ABSTRACT

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

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

KEYWORDS

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

1 INTRODUCTION

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

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

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

On the formal verification side, several works verify the security of masked software implementations under specific masking-related security notions. MaskVerif performs algorithmic software
We then establish a technique to verify working with vendor-supplied contracts of CPUs whose design is with a contract. A processor is compliant when the leakage of each which may not be available, e.g., for most Arm-based embedded de-vices. All other tools operate in fixed software leakage models and their method requires the processor’s netlist, which may not be available, e.g., for most Arm-based embedded devices. All other tools operate in fixed software leakage models and are inadequate to protect software against device-specific leakage.

Our Contribution. We answer to the question of leakage model completeness and establish end-to-end (E2E) security for software executing on a processor. First, we introduce a contract between the hardware and the software that defines precise semantics and models side-channel behavior of assembly instructions. We then establish a technique to verify compliance of a processor with a contract. A processor is compliant when the leakage of each of its gates and the semantic of instruction is correctly specified in the contract. Put vice-versa, we prove the contract’s model of instructions correct and its model of leakages complete. We pave the way for provable E2E security by defining software compliance for threshold probing security notions so that the approach of Barthe et al. can easily be adopted to prove the absence of order-reducing leakage of masked software that is executed on a compliant CPU hardware. The contract, which is based on the official RISC-V reference models, is provided as an artifact.

2 SIDE-CHANNEL RESILIENCE

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

2.1 Hardware Model and Gate-level Leakage

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

The execution of a circuit happens in clock cycles. For a state \(\sigma^h\), we denote the next state as \(\sigma^{h+1}\). The registers of the next state have values reflecting the values of their inputs in the previous cycle, i.e., \(g(\sigma^h_{\tau_{in}}) = g' \cdot (\sigma^h_{\tau}) \quad \text{with} \quad (g', \sigma) \in W\), whereas the next state of circuit inputs is determined by the environment.
We now proceed to define the power side-channel leakage that is exposed to an adversary. The root cause of power side-channels is that CMOS logic draws power, or emits electromagnetic radiation, mainly if a transistor switches its state. Thus, CMOS gates have a data-dependent power consumption. The leakage behavior of CMOS gates themselves is relatively well understood and can be modeled by a few simple leakage effects which allow an (idealized) probing adversary to observe the (intermediate) values of gates and wires without any loss due to measurement noise [5, 20, 30]. The seminal work of Ishai et al. [30] introduced value leakage which allows an idealized adversary to observe the value of any wire connected to a gate at the beginning or end of a cycle, i.e., its stable value. The value leakage \( \lambda_g \) exposed by the gate is its value \( \lambda_g(s^h_g) = g(s^h_{j-1}) \) in the state \( s^h_g \). Besides value leakage, additional leakage effects are also observable in hardware. We define our extended probing model in close relation to the robust probing model of Faust et al. [20]. Transition leakage refers to the phenomenon that the power consumption of CMOS gates depends on the charges (state) of the gate before computation. As such transition leakage allows observing whether the value of a gate changed during a clock cycle but also whether the value changed from zero to one or vice versa. Formally, our idealized adversary is able to observe the initial value and the resulting value of each gate. The observable gate leakage is then the concatenation of the old and new gate values, i.e., \( \lambda_g(s^h_{j-1}, s^h_j) = g(s^h_{j-1}) | g(s^h_j) \). This sufficiently captures any real-world transition leakage function computable from the old and new gate values. In addition to these main phenomena, there are glitches caused by propagation delay in the temporary logic states of combinatorial circuits within one clock cycle (and thus rather ephemeral) [35] and couplings caused by inductive coupling of adjacent wires [17]. These can be modeled by defining \( \lambda_g(s^h_{j-1}, s^h_j) \) for non-register gates as the concatenation of all possible values the gate \( g \) could take on due to these effects. We use \( L^h_{0,m} \) to denote the observable gate-level leakage throughout the execution starting in state \( s^h_0 \) and ending in state \( s^h_m \). While the techniques described in the following apply to all effects, for the purpose of this paper, we focus on value leakage and transition leakage. Hence, we define \( L^h_{0,m} = \{ \lambda_g(s^h_{j-1}, s^h_j) : g \in G, 1 \leq j < m \} \).

### 2.2 Provable Security and Simulatability

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

We give an overview of the notation associated with masking, and formalize gadgets and their security. Masking heavily relies on random variables. We write the names of random variables in lowercase, e.g., \( x \), and use lowercase boldface names for sets of variables, e.g., \( x = \{x_0, \ldots, x_n\} \). Each random variable \( x_i \) respectively set \( x \) is associated with a probability distribution \( \Pr[x_i] \), respectively \( \Pr[x] \). Each secret \( x_i \) is encoded (masked) using \( d \geq t + 1 \) shares and we write \( x_i^d = \{x^d_0, \ldots, x^d_d\} \) for the shares which encode \( x_i \), where \( x^d_j \) denotes for \( 0 \leq i < n \) and \( 0 \leq j < d \) the \( j \)th share of the \( i \)th secret. The set of all shares is denoted by \( x = \{x_0, \ldots, x_{n-1}\} \).

A gadget operates on input tuple \((X, r, p)\) returning tuple \((\bar{y}, o, L)\), each consisting of random variables. The input shares \( X \) are a set of \( t \)-wise independent encodings \( x_i \) of each encoding a secret variable \( x_i \), \( r \) represents a set of independent and uniformly random variables, \( p \) are public inputs independent of secrets. The output of a gadget consists of output shares \( \bar{y} \), public outputs \( o \), and observable leakage \( L \). Each individual output is a random variable \( y_i \) (respectively \( o_i \)) and computed as a function of the gadget’s inputs, i.e., \( y_i^t = f_i^t(X, r, p) \). During its execution a gadget produces observable leakage \( L = \{l_0(X, r, p), \ldots, l_m(X, r, p)\} \), which an attacker can observe, e.g., through power measurements. The attacker’s goal is to learn information about the unshared secret inputs \( x \).

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

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

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

**Definition 2 (Modeling Function).** Let \( f_H : H \rightarrow V \) and \( f_C : C \rightarrow U \) be deterministic functions. We say that a deterministic function \( f_S : U \rightarrow V \) is a modeling function which models \( f_H \) from \( f_C \) under deterministic \( \Psi : H \times C \rightarrow \Xi \) whenever
\[
\forall h \in H, c \in C : \Psi(h, c) \Rightarrow f_S(f_C(c)) = f_H(h). \tag{1}
\]

Definition 2 is strong: whenever modeling function \( f_S \) models \( f_H \) then it also simulates it, captured by Lemma 1.
LEMMMA 1 (MODELING FUNCTIONS SIMULATE). \( \text{Let } h \text{ and } c \text{ be sets of possibly dependent random variables with deterministic function } f_{\text{hs}} \text{ computing a set of dependent random variables } v \text{ and function } f_{\text{cc}} \text{ computing dependent random variables } u. \) \( \text{If function } f_{\text{hs}} \text{ models } f_{\text{cc}} \text{ whenever } \Psi(h, c) \text{ then it also simulates } v: \)
\[
Pr [f_{\text{hs}} \circ f_{\text{cc}}(c) | \Psi(h, c) = \top] = Pr[h | \Psi(h, c) = \top].
\]

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

DEFINITION 3 (STATEFUL \( t \sim (S)NI \) [7–9]). \( \text{Gadget } G(X, r, p) = (\tilde{y}, o, \mathcal{L}) \text{ is Stateful } t \sim (S)NI \text{ if for every set } e \subseteq \mathcal{L} \cup \tilde{y}, \text{ with } |e| \leq t, \text{ there exists a subset of input shares } s \subseteq X, \text{ with } \forall i : |s \cap \tilde{x}| \leq t', \text{ a set of uniformly random variables } r', \text{ a modeling function } F : \text{Dom}(p) \rightarrow \text{Dom}(o) \text{ modeling public outputs } o \text{ from public inputs } p \text{ and a simulator } S : \text{Dom}(s, r', p) \rightarrow \text{Dom}(e) \text{ simulating observations } e \text{ from a subset of shares } s, \text{ random } r' \text{ and public inputs } p \text{ such that } Pr[S(s, r', p), F(p)] = Pr[e, o] \text{ and } o = F(p). \) For \( t\sim NI \) \( t' = \emptyset \) while the stricter \( t \sim SNI \) notion requires \( t' = |e| \).

