IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 0, No. 0, pp. 0–0. DOI:10.46586/tches.v0.i0.0-0

Closing the Gap: Leakage Contracts for Processors with Transitions and Glitches [*]

Johannes Haring[1], Vedad Hadžić[1] and Roderick Bloem[1]

Graz University of Technology, Graz, Austria, first.last@iaik.tugraz.at

Abstract. Security verification of masked software implementations of cryptographic algorithms must account for microarchitectural side-effects of CPUs. Leakage contracts were proposed to provide a formal separation between hardware and software verification, ensuring interoperability and end-to-end security for independently verified components. However, previously proposed leakage contracts did not consider a class of ephemeral hardware effects called glitches, which leaves a considerable gap between security models and the capabilities of real-world attackers. We address this issue by extending the model for leakage contracts to account for glitches and transitions. We further present the first end-to-end verification tool for transient leakage contracts. Our hardware and software verification rely on the same contract as a single source of truth, facilitating fully machine-checked verification from the hardware gate level to the software. By allowing contracts to be written in the C programming language we make power contracts more accessible and intuitive for system-level engineers. To showcase the efficacy of our approach, we apply it to the RISC-V Ibex core. We show that it is possible to write a power contract for Ibex without any modifications to the hardware design. Using this contract, we prove end-to-end security between masked software and gate-level hardware.

Keywords: Hardware-Software Contracts · Power Analysis · Co-Verification

1 Introduction

Physical side-channel attacks such as power [KJJ99] or electromagnetic emanation [QS01] analysis violate the security assumptions of cryptographic algorithms, allowing attackers to extract sensitive information from devices they have physical access to. A widely used countermeasure against such attacks on physical implementations of cryptographic algorithms is masking [CRB+16, GMK16, ISW03]. Like any secret-sharing technique, masking splits an algorithm's inputs and outputs into multiple intermediate values called shares. When splitting a secret into $d \geq t + 1$ random shares, any set of at most t intermediate values of the computation must be statistically independent of any secret value, implying that an attacker with access to at most t intermediate values cannot learn any information about the secret.

In their seminal paper, Ishai et al. [ISW03] introduce a method to transform arbitrary algorithms into functionally equivalent masked algorithms. Their method requires that each successive computation emit leakage independent of previous computations. However, this is not the case in typical hardware implementations. Due to various physical effects occurring in integrated circuits, the power consumption of a design correlates with computations from consecutive clock cycles, potentially breaking the security guarantees of concrete

[*]This work was supported by the *Austrian Science Fund* (FWF SFB project SPyCoDe 10.55776/F85), and the *Austrian Research Promotion Agency* (FFG) through the BRIDGE project AWARE.

implementations. To address this issue, extended leakage models were proposed that incorporate physical effects to prevent the reduction of security order [FGP+18]. Software-based masking implementations additionally suffer from leakages caused by microarchitectural components, such as register files, load store units, and arithmetic logic units. While results of ephemeral computations in those components are immediately discarded, they can cause unexpected leakage behavior, further reducing security guarantees.

The underlying physical phenomena causing leakage in integrated circuits, known as physical defaults [dGPdlP+16, MMT20, GHP+21], are well known and extensively discussed in circuit design literature. The most straightforward effect is the correlation between static power consumption and the values present in the gates of a circuit design. Early publications [ISW03] formalize this effect as *value leakage*, which allows an attacker to observe the values of computations at the end of a clock cycle.

Besides value leakage, other physical defaults stem from the dynamic power consumption of circuit designs. A CMOS gate predominantly draws power when its output state switches. An attacker may also be able to discern whether the state switches from high to low or the opposite. *Transition leakage* formally describes this behavior as leaking the value of the gate's output both at the start and end of a clock cycle, allowing an attacker to observe both values [CGP+12, BGG+14].

This model assumes that transitions only happen once per clock cycle. More precise models account for state transitions caused by inconsistent signal propagation timing. Such ephemeral state transitions that happen before a gate switches to the correct output are called *glitches* [MPG05, BGG+14]. They can happen multiple times per clock cycle, significantly impacting the design's power consumption. There exist digital circuit designs that prevent glitches, however, they often have significant downsides, such as increased area requirements. As glitches are ubiquitous in hardware designs, it is vital to consider them in power-side-channel analysis.

To verify the absence of order-reducing leakage effects, some works propose to use the ISA description to estimate the leakage effects inside of CPUs. Unfortunately, the microarchitectural details of a CPU design significantly impact the actual leakage behavior, which renders models purely based on the ISA incomplete, reducing the actual security order by up to a factor of 2 [BGG+14] and for pipelined processor designs, the security order scales with the number of pipeline stages [GPM21]. Recently, leakage contracts were proposed as a way of accurately formalizing the leakage behavior of CPUs by combining the functional ISA specification with the specification of gate-level leakage characteristics. Software and hardware compliance with a contract can be verified independently, guaranteeing that any compliant program can be securely executed on any compliant CPU. While this allows for very efficient and flexible verification, previously proposed leakage contract formalisms [BGG+22] lack support for glitches and are, therefore, inherently incomplete.

Our Contribution. We address the issue of previously unconsidered physical defaults by introducing an extended contract language that accounts for transitions and glitches, thereby closing the gap between formal verification and physical hardware effects. We also introduce an end-to-end power-contract-verification tool that harnesses user-supplied contracts as a single source of truth for hardware and software verification. Our tool enables automated end-to-end verification from gate-level leakage to software.

1. Extended Gate-level Leakage Model. We extend the leakage model of Bloem et al. [BGG+22] to account for glitches. Previously proposed glitch models [FGP+18] define glitches in a fashion that allows glitch propagation through the entire cone of a gate. We propose a tighter model that accounts for scenarios in which glitch propagation is terminated, considerably reducing the leakage that can be caused by glitches. We identify such scenarios by utilizing knowledge about the concrete values of a gate's inputs.

We obtain our model by formalizing the conditions for glitch propagation. We first formally describe the leakage of common CMOS gates. Deriving from our leakage model, we obtain a notion of signal stability for each type of gate. We then define a notion of detectable difference for each type of gate with respect to the values and stability of its inputs. Applying this notion recursively to the netlist of a hardware design allows our model to capture any potentially occurring glitches. The resulting security notion can be checked using the same 2-safety hyperproperty verification introduced by Bloem et al. [BGG+22].

2. End-to-End Verification. To showcase the feasibility of our method, we introduce a tool[1] to verify compliance of hardware and software with glitch-aware leakage contracts.

During hardware verification, we show that for every observable leak in hardware, there is a corresponding leak emitted by the contract. This property can be easily expressed as an SMT problem with bitvector theories [BGG+22]. We utilize the CBMC frontend [CKL04] to obtain an SMT encoding of the contract, which is then verified against an SMT encoding of the CPU circuit via the Boolector SMT solver [NPWB18].

To verify software compliance with a contract, the user needs to provide the assembly code of the program and information about the location of secret values in the memory and registers. Verification then happens in two steps. First, all leakages emitted by the contract are recorded by executing the program according to the semantics of the contract. Second, to show order t probing security, all combinations of at most t observations are proven to be independent from any of the secrets. For this, a verification approach based on the approximation of Fourier coefficients [BGI+18] is used. The resulting problem can then be efficiently solved with modern SAT solvers.

We chose the C programming language for our contracts because it is widely used in low-level engineering. As contracts are intended to aid the development of secure masked software implementations, choosing a language most system engineers are already familiar with is beneficial. Due to the prevalence of C/C++, implementations for many instruction-set architectures already exist, reducing the effort of contract creation.

3. Case Study. To illustrate the effectiveness of our approach, we apply it to the RISC-V Ibex core [low], which is widely used in embedded systems research. In our experiments, we apply our approach to multiple masked implementations of gadgets and ciphers for several masking orders. Our results demonstrate that our method can be significantly faster than state-of-the-art methods that directly verify software against hardware netlists, considering that hardware verification has to be performed only once.

Related Work. Previous papers addressing power-side-channel leakage typically approach the issue either empirically, by performing power analysis on physical devices, or by means of formal verification.

Empirical analysis is conducted by measuring power consumption and analyzing the resulting power traces with statistical methods. This approach enables the study of observable leakage effects, which can guide the development of more resilient masked implementations [MOW17, PV17, GO22, SCS+21, BDM+20]. Methods based on empirical or generalized leakage models rely on a high practical effort to obtain confidence in the security assurances they provide.

Formal verification approaches prove security with respect to specific masking security notions, which provide an abstraction of the leakage effects observable in the real world.

MaskVerif [BBC+19] is a software and hardware masking verification tool that can verify probing security and the security notions t–NI and t–SNI. scVerif [BGG+21] extends the verification approach, allowing the verification of user-provided leakage models. Both tools require the leakage model to cover all physical defaults comprehensively.

Coco [GHP+21] is a masking verification tool that directly models the leakage of every gate in the hardware circuit netlist, avoiding the reliance on generalized leakage models. Their extended hardware leakage model covers glitches and transitions, giving strong guarantees about the completeness of the verification. Due to the direct verification of software against hardware, the approach is more time consuming and must be performed

[1] https://github.com/IAIK/glitch-contracts

separately for every program on every version of every compatible CPU design. In contrast, the modular verification approach of leakage contracts allows to test each program and CPU design individually, reducing the overall verification effort while maintaining security.

Bloem et al. [BGG+22] propose leakage contracts to specify the leakage behavior of CPUs formally. They allow the verification to be split into two parts. Hardware and software are independently checked against the contract, ensuring that any compliant software can be executed securely on any compliant processor. They introduce Genoa, a DSL for the specification of leakage contracts. To verify software against a contract, they rely on scVerif. In their evaluation, they manually extract the leakage model from their Genoa contract and translate it into the DSL of scVerif. While the overall verification process is sound, this manual translation makes the process vulnerable to user errors. Moreover, they only consider transition leakage, leaving out leakage caused by glitches.

Wang et al. [WMvG+23] propose a system with a similar notion of leakage contracts. Their contract notion is aimed at verifying classic non-interference properties to prove resistance against software-visible microarchitectural leaks. This leads to a different notion of contract satisfaction than ours, as we use contracts to verify power-side-channel security with a focus on glitches and transitions.

2 Preliminaries

In this section, we present the foundational concepts pertaining to side-channel security. Section 2.1 provides a comprehensive overview of the masking countermeasure and outlines the formal definitions of provable side-channel resilience. Section 2.2 formally defines contracts and the abstract semantics of their execution. Section 2.3 establishes a model for hardware circuits and their associated power side-channel leakage. Section 2.4 introduces the formalisms of physical defaults into gate-level leakage models. Section 2.5 formally defines compliance of a hardware circuit with a contract. Section 2.6 gives an overview of the steps involved to verify hardware compliance. Finally, Section 2.7 formally defines compliance of software implementations with a contract.

2.1 Masked Computation and Security

In the following, we formally define masked computations. Masking is a countermeasure against power-analysis attacks, where all secret values are encoded as a linear combination of a set of d *shares*, where d is higher than the expected attack order t. The rationale, which we formalize later, is that an attacker with access to at most t intermediate values in a computation cannot learn any secrets no matter how many times they observe the masked computation.

