: $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$). *A hardware implementation is compliant with a contract under simulation mapping* $\mathcal{M}$ *if for every program* P *and starting hardware and contract states* σ_0^h *and* σ_0^c, *the program executions*

$$[\![\mathsf{P}]\!]^c\left(\sigma_0^c\right) = \left(\sigma_n^c, \boldsymbol{\mathcal{L}}_{0,n}^c\right) \quad \text{and} \quad [\![\mathsf{P}]\!]^h\left(\sigma_0^h\right) = \left(\sigma_m^h, \boldsymbol{\mathcal{L}}_{0,m}^h\right)$$

fulfill the following conditions:

(1) ***States remain similar:*** *Whenever* σ_0^h *and* σ_0^c *are similar under* $\mathcal{M}$, *so are* σ_m^h *and* σ_n^c:

$$\forall \sigma_0^h, \sigma_0^c : \sigma_0^h \simeq_{\mathcal{M}} \sigma_0^c \Rightarrow \sigma_m^h \simeq_{\mathcal{M}} \sigma_n^c. \tag{11}$$

(2) ***Leaks are modeled:*** *For every leak* $\lambda_g(\sigma_{j-1}^h, \sigma_j^h) \in \boldsymbol{\mathcal{L}}_{0,m}^h$ *observable in hardware, there exists a leak* $\lambda(\sigma_i^c) \in \boldsymbol{\mathcal{L}}_{0,n}^c$ *in the contract and a function* $f_\lambda : Dom(\lambda) \to Dom(\lambda_g)$ *that models* λ_g *from* λ *under relation* $\sigma_0^h \simeq_{\mathcal{M}} \sigma_0^c$ *according to Definition 2:*

$$\forall \sigma_0^h, \sigma_0^c : \sigma_0^h \simeq_{\mathcal{M}} \sigma_0^c \Rightarrow f_\lambda \circ \lambda\left(\sigma_i^c\right) = \lambda_g\left(\sigma_{j-1}^h, \sigma_j^h\right). \tag{12}$$

The notion of similar states allows to express a key ingredient for the relational definition of compliance: if execution in a contract and hardware start in a similar state, then execution must end in similar states such that the hardware execution's results can be modeled according to the simulation mapping (Clause 1 of Definition 6). Further, the second part of compliance expresses that every gate-level leak observable during execution in hardware must be modeled by a single, fixed leak observable during execution in the contract (Clause 2 of Definition 6). Combined, this guarantees that software which is Stateful $t-(\mathsf{S})\mathsf{NI}$ secure when executed in the contract, is necessarily Stateful $t-(\mathsf{S})\mathsf{NI}$ when executed on compliant hardware.

3.5 End-to-end security

It remains to prove our E2E security claim: any implementation P of gadget G that is Stateful $t-(\mathsf{S})\mathsf{NI}$ w.r.t. the leakages of a contract must be Stateful $t-(\mathsf{S})\mathsf{NI}$ w.r.t. all gate-level leakage when executed on any compliant hardware and as such its security order cannot be decreased by leakage of the processor.

However, E2E security is claimed for the execution of the same gadget implementation in hardware and contract, i.e., both executions use the same structured inputs and outputs. Since the states in hardware and contract may have different structures we introduce a definition to ensure that the placement of inputs in hardware $\pi^h_{\text{in}}, \pi^h_{\text{out}}$ is similar to the ones $\pi^c_{\text{in}}, \pi^c_{\text{out}}$ for which $t-(\mathsf{S})\mathsf{NI}$ was shown in the contract. A hardware policy can be derived from a contract policy by substituting the locations which define where a value resides in the state according to the simulation mapping.

Definition 7 (Similar Policy ($\pi^h \triangleq_{\mathcal{M}} \pi^c$)). *Let contract policy $\pi^c : (\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n) \leftrightarrow \sigma^c$ link sets of values $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n$ to contract state σ^c. Hardware policy $\pi^h : (\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n) \leftrightarrow \sigma^h$ is similar to π^c, denoted $\pi^h \triangleq_{\mathcal{M}} \pi^c$ if any pair of contract and hardware states constructed from the same sets of values are similar under mapping $\mathcal{M}$:*

$$\forall \sigma^h = \pi^h(\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n), \sigma^c = \pi^c(\boldsymbol{d}_1, \ldots, \boldsymbol{d}_n) \; : \; \sigma^h \simeq_{\mathcal{M}} \sigma^c. \quad (13)$$

Instead of proving the security reduction for $t-(\mathsf{S})\mathsf{NI}$ directly we prove a general *model reduction*: any observations made by an adversary interacting with hardware may be *modeled* with a contract the hardware complies with instead. We emphasize the difference: $t-(\mathsf{S})\mathsf{NI}$ requires the existence of a *simulation procedure* whereas compliance guarantees the existence of a (stronger) *modeling function* easing the subsequent security reduction.

Theorem 2 (Model Reduction). *Let P be a program, and the gadgets $\mathrm{G}^c(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}) = \left(\overline{\boldsymbol{y}}^c, \boldsymbol{o}^c, \boldsymbol{\mathcal{L}}^c_{0,n}\right)$ and $\mathrm{G}^h(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p}) = \left(\overline{\boldsymbol{y}}^h, \boldsymbol{o}^h, \boldsymbol{\mathcal{L}}^h_{0,m}\right)$ correspond to the program executions $[\![\mathrm{P}]\!]^c(\sigma^c_0) = (\sigma^c_n, \boldsymbol{\mathcal{L}}^c_{0,n})$, respectively $[\![\mathrm{P}]\!]^h(\sigma^h_0) = (\sigma^h_m, \boldsymbol{\mathcal{L}}^h_{0,m})$, under policies π^h_{in} and π^h_{out}, respectively π^c_{in} and π^c_{out}, with $\boldsymbol{\sigma}^c_0 = \pi^c_{in}(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$, $\boldsymbol{\sigma}^h_0 = \pi^h_{in}(\overline{\boldsymbol{x}}, \boldsymbol{r}, \boldsymbol{p})$, $\boldsymbol{\sigma}^c_n = \pi^c_{out}(\overline{\boldsymbol{y}}^c, \boldsymbol{o}^c)$, and $\boldsymbol{\sigma}^h_m = \pi^h_{out}\left(\overline{\boldsymbol{y}}^h, \boldsymbol{o}^h\right)$. Furthermore, let $[\![\cdot]\!]^h \vdash_{\mathcal{M}} [\![\cdot]\!]^c$, $\pi^h_{in} \triangleq_{\mathcal{M}} \pi^c_{in}$ and $\pi^h_{out} \triangleq_{\mathcal{M}} \pi^c_{out}$ under complete mapping $\mathcal{M}$. For every set of observations in hardware on $\overline{\boldsymbol{y}}^h$ or $\boldsymbol{o}^h$ there is an equally sized set of observations in the contract on $\overline{\boldsymbol{y}}^c$ or $\boldsymbol{o}^c$ which allows to model the observations under the identity function:*

$$\forall \mathbf{e}^h_{\overline{\boldsymbol{y}}} \subseteq \overline{\boldsymbol{y}}^h \; \exists \, \mathbf{e}^c_{\overline{\boldsymbol{y}}} \subseteq \overline{\boldsymbol{y}}^c \; : \; \mathbf{e}^h_{\overline{\boldsymbol{y}}} = \mathbf{e}^c_{\overline{\boldsymbol{y}}}, \quad (14)$$

$$\forall \mathbf{e}^h_{\boldsymbol{o}} \subseteq \boldsymbol{o}^h \; \exists \, \mathbf{e}^c_{\boldsymbol{o}} \subseteq \boldsymbol{o}^c \; : \; \mathbf{e}^h_{\boldsymbol{o}} = \mathbf{e}^c_{\boldsymbol{o}}. \quad (15)$$

In addition, for every set of observations in hardware on $\boldsymbol{\mathcal{L}}^h_{0,m}$, a modeling function $T^{\boldsymbol{\mathcal{L}}}$ and a (potentially smaller) set of observations in the contract on $\boldsymbol{\mathcal{L}}^c_{0,n}$ allow to model the observations in hardware:

$$\forall \mathbf{e}^h_{\boldsymbol{\mathcal{L}}} \subseteq \boldsymbol{\mathcal{L}}^h_{0,m} \; \exists \, \mathbf{e}^c_{\boldsymbol{\mathcal{L}}} \subseteq \boldsymbol{\mathcal{L}}^c_{0,n} : \left|\mathbf{e}^c_{\boldsymbol{\mathcal{L}}}\right| \leq \left|\mathbf{e}^h_{\boldsymbol{\mathcal{L}}}\right| \wedge \mathbf{e}^h_{\boldsymbol{\mathcal{L}}} = T^{\boldsymbol{\mathcal{L}}}\left(\mathbf{e}^c_{\boldsymbol{\mathcal{L}}}\right). \quad (16)$$

Proof. The gadgets G^c and G^h operate on equally distributed inputs and the policies for hardware are similar, thus for every initial state σ^h_0 there must be a starting state σ^c_0 similar under mapping $\mathcal{M}$, *i.e.*, $\sigma^h_0 \simeq_{\mathcal{M}} \sigma^c_0$. Since hardware is compliant with the contract, the resulting states are similar as well, *i.e.*, $\sigma^h_m \simeq_{\mathcal{M}} \sigma^c_n$ and since every observation in $\mathbf{e}^h_{\boldsymbol{o}}$, respectively $\mathbf{e}^h_{\overline{\boldsymbol{y}}}$, is an observation on the value of a location in σ^h_m it follows directly that there exists a single location in the contract $\mathbf{e}^c_{\boldsymbol{o}}$, respectively $\mathbf{e}^h_{\overline{\boldsymbol{y}}}$, according to the mapping $\mathcal{M}$ which models the observation, fulfilling (14) and (15). From Lemma 1 and the second compliance clause (12) it follows that every observation $\lambda_g(\boldsymbol{\sigma}^h_{j-1}, \boldsymbol{\sigma}^h_j) \in \mathbf{e}^h_{\boldsymbol{\mathcal{L}}}$ can be modeled from some contract leak $\lambda(\boldsymbol{\sigma}^c_i) \in \boldsymbol{\mathcal{L}}^c_{0,n}$ using f_λ as modeling function. Grouping the necessary $\lambda(\boldsymbol{\sigma}^c_i)$ as the set of random variables $\mathbf{e}^c_{\boldsymbol{\mathcal{L}}}$, results in $\left|\mathbf{e}^c_{\boldsymbol{\mathcal{L}}}\right| \leq \left|\mathbf{e}^h_{\boldsymbol{\mathcal{L}}}\right|$, and defining $T^{\boldsymbol{\mathcal{L}}}$ as the set of respective f_λ implies (16), completing the proof. □

From Theorem 3, we derive simulatability of mixed observations in Corollary 3. Furthermore, the reduction from Stateful $t-(\mathsf{S})\mathsf{NI}$ in hardware to Stateful $t-(\mathsf{S})\mathsf{NI}$ in contract stated in Corollary 4 is a direct consequence of Theorem 3 and Corollary 3.