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

3 HARDWARE-SOFTWARE CONTRACTS

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

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

3.1 Expressing Contracts in Genoa

A contract defines the instruction semantics and exposed side-channel information of a processor from the perspective of a software developer, i.e., which data is leaked via power side-channels when an instruction is executed in conjunction with the semantic of the instruction. Contracts must specify correct instruction semantics to be able to express accurate data leakage. Besides the instruction perspective contracts allow to execute and thereby model entire programs. In practice, a contract is a user-supplied text file containing specifications of instructions written in Genoa, which is an extension of the industry-grade \textsc{Sail} DSL [4]. In a nutshell, \textsc{Sail} is designed to model the semantic of instructions of arbitrary ISAs and we add the ability to specify leakage.

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

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

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

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

// adopted from RISCV Sail Model, see license in Listing 6
type regidx = bits(5) // index of register 0b00000 = x1
enum rop = {RISCV_ADD, RISCV_SUB, RISCV_SLL, RISCV_SLT,
            RISCV_SLLU, RISCV_XOR, RISCV_SRL, RISCV_SRA,
            RISCV_OR, RISCV_AND}
union clause ast = RTYPE : (regidx, regidx, regidx, rop),
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
<= 0b00000000 @ rs2 @ rs1 @ 0b0000 @ rd @ 0b00100011
    mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLL)
<= 0b00000000 @ rs2 @ rs1 @ 0b0010 @ rd @ 0b00100011

let result : bits(32) = match op {
    RISCV_ADD => rs1_val + rs2_val,
    RISCV_SLL => EXT2(bool_to_bits(rs1_val <s rs2_val)),
    ...};
let ret = execute(instruction);
ife ret {
    tick_pc();
    nextPC = PC + 4;
    return RETIRE_SUCCESS
}

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

3.2 Contract Formalization

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

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

\[
(\delta, s, L) \mapsto (\delta', s', L').
\]

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

\[
(\delta, \text{leak} (v_1, \ldots, v_n); s, L) \mapsto (\delta, s, L \cup \{v_1(\sigma)|\ldots|v_n(\sigma')\}).
\]

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

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

\[
\chi (\sigma_i^c) = (\sigma_{i+1}^c, L^c_i)
\]

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

\[
\sigma_i^c \xrightarrow{L^c_i} \sigma_{i+1}^c
\]

Finally, we define the execution of programs in a contract. Contract \( \prod P \) models the execution of program \( P \) starting in initial state \( \sigma_0^c \) and resulting in state \( \sigma_n^c \) while producing the accumulated observable side-channel information \( L_{\sigma_0}^c = \bigcup_{i=0}^{n-1} L^c_i \), i.e.,

\[
\prod P \xRightarrow{\{\sigma_0^c\}} (\sigma_n^c, L_{\sigma_0}^c).
\]

3.3 Software Security

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

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

**Definition 4** (\( t \mapsto (S)N \) of \( \llbracket P \rrbracket \) under \( \pi_{in}, \pi_{out} \)). An implementation \( P \) of gadget \( G \) is \( t \mapsto (S)N \) secure w.r.t. semantic \( \llbracket P \rrbracket \) and placement policies \( \pi_{in}, \pi_{out} \) if the gadget \( G \) is \( (x, r, p) = (y, o, L_{0,n}) \) is \( t \mapsto (S)N \) equivalent to Definition 3, where the inputs of the gadget correspond to the starting states \( \sigma_0 = \pi_{in} (x, r, p) \) while leakages and gadget outputs correspond to the random variables \( \pi_{out} (y, o) = \sigma_n \) resulting from execution \( \llbracket P \rrbracket (\sigma_0) = (\sigma_n, L_{0,n}) \).

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

### 3.4 Hardware Compliance With a Contract

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

A program \( P \) executed in initial hardware state \( \sigma_0^h \) leads to leakages \( L_{0,m}^h \), and final state \( \sigma_m^h \) when executed on processor \( \llbracket h \rrbracket \):

\[
\llbracket P \rrbracket^h (\sigma_0^h) = (\sigma_m^h, L_{0,m}^h) \quad (8)
\]

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

\[
\sigma_j^h \xrightarrow{L_{j,im}^h} \sigma_{j+1}^h. \quad (9)
\]

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

**Figure 1: Compliance for a full program execution**

In Definition 5 we introduce a Boolean relation between hardware state \( \sigma^h \) and contract state \( \sigma^c \) expressing that the values contained at a specific location \( \sigma^h \) in the hardware can be modeled from a location \( \sigma^c \) in the contract, e.g., register \( x \) models its counterpart in hardware. Which contract locations model some hardware location is defined in the simulation mapping \( M \) provided by users alongside every contract and checked by our tool. The mapping specifies for all registers in the hardware (including finite state machines, decode stages, etc.) a location in the contract modeling the hardware location. To ease notation assume there are contract locations \( \sigma_0^c, \sigma_1^c \in V^c \) which are constant zero, respectively one, and are later on used to express constraints on hardware execution.

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

\[
\bigwedge_{(\sigma^h, \sigma^c) \in M} \sigma^c (\sigma^h) = \sigma^c (\sigma^c) \quad (10)
\]

Simulation mapping \( M \) is said to be complete if for all hardware locations \( V^h \) a mapping is defined.

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

**Definition 6** (Compliance: \( \llbracket P \rrbracket^h \vdash_M \llbracket P \rrbracket^c \)). A hardware implementation is compliant with a contract under simulation mapping \( M \) if for every program \( P \) and starting hardware and contract states \( \sigma_0^h \) and \( \sigma_0^c \), the program executions

\[
\llbracket P \rrbracket^c (\sigma_0^c) = (\sigma_m^c, L_{0,m}^c) \quad \text{and} \quad \llbracket P \rrbracket^h (\sigma_0^h) = (\sigma_m^h, L_{0,m}^h)
\]

fulfill the following conditions:

1. **States remain similar**: Whenever \( \sigma_0^h \) and \( \sigma_0^c \) are similar under \( M \), so are \( \sigma_m^h \) and \( \sigma_m^c \):

\[
\forall \sigma_0^h, \sigma_0^c : \sigma_0^h \approx_M \sigma_0^c \Rightarrow \sigma_m^h \approx_M \sigma_m^c. \quad (11)
\]

2. **Leaks are modeled**: For every leak \( \lambda_g (\sigma_{g-1}^h, \sigma_g^h) \in L_{0,m}^h \) observable in hardware, there exists a leak \( \lambda (\sigma_g^h) \in L_{0,m}^c \) in the contract and a function \( f_x : \text{Dom}(\lambda) \rightarrow \text{Dom}(\lambda_g) \) that models \( \lambda_g \) from \( \lambda \) under relation \( \sigma_0^h \approx_M \sigma_0^c \) according to Definition 2:

\[
\forall \sigma_0^h, \sigma_0^c : \sigma_0^h \approx_M \sigma_0^c \Rightarrow f_x \circ \lambda (\sigma_g^h) = \lambda_g (\sigma_{g-1}^h, \sigma_g^h). \quad (12)
\]

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

### 3.5 End-to-end security

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

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

**Definition 7 (Similar Policy $\langle \pi^h \equiv_M \pi^c \rangle$).** Let contract policy $\pi^c : (d_1, \ldots, d_n) \leftrightarrow \sigma^c$ link sets of values $d_1, \ldots, d_n$ to contract state $\sigma^c$. Hardware policy $\pi^h : (d_1, \ldots, d_n) \leftrightarrow \sigma^h$ is similar to $\pi^c$, denoted $\pi^h \equiv_M \pi^c$ if any pair of contract and hardware states constructed from the same sets of values are similar under mapping $M$:

$$\forall \sigma^h = \pi^h(d_1, \ldots, d_n), \pi^c = \pi^c(d_1, \ldots, d_n) : \sigma^h \equiv_M \sigma^c. \quad (13)$$