In the following, we use lowercase letters (e.g., x) for variables, and subscripts for secret- and share indices (e.g., $x_{i,j}$ is the j-th share of the i-th secret variable x_i). We denote a tuple of unshared variables with $\boldsymbol{x} := \langle x_0, \dots, x_{n-1} \rangle$. A secret variable x_i can be encoded (masked) as a tuple of shares $\overline{x}_i := \langle x_{i,0}, \dots, x_{i,d-1} \rangle$, with $x_i = x_{i,0} \oplus \dots \oplus x_{i,d-1}$, where $\oplus$ is addition in the respective field. Finally, we use the shorthand $\overline{\boldsymbol{x}} := \overline{x}_0 \parallel \dots \parallel \overline{x}_{n-1}$ for the concatenation of share tuples $\overline{x}_i$ corresponding to unshared variables x_i in $\boldsymbol{x}$.

Formally, a *masked* computation is a function $f : \langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}, \boldsymbol{q}, \boldsymbol{\lambda} \rangle$ mapping an input tuple of shared secrets $\overline{\boldsymbol{x}}$, public values $\boldsymbol{p}$, and masks $\boldsymbol{r}$ to an output tuple of shared secrets $\overline{\boldsymbol{y}}$, public values $\boldsymbol{q}$, and leaks $\boldsymbol{\lambda}$. Each output $y_{i,j}$, q_i, and λ_i is computed by a corresponding coordinate function of f given the inputs, e.g., $y_{i,j} := y^f_{i,j}(\overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r})$. The leaks λ_i are of particular interest for the security of the masked computation, because they represent the values an adversary is able to observe, e.g., through power side-channel measurements. Note that the individual leaks λ_i are not necessarily in the same field as each other or the secrets, e.g., they can represent concatenations of a varying number of field elements. The goal of the adversary is to recover information about the input secrets $\boldsymbol{x}_i$ from at most t observed leaks λ_i.

In order to properly specify what it means for a masked computation to be secure, we first have to interpret its inputs and outputs as random variables. Each random variable x, respectively tuple of random variables like $\overline{x}_i$ and $\overline{\boldsymbol{x}}$, is then associated with a distribution. Moreover, mutual information $I\left(x;y\right)$ between variables x and y measures their dependence, with $I\left(x;y\right)=0$ if and only if they are statistically independent. Using this notation, Definition 1 formalizes *t-probing security* for masked computations.

Definition 1 (t-probing security ([ISW03, RP10, KSM20, BGG+22]))**.** A masked computation $f(\overline{\boldsymbol{x}},\boldsymbol{p},\boldsymbol{r})=\langle\overline{\boldsymbol{y}},\boldsymbol{q},\boldsymbol{\lambda}\rangle$ is t-probing secure if and only if for every set of observations $\boldsymbol{e}\subseteq\boldsymbol{\lambda}$, with $|\boldsymbol{e}|\leq t$, the joint distribution of observations $\boldsymbol{e}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ is statistically independent from the distribution of unshared input secrets $\boldsymbol{x}$, i.e.,

$$I\left(\boldsymbol{x};\boldsymbol{p},\boldsymbol{q},\boldsymbol{e}\right)=0.$$

2.2 Contracts as Programs in a Formal Language

A leakage contract, as introduced by Bloem et al. [BGG+22], is a formal specification of a processor's functionality (ISA), as well as its side-channel leakage behavior. Just like conventional ISA descriptions, a leakage contract precisely defines instruction semantics and their effects on the processors state. It furthermore describes some micro-architectural elements of the processor as well as an over-approximation of the leakage an adversary may observe during the execution of an instruction. For processor hardware to be compliant with a leakage contract, it must be functionally equivalent to the ISA of the leakage contract, and it must not exhibit any side-channel leakage not covered in the contract.

At its core, a leakage contract is just a program, implemented in a some programming language, that specifies the execution of an instruction and the leakage it emits. As a matter of fact, a leakage contract can be expressed in any commonplace turing-complete programming language. From a practical point of view, one should aim for a programming language easily understood by both software and hardware developers, with an efficient translation into symbolic formulas for the purposes of compliance verification. In the rest of this section, we discuss the formalisms of a contract written in a C-like language.

Programming languages can be formally specified through so-called small-step operational semantics, describing how the language would execute a program on an abstract machine by continuously reducing an expression until the program "terminates", yielding a return value [Plo81, NN07]. The small-steps semantics "$\longrightarrow$" of a language describe how to transform the current configuration ω into the next configuration ω' by interpreting a single statement, written as $\omega\longrightarrow\omega'$. Usually, the configuration of an abstract machine for a given language consists of its value storage μ and a sequence of statements to be evaluated ρ, i.e., $\omega=\langle\mu,\rho\rangle$. As an example, the following is a valid sequence of reductions in a C-like language:

$$\begin{aligned}\ldots\longrightarrow\langle\{a\mapsto 5\},\text{“int b = a + 7; a = 8;”}\rangle\longrightarrow\langle\{a\mapsto 5\},\text{“int b = 12; a = 8;”}\rangle\longrightarrow\\ \longrightarrow\quad\langle\{a\mapsto 5,b\mapsto 12\},\text{“a = 8;”}\rangle\quad\longrightarrow\quad\langle\{a\mapsto 8,b\mapsto 12\},\text{“”}\rangle.\end{aligned}$$

Therefore, writing a leakage contract for a processor boils down to writing a program 🗎 that specifies the execution of one instruction on the processor, starting with configuration $\langle\mu_j^{\text{🗎}},\text{🗎}\rangle$ and finishing with configuration $\langle\mu_{j+1}^{\text{🗎}},\text{“”}\rangle$ after a sequence of $\longrightarrow$ reductions. Since the contract needs to specify both the functional behavior of the processor and its leakage behavior, the value storage is split into two distinct parts $\mu^{\text{🗎}}=\langle\sigma^{\text{🗎}},\boldsymbol{\lambda}^{\text{🗎}}\rangle$, where $\sigma^{\text{🗎}}$ contains a specification of the "ISA state" of the processor, and $\boldsymbol{\lambda}^{\text{🗎}}$ represents the set of leaks observable by an adversary.

The $\sigma^{\text{🗎}}$ part of the value storage can be thought of as containing the state of all global variable symbols $v\in V^{\text{🗎}}$ in the contract 🗎 that describe the state of the processor. For example, each machine register defined in the ISA would correspond to some global variable v in the contract, also called *locations*. We write $v(\sigma^{\text{🗎}})$ to denote the value of v in state $\sigma^{\text{🗎}}$.

As for the leaks $\boldsymbol{\lambda}^{\text{🗎}}$, they can be thought of as a special global variable in the leakage contract, only accessible through a special variadic function `void leak(...)`. Since leakage cannot disappear, and only accumulates, the `leak` function only appends the concatenation of its arguments to $\boldsymbol{\lambda}^{\text{🗎}}$. Its abstract small-step semantics are

$$\langle \sigma^{\text{🗎}}, \boldsymbol{\lambda}^{\text{🗎}}, \text{“leak}(v_1, \ldots, v_n);\text{”}\rho \rangle \longrightarrow \langle \sigma^{\text{🗎}}, \boldsymbol{\lambda}^{\text{🗎}} \cup \langle v_1(\sigma^{\text{🗎}}) \parallel \ldots \parallel v_n(\sigma^{\text{🗎}}) \rangle, \rho \rangle. \tag{1}$$

Furthermore, unlike calls to the `leak` function shown in (1), the interpretation of any other statements is *leakage free* and only impacts $\sigma^{\text{🗎}}$.

In the rest of this work, we use the notation $\langle \sigma^{\text{🗎}}_j, \boldsymbol{\lambda}^{\text{🗎}}_j \rangle \xrightarrow{\text{🗎}} \langle \sigma^{\text{🗎}}_{j+1}, \boldsymbol{\lambda}^{\text{🗎}}_{j+1} \rangle$ to mean the execution of a single instruction in the contract 🗎 through a sequence of small-step reductions $\langle \sigma^{\text{🗎}}_j, \boldsymbol{\lambda}^{\text{🗎}}_j, \text{🗎} \rangle \longrightarrow \ldots \longrightarrow \langle \sigma^{\text{🗎}}_{j+1}, \boldsymbol{\lambda}^{\text{🗎}}_{j+1}, \text{“”} \rangle$. Moreover, we write $\xrightarrow{n\text{🗎}}$ to mean $n \in \mathbb{N}$ consecutive executions of n instructions in the contract 🗎. Therefore, $\langle \sigma^{\text{🗎}}_0, \langle\rangle \rangle \xrightarrow{n\text{🗎}} \langle \sigma^{\text{🗎}}_n, \boldsymbol{\lambda}^{\text{🗎}}_n \rangle$ represents the execution of a machine program starting in state $\sigma^{\text{🗎}}_0$ for n instruction in the contract 🗎, finishing in state $\sigma^{\text{🗎}}_n$ and having emmitted leakage $\boldsymbol{\lambda}^{\text{🗎}}_n$ visible to an adversary. The machine program is located in memory and therefore contained in the initial state $\sigma^{\text{🗎}}_0$.

2.3 Gate-level Hardware as Graphs

Every synchronous hardware circuit (e.g., a processor) can be thought of as a labeled directed graph $\text{⚙} = \langle G, W, T, P, \theta \rangle$, where the nodes G are the gates in the design, labeled directed edges $W \subseteq G \times P \times G$ are the wires connecting the output of a gate g' to an input port p of gate g. Furthermore, $\theta : G \to T$ is a function that labels each gate $g \in G$ with its type/functionality $t \in T$. Additionally, no two gates can be connected to the same port of the same destination gate.

The gate types T depend on the netlist technology used to realize the circuit. Irrespective of the technology, T contains the special gate types t_{reg} and t_{in}, which represent simple registers and circuit inputs respectively. For all simple logic gates within the technology, i.e., gates with only one output, T contains a corresponding label, e.g., t_{and} for AND ($\wedge$) gates, whereas P contains the appropriate port labels, e.g., p_{in_0} and p_{in_1}. Moreover, without loss of generality, complex logic gates with mutliple outputs are decomposed into multiple simple gates with a single output, and complex register types are decomposed into a gate computing the next state and a simple register labeled t_{reg}. Since we are interested in synchronous hardware designs, we require each register to be driven with the same phase of the same clock $g_{\text{clk}} \in G$. Moreover, this work only considers hardware circuits where all cyclic paths contain at least one register.

