Pantomime: Constructive Leakage Proofs via Simulation

ROBIN WEBBERS, Vrije Universiteit Amsterdam, Netherlands
ROBERT SCHENCK, Vrije Universiteit Amsterdam, Netherlands
WIND WONG, Vrije Universiteit Amsterdam, Netherlands
KRISTINA SOJAKOVA, Vrije Universiteit Amsterdam, Netherlands
KLAUS V. GLEISSENTHALL, Vrije Universiteit Amsterdam, Netherlands

Tools for verifying leakage descriptions of hardware aim to ensure that a given hardware design doesn't leak secrets via its microarchitecture, when executing programs with appropriate countermeasures. However, existing techniques for proving correctness of leakage descriptions are based on non-constructive proofs via non-interference. As a result, they often rely on expensive solvers that offer little help when verification fails or require handwritten invariants, which are difficult to come up with and even harder to debug.

In this paper, we present a new approach to leakage verification which we call *simulation-based leakage proofs.* To show that a leakage description correctly captures a hardware design using a simulation-based proof, the user constructs a simulator—another hardware design that must faithfully replicate all attacker-observable behavior from explicitly leaked secrets. Simulation-based proofs therefore offer a constructive alternative to classic non-interference proofs, exposing a proof object—the simulator, witnessing the correctness claim. As simulators are just programs, we can write, execute and debug them like any other program, making them easy to use. We also show that they can be checked locally, which makes proof checking fast.

We implement simulation-based leakage proofs in Pantomime, a tool that supports writing processors and their leakage proofs in Haskell; we report on using Pantomime to write and verify AIMCore, a 5-stage in-order processor, its leakage description, and simulator, as well as a side-channel hardened version of the core. We show that Pantomime verifies them efficiently (it checks AIMCore in under 40s).

Additional Key Words and Phrases: Hardware, Side-Channels, Leakage

1 Introduction

Context. Side-channel attacks via cache and timing can leak secrets used in private computations [25, 28, 33, 35, 51]. To prevent side-channels in software, programmers use countermeasures like branch balancing [1, 7, 50] or constant-time (or data-oblivious) programming [2, 10, 13, 49]. Whether these countermeasures are effective crucially depends on the underlying hardware. Unfortunately, modern hardware designs are mind-bendingly complex due to their highly parallel nature, their many microarchitectural optimizations like fast paths [3], pre-fetching [14], and speculative execution [24, 31, 37], and their various micro-architectural buffers [45, 46]. This complexity makes it hard to manually audit even simple designs to check whether software with appropriate defenses will truly execute securely.

Problem. To increase our confidence that hardware designs are keeping up their end of the promise, a string of recent work formally verifies existing open-source hardware designs against descriptions of their intended leakage [4, 5, 19–21, 23, 42–44, 47]. However, existing techniques all rely on classic non-interference proofs. As a result, they make heavy use of expensive solvers—either to find inductive invariants [43, 44, 47], or to exhaustively explore the design's state space [23, 42]. As an alternative to fully automated proofs via solvers, one can instead ask the user to supply missing inductive invariants by hand. In fact, several existing methods already require users to supply certain hard-to-find invariants manually [21, 47]. While this helps with scaling, it places a heavy

Authors' Contact Information: Robin Webbers, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, r.m.a.webbers@vu.nl; Robert Schenck, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, r@bert.lol; Wind Wong, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, t.f.w.wong@vu.nl; Kristina Sojakova, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, sojakova.kristina@gmail.com; Klaus v. Gleissenthall, Vrije Universiteit Amsterdam, Amsterdam, Netherlands, k.freiherrvongleissenthal@vu.nl.

burden on the user: inductive invariants are difficult to come up with by hand and even harder to debug when they are wrong [30]. This is especially true for hardware designers, unless they also happen to be experts in formal verification.

Our Solution. In this paper, we propose a new proof technique called simulation-based leakage proofs. Based on insights borrowed from cryptography [11, 27], a simulation-based leakage proof demonstrates the correctness of a leakage description by constructing a *simulator*—another hardware circuit which must faithfully replicate all attacker-observable behavior of the original design, while only being granted access to explicitly leaked secrets. Simulation-based proofs are a constructive alternative to classic non-interference proofs [22], exposing an executable proof object—the simulator, witnessing the correctness claim. As simulators are just programs, they can be written, debugged, and executed like any other hardware design, making them a good candidate for proofs that are written alongside the design by the hardware developers themselves.

Simulation-Based Leakage Proofs. To show that a hardware circuit `c` is secure via a simulation-based leakage proof, we first have to provide a precise description of its intended leakage in the form of a circuit `leak`. For example, if `c` is an adder with a fast path on input `0`, the leakage description `leak` reveals whether or not the input is `0`. Next, we model the attacker's view of the computation as a circuit `obs`. For example, `obs` may reveal the time it takes to complete the computation by indicating whether an output was produced in a given clock cycle. To prove that the leakage description `leak` faithfully captures everything an attacker can learn about potential secret information processed by `c`, we have to construct a *simulator* circuit `sim` such that the original circuit composed with the attacker observation function `obs`, written as `c` ∘ `obs`, is *indistinguishable* from the composition of the leakage description `leak` and the simulator `sim`, written as `leak` ∘ `sim`. In our example, simulator `sim` reproduces the timing behavior of `c` by delaying inputs that don't take the fast path. The existence of a simulator means that we can reconstruct all attacker-observable behavior from information that has been explicitly leaked. In turn, this means the attacker learns no more than what is specified via the explicit leakage description. We prove that simulation-based proofs are equivalent in expressiveness to traditional non-interference, establishing them as a sound and complete proof method.