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

**Theorem 2 (Model Reduction).** Let $P$ be a program, and the gadgets $G^h(\exists, r, p) = (\vec{g}^h, \sigma^h, L^h_{0,n})$ and $G^c(\exists, r, p) = (\vec{g}^c, \sigma^c, L^c_{0,m})$ correspond to the program executions $[P]^{\pi^c}(\sigma^c)^c$ respectively $[P]^{\pi^h}(\sigma^h)^h$, under policies $\pi^c$ and $\pi^h$ respectively $\pi_{\text{in}}^c$ and $\pi_{\text{out}}^c$, with $\sigma^c_0 = \pi_{\text{in}}^c(\exists, r, p)$, $\sigma^h_0 = \pi_{\text{in}}^h(\exists, r, p)$, $\sigma^c_0 = \pi_{\text{out}}^c(\vec{g}^c, \sigma^c$, $\sigma^h_0 = \pi_{\text{out}}^h(\vec{g}^h, \sigma^h)$. Furthermore, let $\varnothing^h \vdash_M \varnothing^c$, $\pi_{\text{in}}^h \equiv_M \pi_{\text{in}}^c$ and $\pi_{\text{out}}^h \equiv_M \pi_{\text{out}}^c$ under complete mapping $M$. For every set of observations in hardware on $\vec{g}^c$ or $\sigma^h$ there is an equally sized set of observations in the contract on $\vec{g}^h$ or $\sigma^c$ which allows to model the observations under the identity function:

$$\forall \sigma^h \equiv_M \sigma^c \exists e^c \equiv e^h : e^h = e^c \quad \text{(14)}$$
$$\forall e^h \equiv_M e^c \exists \sigma^h \equiv \sigma^c : e^h = e^c. \quad \text{(15)}$$

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

$$\forall e^h_L \subseteq L^h_{0,n} \exists e^c_L \subseteq L^c_{0,n} : e^c_L \equiv e^h_L \wedge e^h_L = T^L(e^c_L). \quad \text{(16)}$$

**Proof.** The gadgets $G^c$ and $G^h$ operate on equally distributed inputs and the policies for hardware are similar, thus for every initial state $\sigma^c_0$ there must be a starting state $\sigma^h_0$ similar under mapping $M$, i.e., $\sigma^h_0 \equiv_M \sigma^c_0$. Since hardware is compliant with the contract, the resulting states are similar as well, i.e., $\sigma^h_0 \equiv_M \sigma^c_0$ and since every observation in $e^h_L$ is an observation on the value of a location in $\sigma^h_0$ it follows directly that there exists a single location in the contract $e^c_L$, respectively $e^h_L$ according to the mapping $M$ which models the observation, fulfilling (14) and (15).

From Lemma 1 and the second compliance clause (12) it follows that every observation $\lambda_{\text{in}}(\sigma^h_{j-1}, \sigma^h_j) \in e^h_L$ can be modeled from some contract leak $\lambda(\sigma^c_j) \in L^c_{0,n}$ using $f_j$ as modeling function. Grouping the necessary $\lambda(\sigma^c_j)$ as the set of random variables $e^c_L$, results in $\left| e^c_L \right| \leq \left| e^h_L \right|$, and defining $T^L$ as the set of respective $f_j$ implies (16), completing the proof.

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

**Corollary 3 (Mixed observations).** Let the setting be as in Theorem 2. Every set of mixed observations on leakage and shared outputs $e^h_L \cup \vec{y}^h \subseteq L^h_{0,n}$, can be modeled from some an equally sized set $e^c_L \subseteq L^c_{0,n}$ using some modeling function $T^L$. y

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

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

## 4 VERIFYING HARDWARE COMPLIANCE

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

### 4.1 Verification Concept

In this section, we first suggest that it is possible to prove a processor compliant without looking at full program executions. We argue that looking at all possible single instruction executions is sufficient to form an inductive argument of compliance. Next, we give an overview of the individual verification steps needed to verify that a processor is compliant with a given contract. Finally, we present a method for directly proving the existence of modeling functions.
This method is the backbone of the verification procedure and relies on encoding constraints into SMT formulas and checking their satisfiability with an SMT solver.

**Single instructions.** Checking compliance for all programs and pairs of processors and contracts using SMT solvers is computationally intractable. Instead we prove compliance inductively by showing that Definition 6 holds for all possible executions of a single instruction. Consequently, we require compliant processors to fulfill the outlined properties at the start and end of each instruction. That is, we show that if the relation \( \sigma_j = \chi(\sigma_i) \), every location in \( \sigma_i \) is equal to the corresponding location in \( \sigma_j \) whenever states are similar throughout the whole execution. That is, we show that states remain similar: the states \( \sigma_{j+k} \) and \( \sigma_{i+1} \) are similar under \( M \) (marked red), i.e., every location in \( \sigma_{j+k} \) is equal to the corresponding location in \( \sigma_{i+1} \). Therefore, we look at every possible instruction duration \( k \) on that particular processor.

Concurrent execution of instructions in the pipeline complicates this approach. For simple pipelines, contracts can easily model the produced leakage because the fetch stage does not operate with security-critical data, and the writeback stage can be made the synchronisation point for the induction, instead of its full retirement. In more complex pipelines, the methods described in this paper require checking leakage produced by hardware components directly influenced by the instruction bits.

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

**Finding modeling functions for gates.** Before verifying that leaks are modeled, we require an intermediate verification step that provides information about the old values of gates. This constrains the old values of each gate \( g \), therefore implicitly constraining the possible values of the corresponding leak \( \lambda_g \). Otherwise, the old value could directly leak secrets, trivially breaking leakage modeling. For every gate \( g \in G \) in the hardware, we show that \( g \) can be modeled by some function \( f_g : B^n \rightarrow B \) that only uses a (small) subset of contract state information \( \theta_g \) as possible. Section 4.4 gives exact definitions of the verification checks and the greedy minimization procedure for \( \theta_g \).

**Verifying that leaks are modeled.** In this verification step, we check whether the hardware leakage is properly modeled from contract leakage for any possible instruction execution, starting in any pair of similar states \( \sigma_{j+k} \approx_M \sigma_{i+1} \). If the check succeeds, the proper modeling throughout any program execution is implied by composition of single instructions. Because we consider transition leakage, we constrain the possible values of gates at the end of the previous instruction. As mentioned before, we use the existence of a modeling function \( f_g \) according to (17) from the previous step.

The hardware leak \( \lambda_g(\sigma_{j+k-1} \cdot \sigma_{j+l}) = g(\sigma_{j+l-1}) || g(\sigma_{j+l}) \) contains information about both the old, and the new values of gate \( g \) for any clock cycle \( l \) of the executed instruction. We analyze each hardware leak function \( \lambda_g \) separately by going through all leakage functions \( \lambda : B^{|V|} \rightarrow B^m \) and checking if there is a function \( f_\lambda : B^m \rightarrow B^2 \) that models \( \lambda_g \), i.e., \( \lambda(\sigma_i) \in L^i \), and \( f_\lambda \) models the previous value of the gate \( g \) from \( \theta_g \). Written formally:

\[
\forall \sigma_{j+l}^h, \sigma_{j+l}^c : \sigma_{j+l}^h \approx_M \sigma_{j+l}^c \land f_g \circ \theta_g \circ \chi(\sigma_i) = g \circ \chi^{k-1}(\sigma_i) \land \\
\lambda(\sigma_i) \in L^i \Rightarrow f_\lambda \circ \lambda(\sigma_i) = \lambda_g(\sigma_{j+l-1}^h, \sigma_{j+l}^h).
\]  

Additionally, we require that for any possible state \( \sigma_i \) at least one leak \( \lambda \) fulfills (18), guaranteeing that Clause 2 of the compliance definition is fulfilled. Section 4.5 gives a more detailed description.