Corollary 3 (Mixed observations). *Let the setting be as in Theorem 2. Every set of mixed observations on leakage and shared outputs $\mathbf{e}^h_{\boldsymbol{\mathcal{L}},\overline{\boldsymbol{y}}} \subseteq \boldsymbol{\mathcal{L}}^h \cup \overline{\boldsymbol{y}}^h$, can be modeled from some an equally sized set $\mathbf{e}^c_{\boldsymbol{\mathcal{L}},\overline{\boldsymbol{y}}} \subseteq \boldsymbol{\mathcal{L}}^c \cup \overline{\boldsymbol{y}}^c$ by some modeling function $T^{\boldsymbol{\mathcal{L}},\overline{\boldsymbol{y}}}$.*

Corollary 4 (End-to-End Security). *Let the setting be as in Theorem 2. If gadget G^c is $t-(\mathsf{S})\mathsf{NI}$ then gadget G^h is also $t-(\mathsf{S})\mathsf{NI}$ since there exist simulators $T^{\boldsymbol{\mathcal{L}},\overline{\boldsymbol{y}}} \circ S$ and F which simulate the outputs of G^h according to Definition 3.*

This proof is valid for higher-order masking, *i.e.*, $t \geq 1$, as *each* of the t hardware observations in $\mathbf{e}^h$ can be simulated from *one* observation in $\mathbf{e}^c$ in the contract. The presented model reduction can be of help in proving the preservation of other security notions like PINI [12] or Threshold Implementations [41].

4 VERIFYING HARDWARE COMPLIANCE

Whereas Section 3 introduces *contracts* and what it means for hardware to be *compliant*, this section presents a method to actually check hardware compliance for a given processor. The method is broken down into verification steps. Each step checks if the processor satisfies some part of Definition 6. First, we check whether similar hardware and contract states stay similar after executing an instruction according to Clause 1. Then, we check that each hardware leak can be modeled from a single leak emitted in the contract, according to Clause 2.

4.1 Verification Concept

In this section, we first suggest that it is possible to prove a processor compliant without looking at full program executions. We argue that looking at all possible single instruction executions is sufficient to form an inductive argument of compliance. Next, we give an overview of the individual verification steps needed to verify that a processor is compliant with a given contract. Finally, we present a method for indirectly proving the existence of modeling functions.

$$\begin{tikzcd}
 & \sigma_i^c \arrow[rr, "\mathcal{L}_i^c"] & & \sigma_{i+1}^c \arrow[dr, equal, "\mathcal{M}"] & \\
\sigma_j^h \arrow[ur, equal, "\mathcal{M}"] \arrow[r, "\mathcal{L}_j^h"] & \dots \arrow[rr, "\mathcal{L}_{j+k-1}^h"] & & & \sigma_{j+k}^h
\end{tikzcd}$$

Figure 2: Compliance for single instruction execution

This method is the backbone of the verification procedure and relies on encoding constraints into SMT formulas and checking their satisfiability with an SMT solver.

Single instructions. Checking compliance for all programs and pairs of processors and contracts using SMT solvers is computationally intractable. Instead we prove compliance inductively by showing that Definition 6 holds for all possible executions of a single instruction. Consequently, we require compliant processors to fulfill the outlined properties at the start and end of each instruction, as shown in Figure 2. Starting with similar states σ_j^h and σ_i^c, the hardware executes k clock cycles and the contract executes one instruction step. The executions produce leaks $\mathcal{L}_{j,j+k}^h$, respectively $\mathcal{L}_i^c$, and state σ_{j+k}^h, respectively σ_{i+1}^c, for which we need to show:

(a) *States remain similar:* The states σ_{j+k}^h and σ_{i+1}^c are similar under $\mathcal{M}$ (marked red), *i.e.*, every location in σ_{j+k}^h is equal to the corresponding location in σ_{i+1}^c.

(b) *Leaks are modeled:* Every leak $\lambda_g(\sigma_{j+l-1}^h, \sigma_{j+l}^h) \in \mathcal{L}_{j,j+k}^h$ produced by the processor (marked red) must be modeled by a single leak $\lambda(\sigma_i^c) \in \mathcal{L}_i^c$ emitted in the contract execution.

These conditions are inductive and much stricter than the corresponding clauses in Definition 6, *i.e.*, if the execution of a single instruction in an arbitrary valid starting state maintains the compliance properties, the processor complies with the contract for all possible program executions.

Unfortunately, as seen in Figure 2, the hardware might require multiple clock cycles to execute an instruction, and we do not have a lock-step execution with the contract. For the purposes of this paper, we define the starting point of an instruction as the moment it becomes in-flight, *i.e.*, it reaches the decode stage, and its end point when it retires, *i.e.*, the writeback completes. Therefore, we look at every possible instruction duration k on that particular processor. Concurrent execution of instructions in the pipeline complicates this approach. For simple pipelines, contracts can easily model the produced leakage because the fetch stage does not operate with security-critical data, and the writeback stage can be made the synchronisation point for the induction, instead of its full retirement. In more complex pipelines, the methods described in this paper require checking leakage produced by hardware components directly influenced by the instruction bits.

Verifying that states remain similar. As the very first step in the verification procedure, we show that the hardware and contract states are similar throughout the whole execution. That is, we show that if the relation $\sigma_j^h \simeq_{\mathcal{M}} \sigma_i^c$ holds, the relation $\sigma_{j+k}^h \simeq_{\mathcal{M}} \sigma_{i+1}^c$ must hold after the execution of a k-cycle instruction *i.e.*, $\sigma_{j+k}^h = \chi^k(\sigma_j^h)$ and $\sigma_{i+i}^c = \chi(\sigma_i^c)$, no matter what the starting states were. This is essentially a full-fledged functional equivalence proof between the hardware and the contract. If this check succeeds, we have shown that the processor satisfies Clause 1 of Definition 6 because $\simeq_{\mathcal{M}}$ is conserved over the execution of an instruction. Section 4.3 formalizes the verification step and gives a verification method.

Finding modeling functions for gates. Before verifying that leaks are modeled, we require an intermediate verification step that provides information about the old values of gates. This constrains the old values of each gate g, therefore implicitly constraining the possible values of the corresponding leak λ_g. Otherwise, the old value could directly leak secrets, trivially breaking leakage modeling. For every gate $g \in G$ in the hardware, we show that g can be modeled by some function $f_g : \mathbb{B}^n \to \mathbb{B}$ that only uses a (small) subset of contract state bits $\theta_g : \mathbb{B}^{|V^c|} \to \mathbb{B}^n$, *i.e.*,

$$\exists f_g : \forall \sigma_j^h, \sigma_i^c : \sigma_j^h \simeq_{\mathcal{M}} \sigma_i^c \Rightarrow f_g \circ \theta_g \circ \chi\left(\sigma_i^c\right) = g \circ \chi^{k-1}\left(\sigma_j^h\right). \quad (17)$$

Ideally, we want to prove the existence of a modeling function that uses as little contract state information θ_g as possible. Section 4.4 gives exact definitions of the verification checks and the greedy minimization procedure for θ_g.

Verifying that leaks are modeled. In this verification step, we check whether the hardware leakage is properly modeled from contract leakage for any possible instruction execution, starting in any pair of similar states $\sigma_j^h \simeq_{\mathcal{M}} \sigma_i^c$. If the check succeeds, the proper modeling throughout any program execution is implied by composition of single instructions. Because we consider transition leakage, we constrain the possible values of gates at the end of the previous instruction. As mentioned before, we use the existence of a modeling function f_g according to (17) from the previous step.

The hardware leak $\lambda_g(\sigma_{j+l-1}^h, \sigma_{j+l}^h) = g(\sigma_{j+l-1}^h)||g(\sigma_{j+l}^h)$ contains information about both the old, and the new values of gate g for any clock cycle l of the executed instruction. We analyze each hardware leak function λ_g separately by going through all leakage functions $\lambda : \mathbb{B}^{|V^c|} \to \mathbb{B}^m$ and checking if there is a function $f_\lambda : \mathbb{B}^m \to \mathbb{B}^2$ that models λ_g from λ, whenever states σ_j^h and σ_i^c are similar, the contract leak is emmitted, *i.e.*, $\lambda(\sigma_i^c) \in \mathcal{L}_i^c$, and f_g models the previous value of the gate g from θ_g. Written formally:

$$\begin{gathered}\forall \sigma_j^h, \sigma_i^c : \sigma_j^h \simeq_{\mathcal{M}} \sigma_i^c \wedge f_g \circ \theta_g \circ \chi\left(\sigma_i^c\right) = g \circ \chi^{k-1}\left(\sigma_j^h\right) \wedge \\ \lambda\left(\sigma_i^c\right) \in \mathcal{L}_i^c \Rightarrow\ f_\lambda \circ \lambda\left(\sigma_i^c\right) = \lambda_g\left(\sigma_{j+l-1}^h, \sigma_{j+l}^h\right).\end{gathered} \quad (18)$$

Additionally, we require that for any possible state σ_i^c at least one leak λ fulfills (18), guaranteeing that Caluse 2 of the compliance definition is fulfilled. Section 4.5 gives a more detailed description.

Existence of modeling functions. Within the last two verification steps, we check that functions over the hardware state σ^h can be modeled from functions over the contract state σ^c whenever $\sigma^h \simeq_{\mathcal{M}} \sigma^c$. This involves proving the existence of modeling functions from Definition 2. However, automatically finding modeling functions is intractable in general [21]. We circumvent this issue by proving the existence of modeling functions without finding their definitions. Theorem 5 presents the condition we need to check.