At any point during the execution of the hardware, its stable state is uniquely determined by the value of its inputs and registers. We term these gates *locations*, defined as $V^{\text{⚙}} = \{g \mid g \in G, \theta(g) \in \{t_{\text{reg}}, t_{\text{in}}\}\}$. We use $\sigma^{\text{⚙}} \in \{\bot, \top\}^{|V^{\text{⚙}}|}$ to represent the state of the hardware circuit, and it can be thought of as a concatenation of location values according to some total order $<_G$. In this interpretation, all locations $v^{\text{⚙}} \in V^{\text{⚙}}$ are just functions returning the appropriate bit of the state $\sigma^{\text{⚙}}$. Similarly, every other gate $g \in G \setminus V^{\text{⚙}}$ is then just a function of the state, i.e., $g : \{\bot, \top\}^{|V^{\text{⚙}}|} \to \{\bot, \top\}$, that is recursively defined through its type $\theta(g)$ and the values arriving at its ports. With slight abuse of notation, we also define the value for any subset of gates $G' \subseteq G$ as a function $G' : \{\bot, \top\}^{|V^{\text{⚙}}|} \to \{\bot, \top\}^{|G'|}$ returning a concatenation of gate function values $g \in G'$ according to the total order $<_G$. This notation is especially useful in the context of computation bases, where a computation base C_g of a gate g is the set of all locations $v \in V^{\text{⚙}}$ necessary to compute the value of g

in the current circuit state $\sigma^{\square}$. More formally, we can define C_g recursively as

$$C_g = \begin{cases} \{g\} & \text{if } g \in V^{\square} \\ \|_{g' \in G, \exists p: \langle g', p, g \rangle \in W} C_{g'} & \text{otherwise} \end{cases} .$$

Synchronous hardware circuits are executed in clock cycles. Therefore, we write $\sigma_i^{\square}$ for the state of the circuit in the i-th clock cycle. At the start of a clock cycle, the values of each location is updated in accordance with its type. While the value of every circuit input is determined by the circuits environment (e.g., a testbench or driver), every register g is updated in accordance to its next-state input $g' \in G$, with $\langle g', p_{\text{ns}}, g \rangle \in W$, as $g(\sigma_i^{\square}) = g'(\sigma_{i-1}^{\square})$. We write $\sigma_{i-1}^{\square} \xrightarrow{\square} \sigma_i^{\square}$ to denote that the states $\sigma_{i-1}^{\square}$ and $\sigma_i^{\square}$ are related through such a state update of the circuit $\square$.

2.4 Gate-level Leakage Models

Power side channels in integrated circuits primarily originate from CMOS gates altering their state, which impacts their power consumption and electromagnetic radiation emission. Consequently, the power consumption of CMOS gates on the data they process, which leads to side-channel leakage. The leakage behavior of CMOS gates has been studied in depth and can be modeled in terms of simple physical leakage effects [dGPdlP+16, MMT20, GHP+21].

In the following, we denote the leakage of a gate g with a function $\lambda_g^{\square}$, where the domain and co-domain of the function depend on the considered physical leakage effect. For simpler notation, we say that $\lambda_g^{\square}$ gets both the previous state $\sigma_{i-1}^{\square}$ and the current state $\sigma_i^{\square}$ as arguments, even if it ignores $\sigma_{i-1}^{\square}$ in the given leakage model.

Value leakage was the first physical leakage effect that was considered in the side-channel literature [ISW03]. An idealized adversary with access to value leakage can observe the value of any wire connected to a gate at the end of a clock cycle. Therefore, the value leakage $\lambda_g^{\square}$ of a gate g is its value $\lambda_g^{\square}(\sigma_{i-1}^{\square}, \sigma_i^{\square}) := g(\sigma_i^{\square})$ in the state $\sigma_i^{\square}$.

Transition leakage is another physical leakage effect that describes the phenomenon where the dynamic power consumption of CMOS gates depends on their previous and current value [BGG+14]. An observer would not only be able to tell what the value of the gate is in the current clock cycle, but also whether it changed from zero to one or vice versa. Formally, an idealized adversary can observe the initial value and the resulting value of each gate, with observable gate leakage $\lambda_g^{\square}(\sigma_{i-1}^{\square}, \sigma_i^{\square}) := g(\sigma_{i-1}^{\square}) \parallel g(\sigma_i^{\square})$ representing the concatenation of old and new gate values, thereby capturing all possible transitions.

Glitch leakage is a physical leakage effect that arises due to the change in value a gate experiences within a clock cycle [FGP+18]. The evaluation of a gate does not happen instantaneously, and neither is its output value propagated instantaneously through the outgoing wires. Instead, whenever a change occurs in a gate's input, it must potentially update the output accordingly, leading to a cascade of adjustments in other gates connected to its output. For convenience, a common approximation of glitch leakage effects is to assume an adversary observing a logic gate g can gain information about all registers and circuit inputs in the gate's computational base C_g. The glitch-extended leakage $\lambda_g^{\square}$ of gate g in state $\sigma_i^{\square}$ is then defined as $\lambda_g^{\square}(\sigma_{i-1}^{\square}, \sigma_i^{\square}) := C_g(\sigma_i^{\square})$.

The overall leakage emitted by circuit $\square$ during the execution of a single clock cycle i corresponds to the concatenation of all the individual leakages of every single gate $g \in G$, written as $\langle \lambda_g^{\square}(\sigma_{i-1}^{\square}, \sigma_i^{\square}) \mid g \in G \rangle$. We extend the meaning of the state update relation $\xrightarrow{\square}$ to also capture the leakage emitted during clock cycle i, with $\langle \sigma_{i-1}^{\square}, \sigma_i^{\square}, \boldsymbol{\lambda}_i^{\square} \rangle \xrightarrow{\square} \langle \sigma_i^{\square}, \sigma_{i+1}^{\square}, \boldsymbol{\lambda}_{i+1}^{\square} \rangle$ meaning the relation where $\sigma_{i-1}^{\square} \xrightarrow{\square} \sigma_i^{\square} \xrightarrow{\square} \sigma_{i+1}^{\square}$ and $\boldsymbol{\lambda}_{i+1}^{\square} = \boldsymbol{\lambda}_i^{\square} \cup \langle \lambda_g^{\square}(\sigma_{i-1}^{\square}, \sigma_i^{\square}) \mid g \in G \rangle$. Furthermore, we write $\xrightarrow{m\square}$, with $m \in \mathbb{N}$ for the m consecutive applications of the $\xrightarrow{\square}$ relation. With this shorthand notation,

$\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}, \langle\rangle\rangle \xrightarrow{m\faMicrochip} \langle \sigma^{\faMicrochip}_{m-1}, \sigma^{\faMicrochip}_{m}, \boldsymbol{\lambda}^{\faMicrochip}_{m}\rangle$ represents an m cycle execution of circuit $\faMicrochip$, starting in state $\sigma^{\faMicrochip}_{0}$, terminating in state $\sigma^{\faMicrochip}_{m}$, and producing leakage $\boldsymbol{\lambda}^{\faMicrochip}_{m}$ throughout the whole execution.

2.5 Hardware Compliance with a Contract

A contract $\faFile*$ specifies the functional and leakage behavior of a processor hardware circuit $\faMicrochip$. We say that processor $\faMicrochip$ is compliant with contract $\faFile*$ if and only if (a) they perform the same functional computation and (b) any leakage detectable in the hardware is also leaked in the contract when computing with the same data [BGG+22]. This notion is formalized through the use of a relation $\simeq_{\mathcal{M}}$ defined by a mapping $\mathcal{M}$ between hardware locations $V^{\faMicrochip}$ in $\faMicrochip$ and contract locations $V^{\faFile*}$ in $\faFile*$.

Definition 2 ($\langle \sigma^{\faMicrochip}_{i-1}, \sigma^{\faMicrochip}_{i}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}$)**.** Let $\faMicrochip$ be a circuit with locations $V^{\faMicrochip}$, and $\faFile*$ be a contract with locations $V^{\faFile*}$, and let $\mathcal{M} \subseteq \{-1, 0\} \times V^{\faMicrochip} \times V^{\faFile*}$ be a timed mapping between their locations. States $\langle \sigma^{\faMicrochip}_{i-1}, \sigma^{\faMicrochip}_{i}\rangle$ and $\sigma^{\faFile*}$ are related, written as $\langle \sigma^{\faMicrochip}_{i-1}, \sigma^{\faMicrochip}_{i}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}$, if and only if $(\sigma^{\faMicrochip}_{i-1} \xrightarrow{\faMicrochip} \sigma^{\faMicrochip}_{i}) \wedge \bigwedge_{\langle l, v^{\faMicrochip}, v^{\faFile*}\rangle \in \mathcal{M}} v^{\faMicrochip}(\sigma^{\faMicrochip}_{i+l}) = v^{\faFile*}(\sigma^{\faFile*})$.

The relation $\simeq_{\mathcal{M}}$, as given in Definition 2, is slightly different from the definition given by Bloem et al. [BGG+22] and includes timing information. Their work only maps values from the contract to the hardware state at the beginning of the first cycle of instruction execution (i.e., $l = 0$), while our glitch-aware leakage model necessitates additional mappings to the clock cycle before an instruction started executing (i.e., $l = -1$).

Definition 3 (Compliance)**.** A hardware implementation $\faMicrochip$ is compliant with contract $\faFile*$ under relation $\simeq_{\mathcal{M}}$ if for every sequence of n instructions, executing for m clock cycles, and starting hardware and contract states $\sigma^{\faMicrochip}_{0}$ and $\sigma^{\faFile*}_{0}$, the executions $\langle \sigma^{\faFile*}_{0}, \langle\rangle\rangle \xrightarrow{n\faFile*} \langle \sigma^{\faFile*}_{n}, \boldsymbol{\lambda}^{\faFile*}_{n}\rangle$ and $\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}, \langle\rangle\rangle \xrightarrow{m\faMicrochip} \langle \sigma^{\faMicrochip}_{m-1}, \sigma^{\faMicrochip}_{m}, \boldsymbol{\lambda}^{\faMicrochip}_{m}\rangle$ fulfill:

(a) **Related states remain related:** Whenever $\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}\rangle$ and $\sigma^{\faFile*}_{0}$ are related under $\simeq_{\mathcal{M}}$, so are the resulting states $\langle \sigma^{\faMicrochip}_{m-1}, \sigma^{\faMicrochip}_{m}\rangle$ and $\sigma^{\faFile*}_{n}$:

$$\forall \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}, \sigma^{\faFile*}_{0} : \left(\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}_{0}\right) \Rightarrow \left(\langle \sigma^{\faMicrochip}_{m-1}, \sigma^{\faMicrochip}_{m}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}_{n}\right).$$

(b) **All emitted leaks are modeled:** For every leak $\lambda^{\faMicrochip}(\sigma^{\faMicrochip}_{i-1}, \sigma^{\faMicrochip}_{i}) \in \boldsymbol{\lambda}^{\faMicrochip}_{m}$ observable in hardware, there exists a leak $\lambda^{\faFile*}(\sigma^{\faFile*}_{j}) \in \boldsymbol{\lambda}^{\faFile*}_{n}$ in the contract and a function f_{λ} that produces the output of $\lambda^{\faMicrochip}$ from the output of $\lambda^{\faFile*}$, whenever $\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}_{0}$:

$$\forall \lambda^{\faMicrochip} \exists \lambda^{\faFile*}, f_{\lambda} \forall \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}, \sigma^{\faFile*}_{0} : \left(\langle \sigma^{\faMicrochip}_{-1}, \sigma^{\faMicrochip}_{0}\rangle \simeq_{\mathcal{M}} \sigma^{\faFile*}_{0}\right) \Rightarrow f_{\lambda}\left(\lambda^{\faFile*}\left(\sigma^{\faFile*}_{j}\right)\right) = \lambda^{\faMicrochip}\left(\sigma^{\faMicrochip}_{i-1}, \sigma^{\faMicrochip}_{i}\right).$$

2.6 Hardware Verification Procedure

As directly proving compliance of every possible program execution is computationally infeasible, Bloem et al. [BGG+22] showed that verification of a single instruction execution is sufficient to inductively prove compliance for arbitrary programs as defined in Definition 3. Their proposed verification procedure consists of the following steps:

1. Verifying that states remain related. The first verification step is to ensure that starting from similar states $\sigma^{\faMicrochip}_{j-1}$, $\sigma^{\faMicrochip}_{j}$ and $\sigma^{\faFile*}_{j}$, contract and hardware states remain related after executing any valid instruction. This is achieved by encoding the condition as one SAT query per cycle, up to the longest possible cycle count for a single instruction.