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

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

\[
\forall h, h' \in H, c, c' \in C : \\
\Psi(h, c) \land \Psi(h', c') \land f_{c'}(c) = f_{c'}(c') \Rightarrow f_H(h) = f_H(h').
\]
We see that if all three premises are fulfilled simultaneously, then
\( f \phi \phi \phi \in k \phi \phi \). In this section, we introduce predicates
constraints related to the mapping \( M \) operating conditions
checking the compliance properties. In particular, we define
nal assumptions and restrictions that need to be considered when
fragment and efficiently checked for satisfiability with modern
then there must also be a function mapping between them. More-
\( f \circ f_c (c) \phi \phi \phi \phi \). We aggregate these conditions into \( \phi_{noc} \)
as
\[ \phi_{noc} (\sigma_{\phi}^b, \sigma_{\phi}^c) := \phi_{ret} (\sigma_{\phi}^c) \land \bigwedge_{l=0}^{k-1} \phi_{dev} (\sigma_{\phi}^h) \land \phi_{instr} (\sigma_{\phi}^h). \]

There are also some constraints that concern multiple executions
of the hardware and contract. For such predicates we write \( \phi^* \)
instead. We define \( \phi^*_{ports}(\sigma^h, \sigma^k) \) as the constraint that certain processor
input ports only contain public values. More concretely, for
two executions of the hardware, these input ports are required to
produce identical values. There are also similar execution-spanning
conditions for the contract. For instance, the contract should forbid
the program counter from becoming secret dependent. The predicate
\( \phi^*_{ret}(\sigma^h, \sigma^k) \) expresses these constraints, and is stricter than
both \( \phi_{ret}(\sigma^h) \) and \( \phi_{ret}(\sigma^k) \) separately. Finally, we extend \( \phi_{noc} \) to
\( \phi_{noc} \) over several executions as
\[ \phi_{noc}^* (\sigma_{\phi}^b, \sigma_{\phi}^c, \sigma_{\phi}^k) := \phi_{ret} (\sigma_{\phi}^c) \land \bigwedge_{i=0}^{k-1} \phi_{dev} (\sigma_{\phi}^{h+i}) \land \phi_{instr} (\sigma_{\phi}^{h+i}) \land \phi_{ports} (\sigma_{\phi}^{h+i}). \]

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

4.2 Verification Prerequisites

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

Normal operating conditions. The hardware of a processor has
many input ports and internal registers that are invisible to a soft-
ware developer and are subject to hidden assumptions under normal
operating conditions. In this section, we introduce predicates
\( \phi \) to explicitly represent these assumptions. We use the predicate
\( \phi_{dev}(\sigma^h) \) to represent the usual assumptions a software developer
might have, such as the processor not getting reset, triggering an
interrupt, going into debug mode, or getting memory access errors.
Similarly, there are several internal conditions for the processor
to fetch, start the execution of, and retire an instruction. We
formalize these conditions as \( \phi_{inst}^l (\sigma^h), 0 \leq l < k \) for the \( l \)-th cycle
in \( k \)-cycle instructions and apply them for the intermediate states
\( \sigma_{f_H^k} = \chi_l (\sigma^h) \). Sometimes, the contract is not able to execute an
instruction because it violates some sanity conditions such as the
instruction not being implemented or triggering a fault. We formal-
ize the condition of the contract successfully retiring an instruction
as \( \phi_{ret}(\sigma^c) \). We aggregate these conditions into \( \phi_{noc} \) as
\[ \phi_{noc}(\sigma_{\phi}^b, \sigma_{\phi}^c) := \phi_{ret}(\sigma_{\phi}^c) \land \bigwedge_{l=0}^{k-1} \phi_{dev}(\sigma_{\phi}^h) \land \phi_{instr}(\sigma_{\phi}^h). \]

As introduced in Section 3.4, hardware and contract
states can be similar under a mapping. For expressing that
two states \( \sigma^h \) and \( \sigma^k \) are similar under mapping \( M \), i.e., \( \sigma^k =_{M} \sigma^k \),
we use the predicate \( \phi_{ret}(\sigma^h, \sigma^k) \) as defined in (10). Conversely, we
also require a predicate expressing that all registers, which are not
in the mapping \( M \), are equivalent across hardware executions
of the same program. We specify this property of two hardware states
\( \sigma^h \) and \( \sigma^k \) for the mapping \( M \) and hardware locations \( V^h \) as
\[ \phi_{pub}(\sigma^h, \sigma^k) := \bigwedge_{v^h \in V^h, (\sigma^h, \sigma^k) \in M} \phi_{dev}(\sigma^h) = \phi_{dev}(\sigma^k). \]

4.3 Verifying that States Remain Similar

As introduced in Section 4.1, we verify that the hardware and con-
tract states resulting from the execution of a program are similar by
showing similarity after every instruction. Our inductive argument
assumes that the states \( \sigma_{\phi}^b \) and \( \sigma_{\phi}^c \) are similar at the start of an
instruction, and proves that the states \( \sigma_{\phi}^{j+1} \) and \( \sigma_{\phi}^{j+2} \) are also similar
after the \( k \)-cycle instruction terminates. This is a straightforward
functional equivalence check under the assumption that \( \phi_{noc} \) holds.

Proposition 1. Let \( \sigma_{\phi}^b \) be a hardware state and \( \sigma_{\phi}^c \) the corre-
sponding contract state under mapping \( M \). Furthermore, let \( \sigma_{\phi}^{j+1} = \)
stead, we propose the greedy minimization procedure shown in Algorithm 1.

Algorithm 1 Greedy minimization of required state bits.

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

\(\lambda^k(\sigma^h_j)\) and \(\sigma^c_{i+1} = \chi(\sigma^c_i)\) be the hardware and contract state after the execution of an instruction. Inductively,

\[
\begin{aligned}
\mathcal{V}^h_{\{\sigma^h_j, \sigma^h_j', \sigma^c_i, \sigma^c_i'\} : \phi_{\text{noc}}(\sigma^h_j, \sigma^c_i) \land \psi_{\text{rel}}^h(\sigma^h_j, \sigma^c_i) \land \neg \psi_{\text{rel}}^c(\sigma^h_{j+k}, \sigma^c_{i+1})} \quad (21)
\end{aligned}
\]

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

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

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

\[
\begin{aligned}
\mathcal{V}^h_{\sigma^h_j, \sigma^h_j', \sigma^c_i, \sigma^c_i' : \phi_{\text{noc}}^h(\sigma^h_j, \sigma^c_i) \land \phi_{\text{rel}}^h(\sigma^h_j, \sigma^c_i) \land \psi_{\text{rel}}^c(\sigma^h_j, \sigma^c_i) \land \phi_{\text{pub}}^c(\sigma^h_j, \sigma^c_i)} \quad (22)
\end{aligned}
\]

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

4.4 Finding Modeling Functions for Gates

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

Corollary 6. Let \(\sigma^h_g\) be a hardware state and \(\sigma^c_g\) the corresponding contract state under mapping \(M\) that fulfill both (21) and (22). Furthermore, let \(\sigma_{g,j+k} = \chi(\sigma^h_{g})\) be the last hardware state before the instruction terminates, and \(\sigma^c_{i+1} = \chi(\sigma^c_{i})\) be the contract state after the instruction terminates. Under normal operating conditions, the value of gate \(g\) in cycle \(k - 1\) can be modeled from contract function \(\theta_g\) if and only if

\[
\begin{aligned}
\mathcal{V}^h_{\sigma^h_g, \sigma^h_g', \sigma^c_i, \sigma^c_i' : \phi_{\text{noc}}^h(\sigma^h_g, \sigma^c_i) \land \phi_{\text{rel}}^h(\sigma^h_g, \sigma^c_i) \land \phi_{\text{rel}}^c(\sigma^h_g, \sigma^c_i) \land \phi_{\text{pub}}^c(\sigma^h_g, \sigma^c_i)} \quad (23)
\end{aligned}
\]

\(\theta_g(\sigma^c_{i+1}) = \theta_g(\sigma^c_i) \land g(h^c_{g,j+k-1}) \neq g(h^c_{g,j+k-1})\).