Theorem 5 (Existence of Modeling Function). *There exists a modeling function $f : U \to V$ according to Def. 2 if and only if*

$$\begin{gathered}\forall h, h' \in H, c, c' \in C : \\ \Psi\left(h, c\right) \wedge \Psi\left(h', c'\right) \wedge f_C\left(c\right) = f_C\left(c'\right) \Rightarrow f_H\left(h\right) = f_H\left(h'\right).\end{gathered} \quad (19)$$

Proof. We prove the equality of the two statements by showing an implication in both directions. First, we prove that (19) follows from (1). From the functional congruence of f, we have:

$$\forall c, c' \in C : \left(f_C\left(c\right) = f_C\left(c'\right)\right) \Rightarrow \left(f \circ f_C\left(c\right) = f \circ f_C\left(c'\right)\right).$$

After instantiating the statement (1) separately for the primed and non-primed versions of $h \in H$ and $c \in C$, we get:

$$\forall h \in H, c \in C : \Psi\left(h, c\right) \Rightarrow f \circ f_C\left(c\right) = f_H\left(h\right),$$
$$\forall h' \in H, c' \in C : \Psi\left(h', c'\right) \Rightarrow f \circ f_C\left(c'\right) = f_H\left(h'\right).$$

We see that if all three premises are fulfilled simultaneously, then also all consequences of the implication must be fulfilled simultaneously. Therefore, we consolidate the left- and right-hand sides. Afterwards, we simplify the right-hand side by substituting $f \circ f_C\left(c\right)$ with $f_H\left(h\right)$, and respectively $f \circ f_C\left(c'\right)$ with $f_H\left(h'\right)$, to get (19).

For the second direction of the proof, we assume (19) and construct f so that it fulfills (1) and is well defined for all $u \in U$. First, we define the subset $\widehat{U} \subseteq U$ of function inputs as

$$\widehat{U} := \left\{u \mid \exists h \in H, c \in C : u = f_C\left(c\right) \wedge \Psi\left(h, c\right)\right\}. \tag{20}$$

For inputs $u \in U \setminus \widehat{U}$, we define $f\left(u\right)$ as an arbitrary result $v \in V$. This partial definition trivially fulfills (1). For all other $u \in \widehat{U}$, we define $f\left(u\right) := f_H\left(h\right)$, for an arbitrary qualified h and c as in (20). We now argue that this portion of f is well defined, because $f_H\left(h\right)$ is fixed for u. Consider the case where we can pick two such pairs:

$$\exists h, h' \in H, c, c' \in C : u = f_C\left(c\right) \wedge \Psi\left(h, c\right) \wedge$$
$$u = f_C\left(c'\right) \wedge \Psi\left(h', c'\right).$$

Because $f_C\left(c\right) = f_C\left(c'\right) = u$, the assumption (19) implies that $f_H\left(h\right)$ is unique since we always get $f_H\left(h'\right) = f_H\left(h\right)$. □

The underlying principle behind Theorem 5 can be thought of as partial functional congruence. Plainly speaking, if equal *inputs* $f_C\left(c\right)$ and $f_C\left(c'\right)$ always result in equal *outputs* $f_H\left(h\right)$ and $f_H\left(h'\right)$, then there must also be a function mapping between them. Moreover, Theorem 5 can be translated into the quantifier-free SMT fragment and efficiently checked for satisfiability with modern SMT solvers.

4.2 Verification Prerequisites

Real program execution within a processor is subject to many internal assumptions and restrictions that need to be considered when checking the compliance properties. In particular, we define *normal operating conditions* for the execution of an instruction, as well as constraints related to the mapping $\mathcal{M}$ from Section 3.4.

Normal operating conditions. The hardware of a processor has many input ports and internal registers that are invisible to a software developer and are subject to hidden assumptions under *normal operating conditions.* In this section, we introduce predicates $\phi_{\_}$ to explicitly represent these assumptions. We use the predicate $\phi_{\text{dev}}(\sigma^h)$ to represent the usual assumptions a software developer might have, such as the processor not getting reset, triggering an interrupt, going into debug mode, or getting memory access errors. Similarly, there are several internal conditions for the processor to fetch, start the execution of, and retire an instruction. We formalize these conditions as $\phi^l_{\text{instr}}(\sigma^h), 0 \leq l < k$ for the l-th cycle in k-cycle instructions and apply them for the intermediate states $\sigma^h_{j+l} = \chi^l(\sigma^h_j)$. Sometimes, the contract is not able to execute an instruction because it violates some sanity conditions such as the instruction not being implemented or triggering a fault. We formalize the condition of the contract successfully retiring an instruction as $\phi_{\text{ret}}(\sigma^c)$. We aggregate these conditions into ϕ_{noc} as

$$\phi_{\text{noc}}\left(\sigma^h_j, \sigma^c_i\right) := \phi_{\text{ret}}\left(\sigma^c_i\right) \wedge \bigwedge_{l=0}^{k-1} \phi_{\text{dev}}\left(\sigma^h_{j+l}\right) \wedge \phi^l_{\text{instr}}\left(\sigma^h_{j+l}\right).$$

There are also some constraints that concern multiple executions of the hardware and contract. For such predicates we write $\phi^*_{\_}$ instead. We define $\phi^*_{\text{ports}}(\sigma^h, \sigma^{h'})$ as the constraint that certain processor input ports only contain public values. More concretely, for two executions of the hardware, these input ports are required to produce identical values. There are also similar execution-spanning conditions for the contract. For instance, the contract should forbid the program counter from becoming secret dependent. The predicate $\phi^*_{\text{ret}}(\sigma^c, \sigma^{c'})$ expresses these constraints, and is stricter than both $\phi_{\text{ret}}(\sigma^c)$ and $\phi_{\text{ret}}(\sigma^{c'})$ separately. Finally, we extend ϕ_{noc} to ϕ^*_{noc} over several executions as

$$\phi^*_{\text{noc}}\left(\sigma^h_j, \sigma^{h'}_j, \sigma^c_i, \sigma^{c'}_i\right) := \phi^*_{\text{ret}}\left(\sigma^c_i, \sigma^{c'}_i\right) \wedge \bigwedge_{l=0}^{k-1} \phi^*_{\text{ports}}\left(\sigma^h_{j+l}, \sigma^{h'}_{j+l}\right)$$
$$\bigwedge_{l=0}^{k-1} \phi_{\text{dev}}\left(\sigma^h_{j+l}\right) \wedge \phi_{\text{dev}}\left(\sigma^{h'}_{j+l}\right) \wedge \phi^l_{\text{instr}}\left(\sigma^h_{j+l}\right) \wedge \phi^l_{\text{instr}}\left(\sigma^{h'}_{j+l}\right).$$

Breaking any of the conditions from ϕ^*_{noc} breaks the guarantees provided in this work. Because these assumptions are instrumental for correct execution, we make sure that the restrictions imposed on the hardware still permit the execution of all instructions defined in the contract. This sanity check confirms that software can still execute within both the hardware and the contract, allowing the implementation of a sensible software verifier.

Applying mappings. As introduced in Section 3.4, hardware and contract states can be similar under a mapping. For expressing that two states σ^h and σ^c are similar under mapping $\mathcal{M}$, *i.e.*, $\sigma^h \simeq_{\mathcal{M}} \sigma^c$, we use the predicate $\phi^{\mathcal{M}}_{\text{rel}}(\sigma^h, \sigma^c)$ as defined in (10). Conversely, we also require a predicate expressing that all registers, which are not in the mapping $\mathcal{M}$, are equivalent across hardware executions of the same program. We specify this property of two hardware states σ^h and $\sigma^{h'}$ for the mapping $\mathcal{M}$ and hardware locations V^h as

$$\phi^{\mathcal{M}*}_{\text{pub}}\left(\sigma^h, \sigma^{h'}\right) := \bigwedge_{v^h \in V^h, \nexists (v^h, v^c) \in \mathcal{M}} v^h\left(\sigma^h\right) = v^h\left(\sigma^{h'}\right).$$

4.3 Verifying that States Remain Similar

As introduced in Section 4.1, we verify that the hardware and contract states resulting from the execution of a program are similar by showing similarity after every instruction. Our inductive argument assumes that the states σ^h_j and σ^c_i are similar at the start of an instruction, and proves that the states σ^h_{j+k} and σ^h_{i+1} are also similar after the k-cycle instruction terminates. This is a straightforward functional equivalence check under the assumption that ϕ_{noc} holds.

Proposition 1. *Let σ^h_j be a hardware state and σ^c_i the corresponding contract state under mapping $\mathcal{M}$. Furthermore, let σ^h_{j+k} =*

$\chi^k(\sigma_j^h)$ and $\sigma_{i+1}^c = \chi(\sigma_i^c)$ be the hardware and contract state after the execution of an instruction. Inductively,

$$\nexists\, \sigma_j^h, \sigma_i^c : \phi_{noc}\left(\sigma_j^h, \sigma_i^c\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^h, \sigma_i^c\right) \wedge \neg\phi_{rel}^{\mathcal{M}}\left(\sigma_{j+k}^h, \sigma_{i+1}^c\right) \quad (21)$$

implies the first hardware compliance condition (11) from Definition 6 under normal operating conditions.

Proposition 1 specifies how exactly this check is performed. We use an SMT solver to check for states that satisfy both ϕ_{noc} and $\phi_{\text{rel}}^{\mathcal{M}}$, but their successors break $\phi_{\text{rel}}^{\mathcal{M}}$. Any such case is a counterexample to the state similarity property of Definition 6. Otherwise the property is inductive, and we can use it as an assumption in all further verification checks.

We also check that $\phi_{\text{pub}}^{\mathcal{M}*}(\cdot)$ is inductive because all of the further verification targets require this as an assumption. We check the inductiveness by asking an SMT solver

$$\begin{aligned}&\nexists\sigma_j^h, \sigma_j^{h'}, \sigma_i^c, \sigma_i^{c'} : \phi_{\text{noc}}^*\left(\sigma_j^h, \sigma_j^{h'}, \sigma_i^c, \sigma_i^{c'}\right) \wedge \phi_{\text{rel}}^{\mathcal{M}}\left(\sigma_j^h, \sigma_i^c\right) \wedge \\ &\quad \phi_{\text{rel}}^{\mathcal{M}}\left(\sigma_j^{h'}, \sigma_i^{c'}\right) \wedge \phi_{\text{pub}}^{\mathcal{M}*}\left(\sigma_j^h, \sigma_j^{h'}\right) \wedge \neg\phi_{\text{pub}}^{\mathcal{M}*}\left(\sigma_{j+k}^h, \sigma_{j+k}^{h'}\right).\end{aligned} \quad (22)$$

If the solver is not able to find a solution, $\phi_{\text{pub}}^{\mathcal{M}*}$ is inductive and we assume $\phi_{\text{pub}}^{\mathcal{M}*}$ in addition to $\phi_{\text{rel}}^{\mathcal{M}}$ whenever we check properties under normal operating conditions over multiple executions.