2. Finding modeling functions for gates. Before verifying gate leakage, an intermediate step is required to restrict the values of gates in the previous state. This is necessary as the contract does not model every aspect of execution and leaving gates as unrestricted variables would allow the solver to assume gates as functionally dependent on shares, even if they are not. To determine which gate depends on which state register, at most one SMT query per pair of gate and state register is required.

3. Verifying that leaks are modeled. The actual verification of the leakage of each gate is modeled by a single leak statement in the contract is performed by encoding this 2-safety hyperproperty as two instances of the hardware and contract, unrolling the hardware for each cycle of the instruction execution. We discuss the encoding of glitch leakage in Chapter 3. This step requires at most one SMT query per combination of gate, cycle, and leak statement.

2.7 Software Compliance with a Contract or Hardware

In this section we define what it means for software to be compliant with a contract for t-probing security. We verify t-probing security of the software with respect to a contract, closely following the definition of probing security by Bloem et al. [BGG+22]. While they use the t-(S)NI security notion, we opt for verifying t-probing security, as t-(S)NI implementations trade ease-of-verification at the cost of higher randomness requirements.

For a masked computation $f : \langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}, \boldsymbol{q}, \boldsymbol{\lambda} \rangle$, its inputs consist of shared secret input values, public values and uniform random values, whereas its output consists of shared secret outputs, public values and leaks. In a concrete machine code implementation of a masked computation, its inputs are located in the processor state (either $\sigma^{\text{🗎}}$ for the contract, or $\sigma^{\text{⚙}}$ for hardware), which consists of locations. The initial and final position of these variables within the state $\sigma^{\text{🗎}}$ can be formalized through input (respectively output) policies $\pi_{\text{in}} : \langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \leftrightarrow \sigma$ (respectively $\pi_{\text{out}} : \sigma \leftrightarrow \langle \overline{\boldsymbol{y}}, \boldsymbol{q} \rangle$), which link the inputs (respectively outputs) to corresponding locations in the contract. A machine code execution, together with the input and output policies, defines a masked computation.

Definition 4 ($\langle \xrightarrow{n\text{🗎}}, \pi^{\text{🗎}}_{\text{in}}, \pi^{\text{🗎}}_{\text{out}} \rangle$ defines a masked computation)**.** Let 🗎 be a contract, $\pi^{\text{🗎}}_{\text{in}} : \langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \leftrightarrow \sigma^{\text{🗎}}$ be an input policy and $\pi^{\text{🗎}}_{\text{out}} : \sigma^{\text{🗎}} \leftrightarrow \langle \overline{\boldsymbol{y}}^{\text{🗎}}, \boldsymbol{q}^{\text{🗎}} \rangle$ be an output policy. Furthermore, let $\sigma^{\text{🗎}}_0 = \pi^{\text{🗎}}_{\text{in}}(\overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r})$ be a state, $\langle \sigma^{\text{🗎}}_0, \langle\rangle \rangle \xrightarrow{n\text{🗎}} \langle \sigma^{\text{🗎}}_n, \boldsymbol{\lambda}^{\text{🗎}}_n \rangle$ be an execution of n instructions, and $\langle \overline{\boldsymbol{y}}^{\text{🗎}}, \boldsymbol{q}^{\text{🗎}} \rangle = \pi^{\text{🗎}}_{\text{out}}(\sigma^{\text{🗎}}_n)$. Then $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\text{🗎}}, \boldsymbol{q}^{\text{🗎}}, \boldsymbol{\lambda}^{\text{🗎}}_n \rangle$ is a masked computation in contract 🗎.

Definition 5 ($\langle \xrightarrow{m\text{⚙}}, \pi^{\text{⚙}}_{\text{in}}, \pi^{\text{⚙}}_{\text{out}} \rangle$ defines a masked computation)**.** Let ⚙ be a hardware circuit, $\pi^{\text{⚙}}_{\text{in}} : \langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \leftrightarrow \sigma^{\text{⚙}}$ be an input policy and $\pi^{\text{⚙}}_{\text{out}} : \sigma^{\text{⚙}} \leftrightarrow \langle \overline{\boldsymbol{y}}^{\text{⚙}}, \boldsymbol{q}^{\text{⚙}} \rangle$ be an output policy. Furthermore, let $\sigma^{\text{⚙}}_0 = \pi^{\text{⚙}}_{\text{in}}(\overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r})$ be a state, $\langle \sigma^{\text{⚙}}_{-1}, \sigma^{\text{⚙}}_0, \langle\rangle \rangle \xrightarrow{m\text{⚙}} \langle \sigma^{\text{⚙}}_{m-1}, \sigma^{\text{⚙}}_m, \boldsymbol{\lambda}^{\text{⚙}}_m \rangle$ be an execution of m clock cycles, and $\langle \overline{\boldsymbol{y}}^{\text{⚙}}, \boldsymbol{q}^{\text{⚙}} \rangle = \pi^{\text{⚙}}_{\text{out}}(\sigma^{\text{⚙}}_m)$. Then $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\text{⚙}}, \boldsymbol{q}^{\text{⚙}}, \boldsymbol{\lambda}^{\text{⚙}}_m \rangle$ is a masked computation in hardware circuit ⚙.

Finally, Bloem et al. [BGG+22] show that all information accessible to an attacker observing side-channel information of a machine code execution on hardware ⚙ is also accessible by an attacker observing an execution of the same machine code on a contract 🗎 the hardware ⚙ complies with.

Theorem 1 (Model Reduction [BGG+22])**.** *Let 🗎 be a contract and ⚙ be a compliant hardware circuit. Furthermore, let $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\text{🗎}}, \boldsymbol{q}^{\text{🗎}}, \boldsymbol{\lambda}^{\text{🗎}}_n \rangle$ and $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\text{⚙}}, \boldsymbol{q}^{\text{⚙}}, \boldsymbol{\lambda}^{\text{⚙}}_m \rangle$ be masked computations and $\langle \sigma^{\text{🗎}}_0 \rangle$, respectively $\langle \sigma^{\text{⚙}}_{-1}, \sigma^{\text{⚙}}_0 \rangle$ be the initial states of their execution in 🗎 and ⚙, as was the case in Definitions 4 and 5. Whenever $\langle \sigma^{\text{⚙}}_{-1}, \sigma^{\text{⚙}}_0 \rangle \simeq_{\mathcal{M}} \sigma^{\text{🗎}}_0$, then*

$$\overline{\boldsymbol{y}}^{\text{⚙}} = \overline{\boldsymbol{y}}^{\text{🗎}}, \boldsymbol{q}^{\text{⚙}} = \boldsymbol{q}^{\text{🗎}}, \textit{and}^{2} \tag{2}$$

$$\forall \lambda^{\text{⚙}} \left(\sigma^{\text{⚙}}_{i-1}, \sigma^{\text{⚙}}_i \right) \in \boldsymbol{\lambda}^{\text{⚙}}_m \; \exists f_\lambda, \lambda^{\text{🗎}} \left(\sigma^{\text{🗎}}_j \right) \in \boldsymbol{\lambda}^{\text{🗎}}_n : f_\lambda \left(\lambda^{\text{🗎}} \left(\sigma^{\text{🗎}}_j \right) \right) = \lambda^{\text{⚙}} \left(\sigma^{\text{⚙}}_{i-1}, \sigma^{\text{⚙}}_i \right). \tag{3}$$

Proof. The proof follows directly from compliance of ⚙ with 🗎 (cf. Definition 3). □

[2] In reality, $\boldsymbol{q}^{\text{⚙}}$ can also contain constant control signals not in $\boldsymbol{q}^{\text{🗎}}$ and ignored in this equality.

3 Leakage Models for Glitches and Transitions

In this section we introduce our extended leakage model for glitches and transitions. Using this model, we construct a contract for the Ibex [low] processor and show how it is modeled in the C language. We further show how to efficiently encode our glitch-aware leakage model in the hardware compliance verification procedure introduced by Bloem et al. [BGG+22].

3.1 A Precise Transition-Glitch Leakage Model

A natural way to define a leakage function λ_g that over-approximates the two common notions of glitch and transition leakage is to define $\lambda_g(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}) := C_g(\sigma^{\square}_{i-1}) \parallel C_g(\sigma^{\square}_{i})$. However, this definition is agnostic to the actual state values $\sigma^{\square}_{i-1}$ and $\sigma^{\square}_{i}$, leading to an over-approximation that also considers physically impossible leakage scenarios, as noted by Gigerl et al. [GHP+21]. We illustrate this in Example 1, where an And gate does not allow a glitch to propagate, and avoids this worst-case leakage.

Example 1. Let us consider a circuit $\square = \langle G, W, T, P\rangle$ with gates $\{a, b, c\} \subset G$ and $\{\langle a, p_{\text{in}_0}, c\rangle, \langle b, p_{\text{in}_1}, c\rangle\} \subset W$. Moreover, let $\theta(a) = t_{\text{in}}$, $\theta(b) = t_{\text{reg}}$, and $\theta(c) = t_{\text{and}}$. Here, $\{a, b\} \subset V^{\square}$ and $C_c = \{a, b\}$, with $c(\sigma^{\square}) = a(\sigma^{\square}) \wedge b(\sigma^{\square})$. For an execution $\sigma^{\square}_0 \xrightarrow{\square} \sigma^{\square}_1$, the naive combined transition-glitch leakage model would claim leakage $\lambda_c(\sigma^{\square}_0, \sigma^{\square}_1) = a(\sigma^{\square}_0) \parallel a(\sigma^{\square}_1) \parallel b(\sigma^{\square}_0) \parallel b(\sigma^{\square}_1)$. While this is exact when both the values of a and b are unknown in both clock cycles, it overapproximates heavily when either of them is known to be $\perp$ in both clock cycles (e.g., it is a control signal independent of data). Assume that $a(\sigma^{\square}_0) = \perp$ and $a(\sigma^{\square}_1) = \perp$. In this case, signal a would not experience a transition, meaning its value would be stable. Because of the way And gates are built in CMOS, this would result in c also remaining stable with value $\perp$, not leaking information about b due to glitches. In reality, an adversary observing c would not learn any information about the value of b from glitches in state $\sigma^{\square}_1$, whereas the naive combined transition-glitch leakage model states they would.