Functional Programs as Hardware. While simulation-based leakage proofs can be applied to hardware designs written in any language, we present a concrete instance of our proof method for circuits expressed as *functional programs*. Building on a long line of work connecting functional programs and hardware [9, 40], we represent the hardware's single clock tick transition function as a functional program that takes a state and input to a state and output.

Equivalence Via State Projection. Next, we construct a proof system for proving equivalence between circuits via local single-step reasoning. As our proofs generally require us to prove an equivalence between functions of *different state types* (*i.e.*, simulator and implementation), this proof system is centered around a new *state projection* rule, which allows changing the type of a circuit's state as long as this change doesn't affect its observable behavior, thereby establishing a refinement relation between the two circuits [26]. State projection proofs offer an executable alternative to classic invariant-based reasoning, which allows for easy debugging. Except for state projection functions—which translate between the states of different types and have to be provided by the user—checking correctness of a simulator using the state projection rule can be fully automated via an SMT solver.

Automation via SMT. While writing simulators by hand is often instructive—especially when debugging faulty leakages—we show that one can further reduce the proof burden of simulation-based proofs. We propose a sound and complete check for the existence of simulators in our proof system, which can be discharged via an SMT query. With this check, the user only has to provide the type of the simulator, but not the simulator itself.

Implementation and Evaluation. We implemented our method in PANTOMIME, a tool for writing processors and proofs in Haskell (including support for higher-order functions, type classes, abstract data types, and monads). Hardware written with PANTOMIME can be extracted to hardware description languages like Verilog and VHDL using CλaSH [8]. To prove equivalence between simulator and implementation, we developed a new symbolic execution engine based on Grisette [29]. The engine proves function equivalence between expressions in GHC Core using the state projection rule and interprets CλaSH data types to accurately model the behavior of the extracted hardware. We used PANTOMIME to write AIMCORE, a 5-stage RISC-V CPU that supports the full base integer instruction set, and a side-channel hardened version of the processor. We show that PANTOMIME can efficiently verify their leakage descriptions, and that leakage descriptions for more secure processors are smaller. Unlike existing work [42, 47], our proofs are constructive exhibiting leakage descriptions and their simulators as a proof artifact. Our leakages are executable and can be run alongside the hardware to log or monitor leakage (even in silicone). Proof checking takes seconds, rather than hours, and proofs are computational, which makes them easier to debug.

Contributions. In summary, we make the following contributions.

- **Simulation-Based Proofs:** A new, constructive approach for verifying leakage descriptions of hardware circuits.
- **Soundness and Completeness:** A proof showing that simulation-based proofs are sound and complete with respect to classic non-interference.
- **State Projection and Automation:** A proof rule which allows us to prove the equivalence of circuits with different types and a condition that proves existence of a simulator without having to write its implementation.
- **PANTOMIME:** An implementation of simulation-based leakage proofs in Haskell via a GHC plugin that symbolically executes GHC Core and interprets hardware data types in CλaSH.
- **AIMCORE:** A 5-stage RISC-V processor, its leakage description and simulator, resulting in the first processor with a constructive and executable leakage proof.

2 Overview

We illustrate our technique on a simple adder with a fast path (§ 2.1) and discuss how to state (§ 2.2) and verify (§ 2.3) its leakage proof.

2.1 A Simple Adder and its Leakage Description

Circuits. We treat hardware designs as functional programs. A *circuit* is a pair of a step function of type `(S,I) -> (S,O)` and an initial state of type `S`. Here `S` represents the circuit's state (*e.g.*, the current values of all its registers), `I` and `O` represent its inputs and outputs, and `(S,I)` and `(S,O)` represent pairs combining the circuit's state with its input of type `I` and output of type `O`, respectively—see Figure 1. We define an alias for circuits as follows:

```
type Circuit s i o
  = ((s, i) -> (s, o), s)
```

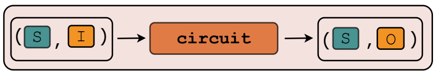

Fig. 1. Circuits take a pair of type `(S,I)`, and return a pair of type `(S,O)`; they also have an initial state of type `S`.

Adder. Our running example is an adder that computes the sum of two integer inputs `a` and `b`. The adder has a *fast path*: if its first input `a` is `0` (in which case the result is just its second input `b`), the adder produces an output right away in the same clock cycle. Otherwise, the adder needs some additional time to compute a + b and outputs the result one cycle later. The timing difference between the paths may leak information about the inputs to an attacker. The step function add for the adder is defined as follows:

```
1 add :: (Maybe Int, (Maybe Int, Maybe Int)) -> (Maybe Int, Maybe Int)
2 add (_, (Just 0, Just b)) = (Nothing, Just b)
3 add (_, (Just a, Just b)) = (Just (a + b), Nothing)
4 add (s, _)                = (Nothing, s)
```

The adder's internal state S is an optional integer of type Maybe Int, whose values are either Nothing or Just v, where v has type Int. When taking the slow path, the adder stores the intermediary result of the computation in its state. The adder's input is a pair of optional integers and its output is a single optional integer. An input with value Nothing means no input is available; for example, when the client of the adder is waiting for the result of another computation.

Computing the Output. When add's first input component is Just 0, we take the fast path (line 2): we update the state to Nothing—there's nothing to save—and output Just b. If it's non-zero, we take the slow path (line 3): we set the state to Just (a + b) to keep track of the sum that will be output in the next clock cycle and output Nothing to signal a pending result. If either of the inputs is Nothing (line 4), neither of the branches applies: we output the stored value, if any, and reset the state to Nothing.