Proposition 2. Let \(\sigma^h_{g,j-1}\) be the predecessor of hardware state \(\sigma^h_g = \chi(\sigma^h_{g,j-1})\), and \(\sigma^c_g\) be similar to contract state \(\sigma^c_g\) under mapping \(M\), fulfilling both (21) and (22). Furthermore, let \(\sigma^h_{g,j} = \chi(\sigma^h_g)\) with \(0 \leq l < k\) be the hardware states reached throughout the execution of a \(k\)-cycle instruction. Let \(\lambda^l_g\) be the leakage function of a hardware gate \(g\), and \(\theta_g\) be a contract function such that (23) holds. Crucially, let \(L^h \subseteq L^c\) be a set of contract leaks, such that for every \(\lambda(\sigma^c_g) \in L^h\):

\[
\begin{aligned}
\mathcal{V}^h_{\sigma^h_{g,j}, \sigma^h_{g,j'}, \sigma^c_i, \sigma^c_i' : \phi_{\text{noc}}^h(\sigma^h_{g,j}, \sigma^c_i) \land \phi_{\text{rel}}^h(\sigma^h_{g,j}, \sigma^c_i) \land \phi_{\text{rel}}^c(\sigma^h_{g,j}, \sigma^c_i) \land \phi_{\text{pub}}^c(\sigma^h_{g,j}, \sigma^c_i) \land \lambda(\sigma^c_i) = \lambda(\sigma^c_i') \land \lambda^l_g(\sigma^h_{g,j+1}, \sigma^h_{g,j+1'}) \neq \lambda^l_g(\sigma^h_{g,j+1}, \sigma^h_{g,j+1'})}
\end{aligned}
\]

The leak function \(\lambda_g\) in cycle \(l\) of a \(k\)-cycle instruction is modeled by a single contract leak function \(\lambda\) under relation \(\phi_{\text{rel}}^h\) and normal
operating conditions $\phi_{noc}$, according Definition 6 if
\[ \forall \sigma^h_i, \sigma^c_i : \phi_{noc}(\sigma^h_i, \sigma^c_i) \land \phi^M_{rel}(\sigma^h_i, \sigma^c_i) \Rightarrow \bigvee_{\lambda(\sigma^c_i) \in \mathcal{L}^s} \phi_{emit}(\sigma^c_i, \lambda). \]

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

### 4.6 Modeling and Implementation

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

**Unfolding circuits into SMT.** Our verification method works with the processor netlist, and uses it to build the SMT formulas previously shown in Section 4. For the most part, this is standard procedure and has been elaborated in the model checking community. In short, the hardware state $\sigma^h_i$ is represented symbolically using propositional variables. Each gate $g$ in the processor is a symbolic expression of the variables representing hardware locations $V$. The expressions are generated by topologically iterating through the circuit and building the representation of each gate $g$ from its inputs and gate type. With regard to clock cycles, the registers of the very first state $\sigma^h_0$, respectively $\sigma^c_0$, are variables. In successor states $\sigma^h_{i+1}$, the registers are determined by their writebacks from the previous cycle. In a sense, we unfold the processor circuit symbolically $k$ times for our verification.

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

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

### 5 VERIFICATION PROCESS

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

The verification requires two manual and four automated steps:

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

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

### 5.1 IBEX Configuration

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

Additionally, we need to make sure that instructions are only retired in the last cycle $k - 1$ of a $k$-cycle instruction by constraining $\text{instr}_{\text{id}}$ done to be $\top$ in the last cycle and $\bot$ otherwise. Similarly, we enforce that the next instruction is fetched exactly in cycle $k - 1$ by constraining $\text{fetch}_{\text{valid}}$ and $\text{id}_{\text{in}}$ ready. We assert that there are no outstanding errors caused by the previous instruction by constraining registers $\text{lsu}_{\text{err}}$, $\text{pm}_{\text{err}}$, $\text{branch}_{\text{set}_{\text{raw}}}$, and $\text{data}_{\text{err}}$ to be $\bot$. To make sure that the state machines in the LSU and control unit start off in a valid state when the instruction starts executing, we add several further constraints. Finally, we also assert that there is no reset through $\text{rst}_{\text{mi}}$ and no interrupt signals $\text{irq}_{\text{x}}$, $\text{debug}_{\text{req}}$ are triggered to match the developers expected behavior.

One of the main challenges in modeling the processor environment is the memory interface. Whenever the processor requests data by setting $\text{data}_{\text{req}}$ to $\top$, the next cycle provides a grant with $\text{data}_{\text{rvalid}}$ set to $\top$ and the corresponding read data being available at $\text{data}_{\text{rdata}}$. Here, we additionally require memory to only provide acknowledgement through $\text{data}_{\text{rvalid}}$.
Listing 4: Common leakage occurring in every instruction.

```javascript
function common_leakage(rs1_val : xlenbits, rs2_val : xlenbits) = {
    leak(rs1_val, rs2_val, rf_pA, rf_pB, mem_last_addr, mem_last_read);
    rf_pA = rs1_val; rf_pB = rs2_val; /* read port */
}
```

Listing 5: Specialized leakage occurring during loads.

```javascript
function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits)
    addr : xlenbits, req_data : xlenbits) = {
    leak(rf_pA, rf_pB, rs1_val, rs2_val);
    leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
    rf_pA = rs1_val; rf_pB = rs2_val;
    leak(addr, req_data, mem_last_addr, mem_last_read);
    mem_last_read = req_data; mem_last_addr = addr; }
```

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

5.2 Complete Power Contract for IBEX

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

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

This common leakage covers leak effects previously discussed in related works. It models transition leakage produced in the ALU, whose source are the two read ports of the register file. Transitions in the first, respectively second, operand of instructions are well-known [36, 37]. Furthermore, the leakage is even caused by invisible outputs of the register file (modeled in leakage states `rf_pA`, `rf_pB`). This leak statement models all transition leakage and value leakage produced in the ALU and the writeback logic. None of the operands in the `leak` statement can be removed without breaking compliance since distinct parts of IBEX cause these combinations. The writeback logic causes additional combinations: ALU or branch instructions following a `load` cause a transition between data from memory access and the current ALU result.

5.3 Discussion

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

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

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

6 Conclusion

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

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

Acknowledgments

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

6.1 RISC-V Model for IBEX

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

Listing 6: Contract model of remaining instructions for IBEX.

```plaintext
/* RISC-V architecture model, comprising all files */
/* and directories except for the snapshots of the */
/* Len and Sail libraries in the prove_snapshots */
/* directory (which include copies of their licences), */
/* is subject to the BSD two-clause licence below. */

/* Copyright (c) 2017-2021 Prashanth Mundkur, Rishiyur S. */
/* Nikhil and Bluespec Inc., Jon French, Brian Campbell, */
/* Robert Norton-Wright, Alasdair Armstrong, Thomas Bauerweis, Shaked Flur, Christopher Pulte, */
/* Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Microsoft, Nathaniel Wesley Filarido, Peter Rugg, and Aril Computer Corp. */
/* and contributions by Scott Johnson */

/* All rights reserved. */
```