The way to improve the leakage over-approximation is to make λ_g dependent on whether or not the inputs of g can experience value fluctuations due to transitions and glitches, as well as the gate type $\theta(g)$ of g. In the following, we propose a more precise encoding for leakage $\lambda_g(\sigma^{\square}_{i-1}, \sigma^{\square}_{i})$. For any hardware location $V^{\square} \subseteq G$, an idealized adversary can observe transitions between the previous and current state, same as in the transition leakage model, i.e., $\lambda_g(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}) := g(\sigma^{\square}_{i-1}) \parallel g(\sigma^{\square}_{i})$ for g with $\theta(g) \in \{t_{\text{reg}}, t_{\text{in}}\}$.

For And gates, we model the forwarding of input leakage through transitions and glitches depending on the values of the other input. Because of the CMOS properties of And gates, the leakage of input at port p_{in_0} is forwarded only if the other input can attain the value $\top$ at any point throughout the clock cycle. For an And gate c, with input wires $\langle a, p_{\text{in}_0}, c\rangle$ and $\langle b, p_{\text{in}_1}, c\rangle$, this forwarding behavior for the input at port p_{in_0} is described by the higher order function

$$l_{\text{and}}\left(\lambda_a, \lambda_b, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) := \left(\lambda_b\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) = \mathbb{0}\right) \ ?\ \mathbb{0} : \lambda_a\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right), \tag{4}$$

where $\mathbb{0}$ is a vector of $\perp$ values with appropriate size, and $x\ ?\ y : z$ is the bitwise conditional operator that returns y if $x = \top$ and z otherwise. The behavior for the other port is symmetric. The transition-glitch leakage of And gate c is then defined as

$$\lambda_c\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) := l_{\text{and}}\left(\lambda_a, \lambda_b, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) \parallel l_{\text{and}}\left(\lambda_b, \lambda_a, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right). \tag{5}$$

Since Or gates are also non-linear and symmetric, they have a very similar leakage behavior to And gates. If c were instead an Or gate (i.e., $\theta(c) = t_{\text{or}}$), we would have

$$l_{\text{or}}\left(\lambda_a, \lambda_b, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) := \left(\lambda_b\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) = \mathbb{1}\right) \ ?\ \mathbb{1} : \lambda_a\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right), \text{ and} \tag{6}$$

$$\lambda_c\left(\sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) := l_{\text{or}}\left(\lambda_a, \lambda_b, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right) \parallel l_{\text{or}}\left(\lambda_b, \lambda_a, \sigma^{\square}_{i-1}, \sigma^{\square}_{i}\right), \tag{7}$$

where $\mathbb{1}$ is a vector of $\top$ values with appropriate size.

Unlike the non-linear gate types And and Or, linear gate types like Xor and Not always forward the leakage of their inputs and make them observable through glitches. Therefore, if c was a Not gate, with $\langle b, p_{\mathrm{in}_1}, c\rangle \notin W$ since there is only one input port, an adversary would observe $\lambda_c(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}) := \sim \lambda_a(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}})$, with operator $\sim x$ representing a bitwise negation of vector x. Similarly, if c were instead an Xor gate (i.e., $\theta(c) = t_{\mathrm{xor}}$), we would have

$$l_{\mathrm{xor}}\left(\lambda_a, \lambda_b, \sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right) := \left(\lambda_b\left(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right) = \mathbb{1}\right)\ ?\ \sim \lambda_a\left(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right) : \lambda_a\left(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right), \text{ and} \tag{8}$$

$$\lambda_c\left(\sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right) := l_{\mathrm{xor}}\left(\lambda_a, \lambda_b, \sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right) \parallel l_{\mathrm{xor}}\left(\lambda_b, \lambda_a, \sigma_{i-1}^{\text{▣}}, \sigma_i^{\text{▣}}\right). \tag{9}$$

Here, the bitwise negation in case one of the inputs is $\mathbb{1}$ is necessary to correctly capture the behavior of an Xor gate when both inputs are stable.

The leakage behavior of negated gate variants like Nand, Nor and Xnor can be achieved by applying a bitwise negation $\sim$ to the definitions for the And, Or and Xor gates. Similarly, the behavior of multiplexer gates can derived from the formulas for And, Or and Not gates using the transformation $(x\ ?\ y : z) \Leftrightarrow ((x \wedge y) \vee (\neg x \wedge z))$.

3.2 Expressing Contracts in C

A hardware contract implements the semantics of the execution of a single instruction $\langle \sigma_j^{\text{🗎}}, \boldsymbol{\lambda}_j^{\text{🗎}}\rangle \xrightarrow{\text{🗎}} \langle \sigma_{j+1}^{\text{🗎}}, \boldsymbol{\lambda}_{j+1}^{\text{🗎}}\rangle$, specifying the architectural behavior of the instruction set and the corresponding leakage behavior. Contracts written in C do this by implementing the `step_cpu` function. The contract states $\sigma_j^{\text{🗎}}$ and $\sigma_{j+1}^{\text{🗎}}$ consist of the values of C variables before and after execution of the `step_cpu` function. Leaks are modeled by invocations of the variadic function `leak`. For example, calling `leak(a, b)` emits a leak of the concatenation of its parameters `a` and `b`. The accumulation of leaks works according to the small step semantics specified in Equation 1. Additionally, the `step_cpu` function's boolean return value specifies if the contract permits the execution of a given instruction in a given initial state. This is used to ensure that normal operating conditions are maintained, i.e., no invalid instructions may be executed.

Addressing C's lack of native support for arbitrary length numeric types, the contract runtime library provides the bitvector datatype `bv`, which is parametrizable to any length. It allows developers to emulate register-transfer-level constructs analogous to System Verilog's `logic`. The bv datatype mimics C's built-in numeric types, supporting both logical and arithmetic operations. Constants are sourced using the `bv_const` macro, and the `slice` macro extracts bit ranges, creating new bitvectors of the desired size.

Listing 1 shows the state variables used to model the Ibex processor. Both registers from the ISA and shadow registers used to model the leakage behavior are defined as global variables. In Listing 2 we define the `step_cpu` function for the Ibex processor, which sets the default value for `next_pc` and defers the actual decoding and execution to the execute function. There the seven bit opcode is extracted from the instruction and the appropriate implementation is called.

3.3 Modeling Glitches of the Ibex Processor

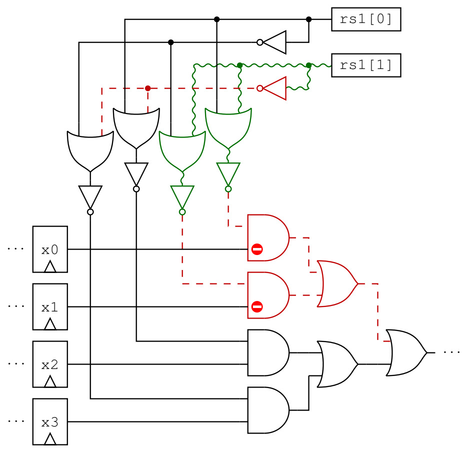
In addition to the functional specification, a contract must model the leakage behavior of CPUs. To accurately describe the leakage behavior with respect to glitches, the implementation details of specific microarchitectural components, such as the read and write ports of register files and memory, are essential. We analyze the variant of Ibex that performs the instruction decode, execution and writeback stages in a single cycle. Thus, glitches that arise anywhere in those stages can potentially propagate to successive

Listing 1: Contract model of state for the RV32E instruction set.

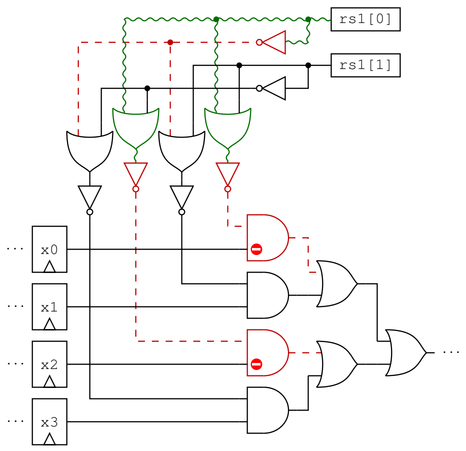
```
typedef struct regfile {
    bv<32> x1, x2, ..., x15;
} regfile_t;
// architectural registers
reg pc;
reg next_pc;
regfile_t regs;
// leakage registers
regfile_t prev_regs;
bv<32> mem_last_addr;
bv<32> mem_last_read;
bv<5> prev_rd;
bv<5> prev_rs1;
bv<5> prev_rs2;
```

Listing 2: Step function of IBEX the contract.

```
bool step_cpu(bv<32> op) {
    next_pc = pc + 4;
    const bool ret = execute(op);
    pc = next_pc;
    return ret;
}
```



(a) Transition from `rs1 = 2` to `rs1 = 3`



(b) Transition from `rs1 = 3` to `rs1 = 1`

Figure 1: Glitch propagation in the register file read port in the IBEX processor when changing the value of `rs1`. Green wavy wires represent $\lambda_g(\sigma_{i-1}^{\text{⯐}}, \sigma_i^{\text{⯐}}) = \mathbb{1}$, whereas red dashed wires represent $\lambda_g(\sigma_{i-1}^{\text{⯐}}, \sigma_i^{\text{⯐}}) = \mathbb{0}$. The ⛔ symbol specifies where value propagation through glitches is stopped.

stages. In the worst case, glitches arising from the read ports of register file or memory can propagate to the write ports of the register file or memory.

In the RISC-V ISA, each instruction may specify up to two source registers, which are read from the register file read ports. Those source registers are specified on fixed locations in the instruction opcode. Bits 15 to 19 specify source register 1, while bits 20 to 24 specify source register 2. The IBEX implementation of RISC-V fetch these values from the register file regardless of the instruction currently executing, Figure 1 illustrates the implementation of register file read ports on the netlist level, only showing the first four registers and two bits of the register index. First, the register index from the instruction opcode is decoded to a one-hot signal for each register using a series of NOT and OR gates. The resulting one-hot encoding is combined with the output ports of the corresponding registers through a layer of AND gates. Finally, a tree of OR gates produces the value of the selected register.

Modeling the transition leakage of this construct would require leaking a combination of the values read in the current and previous instructions. In our extended leakage model, glitches can cause registers that were not used in either instruction to propagate their values to the output of the register file read port. In the worst case, all registers could be combined in a single cycle, leaking the combination of the entire register file. For example, if the value of the read port switches from `x1` to `x2`, `rs1[0]` would transition from $\top$ to $\bot$ and `rs1[1]` would transition from $\bot$ to $\top$. Those transitions cause the outputs of the one-hot decoding Or gates to become unstable. This can lead to all one-hot selector bits to ephemerally assume the value $\top$, causing all registers to propagate to the output of the read port.