From Circuits to Transducers. Circuits describe how hardware evolves from one clock tick to the next. The operator mealy, shown in Figure 2, transforms a circuit into a *transducer*—a function of type [I] -> [O], which maps streams of inputs to streams of outputs. On input, mealy applies the circuit to update its state and to produce an output, which is then prepended onto the outputs produced by recursively processing the remaining inputs.

Running the Adder. For circuit (add, Nothing) and inputs

```
is = [(Just 1, Just 1), (Nothing, Nothing),
      (Just 0, Just 1), (Nothing, Nothing)]
```

the operator mealy produces the following output:

```
mealy (add, Nothing) is
> [Nothing, Just 2, Just 1, Nothing]
```

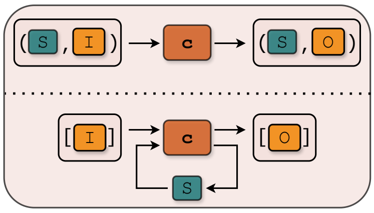

Fig. 2. mealy transforms a circuit into transducer by saving the output state as input for the next cycle.

Attacker View. Since we are interested in the *timing behavior* of the circuit, we model an attacker that can observe whether add produces an output in a given clock cycle. We model this with function isJust, shown below, defining our observation function as obs = isJust.

```
isJust (Just _) = True
isJust _        = False
```

Composing the circuit add with obs produces the combined circuit add_obs, which records the attacker-observable view of a computation. This circuit outputs True in a given clock cycle if and only if an output was produced by add. To build add_obs, we must first define circuit composition.

Sequential Composition. Two circuits c1 :: Circuit s1 i io and c2 :: Circuit s2 io o may be composed to form a new circuit c1 ∘ c2 :: Circuit (s1, s2) i o that first executes c1 and then c2. Figure 3 illustrates circuit composition; in short, circuit composition uses circuit c1's output as input to circuit c2, and combines their individual states.

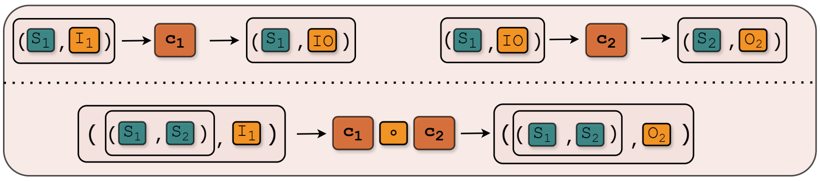

Fig. 3. For circuits c1, and c2, we write c1 ∘ c2 for their sequential composition; we omit initial states.

Lifting. We cannot directly apply circuit composition to obs since it is not a circuit: it doesn't have state. Instead, for any function `f :: i -> o`, we write `lift f` to turn f into a circuit by augmenting it with an empty state: `lift f :: Circuit () i o`.

Combining the Circuits. We now have the machinery to produce add_obs, capturing the attacker-observable view of a computation:

```
add_obs :: Circuit (Maybe Int, ()) (Maybe Int, Maybe Int) Bool
add_obs = (add, Nothing) ∘ (lift obs)
```

The obs observation function describes an attacker that can observe the transducer corresponding to circuit add_obs. Observation function obs models a *timing attacker*: for a given initial state and sequence of inputs, the attacker can see whether an output has been produced in each clock cycle. For example, running add_obs's transducer on the inputs is produces the following output list:

```
mealy add_obs is
> [False, True, True, False]
```

Modeling Leakage. Next, we want to capture which information about the circuit's inputs is leaked to the attacker via an explicit leakage description. The adder leaks information about its operands via timing: if a is 0, the fast-path computation produces an output in the same clock cycle; otherwise the output is delayed by one cycle. By observing the presence (or absence) of an output, the attacker can therefore determine whether a is 0. We capture this leakage via function leak, which takes the same inputs as add—a pair of optional integers—and returns a pair of type `(Maybe Bool, Bool)` that describes the component-wise input leakage. The first component of the leakage says whether the first input was a proper integer a, in which case we also learn whether `a == 0`. The second component only says whether the second input was a proper integer, but we learn nothing about its value. We give the definition of leak below:

```
leak :: (Maybe Int, Maybe Int) -> (Maybe Bool,Bool)
leak (Just a, i2)  = (Just (a == 0), isJust i2)
leak (Nothing, i2) = (Nothing, isJust i2)
```

Applying `lift leak` on inputs is using mealy produces the following outputs:

```
mealy (lift leak) is
>[(Just False, True),(Nothing, False),(Just True, True),(Nothing, False)]
```

While the leakage and observation are pure, stateless functions in this example, they can be arbitrary stateful computations, in general.

2.2 Proving Correctness via a Simulator Circuit

Our leakage description must contain enough information to reconstruct the full observable behavior of the adder for an attacker observing timing. To prove this, we build a *simulator*—a circuit that computes the observable behavior of the adder (as defined by obs) solely from the leakage description `leak`. The existence of a simulator guarantees that a timing attacker can learn no information beyond what is leaked explicitly via `leak` when observing the circuit `add`. Indeed, if `leak` were missing any relevant information, the simulator could not faithfully reproduce `add`'s observable behavior.