4.4 Finding Modeling Functions for Gates

In this section, we introduce a method that finds a small number of contract registers from which the value of a hardware gate is modeled. We require this as an intermediate step whose aim it is to restrict the values a gate can have at the end of the previous instruction. Corollary 6 instantiates the general statements from Theorem 5 under normal operating conditions and presents a method for checking whether the value of a hardware g can be modeled by contract state bits θ_g.

Corollary 6. *Let σ_j^h be a hardware state and σ_i^c the corresponding contract state under mapping $\mathcal{M}$ that fulfill both (21) and (22). Furthermore, let $\sigma_{j+k-1}^h = \chi^{k-1}(\sigma_j^h)$ be the last hardware state before the instruction terminates, and $\sigma_{i+1}^c = \chi(\sigma_i^c)$ be the contract state after the instruction terminates. Under normal operating conditions, the value of gate g in cycle $k-1$ can be modeled from contract function θ_g if and only if*

$$\begin{aligned}&\nexists\sigma_j^h, \sigma_j^{h'}, \sigma_i^c, \sigma_i^{c'} : \phi_{noc}^*\left(\sigma_j^h, \sigma_j^{h'}, \sigma_i^c, \sigma_i^{c'}\right) \wedge \\ &\phi_{pub}^{\mathcal{M}*}\left(\sigma_j^h, \sigma_j^{h'}\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^h, \sigma_i^c\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^{h'}, \sigma_i^{c'}\right) \wedge \\ &\theta_g\left(\sigma_{i+1}^c\right) = \theta_g\left(\sigma_{i+1}^{c'}\right) \wedge g\left(\sigma_{j+k-1}^h\right) \neq g\left(\sigma_{j+k-1}^{h'}\right).\end{aligned} \quad (23)$$

Corollary 6 instantiates Theorem 5 under assumption of ϕ_{noc} and $\phi_{\text{pub}}^{\mathcal{M}*}$. Here, g is the function f_H to be modeled, θ_g is the function f_C whose results are used as inputs for the modeling function, and $\phi_{\text{rel}}^{\mathcal{M}}$ is the relation Ψ between the hardware and contract states.

However, not all functions θ_g are useful, so a function like $\theta_g(\sigma^c) = \sigma^c$ would not really restrict the initial values of g. Instead, we propose the greedy minimization procedure shown in Algorithm 1. Here, we first check whether the hardware gate g can be modeled from the complete contract state. If this fails the contract does not model the hardware properly and the verification fails because there is no way of telling the value of g when starting the execution of an instruction. Algorithm 1 iterates over all locations v^c in the contract state and checks whether they are needed for modeling g. In case they are not, *i.e.*, formula (23) is unsatisfirable, they are removed from θ_g. At the end, we have a (locally) minimal θ_g, where removing even one of its component locations would not allow to model gate g.

Algorithm 1 Greedy minimization of required state bits.

```
1: procedure FIND_THETA(gate $g$)
2:     $\Theta \leftarrow V^c$; $\theta_g \leftarrow$ concat $(\Theta)$;
3:     if formula (23) is SAT then
4:         error("gate $g$ cannot be modeled");
5:     for $v^c \in V^c$ do
6:         $\theta_g \leftarrow$ concat $(\Theta \setminus \{v^c\})$;
7:         if formula (23) is UNSAT then
8:             $\Theta \leftarrow \Theta \setminus \{v^c\}$;
9:     return $\theta_g$;
```

4.5 Verifying that Leaks are Modeled

Lastly, we verify that the hardware leakage produced during the execution of an instruction can be modeled by the contract leakage emitted during the execution of the same instruction. The set of transition leaks produced in the hardware, starting in state σ_j^h and executing a k-cycle instruction is given by

$$\mathcal{L}_{j,j+k}^h = \left\{\lambda_g\left(\sigma_{j+l-1}^h, \sigma_{j+l}^h\right) \mid g \in G, 0 \le l < k\right\}.$$

As established in Section 4.1, we analyze every hardware leak $\lambda_g(\sigma_{j+q-1}^h, \sigma_{j+q}^h)$ separately and show that there is a set of leak statements $\mathcal{L}^g \subseteq \mathcal{L}_i^c$ such that every λ_g can be modeled by $\lambda(\sigma_i^c) \in \mathcal{L}^g$ whenever the corresponding `leak` statement is reached in the contract, written as $\phi_{\text{emit}}(\sigma_i^c, \lambda)$. Here, we use the intermediate proof of $g(\sigma_{j-1}^h)$ being modeled by $\theta_g(\sigma_i^c)$, which we have shown through Corollary 6.

Proposition 2. *Let σ_{j-1}^h be the predecessor of hardware state $\sigma_j^h = \chi(\sigma_{j-1}^h)$, and σ_j^h be similar to contract state σ_i^c under mapping $\mathcal{M}$, fulfilling both (21) and (22). Furthermore, let $\sigma_{j+l}^h = \chi^l(\sigma_j^h)$ with $0 \le l < k$ be the hardware states reached throughout the execution of a k-cycle instruction. Let λ_g be the leakage function of a hardware gate g, and θ_g be a contract function such that (23) holds. Crutially, let $\mathcal{L}^g \subseteq \mathcal{L}_i^c$ be a set of contract leaks, such that for every $\lambda(\sigma_i^c) \in \mathcal{L}^g$:*

$$\begin{aligned}&\nexists\sigma_{j-}^h, \sigma_{j-1}^{h'}, \sigma_i^c, \sigma_i^{c'} : \phi_{noc}^*\left(\sigma_j^h, \sigma_j^{h'}, \sigma_i^c, \sigma_i^{c'}\right) \wedge \phi_{emit}\left(\sigma_i^c, \lambda\right) \wedge \\ &\quad \phi_{pub}^{\mathcal{M}*}\left(\sigma_j^h, \sigma_j^{h'}\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^h, \sigma_i^c\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^{h'}, \sigma_i^{c'}\right) \wedge \\ &\quad \left(\theta_g\left(\sigma_i^c\right) = \theta_g\left(\sigma_i^{c'}\right) \Rightarrow g\left(\sigma_{j-1}^h\right) = g\left(\sigma_{j-1}^{h'}\right)\right) \wedge \\ &\lambda(\sigma_i^c) = \lambda(\sigma_i^{c'}) \wedge \lambda_g\left(\sigma_{j+l-1}^h, \sigma_{j+l}^h\right) \neq \lambda_g\left(\sigma_{j+l-1}^{h'}, \sigma_{j+l}^{h'}\right).\end{aligned}$$

The leak function λ_g in cycle l of a k-cycle instruction is modeled by a single contract leak function λ under relation $\phi_{rel}^{\mathcal{M}}$ and normal *operating conditions ϕ_{noc}, according Definition 6 if*

$$\forall \sigma_j^h, \sigma_i^c : \phi_{noc}\left(\sigma_j^h, \sigma_i^c\right) \wedge \phi_{rel}^{\mathcal{M}}\left(\sigma_j^h, \sigma_i^c\right) \Rightarrow \bigvee_{\lambda(\sigma_i^c)\in\mathcal{L}^g} \phi_{emit}\left(\sigma_i^c, \lambda\right).$$

Again, the method outlined in Proposition 2 uses an SMT solver to show that the hardware cannot leak more information than the contract. If the solver is able to find a pair of states σ_j^h, σ_i^c for which the check fails, it has found a counterexample and the hardware does not comply with the leakage specified in the contract.

4.6 Modeling and Implementation

In this section, we briefly discuss the implementation and modeling details enabling our verification method. In particular, we discuss how all the formulas given to the SMT solver are constructed.

Unfolding circuits into SMT. Our verification method works with the processor netlist, and uses it to build the SMT formulas previously shown in Section 4. For the most part, this is standard procedure and has been elaborated in the model checking community. In short, the hardware state σ_j^h is represented symbolically using propositional variables. Each gate g in the processor is a symbolic expression of the variables representing hardware locations V^h. The expressions are generated by topologically iterating through the circuit and building the representation of each gate g from its inputs and gate type. With regard to clock cycles, the registers of the very first state σ_j^h, repsectively σ_{j-1}^h, are variables. In successor states σ_{j+l}^h, the registers are determined by their writebacks from the previous cycle. In a sense, we unfold the processor circuit symbolically k times for our verification.

Genoa to SMT translation. The translation of a contract to a SMT formula is based on an existing Sail back-end which allows to generate SMT formulas for custom predicates. However, the back-end cannot handle `leak` statements. We perform two code-rewriting passes from Genoa to Genoa. The first adds global state for each value in a `leak` statement and replaces the `leak` by an assignment to the respective global state. This reduces the Genoa DSL to the Sail subset supported by the SMT back-end. The second pass duplicates the variables representing contract state σ^c and leakages into prime and non-primed variants and duplicates the instruction-step function χ by rewriting it to operate on either σ_i^c or $\sigma_i^{c'}$ and resulting in σ_{i+1}^c, respectively $\sigma_{i+1}^{c'}$. Finally, a predicate is defined to ensure the initial and final states, including global leakages are preserved by the SMT back-end and that the predicates ϕ_{ret}, ϕ_{ret}^* and ϕ_{emit} hold. Our tool receives the SMT as input.

Gluing it all together. Configuration files play a central role in the generation of formulas. In particular, our verification procedure expects an input where all of the hardware locations are declared, and either mapped onto contract registers with $\phi_{\text{rel}}^{\mathcal{M}}$, subjected to developer assumptions ϕ_{dev}, port restrictions ϕ_{ports}^*, or instruction execution constraints ϕ_{instr}^l. As previously mentioned, everything specified in the configuration is heavily sanity-checked, making sure that execution still works properly, public signals $\phi_{\text{pub}}^{\mathcal{M}*}$ remain public, and every hardware location is declared. Similarly, intermediate results such as θ_g are cached in configuration files and checked upon loading a configuration.

5 VERIFICATION PROCESS

We apply the verification method presented to the IBEX processor and detail the process and results. IBEX is an open source RISC-V processor that supports the **I**nteger, **E**mbedded, **M**ultiplication, **C**ompressed and **Bi**t manipulation ISA extensions [32]. For the purpose of our paper, we mainly target the **E** extension, although adding support for the others is possible. The IBEX pipeline consists of two stages, Instruction Fetch (IF) and Instruction Decode/Execute (ID/EX). Computations take place in the ID/EX stage, which consists of a decoder, a controller, and the register file, which forward the data into the arithmetic-logic unit (ALU), and the load-store unit (LSU). In the same pipeline stage, and hence in the same clock cycle, the result is written back into the register file.