Efficient implementations of software masking require multiple shares of the same secret to reside in the register file simultaneously, which would be insecure if the entire register file is leaked in every cycle. To enable such optimized implementations, it is necessary to identify conditions under which the glitch propagation is terminated before the register file output, resulting in leakage that only combines parts of the register file. This is the case whenever a signal of the one-hot encoding is stable with the value $\bot$, causing the And gates to terminate glitch propagation of the respective register. Figure 1a shows this behavior when switching the register index from `x2` to `x3`, which only causes those two registers to be combined, while the one-hot decoded signals for the other registers remains stable. Figure 1b shows the same effect when switching from `x3` to `x1`, where `rs1[0]` remains stable, preventing the propagation of `x0` to `x2`. While this example only shows four registers, the principle applies to the entire register file of the Ibex, which can be configured with 32 registers and five index bits for RV32I or 16 registers and four index bits for RV32E. While the described system scales with the number of register bits, `x0` is not an actual hardware register but rather a zero constant, which is why this value is hardwired to zero in the Ibex processor. Each register bit that is stable, i.e., does not change between instruction opcodes, halves the amount of unstable one-hot signals, halving the number of register values that are propagated. In the worst case, this still leads to a leak of all registers of the register file, e.g., when switching from `x0` to `x15` or from `x1` to `x14`, as all index bits transition in these cases.

To model the leakage in our contract, we emit three kinds of leakages. The *common leakage* models all gates before the register writeback, while *writeback leakage* models the remaining leaks emitted by the register writeback logic. Load instructions additionally leak the load value from the memory bus, which is covered through the *load leakage*. In our contract, common leakage and writeback leakage are modeled in the `regfile_glitches` function shown in Listing 4, which is called regardless of the instruction. The load leakage is modeled by the `load_leakage` function shown in Listing 5, which is only called when a load instruction is executed.

Common Leakage. To model this common leakage of register file reads in the contract, we define the `glitchy_decode` function, which uses bit-vector arithmetic to determine which registers will be combined. Listing 3 shows the implementation, which forms a mask of bits that have transitioned from the previous instruction (only four index bits are used if the contract is compiled for RV32E). The function then returns a vector of registers, where each entry either contains the register value or zero if the register is not leaked. We apply this to the values of both register read port values in the current and previous cycle and jointly leak the results with the last memory address and memory read value. For load instructions, a variant of this leak statement is emitted, which also leaks the load value from the memory bus. Together, those two leak statements form the *common leakage*, which can simulate the leakage of the entire Ibex CPU, except for the register writeback.

Writeback Leakage. Figure 2 illustrates the writeback stage of the Ibex processor. The register writeback has a similar structure and leakage behavior to the register read ports. The destination register index `rd`, which is part of the instruction's opcode, is one-hot decoded through the same scheme of Not and Or gates as the register read ports. The results are used as the selector inputs of multiplexer gates that either select the previous value of the register or the writeback value if the register is to be assigned a new value. If the one-hot decoding of `rd` for a register is unstable, the multiplexers

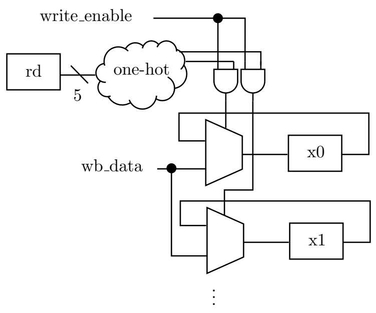


Figure 2: Register file write port in the IBEX processor.

Listing 3: Model of register file read port leakage in the IBEX contract.

```
regfile_t glitchy_decode(const regfile_t* regfile, bv<5> idx,
                         bv<5> prev_idx) {
    bv<5> mask = (idx ^ prev_idx ^ bv_const(0b11111, 5)) & REGIDX_BITS;
    const bv<5> pattern = (idx & mask);
    regfile_t res = {
        ((bv_const( 1, 5) & mask) == pattern) ? regfile->x1  : 0,
        ((bv_const( 2, 5) & mask) == pattern) ? regfile->x2  : 0,
        // omitted ...
        ((bv_const(15, 5) & mask) == pattern) ? regfile->x15 : 0
    };
    return res;
}
```

for the corresponding registers emit joint leakage of the writeback value and the register output. If the one-hot decoding is stable and the value is $\perp$, only the register value is leaked. This behavior is modeled in the contract by emitting one leak per register, which leaks the register value and, depending on the transitions in `rd`, appends the values from the common leakage.

Load Leakage. Load instructions on the IBEX processor take at least two cycles to process. In the first cycle of execution a load instruction, the LSU issues a request to load a specific address to the memory bus. One or more cycles later, the memory responds with the value stored at the requested address. To model the leakage of load instructions, both cycles have to be considered separately. To simulate the first cycle of a load instruction, it is sufficient to emit the common leakage, as the response from the memory bus cannot leak into the CPU before the second cycle. To simulate the second cycle, only values from the first and second cycle can be leaked, while values from the previous instruction have no influence. Most of the control signals in the CPU are stable after the first cycle, as execution is stalled until the memory bus responds with the requested data. To model the leaks caused in the second cycle of a load instruction, only the stable value of `rs1`, `mem_last_addr` and the load value `req_data` need to be leaked. Similarly to the writeback leakage for other instructions, the write ports of the register file can be covered by emitting one leak per register, conditionally combined with the load leakage.

3.4 Efficient Encoding of Glitches for Hardware Verification

The heart of the hardware compliance (cf. Definition 3) verification procedure outlined by Bloem et al. [BGG+22] is the way they prove the existence of a function f_λ that always maps the value of a contract leak $\lambda^{📄}(\sigma_j^{📄})$ to the value of a hardware leak $\lambda^{🔲}(\sigma_{i-1}^{🔲}, \sigma_i^{🔲})$. Without going into too much detail or formalism, their method relies on showing that there are no state triples $\langle \sigma_{-1}^{🔲}, \sigma_0^{🔲}, \sigma_0^{📄} \rangle$ and $\langle \hat{\sigma}_{-1}^{🔲}, \hat{\sigma}_0^{🔲}, \hat{\sigma}_0^{📄} \rangle$ where $\left(\langle \sigma_{-1}^{🔲}, \sigma_0^{🔲} \rangle \simeq_{\mathcal{M}} \sigma_0^{📄}\right)$, $\left(\langle \hat{\sigma}_{-1}^{🔲}, \hat{\sigma}_0^{🔲} \rangle \simeq_{\mathcal{M}} \hat{\sigma}_0^{📄}\right)$, and $\lambda^{📄}(\sigma_j^{📄}) = \lambda^{📄}(\hat{\sigma}_j^{📄})$ but $\lambda^{🔲}(\sigma_{i-1}^{🔲}, \sigma_i^{🔲}) \neq \lambda^{🔲}(\hat{\sigma}_{i-1}^{🔲}, \hat{\sigma}_i^{🔲})$. Showing this property is sufficient

Listing 4: Ibex Contract leaks for common leakage and writeback leakage.

```
void regfile_glitches(bv<32> op) {
    const bv<5> rd  = slice(op, 11,  7);
    const bv<5> rs1 = slice(op, 19, 15);
    const bv<5> rs2 = slice(op, 24, 20);
    regfile_t glitchy_rs1_val = glitchy_decode(&regs, rs1, prev_rs1);
    regfile_t glitchy_prev_rs1_val = glitchy_decode(&prev_regs, rs1,
        prev_rs1);
    regfile_t glitchy_rs2_val = glitchy_decode(&regs, rs2, prev_rs2);
    regfile_t glitchy_prev_rs2_val = glitchy_decode(&prev_regs, rs2,
        prev_rs2);
    // common leakage
    leak(mem_last_addr, mem_last_read,
         rf_to_bv(glitchy_rs1_val), rf_to_bv(glitchy_prev_rs1_val),
         rf_to_bv(glitchy_rs2_val), rf_to_bv(glitchy_prev_rs2_val));
    // writeback leakage
    bv<5> mask = (prev_rd ^ rd ^ bv_const(0b11111, 5)) & REGIDX_BITS;
    const bv<5> pattern = (rd & mask);
    for (int i = 1; i < NUM_REGS; i++) {
        if ((bv_const(i, 5) & mask) == pattern) {
            leak(rX(&regs, i), rX(&prev_regs, i),
                 mem_last_addr, mem_last_read,
                 rf_to_bv(glitchy_rs1_val), rf_to_bv(glitchy_prev_rs1_val),
                 rf_to_bv(glitchy_rs2_val), rf_to_bv(glitchy_prev_rs2_val));
        } else {
            leak(rX(&regs, i), rX(&prev_regs, i));
        }
    }
}
```

to imply that there must be a function f_λ computing $\lambda^{\boxdot}(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i)$ from $\lambda^{\boxminus}(\sigma^{\boxminus}_j)$.

Directly encoding the leakage model as specified in Section 3.1 is not realistic, since the length of $\lambda^{\boxdot}_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i)$ is exponential in the number of locations in the cone C_g of gate g. However, this exponential encoding is not necessary to express the existance of state triples $\langle \sigma^{\boxdot}_{-1}, \sigma^{\boxdot}_0, \sigma^{\boxminus}_0 \rangle$ and $\langle \hat{\sigma}^{\boxdot}_{-1}, \hat{\sigma}^{\boxdot}_0, \hat{\sigma}^{\boxminus}_0 \rangle$ where $\lambda^{\boxdot}_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \neq \lambda^{\boxdot}_g(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$. It turns out that it is sufficient to encode the stability $\tau_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ of gate g, i.e., the equivalence of $\lambda^{\boxdot}_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \parallel \lambda^{\boxdot}_g(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ with either $\mathbb{0}$ or $\mathbb{1}$, and the difference $\delta_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$, i.e., $\lambda^{\boxdot}_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \neq \lambda^{\boxdot}_g(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$.

Stability. The stability $\tau_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ of locations $g \in V^{\boxdot}$ is encoded as

$$\tau_g\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right) := g\left(\sigma^{\boxdot}_i\right) = g\left(\sigma^{\boxdot}_{i-1}\right) \wedge g\left(\hat{\sigma}^{\boxdot}_i\right) = g\left(\hat{\sigma}^{\boxdot}_{i-1}\right) \wedge g\left(\sigma^{\boxdot}_i\right) = g\left(\hat{\sigma}^{\boxdot}_i\right). \tag{10}$$

Trivially, whenever $\tau_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i) = \top$, then $\lambda^{\boxdot}_g(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \parallel \lambda^{\boxdot}_g(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ is either $\mathbb{1}$ or $\mathbb{0}$. Moreover, negations through NOT gates do not change stability, so a gate c with $\theta(c) = t_{\text{not}}$ and $\langle a, p_{\text{in}_0}, c \rangle \in W$, would have $\tau_c(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i) = \tau_a(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$.