Simulator. In our proof system, the simulator is a proof artifact produced by the user. We define a step function `sim` for the simulator, which will closely follow the definition of `add`. However, `sim`'s internal state is of type `Bool` instead of `Maybe Int`. This is sufficient; the simulator only needs to keep track of whether there is a pending computation from a previous cycle—it doesn't need to know the value. The simulator accepts an input of type `(Maybe Bool, Bool)`, matching `leak`'s output, and produces a `Bool`, matching `add_obs`'s output.

```
1 sim :: (Bool, (Maybe Bool, Bool)) -> (Bool, Bool)
2 sim (_, (Just True, True))  = (False, True)
3 sim (_, (Just False, True)) = (True, False)
4 sim (s, _)                  = (False, s)
```

If the leakage is `(Just True, True)` (line 2), `sim` simulates the fast path: its state is updated to `False`—there is no new pending computation—and it outputs `True` to indicate that the current cycle yielded an output. If the leakage is `(Just False, True)` (line 3), `sim` simulates the slow path: its state is set to `True` to reflect that there is a pending computation and it outputs `False` to indicate that the current cycle yielded no new output. Otherwise (line 4), it outputs the saved state—which is `True` if there's a pending computation and `False` otherwise—and sets the new state to `False`.

Combining the Circuits. Composing `leak` with `sim` produces the combined circuit that forwards the result of `leak` to the simulator:

```
leak_sim :: Circuit ((), Bool) (Maybe Int, Maybe Int) Bool
leak_sim = (lift leak) ∘ (sim, False)
```

Correctness of the Leakage Description. Circuits `add_obs` and `leak_sim` both take a pair of optional integers as an input and produce a Boolean as an output. The transducers `mealy add_obs` and `mealy leak_sim` thus have the same type: `[(Maybe Int, Maybe Int)] -> [Bool]`. Indeed, if the simulator is constructed correctly, the two transducers should coincide as functions. To prove that `leak` correctly captures circuit `add`'s leakage with respect to `obs`, we need to prove that the transducers of circuits `add_obs` and circuit `leak_sim` behave the same.

Different Types. However, when we view `add_obs` and `leak_sim` at the level of circuits, they are *not* equal: indeed, we cannot even compare them as they have different types. `add_obs`'s state has type `(Maybe Int, ())`, whereas `leak_sim`'s state has type `((), Bool)`. Fortunately, equivalence at the circuit level is not our ultimate goal; we only need the underlying *transducers* to be equal.

The State Projection Rule. Based on this observation, we introduce the *state projection rule* to reason about transducer equivalence of circuits with different state types: if information can be removed from the circuit's state—thereby changing its type—without affecting its input/output behavior, then the tranducer's behavior is also unaffected. Indeed, if we look back at the definition of `add`, we can see that the integer value `v` stored in the state of form `Just v` is *irrelevant* in circuit `add_obs`. Lines (2) – (3) do not depend on the state at all. Line (4) does output `v`, but we discard the result in `add_obs` due to the output projection with `obs`.

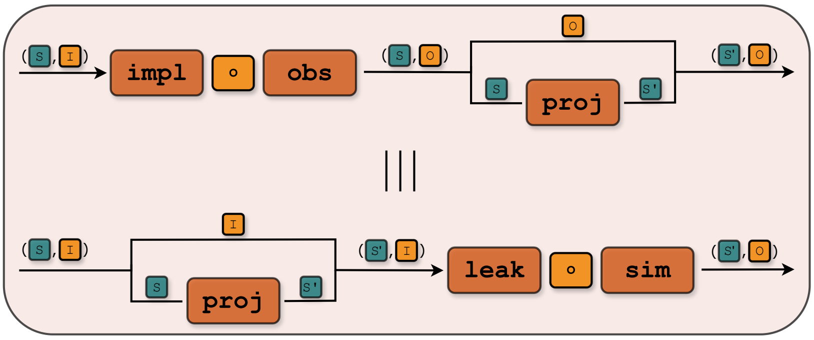

Fig. 4. Proof obligation for the correctness of a leakage leak with respect to an implementation impl and its observation obs, due to the state projection rule. Wires are annotated above with their type. We use ≡ for input/output equality.

2.3 Proof Checking via State Projection

Projection Function. We define the projection function proj using function isJust from before. This function discards add_obs's unit state and applies isJust to discard the actual computation result, only remembering whether there is a value or not; finally it adds leak_sim's unit state.

```
proj :: (Maybe Int, ()) -> ((), Bool)
proj (s, _) = ((), isJust s)
```

Applying the State Projection Rule. The state projection rule asks us to show that applying the projection to the *output state* of add_obs yields the same result as applying it to the *input state* of leak_sim, thereby showing that leak_sim can compute the same result as add_obs *without* the information we removed. We illustrate this proof obligation in Figure 4. As the resulting functions have the same type, we can prove their equivalence using standard methods, e.g., an SMT solver.

Summary. To sum up, we've shown that the simulator reconstructs the observable behavior of add, confirming that function leak correctly captures add's leakage. To apply the state projection rule, a PANTOMIME user only has to supply a state projection function, which removes the irrelevant part of the circuit state. In our implementation, we prove function equivalence using PANTOMIME's symbolic execution engine, described in § 5.

Proving Existence of a Simulator. While simulators allow running and debugging leakages and serve as a proof artifact for the security of the design, it may sometimes be sufficient to show existence of a relevant simulator. We give a set of conditions (Lemma 3.11) that prove existence of a simulator if and only its correctness can be shown using the state projection rule. With these conditions, whose check can be automated via an SMT solver, we can provide significant automation for simulation-based proofs: the user only has to write the state projection, but not the actual implementation of the simulator.