The verification requires two manual and four automated steps:

(1) Configuration of the processor by defining constraints
(2) Definition of a mapping between hardware and contract
(3) Automated sanity-check to ensure instructions defined in the contract can still execute in the processor under constraints
(4) Automated check of similarity for resulting states (Section 4.3)
(5) Automated check for gate modeling functions (Section 4.4)
(6) Automated check for leakage modeling functions (Section 4.5)

In case any of the steps fail, the verification framework produces a detailed counterexample explaining the verification failure. The developer must then adjust the configuration, the annotation, the contract, or even the processor in order to fix the problem and restart verification. Therefore, development and verification form a refinement loop producing improved contracts.

5.1 IBEX Configuration

We align the contract and the hardware by restricting the state of the processor throughout the execution of an instruction. In the configuration files, provided in Listing 7, we precisely constrain the values of all registers with regard to the current instruction length and analyzed cycle. For the verification, we look at instructions when they reach the ID stage. At this point, signal `instr_rdata_id` carries the instruction bits and must be set equivalent to the argument of `step_ibex` in the contract.

Additionally, we need to make sure that instructions are only retired in the last cycle $k-1$ of a k-cycle instruction by constraining `instr_id_done` to be $\top$ in the last cycle and $\bot$ otherwise. Similarly, we enforce that the next instruction is fetched exactly in cycle $k-1$ by constraining `fetch_valid` and `id_in_ready`. We assert that there are no outstanding errors caused by the previous instruction by constraining registers `lsu_err_q`, `pmp_err_q`, `branch_set_raw`, and `data_err_i` to be $\bot$. To make sure that the state machines in the LSU and control unit start off in a valid state when the instruction starts executing, we add several further constraints. Finally, we also assert that there is no reset through `rst_ni` and no interrupt signals `irq_*`, `debug_req_i` are triggered to match the developers expected behavior.

One of the main challenges in modeling the processor environment is the memory interface. Whenever the processor requests data by setting `data_req_o` to $\top$, the next cycle provides a grant with `data_rvalid_i` set to $\top$ and the corresponding read data being available at `data_rdata_i`. Here, we additionally require memory to only provide acknowledgement through `data_rvalid_i`

Listing 4: Common leakage occurring in every instruction.

```
1  function common_leakage(rs1_val, rs2_val) = {
2    leak(rs1_val, rs2_val, rf_pA, rf_pB,
3         mem_last_addr, mem_last_read);
4    rf_pA = rs1_val; rf_pB = rs2_val; /* read port */}
```

Listing 5: Specialized leakage occurring during loads.

```
5  function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits
       ↪ , addr: xlenbits, req_data: xlenbits) = {
6    leak(rf_pA, rf_pB, rs1_val, rs2_val);
7    leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
8    rf_pA = rs1_val; rf_pB = rs2_val;
9    leak(addr, req_data, mem_last_addr, mem_last_read);
10   mem_last_read = req_data; mem_last_addr = addr; }
```

if there was a request, and not provide any data on the input `data_rdata_i` otherwise. This is due to an oversight in IBEX, which causes the `data_rvalid_i` signal to overrule all other signals in the processor and ultimately issue an erroneous write-back.

5.2 Complete Power Contract for IBEX

We have proven that IBEX is compliant with the contract shown in Listings 1 to 6. In this section, we discuss the observed behavior and compare the findings to existing models for other architectures.

Most instructions have a common leakage pattern modeled in `common_leakage` in Listing 4. The IBEX processor combines the previous outputs of the register file (modeled in leakage states `rf_pA`, `rf_pB`) with the current outputs `rs1_val` and `rs2_val`, as well as the address and value of the last memory access `mem_last_addr` and `mem_last_read`. This leak statement models all transition leakage and value leakage produced in the ALU and the writeback logic. None of the operands in the `leak` statement can be removed without breaking compliance since distinct parts of IBEX cause these combinations The writeback logic causes additional combinations: ALU or branch instructions following a `LOAD` cause a transition between data from memory access and the current ALU result.

This common leakage covers leak effects previously discussed in related works. It models transition leakage produced in the ALU, whose source are the two read ports of the register file. Transitions in the first, respectively second, operand of instructions are well-known [36, 37]. Furthermore, the leakage is even caused by instructions which have no register operands like `LUI` (load unsigned immediate) (line 88 in Listing 6). The root cause of this effect is that the register file is always active, decoding specific instruction bits as register addresses. In the case of `LUI`, these bits are actually part of the immediate value. This effect was observed by Gigerl *et al.* [25] but we model it precisely.

The leakage of load instructions modeled by `load_leakage` (Listing 5) differs from all other instructions. Here, the leak statement in `common_leakage` can be broken down into smaller leak statements. First off, because the ALU is always active and performs its computations even when not necessary, the current and previous values of the register file outputs are combined in line 6. Similarly, the last register file outputs are combined with the loaded data and the load address within the writeback logic, as shown in line 7. Papagiannopoulos *et al.* [43] already observe leakage across the value read in two load instructions separated by an arbitrary number of unrelated instructions. We see this effect as well, as shown in line 9.

Surprisingly, IBEX does not expose transition leakages in subsequent memory writes separated by arbitrary other instructions, which was observed for several Arm architectures [9, 36]. Likely, this is related to IBEX not having additional registers in the memory path which many other processors have.

We emphasize that our model is provably complete for branching instructions as well (see Listing 6). The side-channel behavior of these ubiquitous instructions was not characterized so far.

5.3 Discussion

While we demonstrate our approach on the RISC-V IBEX core we emphasize that it is neither limited to RISC-V processors, nor the IBEX core. Verifying contract compliance for similar architectures and processors requires adapting the tool to their pipeline and properly configuring the verification procedure.

Our tool focuses on value leakage and transition leakage, while our theoretical framework supports arbitrary gate-level leakage. Extending our verification tool to include further effects such as glitches makes an interesting future research question, and could be achieved by extending the encoding of leakage from Section 4.5.

Our verification methodology and the contracts themselves support bitsliced and n-sliced masking [11], which are among the most popular implementation techniques for masked software. More exotic concepts like share-slicing require a bit-granular verification. However, since we confirm the empirical results of Gao *et al.* [22] and show the existence of bit-combinations within one 32-bit register, any share-sliced implementation is insecure on IBEX.

6 CONCLUSION

We introduced a methodology for creating software leakage models and proving their completeness based on the netlist of a CPU. Our rigorous approach allows us to treat the model as contract between the software and the hardware which provably guarantees end-to-end security: any implementation secure w.r.t. a contract is also secure on any compliant processor for all leakages exposed at gate-level. Overall the result significantly improve the secure construction of hardened software implementations.

Besides providing strong guarantees of side-channel resistance, easing the safe porting of programs to different CPUs and the most extensive modeling of different instructions's side-channel leakage, we think our approach could be beneficial for other applications as well. In particular, we believe it could be used for leakage emulators or statistical security evaluations that can be derived from the executable GENOA contracts.

ACKNOWLEDGMENTS

This work received funding from the Federal Ministry of Education and Research (BMBF) as part of the VE-Jupiter project (grant number 16ME0231K). This work was supported by the Austrian Research Promotion Agency (FFG) through the FERMION project (grant number 867542).

REFERENCES

[1] Arnold Abromeit, Florian Bache, Leon A. Becker, Marc Gourjon, Tim Güneysu, Sabrina Jorn, Amir Moradi, Maximilian Orlt, and Falk Schellenberg. Automated masking of software implementations on industrial microcontrollers. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 1006–1011. IEEE, 2021.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Kathryn E. Gray, Robert Norton-Wright, Christopher Pulte, Shaked Flur, and Peter Sewell, July 2021 (accessed January 12, 2022). https://raw.githubusercontent.com/rems-project/sail/sail2/manual.pdf.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. Minisail: A core calculus for sail, 2018 (accessed January 12, 2022). https://www.cl.cam.ac.uk/~mpew2/papers/minisail_anf.pdf.

[4] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[5] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[7] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 457–485. Springer, Heidelberg, April 2015.

[8] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

[9] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR TCHES*, 2021(2):189–228, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8792.

[10] Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. Abby: Automating the creation of fine-grained leakage models. Cryptology ePrint Archive, Report 2021/1569, 2021. https://ia.cr/2021/1569.

[11] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 311–341. Springer, Heidelberg, May 2020.

[12] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[13] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[14] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[15] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, Heidelberg, May 2014.

[16] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012.

[17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 1–18. Springer, Heidelberg, April 2017.

[18] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and ARM: friends or foes? In *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*, volume 10098 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2016.

[19] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[20] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR TCHES*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270.

[21] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In Andrei A. Bulatov and Arseny M. Shur, editors, *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings*, volume 7913 of *Lecture Notes in Computer Science*, pages 378–390. Springer, 2013.

[22] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *IACR TCHES*, 2020(1):152–174, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8396.

[23] Si Gao and Elisabeth Oswald. A novel completeness test and its application to side channel attacks and simulators. Cryptology ePrint Archive, Report 2021/756, 2021. https://ia.cr/2021/756.

[24] Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. Cryptology ePrint Archive, Report 2021/794, 2021. https://ia.cr/2021/794.

[25] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1468. USENIX Association, 2021.

[26] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2021.

[27] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, TIS '16, pages 3–3, New York, NY, USA, 2016. ACM.

[28] Hannes Gross, Stefan Mangard, and Thomas Korak. An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order. In Helena Handschuh, editor, *CT-RSA 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*, pages 95–112, Cham, 2017. Springer International Publishing.

[29] Vedad Hadzic and Roderick Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 1–10. IEEE, 2021.

[30] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[31] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[32] lowRISC. Ibex RISC-V Core. https://github.com/lowRISC/ibex.

[33] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, Heidelberg, February 2005.

[34] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *LNCS*, pages 157–171. Springer, Heidelberg, August / September 2005.

[35] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.

[36] Ben Marshall, Dan Page, and James Webb. Miracle: Micro-architectural leakage evaluation: A study of micro-architectural power leakage across many devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):175–220, Nov. 2021.