Non-linear AND and OR gates can have a stable value if one of their inputs is stable with an appropriate value. For an AND gate c with input wires $\langle a, p_{\text{in}_0}, c \rangle, \langle b, p_{\text{in}_1}, c \rangle \in W$, its stability is

$$\tau_c\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right) := \begin{array}{l} \left(\tau_a\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right) \wedge \tau_b\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right)\right) \vee \\ \left(\tau_a\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right) \wedge \neg a\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i\right)\right) \vee \\ \left(\tau_b\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i, \hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i\right) \wedge \neg b\left(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i\right)\right) \end{array}, \tag{11}$$

whereas the definition for an OR gate has the same structure, but the terms $a(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i)$ and $b(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i)$ are not negated. This corresponds to definition (5) where $\lambda^{\boxdot}_c(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \parallel \lambda^{\boxdot}_c(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ is $\mathbb{1}$ if both $\lambda^{\boxdot}_a(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \parallel \lambda^{\boxdot}_a(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ and $\lambda^{\boxdot}_b(\sigma^{\boxdot}_{i-1}, \sigma^{\boxdot}_i) \parallel \lambda^{\boxdot}_b(\hat{\sigma}^{\boxdot}_{i-1}, \hat{\sigma}^{\boxdot}_i)$ are $\mathbb{1}$, and $\mathbb{0}$ if either input is $\mathbb{0}$. The case for OR gates is analogous, corresponding to (7).

Listing 5: Contract leaks for load leakage.

```
void load_leakage(const reg op, reg addr, reg req_data)
{
    const bv<5> rd  = slice(op, 11,  7);
    const bv<5> rs1 = slice(op, 19, 15);
    leak(rX(&regs, rs1), mem_last_addr, req_data);
    for (int i = 1; i < NUM_REGS; i++) {
        if (bv_const(i, 5) == rd) {
            leak(rX(&regs, i), rX(&regs, rs1), mem_last_addr, req_data);
        } else {
            leak(rX(&regs, i));
        }
    }
    mem_last_read = req_data;
    mem_last_addr = addr;
}
```

As for Xor gates, as can be seen in definition (9), the leakage of such a gate c is only stable if its inputs are $\mathbb{0}$ or $\mathbb{1}$. Analogously, we have the stability definition

$$\tau_c\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) := \left(\tau_a\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) \wedge \tau_b\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right)\right). \tag{12}$$

Finally, the stability of negated gate versions Nand, Nor, and Xnor has the same definition as their non-negated counterparts And, Or, and Xor.

Difference. The difference $\delta_g(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i})$ encodes if $\lambda^{\square}_g(\sigma^{\square}_{i-1},\sigma^{\square}_{i}) \neq \lambda^{\square}_g(\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i})$. Appropriately, for locations $g \in V^{\square}$, the difference is encoded as

$$\delta_g\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) := \left(g\left(\sigma^{\square}_{i-1}\right) \neq g\left(\hat{\sigma}^{\square}_{i-1}\right)\right) \vee \left(g\left(\sigma^{\square}_{i}\right) \neq g\left(\hat{\sigma}^{\square}_{i}\right)\right), \tag{13}$$

trivially encoding $\lambda^{\square}_g(\sigma^{\square}_{i-1},\sigma^{\square}_{i}) \neq \lambda^{\square}_g(\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i})$. Similarly, Not gates do not change the difference, with $\delta_c(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}) := \delta_a(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i})$, and Xor gates inherit the difference if either input has a difference, same as in definition (9), with

$$\delta_c(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}) := \delta_a(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}) \vee \delta_b(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}). \tag{14}$$

For And gates, the definitions (4) and (5) suggests that the output c inherits a difference from input a if input b is either unstable or $\mathbb{1}$, and vice versa. Thus, for an And gate c:

$$\delta_c\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) := \begin{array}{l} \delta_a\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) \wedge \left(\neg\tau_b\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) \vee b\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i}\right)\right) \vee \\ \delta_b\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) \wedge \left(\neg\tau_a\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i},\hat{\sigma}^{\square}_{i-1},\hat{\sigma}^{\square}_{i}\right) \vee a\left(\sigma^{\square}_{i-1},\sigma^{\square}_{i}\right)\right). \end{array} \tag{15}$$

The encoding for Or gates is similar, with the $a(\sigma^{\square}_{i-1},\sigma^{\square}_{i})$ and $b(\sigma^{\square}_{i-1},\sigma^{\square}_{i})$ terms being negated. Moreover, the encoding for negated gate veriants Xnor, Nand, Nor is the same as for these non-negated variants listed.

4 Software Compliance Verification

In this section we describe how we verify end-to-end security of a system by verifying compliance of a program with respect to a contract. In Section 4.1 we show that software compliance implies end-to-end security under the t-probing model by proving that when a processor's hardware complies with a contract, the contract effectively models all gate-level leakages. We then give an overview of our software compliance verification approach. Starting from a user-defined initial state, we use symbolic execution to obtain the leakage of a program execution. While the resulting leakage trace can be verified with any masking verification approach, we utilize the verification approach introduced by Gigerl et al. [GHP+21], with is based on the over-approximation of correlations between secrets and leaked values.

4.1 End-to-end t-Probing Security

Software implementations compliant with a contract can be executed on any compliant hardware. While this property has been shown to hold for the t-(S)NI probing model [BGG+22], we provide a proof that the same property holds for the less restricting t-probing security notion of Definition 1.

Theorem 2 (End-to-end t-probing security)**.** *Let the setting be as in Theorem 1, and let* $\langle \sigma^{\boxdot}_{-1}, \sigma^{\boxdot}_{0} \rangle \simeq_{\mathcal{M}} \sigma^{\boxminus}_{0}$. *If* $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\boxminus}, \boldsymbol{q}^{\boxminus}, \boldsymbol{\lambda}^{\boxminus}_{n} \rangle$ *is t-probing secure, then* $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\boxdot}, \boldsymbol{q}^{\boxdot}, \boldsymbol{\lambda}^{\boxdot}_{m} \rangle$ *is also t-probing secure.*

Proof. From Theorem 1, we know that for every tuple $\boldsymbol{e}^{\boxdot} \subseteq \boldsymbol{\lambda}^{\boxdot}_{m}$, there is tuple $\boldsymbol{e}^{\boxminus} \subseteq \boldsymbol{\lambda}^{\boxminus}_{n}$, and functions $f_{\lambda,i}$ such that $e^{\boxdot}_{i} = f_{\lambda,i}(e^{\boxminus}_{i})$. Interpreting functions $f_{\lambda,i}$ as coordinate functions of f_{λ}, we have $\boldsymbol{e}^{\boxdot} = f_{\lambda}(\boldsymbol{e}^{\boxminus})$. Since $\boldsymbol{x} \to \boldsymbol{e}^{\boxminus} \to f_{\lambda}(\boldsymbol{e}^{\boxminus})$ is a markov chain, we can apply the data-processing inequality to get

$$I\left(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{q}^{\boxdot}, \boldsymbol{e}^{\boxdot}\right) = I\left(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{q}^{\boxminus}, f_{\lambda}(\boldsymbol{e}^{\boxminus})\right) \leq I\left(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{q}^{\boxminus}, \boldsymbol{e}^{\boxminus}\right). \tag{16}$$

Whenever $|\boldsymbol{e}^{\boxdot}| = |\boldsymbol{e}^{\boxminus}| \leq t$, t-probing security of $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\boxminus}, \boldsymbol{q}^{\boxminus}, \boldsymbol{\lambda}^{\boxminus}_{n} \rangle$ implies that $I\left(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{q}^{\boxminus}, \boldsymbol{e}^{\boxminus}\right) = 0$. The non-negativity of mutual information, together with (16) gives $I\left(\boldsymbol{x}; \boldsymbol{p}, \boldsymbol{q}^{\boxdot}, \boldsymbol{e}^{\boxdot}\right) = 0$, and therefore $\langle \overline{\boldsymbol{x}}, \boldsymbol{p}, \boldsymbol{r} \rangle \mapsto \langle \overline{\boldsymbol{y}}^{\boxdot}, \boldsymbol{q}^{\boxdot}, \boldsymbol{\lambda}^{\boxdot}_{m} \rangle$ is also t-probing secure. $\square$

4.2 Software Compliance Verification Concept

Verification of software compliance is performed in three consecutive stages:

1. Initial State Definition. Before verification, the initial state of the program must be explicitly defined by the user. This entails specifying the contents of registers and memory. Random shares and masks are specified as symbols, while all other values are concrete. This distinction is necessary as verification must account for arbitrary values of secrets, shares and masks.

2. Generation of Leakage Trace. As we perform verification for arbitrary values of shares and masks, they are treated as symbolic variables while generating the leakage trace. A representation of the leakage exhibited during the execution is obtained by symbolically executing the contract until program termination. The outcome of the symbolic execution is a sequence of leaks, each represented by a computation tree with secrets, masks and constants as leaf nodes. To account for bit-sliced or n-sliced implementations, we perform tracing on a user defined granularity, allowing multiple bits to be treated as one symbolic value. This computational complexity of this step stems from the semantic of the contract, but for practical purposes can be considered linear in the size of the program.

3. Analysis of Leaks. The symbolic representation of the leakage trace is agnostic to the masking security model and verification method used. In this paper we verify t-probing security by constructing correlation sets, which serve as an over-approximation of probing security. The proof is then performed by translating t-probing security over correlation sets into a single SAT problem, which can then be solved with any modern SAT solver.

4.3 Initial State Definition

Before simulation, the initial state of the system must be specified. This includes the location of secrets, random shares and public values in the registers and memory. This is achieved via a label file that has to be supplied for each program. Labels can include

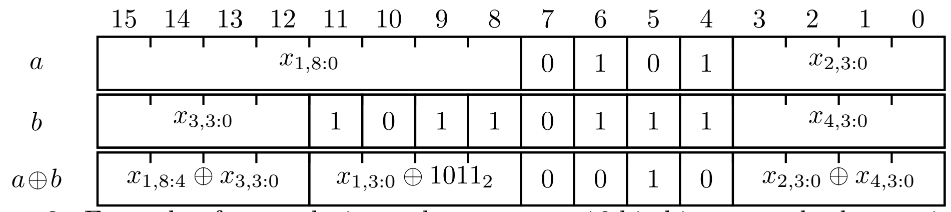


Figure 3: Example of an exclusive-or between two 16-bit bitvectors, both containing different sized slices of symbolic and concrete values.

constants, random, and secret values and can either apply to contract registers or memory locations. Any state variable of the contract may be occupied by multiple slices of different secrets, which is typical for bit-sliced and word-sliced implementations. Figure 3 shows an example configuration of contract variables a and b, each containing a combination of symbolic shares and concrete values.

4.4 Generating Leakage Traces

Execution of machine code in a contract 🗎 is based on its semantics $\langle \sigma_0^{🗎}, \langle\rangle\rangle \xrightarrow{n🗎} \langle \sigma_n^{🗎}, \boldsymbol{\lambda}_n^{🗎}\rangle$, which we implement by combining the contract with our symbolic tracing library to obtain an optimized simulator for the contract. Each time a leak statement in the contract is executed, a symbolic representation of its parameters is recorded. These records form the list of symbolic leak invocations $\boldsymbol{\lambda}_n^{🗎}$ at the end of execution.

The `bv` class represents a concatenation of bitvector slices, each representing either a concrete value or a symbolic expression. The granularity of tracing is user-defined through the initial state configuration. Different levels of granularity provide a trade-off between accuracy and verification time. For bit-sliced implementations it might be necessary to perform tracing for individual bits, while for n-sliced implementations it is sufficient to trace register values in a granularity of n-bit sized slices.