/* This software is developed by the above within the */
/* Rigorous Engineering of Mainstream Systems (REMS) */
/* project, partly funded by EPSRC grant EP/K008528/1, */
/* at the Universities of Cambridge and Edinburgh. */
/* This software was developed by SRI International and the */
/* University of Cambridge Computer Laboratory ( */
/* Department of Computer Science and Technology) */
/* under DARPA/AFRL contract FA8650-18-C-7809 (“CIFV”), */
/* and under DARPA contract HR0011-18-C-0016 (“ECATS”) */
/* as part of the DARPA SSITH research programme. */
/* This project has received funding from the European */
/* Research Council (ERC) under the European Union’s */
/* Horizon 2020 research and innovation programme */
/* (grant agreement 789108, ELVER). */
/* This software has received funding from the Federal */
/* Ministry of Education and Research (BMBF) as part of the VE-Jupiter project grant */
/* 10ME0231K. */
/* This work was supported by the Austrian Research */
/* Promotion Agency (FFG) through the FERMION project */
/* (grant number 867542). */
/* Redistribution and use in source and binary forms, with */
/* or without modification, are permitted provided */
/* that the following conditions are met: */
/* 1. Redistributions of source code must retain the above */
/* copyright notice, this list of conditions and the */
/* following disclaimer. */
/* 2. Redistributions in binary form must reproduce the */
/* above copyright notice, this list of conditions and */
/* the following disclaimer in the documentation and/ */
/* or other materials provided with the distribution. */

/* This software is provided by the author and contributors */
/* AS IS’ AND ANY EXPRESS OR IMPLIED WARRANTIES, */
/* INCLUDING, BUT NOT LIMITED TO, THE IMPLIED */
/* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A */
/* PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT */
/* SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY */
/* DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, */
/* OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT */
/* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR */
/* SERVICES; LOSS OF USE, DATA, OR PROFITS; OR */
/* BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY */
/* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT */
/* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR */
/* OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF */
/* THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY */
/* OF SUCH DAMAGE. */
then (res) else (x2) in leak(x2, x2_n);
let x3_n = if (dest_idx == 0b00011) then {res} else {x3} in leak(x3, x3_n);
let x4_n = if (dest_idx == 0b00010) then {res} else {x4} in leak(x4, x4_n);
let x5_n = if (dest_idx == 0b00110) then {res} else {x5} in leak(x5, x5_n);
let x6_n = if (dest_idx == 0b00111) then {res} else {x6} in leak(x6, x6_n);
let x7_n = if (dest_idx == 0b01000) then {res} else {x7} in leak(x7, x7_n);
let x8_n = if (dest_idx == 0b01001) then {res} else {x8} in leak(x8, x8_n);
let x9_n = if (dest_idx == 0b01010) then {res} else {x9} in leak(x9, x9_n);
let x10_n = if (dest_idx == 0b01011) then {res} else {x10} in leak(x10, x10_n);
let x11_n = if (dest_idx == 0b01100) then {res} else {x11} in leak(x11, x11_n);
let x12_n = if (dest_idx == 0b01101) then {res} else {x12} in leak(x12, x12_n);
let x13_n = if (dest_idx == 0b01110) then {res} else {x13} in leak(x13, x13_n);
let x14_n = if (dest_idx == 0b01111) then {res} else {x14} in leak(x14, x14_n);
let x15_n = if (dest_idx == 0b10000) then {res} else {x15} in leak(x15, x15_n);

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

mapping encdec_uop : uop <-> bits(7) = {
    RISCV_LUI <-> 0b0110111,
    RISCV_AUIPC <-> 0b1101111
}
mapping clause encdec = UTYPE(imm, rd, op)
if (rd[4] == bitzero)
function clause execute UTYPE(imm, rd, op) = {
    let rs1_val = X(0b0 @ imm[6 .. 3]);
    let rs2_val = X(0b0 @ imm[11 .. 8]);
    common_leakage(rs1_val, rs2_val);
    let off : xlenbits = EXTS(imm @ 0x000);
    let ret : xlenbits = match op {
        RISCV_LUI => off,
        RISCV_AUIPC => get_arch_pc() + off
    };
    X(rd) = ret;
    RETIRE_SUCCESS
}

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

/* *************************** */
enum uop = (RISCV_LUI, RISCV_AUIPC)
union clause ast = UTYPE : (bits(20), regidx, uop)
mapping encdec_uop : uop <-> bits(7) = {
    RISCV_LUI <-> 0b0101111,
    RISCV_AUIPC <-> 0b0101111
}
mapping clause encdec = UTYPE(imm, rd, op)
if (rd[4] == bitzero)
function clause execute UTYPE(imm, rd, op) = {
    let rs1_val = X(0b0 @ imm[18 .. 15]);
    let rs2_val = X(0b0 @ imm[3 .. 1])
    @ subrange_bits(imm, 11, 11));
    common_leakage(rs1_val, rs2_val);
    let t : xlenbits = PC + EXTS(imm);
    let rd_next = get_next_pc();
    overwrite_leakage(rd, rd_next);
    X(rd) = rd_next;
    set_next_pc(t);
    RETIRE_SUCCESS
}

/* *************************** */
enum clause ast = RISCV_JAL : (bits(21), regidx)
mapping clause encdec = RISCV_JAL(imm_19 @ imm_7_0 @ imm_8 @ imm_18_13 @ imm_12_9
                        @ 0b0, rd)
                        @ 0b1101111
if (rd[4] == bitzero)
function clause execute (RISCV_JAL(imm, rd)) = {
    let rs1_val = X(0b0 @ imm[18 .. 15]);
    let rs2_val = X(0b0 @ imm[3 .. 1])
    @ subrange_bits(imm, 11, 11));
    common_leakage(rs1_val, rs2_val);
    let t : xlenbits = PC + EXTS(imm);
    let rd_next = get_next_pc();
    overwrite_leakage(rd, rd_next);
    X(rd) = rd_next;
    set_next_pc(t);
    RETIRE_SUCCESS
}

/* *************************** */
enum clause ast = RISCV_JALR : (bits(12), regidx, regidx)
mapping clause encdec = RISCV_JALR(imm, rs1, rd)
                        @ 0b1100111
function clause execute (RISCV_JALR(imm, rs1, rd)) = {
    let rs1_val = X(rs1);
    let rs2_val = X(0b0 @ imm[3 .. 1])
    @ subrange_bits(imm, 11, 11));
    common_leakage(rs1_val, rs2_val);
    let t : xlenbits = PC + EXTS(imm) with 0 = bitzero];
    if t[1 .. 0] == 0b00 then {
        set_next_pc(t);
        RETIRE_SUCCESS
    } else RETIRE_FAIL
}

/* *************************** */
enum bop = (RISCV_BEQ, RISCV_BNE, RISCV_BLT, RISCV_BGE,
            RISCV_BLTU, RISCV_BGEU)
union clause ast = BTYPE : (bits(13), regidx, regidx, bop)
mapping encdec_bop : bop <-> bits(3) = {
    RISCV_BEQ <-> 0b000,
    RISCV_BNE <-> 0b001,
    RISCV_BLT <-> 0b100,
    RISCV_BGE <-> 0b101,
    RISCV_BLTU <-> 0b110,
    RISCV_BGEU <-> 0b111
}

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


function clause execute (BTYPE(imm, rs2, rs1, op)) = {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    common_leakage(rs1_val, rs2_val);
    let taken : bool = match op {
        RISCV_BEQ => rs1_val == rs2_val,
        RISCV_BNE => rs1_val != rs2_val,
        RISCV_BLT => rs1_val <_s rs2_val,
        RISCV_BGE => rs1_val >=_s rs2_val,
        RISCV_BLTU => rs1_val <_u rs2_val,
        RISCV_BGEU => rs1_val >=_u rs2_val
    };
    let t : xlenbits = PC + EXTS(imm);
    if (t[1 .. 0] != 0b00) then
        return RETIRE_FAIL;
    if taken then { set_next_pc(t); };
    return RETIRE_SUCCESS
}