[37] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*,

pages 199–216. USENIX Association, August 2017.
[38] David McCann, Carolyn Whitnall, and Elisabeth Oswald. ELMO: emulating leaks for the ARM cortex-m0 without access to a side channel lab. *IACR Cryptol. ePrint Arch.*, page 517, 2016.
[39] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. *IACR Cryptol. ePrint Arch.*, 2020:1297, 2020.
[40] Prashanth Mundkur, Rishiyur S. Nikhil, Bluespec Inc, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Microsoft, Nathaniel Wesley Filardo, Peter Rugg, and Aril Computer Corp. Riscv sail model, August 2021 (accessed January 17, 2022). https://github.com/riscv/sail-riscv.
[41] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, December 2006.
[42] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, April 2011.
[43] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 282–297. Springer, Heidelberg, April 2017.
[44] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.
[45] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.
[46] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *CRYPTO 2015*, 2015.
[47] Peter Sewell. Isa formal spec public review, June 2020 (accessed January 14, 2022). https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review.
[48] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*. The Internet Society, 2021.

APPENDIX

6.1 RISC-V Model for IBEX

The missing parts of our contract for IBEX are depicted in Listing 6. Comment for reviewers: The non-blinded copyright holders do not deanonymize the authors of this paper.

Listing 6: Contract model of remaining instructions for IBEX.

```
/*==============================*/
/* RISCV Sail Model */
/* This Sail RISC-V architecture model, comprising all files
    ↪  and directories except for the snapshots of the
    ↪ Lem and Sail libraries in the prover_snapshots
    ↪ directory (which include copies of their licences),
    ↪  is subject to the BSD two-clause licence below. */
/* Copyright (c) 2017-2021 Prashanth Mundkur, Rishiyur S.
    ↪ Nikhil and Bluespec Inc., Jon French, Brian
    ↪ Campbell, Robert Norton-Wright, Alasdair Armstrong,
    ↪  Thomas Bauereiss, Shaked Flur, Christopher Pulte,
    ↪ Peter Sewell, Alexander Richardson, Hesham Almatary,
    ↪  Jessica Clarke, Microsoft, for contributions by
    ↪ Robert Norton-Wright and Nathaniel Wesley Filardo,
    ↪ Peter Rugg and Aril Computer Corp., for
    ↪ contributions by Scott Johnson */
/* Copyright 2020-2022 - TUHH, TU Graz */
/* All rights reserved. */
/* This software was developed by the above within the
    ↪ Rigorous Engineering of Mainstream Systems (REMS)
    ↪ project, partly funded by EPSRC grant EP/K008528/1,
    ↪  at the Universities of Cambridge and Edinburgh. */
/* This software was developed by SRI International and the
    ↪ University of Cambridge Computer Laboratory (
    ↪ Department of Computer Science and Technology)
    ↪ under DARPA/AFRL contract FA8650-18-C-7809 ("CIFV"),
    ↪  and under DARPA contract HR0011-18-C-0016 ("ECATS")
    ↪  as part of the DARPA SSITH research programme. */
/* This project has received funding from the European
    ↪ Research Council (ERC) under the European Union's
    ↪ Horizon 2020 research and innovation programme (
    ↪ grant agreement 789108, ELVER). */
/* This software has received funding from the Federal
    Ministry of Education and Research
    (BMBF) as part of the VE-Jupiter project grant
    16ME0231K. */
/* This work was supported by the Austrian Research
    Promotion Agency (FFG) through the FERMION project
    (grant number 867542). */
/* Redistribution and use in source and binary forms, with
    ↪ or without modification, are permitted provided
    ↪ that the following conditions are met: */
/* 1. Redistributions of source code must retain the above
    ↪ copyright notice, this list of conditions and the
    ↪ following disclaimer. */
/* 2. Redistributions in binary form must reproduce the
    ↪ above copyright notice, this list of conditions and
    ↪  the following disclaimer in the documentation and/
    ↪ or other materials provided with the distribution.
    ↪ */
/* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS
    ↪ ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
    ↪ INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    ↪ WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
    ↪ PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
    ↪ SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY
    ↪ DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
    ↪ OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
    ↪ LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
    ↪ SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
    ↪ BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
    ↪ THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    ↪ LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
    ↪ OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
    ↪ THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
    ↪ OF SUCH DAMAGE. */
/*==============================*/

val overwrite_leakage : (regidx, xlenbits) -> unit effect {
    ↪ rreg, leakage}
function overwrite_leakage(dest_idx : regidx, res : xlenbits
    ↪ ) = {
  let x1_n = if (dest_idx == 0b00001)
    then {res} else {x1} in leak(x1 , x1_n );
  let x2_n = if (dest_idx == 0b00010)
```

```
    then {res} else {x2} in leak(x2 , x2_n );
  let x3_n = if (dest_idx == 0b00011)
    then {res} else {x3} in leak(x3 , x3_n );
  let x4_n = if (dest_idx == 0b00100)
    then {res} else {x4} in leak(x4 , x4_n );
  let x5_n = if (dest_idx == 0b00101)
    then {res} else {x5} in leak(x5 , x5_n );
  let x6_n = if (dest_idx == 0b00110)
    then {res} else {x6} in leak(x6 , x6_n );
  let x7_n = if (dest_idx == 0b00111)
    then {res} else {x7} in leak(x7 , x7_n );
  let x8_n = if (dest_idx == 0b01000)
    then {res} else {x8} in leak(x8 , x8_n );
  let x9_n = if (dest_idx == 0b01001)
    then {res} else {x9} in leak(x9 , x9_n );
  let x10_n = if (dest_idx == 0b01010)
    then {res} else {x10} in leak(x10, x10_n);
  let x11_n = if (dest_idx == 0b01011)
    then {res} else {x11} in leak(x11, x11_n);
  let x12_n = if (dest_idx == 0b01100)
    then {res} else {x12} in leak(x12, x12_n);
  let x13_n = if (dest_idx == 0b01101)
    then {res} else {x13} in leak(x13, x13_n);
  let x14_n = if (dest_idx == 0b01110)
    then {res} else {x14} in leak(x14, x14_n);
  let x15_n = if (dest_idx == 0b01111)
    then {res} else {x15} in leak(x15, x15_n);
}

/* **************************** */
enum uop = {RISCV_LUI, RISCV_AUIPC}
union clause ast = UTYPE : (bits(20), regidx, uop)

mapping encdec_uop : uop <-> bits(7) = {
  RISCV_LUI <-> 0b0110111,
  RISCV_AUIPC <-> 0b0010111
}

mapping clause encdec = UTYPE(imm, rd, op)
  <-> imm @ rd @ encdec_uop(op)
  if (rd[4] == bitzero)

function clause execute UTYPE(imm, rd, op) = {
  let rs1_val = X(0b0 @ imm[6 .. 3]);
  let rs2_val = X(0b0 @ imm[11 .. 8]);
  common_leakage(rs1_val, rs2_val);

  let off : xlenbits = EXTS(imm @ 0x000);
  let ret : xlenbits = match op {
    RISCV_LUI => off,
    RISCV_AUIPC => get_arch_pc() + off
  };

  X(rd) = ret;
  RETIRE_SUCCESS
}

/* **************************** */
union clause ast = RISCV_JAL : (bits(21), regidx)

mapping clause encdec =
  RISCV_JAL(imm_19 @ imm_7_0 @ imm_8 @ imm_18_13 @ imm_12_9
       ↪ @ 0b0, rd)
  <-> imm_19 : bits(1) @ imm_18_13 : bits(6) @ imm_12_9 :
       ↪ bits(4) @ imm_8 : bits(1) @ imm_7_0 : bits(8) @ rd
       ↪  @ 0b1101111
  if (rd[4] == bitzero)

function clause execute (RISCV_JAL(imm, rd)) = {
  let rs1_val = X(0b0 @ imm[18 .. 15]);
  let rs2_val = X(0b0 @ imm[3 .. 1]
                @ subrange_bits(imm, 11, 11));
  common_leakage(rs1_val, rs2_val);

  let t : xlenbits = PC + EXTS(imm);
  let rd_next = get_next_pc();

  overwrite_leakage(rd, rd_next);
  X(rd) = rd_next;
  if t[1 .. 0] == 0b00 then {
    set_next_pc(t);
    RETIRE_SUCCESS
  } else RETIRE_FAIL
}

/* **************************** */
union clause ast =
  RISCV_JALR : (bits(12), regidx, regidx)

mapping clause encdec = RISCV_JALR(imm, rs1, rd)
  <-> imm @ rs1 @ 0b000 @ rd @ 0b1100111
  if (rs1[4] == bitzero & rd[4] == bitzero)