To denote a slice of multiple secret bits we write $x_{n,e:s} = (x_{n,e}, \dots, x_{n,s})$. To be consistent with the typical notation in hardware design, the second index is the start and the first is the end of the series. After the initial state of the memory and the contract's register and leakage state variables has been assigned, the contract is executed according to the semantics of the C programming language. Whenever the contract performs operations on bitvectors, new bitvectors are created that preserve as much generality of the symbolic representation as possible. For example, the result of an exclusive or between two bitvectors $a = (x_{1,1}, x_{2,1})$ and $b = (x_{1,2}, x_{2,2})$ is the bitvector $c = (x_{1,2} \oplus x_{2,1}, x_{2,1} \oplus x_{2,2})$ where each slice is calculated individually. This is important for bit-sliced implementations, as merging the slices into a single expression would result in combined leakage of all slices.

Figure 3 shows a more elaborate example. When performing an operation on two bitvectors consisting of slices of different lengths, new slices must be formed. For logical operations (such as Not, And, Or, Xor), each output bit can be computed from the single corresponding bit of each input. The result of logical operations between two `bv` instances are slices that overlap in both input vectors.

For other operations, such as arithmetic operations, a single output bit may depend on multiple input bits. Since arithmetic operations are not needed for the implementation of Boolean masked implementations, we over-approximate the result by combining all input slices through an uninterpreted function. To minimize the complexity and memory consumption of symbolic expressions, we perform a simplification of all slices of a `bv` computation result. Table 1 shows the rules used for simplification.

The leakage behavior of a contract is characterized by invocations of the `leak` function. Each invocation saves a copy of all fragments of all of its arguments as a symbolic expression. Tracing results is a list of symbolic leaks used to prove software compliant with the contract.

Table 1: Simplification rules for symbolic tracing. The symbolic template f is simplified to f'. Vector $\mathbf{a}$ denotes a vector containing secrets or masks, $\mathbb{0}$ and $\mathbb{1}$ denote vectors with all values 0 and 1 respectively.

Operation type	f	f'
Linear	$\mathbf{a} \oplus \mathbb{0}$	$\mathbf{a}$
	$\mathbf{a} \oplus \mathbb{1}$	$\mathbf{a}$
Non-linear (logical)	$\mathbf{a} \wedge \mathbb{0}$	$\mathbb{0}$
	$\mathbf{a} \wedge \mathbb{1}$	$\mathbf{a}$
	$\mathbf{a} \vee \mathbb{0}$	$\mathbf{a}$
	$\mathbf{a} \vee \mathbb{1}$	$\mathbb{1}$

Operation type	f	f'
Arithmetic	$\mathbf{a} + \mathbb{0}$	$\mathbf{a}$
	$\mathbf{a} - \mathbb{0}$	$\mathbf{a}$
Shift (SLL, SRA, SRL)	$\mathbf{a} << \mathbb{0}$	$\mathbf{a}$
	$\mathbb{0} << \mathbf{a}$	$\mathbb{0}$

4.5 Correlation Sets and SAT Encoding

The leakage trace resulting from symbolic simulation can be verified with any suitable software or hardware leakage verification tool, such as Coco [GHP+21], Silver [KSM20], MaskVerif [BBC+19] or VerifMSI [MT23]. To make our results more comparable we chose the verification strategy of COCO, as it verifies security with regard to gate-level leakage. We give a brief overview of the verification approach, while referring to the original publication [GHP+21] for a more in-depth elaboration of the verification process. Directly verifying the statistical independence of secrets and leaks if computationally infeasible [HB21]. Instead it is sufficient to track the over-approximation of the Fourier expansions of Boolean functions originally employed by Bloem et al. [BGI+18]. Each argument of each leak invocation is translated into a corresponding correlation set. Correlation sets are constructed recursively according to the symbolic computations. The verification of t-probing security is then performed by encoding the attackers ability to place t probes as a choice of t leaks during the execution of the contract. We then verify that no combination of t leak statements correlates to the secrets by encoding the problem as a Boolean formula, efficiently solvable via a single query to a modern SAT solver.

5 Experiments

To demonstrate the practicality of our extended leakage model and verification tool, we construct a leakage contract for the Ibex processor. The modular verification approach of contracts allows proving hardware and software compliance individually, ensuring compatibility without requiring reverifying unchanged components.

5.1 Verifying the Ibex processor

To prove hardware compliance of our contract with the Ibex processor with our tool, we provide the contract and additional configuration for the hardware verifier. The configuration defines normal operating conditions of the processor via constraints on the hardware state and includes a mapping between hardware and contract registers. All of the fully automated verification steps were performed on an Intel Xeon E5-4669 CPU with 88 logical cores at 2.2 GHz clock speed. Verification that the contract and hardware registers stay equivalent under normal operating conditions takes approximately 40 minutes. Finding the set of contract variables that can simulate the pre-cycle registers can be achieved in 10 minutes. As a last step, all gate's leakage effects must be proven simulatable by contract leaks, which takes 11.3 hours. In total, verification of the Ibex processor takes 12.1 hours. To the best of our knowledge, the only other published contract verification tool for masking security is from Bloem et al. [BGG+22], who report a total verification time of 35.5 hours, while only verifying transition leakage and ignoring glitches.

Table 2: Verifying t-probing security of software implementations for IBEX results in the same confirmation of security at reduced verification time and validates our approach.

Computation	# Instr.	Input shares	Randomness	Verification runtime		
				COCO	Ours	
					Bit-wise	Word-wise
First-order						
`XOR`	13	4×32 bit	-	92 s	2 s	1 s
`AND` (DOM-dep reg.)	13	4×32 bit	1×32 bit	204 s	7 s	1 s
`AND` (DOM-dep)	24	4×32 bit	1×32 bit	428 s	7 s	1 s
`AND` (HPC1)	31	4×32 bit	2×32 bit	321 s	9 s	1 s
`AND` (TI)	36	6×32 bit	-	384 s	7 s	1 s
`AND` (ISW)	33	4×32 bit	1×32 bit	392 s	8 s	1 s
Keccak S-box (DOM-dep)	424	10×32 bit	5×32 bit	4.9 m	4.2 m	1.1 m
AES S-Box (DOM-dep)	3196	16×16 bit	35×16 bit	4.1 h	3.8 h	15.5 m
Second-order						
`AND` (DOM-dep)	40	6×32 bit	3×32 bit	623 s	15 s	3 s
`AND` (HPC1)	52	6×32 bit	5×32 bit	643 s	18 s	3 s
Third-order						
`AND` (DOM-dep)	71	8×32 bit	6×32 bit	2.7 m	1.9 m	13 s
`AND` (HPC1)	89	8×32 bit	9×32 bit	2.9 m	2.1 m	13 s

5.2 Software Verification Experiments

We evaluate our software verification tool and IBEX contract by verifying multiple masked implementations of gadgets and S-boxes. All implementations initially targeted non-glitch-aware contracts and were insecure under our glitch-aware model, exemplifying the importance of considering glitches in the hardware model. We modified the implementations to adhere to our glitch-aware contract, sometimes requiring more instructions.

To demonstrate the runtime improvements of contracts compared to directly verifying the hardware netlist, we verify the same implementations with COCO [GHP+21], which implements the same verification strategy, but directly targets the design's netlist. While COCO is capable of including the system memory in the side-channel evaluation, we explicitly exclude it in our experiments to provide a fair comparison to our tool. All implementations operate on entire registers to handle shares, allowing bit-sliced processing of up to 32 sharings simultaneously. We test our tools with two input share configurations for each implementation, one that verifies on the word-level and one that verifies each input bit individually, which is closer to the way COCO performs verification.

Table 2 shows the verification runtime results for both tools, all of which deemed the implementations secure. Our bit-wise verification approach is faster than COCO, which needs to parse and evaluate the processor netlist on every evocation, leading to slow verification times for small programs. Verification on a word-granular level is significantly faster than verification on a bit-granular level, as the solver does not need to perform the proof for each bit individually. Compared to COCO, our word-wise verification approach is faster by at least a factor of four.

6 Conclusion

We introduced extended hardware-side-channel leakage contracts that account for glitches and transitions. Our model can be used to verify the compliance of the netlist of CPUs with corresponding leakage contracts. Masked cryptographic software implementations can then be independently verified against the contract, guaranteeing end-to-end security of any compliant implementation on any compliant CPU netlist. By applying it to the RISC-V IBEX core, we provide a practical demonstration of its capabilities, showing that end-to-end verification of our glitch-aware leakage model is practically feasible.

References

[BBC$^+$19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *European Symposium on Research in Computer Security – ESORICS*, pages 300–318, 2019.

[BDM$^+$20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Advances in Cryptology – EUROCRYPT*, pages 311–341, 2020.

[BGG$^+$14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications – CARDIS*, pages 64–81, 2014.

[BGG$^+$21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 189–228, 2021.

[BGG$^+$22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. Power contracts: Provably complete power leakage models for processors. In *Conference on Computer and Communications Security – CCS*, pages 381–395, 2022.

[BGI$^+$18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology – EUROCRYPT*, pages 321–353, 2018.

[CGP$^+$12] 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 – COSADE*, pages 69–81, 2012.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS*, pages 168–176, 2004.

[CRB$^+$16] 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*, pages 194–212, 2016.

[dGPdlP$^+$16] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and ARM: friends or foes? *IACR Cryptol. ePrint Arch.*, page 946, 2016.

[FGP$^+$18] 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 Trans. Cryptogr. Hardw. Embed. Syst.*, pages 89–120, 2018.

[GHP$^+$21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In *USENIX Security Symposium*, pages 1469–1468, 2021.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Conference on Computer and Communications Security – CCS*, page 3, 2016.

[GO22] Si Gao and Elisabeth Oswald. A novel completeness test for leakage models and its application to side channel attacks and responsibly engineered simulators. In *Advances in Cryptology – EUROCRYPT*, pages 254–283, 2022.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In *Advances in Cryptology – ASIACRYPT*, pages 3–32, 2021.

[HB21] 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, 2021.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology – CRYPTO*, pages 463–481, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO*, pages 388–397, 1999.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *Advances in Cryptology – ASIACRYPT*, pages 787–816, 2020.

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

[MMT20] 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.*, page 1297, 2020.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *USENIX Security Symposium*, pages 199–216, 2017.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In *Topics in Cryptology – CT-RSA*, pages 351–365, 2005.

[MT23] Quentin L. Meunier and Abdul Rahman Taleb. Verifmsi: Practical verification of hardware and software masking schemes implementations. In *Proceedings of the 20th International Conference on Security and Cryptography, SECRYPT 2023, Rome, Italy, July 10-12, 2023*, pages 520–527, 2023.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

[NPWB18] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *Computer Aided Verification – CAV*, pages 587–595, 2018.

[Plo81] Gordon D Plotkin. *A structural approach to operational semantics.* Aarhus University, 1981.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design – COSADE*, pages 282–297, 2017.

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, pages 200–210, 2001.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems – CHES*, pages 413–427, 2010.

[SCS$^+$21] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *Conference on Computer and Communications Security – CCS*, pages 685–699, 2021.

[WMvG$^+$23] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. *CoRR*, abs/2305.06979, 2023.