/* **************************** */
enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI,
            RISCV_ORI, RISCV_ANDI}
union clause ast =
            ITYPE : (bits(12), regidx, regidx, iop)
            mapping encdec_iop : iop <-> bits(3) = {
                RISCV_ADDI <-> 0b000,
                RISCV_SLTI <-> 0b010,
                RISCV_SLTIU <-> 0b011,
                RISCV_ANDI <-> 0b100,
                RISCV_XORI <-> 0b101,
                RISCV_ORI <-> 0b1100011
        }

mapping clause encdec = ITYPE(imm, rs2, rd, op)
                                      <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
                                      mapping clause encdec = ITYPE(imm, rs1, rd, op)
                                      <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
                                      mapping clause execute (ITYPE (imm, rs2, rd, op)) = {
                                      let rs1_val = X(rs1);
                                      let rs2_val = X(0b0 @ imm[3 .. 0]);
                                      common_leakage(rs1_val, rs2_val);
                                      let immext : xlenbits = EXTS(imm);
                                      let result : bool = match op {
                                          RISCV_ADDI => rs1_val + immext,
                                          RISCV_SLTI =>
                                          EXTZ(bool_to_bits(rs1_val <_s immext)),
                                          RISCV_SLTIU =>
                                          EXTZ(bool_to_bits(rs1_val <_u immext)),
                                          RISCV_ANDI => rs1_val & immext,
                                          RISCV_XORI => rs1_val ^ immext,
                                          RISCV_ORI => rs1_val | immext
                                      };
                                      overwrite_leakage(rd, result);
                                      X(rd) = result;
                                      RETIRE_SUCCESS
                                      }

/* **************************** */
enum sop = {RISCV_SLLI, RISCV_SRLI, RISCV_SRAI}
union clause ast =
            SHIFTIOP : (bits(6), regidx, regidx, sop)
            mapping encdec_sop : sop <-> bits(3) = {
                RISCV_SLLI <-> 0b001,
                RISCV_SRLI <-> 0b101,
                RISCV_SRAI <-> 0b101
        }

mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SLLI)
                                      <-> 0b000000 @ shamt @ rs1 @ 0b001 @ rd @ 0b0010011
                                      bitzero)
                                      mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRLI)
                                      <-> 0b000000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
                                      bitzero)
                                      mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRAI)
                                      <-> 0b010000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
                                      bitzero)

function clause execute (SHIFTIOP(shamt, rs1, rd, op)) = {
    let rs1_val = X(rs1);
    let rs2_val = X(0b0 @ shamt[3 .. 0]);
    common_leakage(rs1_val, rs2_val);
    /* the decoder guard ensures that shamt[5] = 0 for RV32E */
    let result : xlenbits = match op {
        RISCV_SLLI => if sizeof(xlen) == 32
        then rs1_val << shamt[4 .. 0]
        else rs1_val << shamt,
        RISCV_SRLI => if sizeof(xlen) == 32
        then rs1_val >> shamt[4 .. 0]
        else rs1_val >> shamt,
        RISCV_SRAI => if sizeof(xlen) == 32
    };
}
then shift_right_arith32(rs1_val, shamt <- [4..0])
else shift_right_arith64(rs1_val, shamt);
overwrite_leakage(rd, result);
X(rd) = result;
RETIRE_SUCCESS

*/ ******************************************************* */
enum word_width = {BYTE, HALF, WORD, DOUBLE}
union clause ast = LOAD:
(bits(12), regidx, regidx, bool, word_width, bool, bool)
mapping clause encdec = LOAD(imm7 @ imm5, rs2, rs1, size, 
false, false)
function aligned(vaddr : xlenbits, width : word_width) ->
bool =
{ width == BYTE | (width == HALF & vaddr[0] == bitzero) |
width == WORD & vaddr[1 .. 0] == 0b00 ]
val load_leakage : (xlenbits, xlenbits, xlenbits, xlenbits)
-> unit effect (reg, wreg, leakage)
function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits
<- addr : xlenbits, req_data : xlenbits) =
// as in common_leakage
leak(rf_pa, rf_pb, rs1_val, rs2_val);
leak(rf_pa, rf_pb, mem_last_addr, mem_last_read);
rf_pa = rs1_val;
rf_pb = rs2_val;
leak(addr, req_data, mem_last_addr, mem_last_read);
mem_last_read = req_data;
mem_last_addr = addr;
}

function clause execute(LOAD(imm, rs1, rd, is_unsigned,
<- width, aq, rl)) =
let offset : xlenbits = EXTS(imm);
let rs1_val = X(rs1);
let rs2_val = X(0b0 @ imm[3 .. 0]);
let addr = rs1_val + offset;
let req_addr = addr[(sizeof(xlen) - 1) .. 2] @ 0b00;
let req_data = read_mem(Read_plain, sizeof(xlen), req_addr
<- 4, 4);
load_leakage(rs1_val, rs2_val, addr, req_data);
let req_byte : bits(8) = match (addr[1 .. 0]) {
0b00 => req_data[7 .. 0],
0b01 => req_data[15 .. 8],
0b10 => req_data[23 .. 16],
0b11 => req_data[31 .. 24]);
let req_half : bits(16) = match (addr[1]) {
bitzero => req_data[15 .. 0],
bitone => req_data[31 .. 16]);
match (width, addr[1 .. 0]) {
(BYTE, _ ) => process_load(rd, addr, req_byte,
<- is_unsigned),
(HALF, 0b00) => process_load(rd, addr, req_half,
<- is_unsigned),
(HALF, 0b10) => process_load(rd, addr, req_half,
<- is_unsigned),
(WORD, 0b00) => process_load(rd, addr, req_data,
<- is_unsigned),
(_, _) => RETIRE_FAIL // takes care of misaligned}
*/ ******************************************************* */
union clause ast = STORE:
(bits(12), regidx, regidx, word_width, bool, bool)
mapping clause encdec = STORE(imm7 @ imm5, rs2, rs1, size,
false, false)
if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1
<- imm @ rs1 @ bool_bits(is_unsigned) @ size_bits(size) @
rd @ 0b0000011
if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1
<- [4] == bitzero) & (rd
<- [4] == bitzero)

/* **************************** */
RETIRE_SUCCESS
X(rd) = result;
overwrite_leakage(rd, result);
leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
leak(rf_pA, rf_pB, rs1_val, rs2_val);
// address computation and register file access leakage
leak(mem_last_addr, addr);
leak(mem_last_addr, addr);
leak(mem_last_addr, addr);

function clause execute (STORE(imm, rs2, rs1, width, rl,
<- ) ) =
let offset : xlenbits = EXTS(imm);
let rs1_val = X(rs1);
let rs2_val = X(rs2);
common_leakage(rs1_val, rs2_val);
let addr = rs1_val + offset;
// address computation and register file access leakage
leak(mem_last_addr, addr);
mem_last_addr = addr;
if aligned(addr, width) then {
let result = rs2_val;
overwrite_leakage(0b000000, result);
let success : bool = match(width) {
BYTE => write_mem(Write_plain, sizeof(xlen), addr, 1,
<- result[7 .. 0]),
HALF => write_mem(Write_plain, sizeof(xlen), addr, 2,
<- result[15 .. 0]),
WORD => write_mem(Write_plain, sizeof(xlen), addr, 4,
<- result),
_ => false);
if success then (RETIRE_SUCCESS) else (RETIRE_FAIL)
} else { RETIRE_FAIL }
}