function clause execute (RISCV_JALR(imm, rs1, rd)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(0b0 @ imm[3..0]);
  common_leakage(rs1_val, rs2_val);

  let t : xlenbits =
    [(rs1_val + EXTS(imm)) with 0 = bitzero];
  if t[1 .. 0] == 0b00 then {
    overwrite_leakage(rd, get_next_pc());
    X(rd) = get_next_pc();
    set_next_pc(t);
    RETIRE_SUCCESS
  } else RETIRE_FAIL
}

/* **************************** */
enum bop = {RISCV_BEQ, RISCV_BNE, RISCV_BLT, RISCV_BGE,
     ↪ RISCV_BLTU, RISCV_BGEU}
union clause ast =
  BTYPE : (bits(13), regidx, regidx, bop)
```

```
mapping encdec_bop : bop <-> bits(3) = {
  RISCV_BEQ <-> 0b000,
  RISCV_BNE <-> 0b001,
  RISCV_BLT <-> 0b100,
  RISCV_BGE <-> 0b101,
  RISCV_BLTU <-> 0b110,
  RISCV_BGEU <-> 0b111
}

mapping clause encdec = BTYPE(imm7_6 @ imm5_0 @ imm7_5_0 @
     ↪ imm5_4_1 @ 0b0, rs2, rs1, op)
  <-> imm7_6 : bits(1) @ imm7_5_0 : bits(6) @ rs2 @ rs1 @
       ↪ encdec_bop(op) @ imm5_4_1 : bits(4) @ imm5_0 :
       ↪ bits(1) @ 0b1100011
  if (rs1[4] == bitzero & rs2[4] == bitzero)

function clause execute (BTYPE(imm, rs2, rs1, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  common_leakage(rs1_val, rs2_val);
  let taken : bool = match op {
    RISCV_BEQ => rs1_val == rs2_val,
    RISCV_BNE => rs1_val != rs2_val,
    RISCV_BLT => rs1_val <_s rs2_val,
    RISCV_BGE => rs1_val >=_s rs2_val,
    RISCV_BLTU => rs1_val <_u rs2_val,
    RISCV_BGEU => rs1_val >=_u rs2_val
  };

  let t : xlenbits = PC + EXTS(imm);
  if (t[1 .. 0] != 0b00) then
    return RETIRE_FAIL;
  if taken then { set_next_pc(t); };
  return RETIRE_SUCCESS
}

/* **************************** */
enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI,
     ↪  RISCV_ORI, RISCV_ANDI}
union clause ast =
  ITYPE : (bits(12), regidx, regidx, iop)

mapping encdec_iop : iop <-> bits(3) = {
  RISCV_ADDI <-> 0b000,
  RISCV_SLTI <-> 0b010,
  RISCV_SLTIU <-> 0b011,
  RISCV_ANDI <-> 0b111,
  RISCV_ORI <-> 0b110,
  RISCV_XORI <-> 0b100
}

mapping clause encdec = ITYPE(imm, rs1, rd, op)
  <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
  if (rs1[4] == bitzero) & (rd[4] == bitzero)

function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(0b0 @ imm[3 .. 0]);
  common_leakage(rs1_val, rs2_val);
  let immext : xlenbits = EXTS(imm);
  let result : xlenbits = match op {
    RISCV_ADDI => rs1_val + immext,
    RISCV_SLTI =>
      EXTZ(bool_to_bits(rs1_val <_s immext)),
    RISCV_SLTIU =>
      EXTZ(bool_to_bits(rs1_val <_u immext)),
    RISCV_ANDI => rs1_val & immext,
    RISCV_ORI => rs1_val | immext,
    RISCV_XORI => rs1_val ^ immext
  };
  overwrite_leakage(rd, result);
  X(rd) = result;
  RETIRE_SUCCESS
}

/* **************************** */
enum sop = {RISCV_SLLI, RISCV_SRLI, RISCV_SRAI}
union clause ast =
  SHIFTIOP : (bits(6), regidx, regidx, sop)

mapping encdec_sop : sop <-> bits(3) = {
  RISCV_SLLI <-> 0b001,
  RISCV_SRLI <-> 0b101,
  RISCV_SRAI <-> 0b101
}

mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SLLI)
  <-> 0b000000 @ shamt @ rs1 @ 0b001 @ rd @ 0b0010011
  if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] ==
       ↪ bitzero)
mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRLI)
  <-> 0b000000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
  if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] ==
       ↪ bitzero)
mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRAI)
  <-> 0b010000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
  if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] ==
       ↪ bitzero)

function clause execute (SHIFTIOP(shamt, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(0b0 @ shamt[3..0]);
  common_leakage(rs1_val, rs2_val);
  /* the decoder guard ensures that shamt[5] = 0 for RV32E
       ↪ */
  let result : xlenbits = match op {
    RISCV_SLLI => if sizeof(xlen) == 32
                  then rs1_val << shamt[4..0]
                  else rs1_val << shamt,
    RISCV_SRLI => if sizeof(xlen) == 32
                  then rs1_val >> shamt[4..0]
                  else rs1_val >> shamt,
    RISCV_SRAI => if sizeof(xlen) == 32
```

```
                  then shift_right_arith32(rs1_val, shamt
                       ↪ [4..0])
                  else shift_right_arith64(rs1_val, shamt)};
  overwrite_leakage(rd, result);
  X(rd) = result;
  RETIRE_SUCCESS
}

/* **************************** */
enum word_width = {BYTE, HALF, WORD, DOUBLE}
union clause ast = LOAD :
  (bits(12), regidx, regidx, bool, word_width, bool, bool)

mapping clause encdec = LOAD(imm, rs1, rd, is_unsigned, size
     ↪ , false, false)
  if ((word_width_bytes(size) < sizeof(xlen_bytes)) | (
       ↪ not_bool(is_unsigned) & word_width_bytes(size) <=
       ↪ sizeof(xlen_bytes))) & (rs1[4] == bitzero) & (rd
       ↪ [4] == bitzero)
  <-> imm @ rs1 @ bool_bits(is_unsigned) @ size_bits(size) @
       ↪  rd @ 0b0000011
  if ((word_width_bytes(size) < sizeof(xlen_bytes)) | (
       ↪ not_bool(is_unsigned) & word_width_bytes(size) <=
       ↪ sizeof(xlen_bytes))) & (rs1[4] == bitzero) & (rd
       ↪ [4] == bitzero)

function aligned(vaddr : xlenbits, width : word_width) ->
     ↪ bool =
  { width == BYTE | (width == HALF & vaddr[0] == bitzero) |
       ↪ (width == WORD & vaddr[1 .. 0] == 0b00) }

val load_leakage : (xlenbits, xlenbits, xlenbits, xlenbits)
  -> unit effect {rreg, wreg, leakage}
function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits
     ↪ , addr: xlenbits, req_data: xlenbits) = {
  // as in common_leakage
  leak(rf_pA, rf_pB, rs1_val, rs2_val);
  leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
  rf_pA = rs1_val;
  rf_pB = rs2_val;
  leak(addr, req_data, mem_last_addr, mem_last_read);
  mem_last_read = req_data;
  mem_last_addr = addr;
}

function clause execute(LOAD(imm, rs1, rd, is_unsigned,
     ↪ width, aq, rl)) = {
  let offset : xlenbits = EXTS(imm);
  let rs1_val = X(rs1);
  let rs2_val = X(0b0 @ imm[3 .. 0]);
  let addr = rs1_val + offset;
  let req_addr = addr[(sizeof(xlen) - 1) .. 2] @ 0b00;
  let req_data = read_mem(Read_plain, sizeof(xlen), req_addr
       ↪ , 4);
  load_leakage(rs1_val, rs2_val, addr, req_data);
  let req_byte : bits(8) = match (addr[1 .. 0]) {
    0b00 => req_data[ 7 .. 0],
    0b01 => req_data[15 .. 8],
    0b10 => req_data[23 .. 16],
    0b11 => req_data[31 .. 24]};
  let req_half : bits(16) = match (addr[1]) {
    bitzero => req_data[15 .. 0],
    bitone => req_data[31 .. 16]};
  match (width, addr[1 .. 0]) {
    (BYTE, _) => process_load(rd, addr, req_byte,
         ↪ is_unsigned),
    (HALF, 0b00) => process_load(rd, addr, req_half,
         ↪ is_unsigned),
    (HALF, 0b10) => process_load(rd, addr, req_half,
         ↪ is_unsigned),
    (WORD, 0b00) => process_load(rd, addr, req_data,
         ↪ is_unsigned),
    (_, _) => RETIRE_FAIL // takes care of misaligned}
}
/* **************************** */
union clause ast = STORE :
  (bits(12), regidx, regidx, word_width, bool, bool)

mapping clause encdec = STORE(imm7 @ imm5, rs2, rs1, size,
     ↪ false, false)
  if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1
       ↪ [4] == bitzero) & (rs2[4] == bitzero)
  <-> imm7 : bits(7) @ rs2 @ rs1 @ 0b0 @ size_bits(size) @
       ↪ imm5 : bits(5) @ 0b0100011
  if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1
       ↪ [4] == bitzero) & (rs2[4] == bitzero)

function clause execute (STORE(imm, rs2, rs1, width, aq, rl)
     ↪ ) = {
  let offset : xlenbits = EXTS(imm);
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  common_leakage(rs1_val, rs2_val);
  let addr = rs1_val + offset;
  // address comptation and register file access leakage
  leak(mem_last_addr, addr);
  mem_last_addr = addr;
  if aligned(addr, width) then {
    let result = rs2_val;
    overwrite_leakage(0b00000, result);
    let success : bool = match(width) {
      BYTE => write_mem(Write_plain, sizeof(xlen), addr, 1,
           ↪ result[7..0]),
      HALF => write_mem(Write_plain, sizeof(xlen), addr, 2,
           ↪ result[15..0]),
      WORD => write_mem(Write_plain, sizeof(xlen), addr, 4,
           ↪ result),
      _ => false};
    if success then {RETIRE_SUCCESS} else {RETIRE_FAIL}
  } else { RETIRE_FAIL }
}
/* **************************** */
mapping clause encdec = ILLEGAL(s) <-> s
function clause execute (ILLEGAL(s)) =
  { return RETIRE_FAIL }
```