3 Simulation-Based Leakage Proofs

3.1 Language

We now formalize our proof method in a core language (Figure 5) based on first-order simply-typed lambda calculus (STLC), extended with records (labeled product types) and variants (sum types). Since STLC does not allow recursion, it is a good match for describing the single-step behavior of hardware, which must be loop-free. Our type system is a version of the standard STLC (as described, *e.g.*, in [36]) restricted to first-order and slightly adapted to our purposes. Our language satisfies the *preservation* property: the type of an expression is invariant across evaluation. The language features the usual STLC staples: constants, variables, lambda abstractions, and applications. Constants are composed of bitstrings and of binary, unary, and equality operators—for example, the operator $+_8$ adds two bitstrings of type `Word8`, and the operator $=_8$ compares two `Word8` bitstrings for equality, returning a Boolean. Labels in variant and record types are denoted by $\mathtt{l}$ and are assumed to be unique within the same record or variant type. We omit type annotations on function binders as they are not important in our setting. The substitution operator $[\mathtt{x} = \mathtt{e}]$ performs a capture-avoiding substitution of free occurrences of the variable $\mathtt{x}$ with term $\mathtt{e}$. We write $\overrightarrow{\mathtt{k_i}}$ to represent a vector of expressions $\mathtt{k}_1, \mathtt{k}_2, \ldots, \mathtt{k}_n$. Records are constructed with $\{\overrightarrow{\mathtt{l_i} = \mathtt{e_i}}\}$, wherein each term $\mathtt{e_i}$ is assigned the corresponding label $\mathtt{l_i}$. For a record $\mathtt{e}$, $\mathtt{e.l}$ accesses its field with label $\mathtt{l}$. Term $\mathtt{l\ e}$ constructs an inhabitant of a variant type with label $\mathtt{l}$.

Syntactic Sugar. We elaborate $\mathtt{let\ p\ =\ a\ in\ e}$ according to the choice of pattern $\mathtt{p}$: $\mathtt{let\ x\ =\ a\ in\ e}$ is elaborated to $\mathtt{e}[\mathtt{x} = \mathtt{a}]$, and $\mathtt{let}\ \{\mathtt{l}_1 = \mathtt{p}_1, \ldots, \mathtt{l}_n = \mathtt{p}_n\}\ =\ \mathtt{a\ in\ e}$ elaborates to $\mathtt{let\ p_1\ =\ a.l_1\ in}\ \ldots\ \mathtt{in\ let\ p_n\ =\ a.l_n\ in\ e}$. We elaborate $\lambda\,\mathtt{p.\ e}$ to $\lambda\,\mathtt{x.\ let\ p\ =\ x\ in\ e}$. If the size of a bitstring is clear from the context, we omit leading zeros. Finally, $\mathtt{e_1\ bop\ e_2}$ should be understood as $\mathtt{bop\ e_1\ e_2}$.

Base Types	w	::=	Word8, Word16, ...	*base types*
Core Types	s	::=	w	*base types*
		|	Bool	*Booleans*
		|	$\{\overrightarrow{\mathtt{l_i} : \mathtt{s_i}}\}$	*records*
		|	$\langle\overrightarrow{\mathtt{l_i} : \mathtt{s_i}}\rangle$	*variants*
Types	τ	::=	s	*core types*
		|	$\mathtt{s} \to \tau$	*functions*
Bitstrings	b	::=	0 ... 0, 0 ... 1, ...	*bitstrings*
Operators	bop	::=	$+_8, -_8, +_{16}, \ldots$	*binary ops*
	uop	::=	$\sim_8, \sim_{16}, \ldots$	*unary ops*
	eq	::=	$=_8, =_{16}, \ldots$	*equality*
Constants	c	::=	b | bop	*bitstrings*
		|	uop | eq	*operators*

Terms	e	::=	c	*constants*
		|	x	*variables*
		|	true | false	*Booleans*
		|	if e then $\mathtt{e_1}$ else $\mathtt{e_2}$	*branching*
		|	$\{\overrightarrow{\mathtt{l_i} = \mathtt{e_i}}\}$	*records*
		|	e.l	*fields*
		|	l e	*variants*
		|	case e of $\overrightarrow{\mathtt{l_i\ x_i} \to \mathtt{e_i}}$	*matching*
		|	λx. e	*abstraction*
		|	$\mathtt{e_1\ e_2}$	*application*
Patterns	p	::=	x	*variables*
		|	$\{\overrightarrow{\mathtt{l_i} = \mathtt{p_i}}\}$	*records*

Fig. 5. Syntax of types and terms in our core language.

Evaluation. Selected evaluation rules for the core language are given in Appendix A. Evaluation is based on a reduction relation $\mathtt{e} \rightsquigarrow \mathtt{e}$. The rules are standard; for example, the BETA rule reduces an application: $(\lambda\mathtt{x.\ e_1})\ \mathtt{e_2} \rightsquigarrow \mathtt{e_1}[\mathtt{x} = \mathtt{e_2}]$. We write $[\![\mathtt{e}]\!]$ to represent the unique term obtained by exhaustive application of the evaluation rules to $\mathtt{e}$.

Unit, Product, and Option Types. We encode the unit type $()$ as the empty record type with no fields. For types $\mathtt{t_1}$ and $\mathtt{t_2}$, we encode the product type $(\mathtt{t_1}, \mathtt{t_2})$ as the record type $\{\mathtt{fst} : \mathtt{t_1},\ \mathtt{snd} : \mathtt{t_2}\}$. For type $\mathtt{t}$, we encode the option type $\mathtt{Maybe\ t}$ as the variant type $\langle\mathtt{Just} : \mathtt{t},\ \mathtt{Nothing} : ()\rangle$.