/* **************************** */
function clause execute (ILLEGAL(s)) =

6.2 IBEX Configuration

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

Listing 7: IBEX configuration file

```
// Power Contract for IBEX
// Copyright (c) 2020–2022 - TUHH, TU Graz
// All rights reserved.
// This software has received funding from the Federal Ministry of Education and Research (BMBF) as part of the VE-Jupiter project grant 16M60231K.
// Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
// 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
// 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer.
//    in every copying or redistribution of such material or derivative works.
// 3. Related documentation and/or other materials provided with this software may only be redistributed or published with this software.
// 4. The name and logo of the TuHH (T UHH) or Hradec Králové University may not be used for promotion purposes.
// 5. The TuHH or Hradec Králové University, their employees and/or contributors shall not be liable for any damage caused by this software.
// 6. The software is provided "AS IS" and without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement.
// 7. In no event shall the TuHH, TU Graz, or the contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

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

// specification of architectural registers and HW/CT mapping

// PC
contract register PC BitVec 32
hardware public u_ibex_core.pc_id
mapping register PC u_ibex_core.pc_id

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

// REGISTERS
contract register x1 BitVec 32
contract register x2 BitVec 32
contract register x3 BitVec 32
contract register x4 BitVec 32
contract register x5 BitVec 32
contract register x6 BitVec 32
contract register x7 BitVec 32
contract register x8 BitVec 32
contract register x9 BitVec 32
contract register x10 BitVec 32
contract register x11 BitVec 32
contract register x12 BitVec 32
contract register x13 BitVec 32
contract register x14 BitVec 32
contract register x15 BitVec 32

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

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

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

memory rdata data_rdata_i

hardware const@start u_ibex_core.id_stage_i.decoder_i.

hardware const@pre u_ibex_core.instr_id_done 0b1

hardware const@end-1 u_ibex_core.instr_id_done 0b1

// make sure that an instruction has its last cycle in our last cycle

hardware const@start end-1 u_ibex_core.instr_id_done 0b1

// make sure that an instruction has its last cycle in our last cycle

hardware const@start end-1 u_ibex_core.instr_id_done 0b1

hardware const@start end-1 u_ibex_core.id_stage_i.decoder_i.

hardware const@pre: illegal_insn 0b0

hardware const@pre: id_in_ready_o 0b0

hardware const@end-1 u_ibex_core.if_stage_i.fetch_valid 0b1

hardware const@pre u_ibex_core.if_stage_i.fetch_valid 0b1

// do not load a new instruction until last cycle

hardware const@start end-1 u_ibex_core.id_stage.i.

// make sure that nothing retires before the end of the last cycle

hardware const@start end-1 u_ibex_core.instr_id_done 0b0

// make sure that an instruction has its last cycle in our

hardware const@start end-1 u_ibex_core.instr_id_done 0b1

hardware const@pre u_ibex_core.instr_id_done 0b1

hardware const@start u_ibex_core.id_stage_i.decoder_i.

hardware const@pre: illegal_insn 0b0

// important signals that must be constrained

// never trigger a reset of the core

hardware public rst_ni

hardware const@pre: rst_ni 0b1

// make sure that initially, the ID FSM is in state

// this means that we look at the case where we started executing in 6th cycle

hardware public u_ibex_core.id_stage_i.id_fsm_q

hardware const@start u_ibex_core.id_stage_i.id_fsm_q 0b0

// no compressed (valid or invalid) instructions at the // output of instruction fetch stage

hardware public u_ibex_core.if_stage_i.instr_new_id_q

hardware public u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

hardware const@start u_ibex_core.if_stage_i.

// this is a hidden assumption made by IBEX developers

hardware public u_ibex_core.if_stage_i.instr_new_id_q

hardware public u_ibex_core.if_stage_i.instr_rdata_id

// Instruction memory interface

hardware public boot_addr_i

hardware public hart_id_i

hardware public test_en_i

hardware public clk_i

hardware public ram_cfg_i

hardware public data_rvalid_i

hardware public instr_rdata_i

hardware public instr_rdata_id_o

hardware const@start u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

hardware const@start u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

hardware const@start u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

hardware const@start u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.

// this encodes pre cycle and first cycle assumptions

hardware equiv@pre: start+1 u_ibex_core.if_stage_i.
hardware const@pre: irq_nm_i 0b0
// core debug
hardware public debug_req_i
hardware const@pre: debug_req_i 0b0
hardware public fetch_enable_i
hardware public scan_rst_ni

////////////////////////////////////////////////////////////////////////////
// annotate internal state of ibex
////////////////////////////////////////////////////////////////////////////

hardware public core_busy_q
hardware public u_ibex_core.instr_fetch_err
hardware public u_ibex_core.instr_fetch_err_plus2
// instructions in the prefetch fifo
hardware public u_ibex_core.if_stage_i.instr_valid_id_q
hardware public u_ibex_core.if_stage_i.instr_rdata_c_id_o
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.fifo_i.rdata_q0
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.fifo_i.rdata_q1
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.fifo_i.err_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.fifo_i.valid_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.rdata_pmp_err_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.discard_req_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.branch_discard_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.rdata_outstanding_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.fetch_addr_q
hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
  \rightarrow prefetch_buffer_i.stored_addr_q

// instruction decode
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow ctrl fsm_cs
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow load_err_q
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow store_err_q
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow exc_req_q
hardware public u_ibex_core.id_stage_i.branch_set_raw
hardware const@start u_ibex_core.id_stage_i.branch_set_raw 0
  \rightarrow b0
hardware public u_ibex_core.id_stage_i.
  \rightarrow branch_jump_set_done_q
hardware public u_ibex_core.load_store_unit_i.data_we_q
// since data_pmp_err_i is 0, this should not be 1
hardware public u_ibex_core.load_store_unit_i.pmp_err_q

hardware const@start u_ibex_core.load_store_unit_i.pmp_err_q
  \rightarrow 0b0
// since data_err_i is never 1 and pmp_err_q is also not 1,
// this must be 0
hardware public u_ibex_core.load_store_unit_i.lsu_err_q
hardware const@start u_ibex_core.load_store_unit_i.lsu_err_q
  \rightarrow 0b0
hardware public u_ibex_core.load_store_unit_i.
  \rightarrow handle_misaligned_q
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow illegal_insn_q
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow do_single_step_q
hardware public u_ibex_core.id_stage_i.controller_i.
  \rightarrow enter_debug_mode_prio_q

hardware public u_ibex_core.load_store_unit_i.addr_last_q
hardware variable u_ibex_core.load_store_unit_i.addr_last_q
hardware public u_ibex_core.load_store_unit_i.
  \rightarrow rdata_offset_q

contract leakagestate mem_last_read BitVec 32
contract leakagestate mem_last_addr BitVec 32
// Must be idle when new instruction reaches ID/EX
hardware public u_ibex_core.load_store_unit_i.ls_fsm_cs
hardware const@start u_ibex_core.load_store_unit_i.ls_fsm_cs
  \rightarrow 0b000
hardware variable u_ibex_core.load_store_unit_i.data_type_q
hardware public u_ibex_core.load_store_unit_i.
  \rightarrow data_sign_ext_q

// system registers
hardware public u_ibex_core.cs_registers_i.mie_q
hardware public u_ibex_core.cs_registers_i.mtval_q
hardware public u_ibex_core.cs_registers_i.mcause_q
hardware public u_ibex_core.cs_registers_i.mscratch_q
hardware public u_ibex_core.cs_registers_i.dscratch0_q
hardware public u_ibex_core.cs_registers_i.dscratch1_q
hardware public u_ibex_core.cs_registers_i.mstck_q
hardware public u_ibex_core.cs_registers_i.mstck_cause_q
hardware public u_ibex_core.cs_registers_i.mstck_epc_q
hardware public u_ibex_core.cs_registers_i.mstatus_q
hardware public u_ibex_core.cs_registers_i.dcsr_q
hardware const@start u_ibex_core.cs_registers_i.dcsr_q 0
  \rightarrow b00000000000000000000000000000000
hardware public u_ibex_core.cs_registers_i.mhpncounter[0]
hardware public u_ibex_core.cs_registers_i.mhpncounter[1]
hardware public u_ibex_core.cs_registers_i.mcountinhibit
hardware public u_ibex_corecsr_depc
hardware public u_ibex_corecsr_mepc
hardware public u_ibex_corecsr_mnpc
hardware public u_ibex_core.dummy_instr_en

Bloem, Gigerl, Gourjon, Hadžić, Mangard and Primas
hardware public u_ibex_core.dummy_instr_mask
hardware public u_ibex_core.data_ind_timing
hardware public u_ibex_core.icache_enable
hardware public u_ibex_core.debug_mode
hardware public u_ibex_core.priv_mode_id

contract leakagestate rf_pA BitVec 32
contract leakagestate rf_pB BitVec 32