6.2 IBEX Configuration

In the following, we give the configuration file that specifies the mapping and normal operating conditions simultaneously.

Listing 7: IBEX configuration file

```
// Power Contract for IBEX
//
// Copyright (c) 2020-2022 - TUHH, TU Graz
//
// All rights reserved.
//
// This software has received funding from the Federal
    Ministry of Education and Research (BMBF) as part of
    the VE-Jupiter project grant 16ME0231K.
//
// This work was supported by the Austrian Research
    Promotion Agency (FFG) through the FERMION project
    (grant number 867542).
//
// Redistribution and use in source and binary forms,
// with or without modification, are permitted provided
// that the following conditions are met:
// 1. Redistributions of source code must retain the
// above copyright notice, this list of conditions
// and the following disclaimer.
// 2. Redistributions in binary form must reproduce the
// above copyright notice, this list of conditions
// and the following disclaimer in the documentation
// and/or other materials provided with the
// distribution.
//
// THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND
// CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR
// IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
// TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
// AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR
// CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
// INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
// OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
// LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
// OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS
// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
// SUCH DAMAGE.

/////////////////////////////////////////////////////
// This file contains the configuration of our tool.
// It specifies
// - the state modeled in the contract (registers, memory,
    ↪ leakage state)
// - the registers of the IBEX processor (registers and
    ↪ memory)
// - a mapping between the states
// - which states may contain sensitive data
// - conditions which have to hold before/during execution
    ↪ of an instruction
// - which HW and contract state is printed in
    ↪ counterexamples
/////////////////////////////////////////////////////

/////////////////////////////////////////////////////
// specification of architectural registers and HW/CT
    ↪ mapping
/////////////////////////////////////////////////////
// PC
contract register PC BitVec 32
hardware public u_ibex_core.pc_id
mapping register PC u_ibex_core.pc_id

// next PC
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
    ↪ prefetch_buffer_i.fifo_i.instr_addr_q
contract register nextPC BitVec 32
mapping register nextPC u_ibex_core.if_stage_i.
    ↪ gen_prefetch_buffer.prefetch_buffer_i.fifo_i.
    ↪ instr_addr_q

// REGISTERS
contract register x1 BitVec 32
contract register x2 BitVec 32
contract register x3 BitVec 32
contract register x4 BitVec 32
contract register x5 BitVec 32
contract register x6 BitVec 32
contract register x7 BitVec 32
contract register x8 BitVec 32
contract register x9 BitVec 32
contract register x10 BitVec 32
contract register x11 BitVec 32
contract register x12 BitVec 32
contract register x13 BitVec 32
contract register x14 BitVec 32
contract register x15 BitVec 32
hardware variable register_file_i.rf_reg_q[1]
hardware variable register_file_i.rf_reg_q[2]
hardware variable register_file_i.rf_reg_q[3]
hardware variable register_file_i.rf_reg_q[4]
hardware variable register_file_i.rf_reg_q[5]
hardware variable register_file_i.rf_reg_q[6]
hardware variable register_file_i.rf_reg_q[7]
hardware variable register_file_i.rf_reg_q[8]
hardware variable register_file_i.rf_reg_q[9]
hardware variable register_file_i.rf_reg_q[10]
hardware variable register_file_i.rf_reg_q[11]
hardware variable register_file_i.rf_reg_q[12]
hardware variable register_file_i.rf_reg_q[13]
hardware variable register_file_i.rf_reg_q[14]
hardware variable register_file_i.rf_reg_q[15]
mapping register x1 register_file_i.rf_reg_q[1]
mapping register x2 register_file_i.rf_reg_q[2]
mapping register x3 register_file_i.rf_reg_q[3]
mapping register x4 register_file_i.rf_reg_q[4]
mapping register x5 register_file_i.rf_reg_q[5]
mapping register x6 register_file_i.rf_reg_q[6]
mapping register x7 register_file_i.rf_reg_q[7]
mapping register x8 register_file_i.rf_reg_q[8]
mapping register x9 register_file_i.rf_reg_q[9]
mapping register x10 register_file_i.rf_reg_q[10]
mapping register x11 register_file_i.rf_reg_q[11]
mapping register x12 register_file_i.rf_reg_q[12]
mapping register x13 register_file_i.rf_reg_q[13]
mapping register x14 register_file_i.rf_reg_q[14]
mapping register x15 register_file_i.rf_reg_q[15]

contract opcode op BitVec 32
// instruction bits for the instruction whose last execution
    ↪  cycle is this cycle
// only true if the assertion for the valid_d is present
hardware opcode u_ibex_core.instr_rdata_id

// memory request name_contract name_hardware
contract register read_val_1 BitVec 32
contract register read_addr_1 BitVec 32
hardware variable data_rdata_i

memory raddr u_ibex_core.load_store_unit_i.adder_result_ex_i
    ↪  read_addr_1
memory rdata data_rdata_i read_val_1
memory req data_req_o
memory gnt data_gnt_i
memory ack data_rvalid_i
memory we data_we_o

////////////////////////////////////////////////////
// Non-regport signals that must be constrained
////////////////////////////////////////////////////
// fetching next instruction was successful, ready to
    ↪ continue execution
hardware const@end-1 u_ibex_core.if_stage_i.fetch_valid 0b1
hardware const@pre u_ibex_core.if_stage_i.fetch_valid 0b1
// do not load a new instruction until last cycle
hardware const@start:end-1 u_ibex_core.id_stage_i.
    ↪ id_in_ready_o 0b0
// make sure that nothing retires before the end of the last
    ↪  cycle
hardware const@start:end-1 u_ibex_core.instr_id_done 0b0
// make sure that an instruction has its last cycle in our
    ↪ last cycle
hardware const@end-1 u_ibex_core.instr_id_done 0b1
hardware const@pre u_ibex_core.instr_id_done 0b1
hardware const@start u_ibex_core.id_stage_i.decoder_i.
    ↪ illegal_insn 0b0

////////////////////////////////////////////////////
// important signals that must be constrained
////////////////////////////////////////////////////
// never trigger a reset of the core
hardware public rst_ni
hardware const@pre: rst_ni 0b1
// make sure that initially, the ID FSM is in state
    ↪ instr_first_cycle_i
// this means that we look at the case where we started
    ↪ executing in 0th cycle
hardware public u_ibex_core.id_stage_i.id_fsm_q
hardware const@start u_ibex_core.id_stage_i.id_fsm_q 0b0
// no compressed (valid or invalid) instructions at the
    ↪ output of instruction fetch stage
hardware public u_ibex_core.if_stage_i.instr_new_id_q
hardware public u_ibex_core.if_stage_i.
    ↪ instr_is_compressed_id_o
hardware const@start u_ibex_core.if_stage_i.
    ↪ instr_is_compressed_id_o 0b0
hardware public u_ibex_core.if_stage_i.illegal_c_insn_id_o
hardware const@start u_ibex_core.if_stage_i.
    ↪ illegal_c_insn_id_o 0b0
// this is a hidden assumption made by IBEX developers
hardware public u_ibex_core.if_stage_i.instr_rdata_alu_id_o
hardware public u_ibex_core.if_stage_i.instr_rdata_id_o
// this encodes pre cycle and first cycle assumptions
hardware equiv@pre:start+1 u_ibex_core.if_stage_i.
    ↪ instr_rdata_id_o u_ibex_core.if_stage_i.
    ↪ instr_rdata_alu_id_o

////////////////////////////////////////////////////
// annotation of input ports of ibex_top
////////////////////////////////////////////////////
hardware public clk_i
hardware public ram_cfg_i
hardware public test_en_i
hardware public hart_id_i
hardware public boot_addr_i
// Instruction memory interface
hardware public instr_gnt_i
hardware public instr_rvalid_i
hardware public instr_rdata_i
hardware public instr_err_i
hardware public data_gnt_i
hardware public data_rvalid_i
hardware public data_err_i
hardware const@pre: data_err_i 0b0
// interrupts
hardware public irq_software_i
hardware public irq_timer_i
hardware public irq_external_i
hardware public irq_fast_i
hardware public irq_nm_i
// disabling interrupts
hardware const@pre: irq_software_i 0b0
hardware const@pre: irq_timer_i 0b0
hardware const@pre: irq_external_i 0b0
hardware const@pre: irq_fast_i 0b000000000000000
hardware const@pre: irq_nm_i 0b0
// core debug
hardware public debug_req_i
hardware const@pre: debug_req_i 0b0
hardware public fetch_enable_i
hardware public scan_rst_ni

//////////////////////////////////////
// annotate internal state of ibex
//////////////////////////////////////
hardware public core_busy_q
hardware public u_ibex_core.instr_fetch_err
hardware public u_ibex_core.instr_fetch_err_plus2
// instructions in the prefetch fifo
hardware public u_ibex_core.if_stage_i.instr_valid_id_q
hardware public u_ibex_core.if_stage_i.instr_rdata_c_id_o
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fifo_i.rdata_q0
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fifo_i.rdata_q1
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fifo_i.rdata_q2
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fifo_i.err_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fifo_i.valid_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.rdata_pmp_err_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.discard_req_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.branch_discard_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.rdata_outstanding_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.fetch_addr_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.prefetch_buffer_i.stored_addr_q

// instruction decode
hardware public u_ibex_core.id_stage_i.controller_i.ctrl_fsm_cs
hardware public u_ibex_core.id_stage_i.controller_i.load_err_q
hardware public u_ibex_core.id_stage_i.controller_i.store_err_q
hardware public u_ibex_core.id_stage_i.controller_i.exc_req_q
hardware public u_ibex_core.id_stage_i.branch_set_raw
hardware const@start u_ibex_core.id_stage_i.branch_set_raw 0b0
hardware public u_ibex_core.id_stage_i.branch_jump_set_done_q
hardware public u_ibex_core.load_store_unit_i.data_we_q

// since data_pmp_err_i is 0, this should not be 1
hardware public u_ibex_core.load_store_unit_i.pmp_err_q
hardware const@start u_ibex_core.load_store_unit_i.pmp_err_q 0b0
// since data_err_i is never 1 and pmp_err_q is also not 1, this must be 0
hardware public u_ibex_core.load_store_unit_i.lsu_err_q
hardware const@start u_ibex_core.load_store_unit_i.lsu_err_q 0b0
hardware public u_ibex_core.load_store_unit_i.handle_misaligned_q
hardware public u_ibex_core.id_stage_i.controller_i.illegal_insn_q
hardware public u_ibex_core.id_stage_i.controller_i.do_single_step_q
hardware public u_ibex_core.id_stage_i.controller_i.enter_debug_mode_prio_q
// LSU register handling misaligned memory accesses which are not allowed by the contract
hardware public u_ibex_core.load_store_unit_i.rdata_q
contract leakagestate mem_last_read BitVec 32
// Must be idle when new instruction reaches ID/EX
hardware public u_ibex_core.load_store_unit_i.ls_fsm_cs
hardware const@start u_ibex_core.load_store_unit_i.ls_fsm_cs 0b000
hardware variable u_ibex_core.load_store_unit_i.addr_last_q
hardware variable u_ibex_core.load_store_unit_i.rdata_offset_q
contract leakagestate mem_last_addr BitVec 32
mapping leakagestate mem_last_addr u_ibex_core.load_store_unit_i.addr_last_q
mapping leakagestate mem_last_addr u_ibex_core.load_store_unit_i.rdata_offset_q
hardware public u_ibex_core.load_store_unit_i.data_type_q
hardware public u_ibex_core.load_store_unit_i.data_sign_ext_q

// system registers
hardware public u_ibex_core.cs_registers_i.mie_q
hardware public u_ibex_core.cs_registers_i.mtval_q
hardware public u_ibex_core.cs_registers_i.mcause_q
hardware public u_ibex_core.cs_registers_i.mscratch_q
hardware public u_ibex_core.cs_registers_i.dscratch0_q
hardware public u_ibex_core.cs_registers_i.dscratch1_q
hardware public u_ibex_core.cs_registers_i.mstack_q
hardware public u_ibex_core.cs_registers_i.mstack_cause_q
hardware public u_ibex_core.cs_registers_i.mstack_epc_q
hardware public u_ibex_core.cs_registers_i.mstatus_q
hardware public u_ibex_core.cs_registers_i.dcsr_q
hardware const@start u_ibex_core.cs_registers_i.dcsr_q 0b00000000000000000000000000000000
hardware public u_ibex_core.cs_registers_i.mhpmcounter[0]
hardware public u_ibex_core.cs_registers_i.mhpmcounter[1]
hardware public u_ibex_core.cs_registers_i.mhpmcounter[2]
hardware public u_ibex_core.cs_registers_i.mcountinhibit
hardware public u_ibex_core.csr_depc
hardware public u_ibex_core.csr_mtvec
hardware public u_ibex_core.csr_mepc
hardware public u_ibex_core.dummy_instr_en
hardware public u_ibex_core.dummy_instr_mask
hardware public u_ibex_core.data_ind_timing
hardware public u_ibex_core.icache_enable
hardware public u_ibex_core.debug_mode
hardware public u_ibex_core.priv_mode_id
hardware public u_ibex_core.nmi_mode

contract leakagestate rf_pA BitVec 32
contract leakagestate rf_pB BitVec 32
```