Example 3.1: Consider the term

$$\begin{aligned}&\mathtt{ex}\ :\ (\mathtt{Word32}, \mathtt{Maybe\ Word32})\ \longrightarrow\ (\mathtt{Word32}, \mathtt{Word32})\\ &\mathtt{ex} = \lambda\,(\mathtt{s}, \mathtt{i}).\ (\mathtt{s} + 1, \mathtt{case\ i\ of\ Just\ a} \longrightarrow \mathtt{s} + \mathtt{a},\ \mathtt{Nothing} \longrightarrow \mathtt{s}).\end{aligned}$$

Then we have $[\![\mathtt{ex}\ 0\ \mathtt{Nothing}]\!] =\ (1, 0)$ and $[\![\mathtt{ex}\ 0\ (\mathtt{Just}\ 1)]\!] =\ (1, 1)$.

3.2 Circuits, Transducers, Leakage Description, and State Projection

Circuits. Circuits describe the behavior of a hardware design in a single clock tick. A circuit is a pair of an internal state and a computation that updates the internal state and produces an output based on the current state and an input.

***Definition* 3.1** (Circuit)**.** Let S, I, O be core types in our language. A circuit of input type I and output type O is a term of type $\big((S, I) \rightarrow (S, O), S\big)$, where S is the internal state type.

If the state s is clear from the context, we sometimes drop the state and denote the circuit (c, s) by c.

Example 3.2*:* The pair $(\texttt{ex}, \texttt{0})$ is a circuit with state `Word32`, input `Maybe Word32`, and output `Word32` that increments its state at each clock tick. It outputs the sum of its input and current state when the input is non-empty, and outputs its state otherwise.

Transducers. The *Mealy* function $\mathcal{M}$ transforms a circuit into a *transducer* that applies one input per clock tick from a list of inputs and produces a corresponding list of outputs. Transducers therefore represent the actual input/output behavior of sequential hardware.

***Definition* 3.2** (Mealy)**.** Let $(c, s) : \big((S, I) \rightarrow (S, O), S\big)$ be a circuit, and let $is : [I]$ be a list of inputs. We build a list of outputs $\mathcal{M}_{(c,s)}(is) : [O]$ recursively as follows:

$$\mathcal{M}_{(c,s)}(is) \coloneqq \begin{cases} [] & \text{if } is = [], \\ o : \mathcal{M}_{(c,s')}(is') & \text{if } is = i : is' \text{and } (s', o) = [\![c\ (s, i)]\!] \end{cases}$$

Here [] is the empty list and : denotes the *cons* operator on lists.

Thus, the transducer applies the circuit computation c to the first input i of the input list is to compute a new state s' and an output o. It prepends o to the list of outputs and then recursively computes the remaining outputs using the new state s' and the remaining inputs is'.

Example 3.3*:* Running the circuit $(\texttt{ex}, \texttt{0})$ on the list of inputs $is = [$`Just` 1, `Nothing`, `Just` 2, `Just` 3$]$ yields the outputs $\mathcal{M}_{(\texttt{ex},\texttt{0})}(is) = [1, 1, 4, 6]$.

***Definition* 3.3** (Mealy Equivalence of Circuits)**.** Two circuits $(c_1, s_1) : \big((S_1, I) \rightarrow (S_1, O), S_1\big)$ and $(c_2, s_2) : \big((S_2, I) \rightarrow (S_2, O), S_2\big)$ are *Mealy-equivalent*, denoted $(c_1, s_1) =_{\mathcal{M}} (c_2, s_2)$, if for any input sequence $is : [I]$, we have $\mathcal{M}_{(c_1,s_1)}(is) = \mathcal{M}_{(c_2,s_2)}(is)$.

Sequential Composition. To define leakage descriptions, we first define sequential circuit composition.

***Definition* 3.4** (Sequential Composition)**.** The sequential composition of two circuits (c_1, s_1) : $\big((S_1, I_1) \rightarrow (S_1, O_1), S_1\big)$ and $(c_2, s_2) : \big((S_2, O_1) \rightarrow (S_2, O_2), S_2\big)$ is the circuit $\big(c_1 \circ c_2, (s_1, s_2)\big)$, where $c_1 \circ c_2$ is defined by

$$c_1 \circ c_2 : \big((S_1, S_2), I_1\big) \rightarrow \big((S_1, S_2), O_2\big)$$
$$c_1 \circ c_2 \coloneqq \lambda\, ((\mathtt{s}_1, \mathtt{s}_2), \mathtt{i}_1).\ \mathtt{let}\ (\mathtt{s}'_1, \mathtt{o}_1) = c_1\ \mathtt{s}_1\ \mathtt{i}_1\ \mathtt{in}\ \mathtt{let}\ (\mathtt{s}'_2, \mathtt{o}_2) = c_2\ \mathtt{s}_2\ \mathtt{o}_1\ \mathtt{in}\ ((\mathtt{s}'_1, \mathtt{s}'_2), \mathtt{o}_2).$$

Leakage. We now have the machinery needed to define leakage.

***Definition* 3.5** (Leakage)**.** Let $(c, s_c) : \big((S, I) \rightarrow (S, O), S\big)$ be a circuit, and let $(obs, s_o) : \big((S_o, O) \rightarrow (S_o, O_o), S_o\big)$ be an *observation.* A circuit $(leak, s_l) : \big((S_l, I) \rightarrow (S_l, O_l), S_l\big)$ is a *leakage* for (c, s_c) with respect to (obs, s_o) if there exists a *simulator* $(sim, s_s) : \big((S_s, O_l) \rightarrow (S_s, O_o), S_s\big)$ such that

$$(c, s_c) \circ (obs, s_o) =_{\mathcal{M}} (leak, s_l) \circ (sim, s_s).$$

Next we adapt contract equivalence [47]—a prior non-interference based notion of leakage—to our setting and show that it is equivalent to our definition.

***Definition* 3.6** (Contracts)**.** Let $(c, s_c) : \big((S, I) \longrightarrow (S, O), S\big)$ and $(obs, s_o) : \big((S_o, O) \longrightarrow (S_o, O_o), S_o\big)$ be circuits. A circuit $(leak, s_l) : \big((S_l, I) \longrightarrow (S_l, O_l), S_l\big)$ is a *contract* for (c, s_c) with respect to (obs, s_o) if for all input sequences $is, is' : [I]$, we have that

$$\mathcal{M}_{(leak,\, s_l)}(is) = \mathcal{M}_{(leak,\, s_l)}(is') \text{ implies } \mathcal{M}_{(c,\, s_c)\,\circ\,(obs,\, s_o)}(is) = \mathcal{M}_{(c,\, s_c)\,\circ\,(obs,\, s_o)}(is').$$

***Theorem* 3.7** (Leakage and Contract Equivalence)**.** *Let* $(c, s_c) : \big((S, I) \longrightarrow (S, O), S\big)$ *and* $(obs, s_o) : \big((S_o, O) \longrightarrow (S_o, O_o), S_o\big)$ *be circuits, and assume that the type* O_o *is inhabited. A circuit* $(leak, s_l) : \big((S_l, I) \longrightarrow (S_l, O_l), S_l\big)$ *is a leakage for* (c, s_c) *with respect to* (obs, s_o) *if and only if it is a contract.*

We give the proof of Theorem 3.7 in appendix B. The left–to–right direction can be understood as soundness and the right–to–left direction as completeness of our approach.

State Projections. In practice, establishing the Mealy equivalence of circuits requires an invariant that suitably relates the internal states of the two circuits. Our leakage descriptions as well as the associated simulators proceed in lockstep with the original circuit and share the same stream of inputs. In this setting, a particularly useful invariant for showing Mealy equivalence arises from the *state projection function*, an example of which we have already seen in § 2. Showing that the state projection function indeed gives rise to an invariant requires proving the equivalence of two functions. In particular, we are interested in *observable* function equivalence rather than syntactic equivalence. For example, the functions $\lambda\,$`i. i + 0` and $\lambda\,$`i. 0 + i` should be equivalent.

***Definition* 3.8** (Observational Equivalence)**.** Two closed terms e, e' of the same type τ are *observationally equivalent*, written $\mathsf{e} \simeq \mathsf{e}'$, if one of the following holds:

1. $\tau = \mathsf{s}$ is a core type and $[\![\mathsf{e}]\!] = [\![\mathsf{e}']\!]$.
2. $\tau = \mathsf{s}_1 \longrightarrow \tau_2$ is a function type and for any two closed terms $\mathsf{e}_1, \mathsf{e}_1' : \mathsf{s}_1$ such that $\mathsf{e}_1 \simeq \mathsf{e}_1'$, we have $\mathsf{e}\ \mathsf{e}_1 \simeq \mathsf{e}'\ \mathsf{e}_1'$.

Example 3.4*:* The three functions $\lambda\,$`i. i + 0`, $\lambda\,$`i. 0 + i`, $\lambda\,$`i. i` of type `Word` $\longrightarrow$ `Word` are observationally equivalent since they return the same result on any input. Similarly, the two functions $\lambda\,$`b. b` and $\lambda\,$`b. if b then true else false` of type Bool $\longrightarrow$ Bool are observationally equivalent as each function returns the same result on both `true` and `false`.

***Definition* 3.9** (State Projection Rule)**.** Given two circuits $(c_1, s_1) : \big((S_1, I) \longrightarrow (S, O), S_1\big)$ and $(c_2, s_2) : \big((S_2, I) \longrightarrow (S, O), S_2\big)$, and a function $p : S_1 \longrightarrow S_2$, define

$$\begin{aligned} c_1p &: (S_1, I) \longrightarrow (S_2, O) & \qquad pc_2 &: (S_1, I) \longrightarrow (S_2, O) \\ c_1p &= \lambda\,(\mathsf{s}_1, \mathsf{i}).\ \mathtt{let}\ (\mathsf{s}_1', \mathsf{o}) = c_1\ (\mathsf{s}_1, \mathsf{i})\ \mathtt{in}\ (p\ \mathsf{s}_1', \mathsf{o}) & pc_2 &= \lambda\,(\mathsf{s}_1, \mathsf{i}).\ c_2\ (p\ \mathsf{s}_1, \mathsf{i}). \end{aligned}$$

We say that (c_1, s_1) *state-reduces* to (c_2, s_2), written $(c_1, s_1) \hookrightarrow (c_2, s_2)$, if $p\ s_1 \simeq s_2$ and $c_1p \simeq pc_2$. In this case, we call p the *state projection* function witnessing this reduction.

Example 3.5*:* In the adder example from § 2, we showed that `add_obs` $\hookrightarrow$ `leak_sim`, *i.e.*, that `add_obs` state-reduces to `leak_sim` using the state projection function `proj`, by the observational equivalence between the two functions below:

```
c1p (s, i) = let (s', o) = add_obs s i in (proj s', o)
pc2 (s, i) = leak_sim (proj s) i
```

Our goal was to show that the transducers of `add_obs` and `leak_sim` are equal, which would confirm that `lift leak` is a suitable leakage for `add` with respect to `obs`. This is guaranteed by the following:

***Theorem* 3.10** (State Reduction implies Mealy Equivalence)**.** *Given two circuits* $(c_1, s_1) : \big((S_1, I) \longrightarrow (S, O), S_1\big)$ *and* $(c_2, s_2) : \big((S_2, I) \longrightarrow (S, O), S_2\big)$, *if* $(c_1, s_1) \hookrightarrow (c_2, s_2)$ *then* $(c_1, s_1) =_{\mathcal{M}} (c_2, s_2)$.

We prove Theorem 3.10 in appendix B.

Example 3.6: Revisiting the adder (§ 2), circuit `lift leak` is a leakage description for `add` under `lift obs` since there is a simulator `sim` such that `add` $\circ$ (`lift obs`) $\hookrightarrow$ (`lift leak`) $\circ$ `sim`.

By an argument similar to the proof of Theorem 3.7, we can show that for a *fixed* state projection function, which in particular fixes the simulator states, the existence of a simulator can be determined via a simple non-interference check, using the three conditions below. Intuitively, condition (1) requires that the projection of the initial state matches the initial state of the leakage; condition (2) requires that the leakage updates its state in a way that's consistent with the implementation; condition (3) is a non-interference condition that requires that all pairs of inputs and states that produce the same leakage must produce the same observation. We give the proof in appendix B.

***Lemma* 3.11** (Simulator Existence). *Fix two circuits* $(c, s_c) : \big((S, I) \to (S, O), S\big)$ *and* $(obs, s_o) : \big((S_o, O) \to (S_o, O_o), S_o\big)$ *such that the type* O_o *is inhabited, the intended leakage* $(leak, s_l) : \big((S_l, I) \to (S_l, O_l), S_l\big)$*, and the intended state projection function* $p : (S, S_o) \to (S_l, S_s)$*. A simulator* $(sim, s_s) : \big((S_s, O_l) \to (S_s, O_o), S_s\big)$ *such that*

$$(c, s_c) \circ (obs, s_o) \hookrightarrow (leak, s_l) \circ (sim, s_s)$$

via the state projection function p exists if and only if the following conditions are satisfied:

(1) ***Initial State Correspondence***: *We have* $[\![\texttt{fst}\ (p\ (s_c, s_o))]\!] = [\![s_l]\!]$.

(2) ***Tick State Correspondence***: *For all* $((s_1, s_2), i) : ((S, S_o), I)$*, we have*

$$[\![\texttt{let}\ ((\texttt{t}_1, \texttt{t}_2), \_) = c \circ obs\ ((s_1, s_2), i)\ \texttt{in let}\ (\texttt{v}_1, \_) = p\ (\texttt{t}_1, \texttt{t}_2)\ \texttt{in}\ \texttt{v}_1]\!] =$$
$$[\![\texttt{let}\ (\texttt{u}_1, \_) = p\ (s_1, s_2)\ \texttt{in let}\ (\texttt{v}_1, \_) = leak\ (\texttt{u}_1, i)\ \texttt{in}\ \texttt{v}_1]\!].$$

(3) ***Projection Coherence***: *For all* $((s_1, s_2), i), ((s_1', s_2'), i') : ((S, S_o), I)$*, if*

$$[\![\texttt{let}\ (\texttt{u}_1, \texttt{u}_2) = p\ (s_1, s_2)\ \texttt{in let}\ (\_, \texttt{o}_1) = leak\ (\texttt{u}_1, i)\ \texttt{in}\ (\texttt{u}_2, \texttt{o}_1)]\!] =$$
$$[\![\texttt{let}\ (\texttt{u}_1', \texttt{u}_2') = p\ (s_1', s_2')\ \texttt{in let}\ (\_, \texttt{o}_1') = leak\ (\texttt{u}_1', i')\ \texttt{in}\ (\texttt{u}_2', \texttt{o}_1')]\!]$$

then we have

$$[\![\texttt{let}\ ((\texttt{t}_1, \texttt{t}_2), \texttt{o}_o) = c \circ obs\ ((s_1, s_2), i)\ \texttt{in let}\ (\_, \texttt{v}_2) = p\ (\texttt{t}_1, \texttt{t}_2)\ \texttt{in}\ (\texttt{v}_2, \texttt{o}_o)]\!] =$$
$$[\![\texttt{let}\ ((\texttt{t}_1', \texttt{t}_2'), \texttt{o}_o') = c \circ obs\ ((s_1', s_2'), i)\ \texttt{in let}\ (\_, \texttt{v}_2') = p\ (\texttt{t}_1', \texttt{t}_2')\ \texttt{in}\ (\texttt{v}_2', \texttt{o}_o')]\!].$$

Constructing a Simulator through Inversion. If all conditions hold, we can automatically construct a simulator. For this, we need an inversion function which maps simulator state and leakage output (S_s, O_l) to a preimage $((S, S_o), I)$. Applying $c \circ obs$ to the preimage and projecting the resulting state back to S_s using p, then yields the simulator. In theory, one can always construct a simulator using this method by defining an inversion function which iterates all possible values of $((S, S_o), I)$ until it finds a preimage that maps to the given (S_s, O_l). Indeed, this is intuition behind our proof. In practice, the resulting simulator is of little use due to the potentially huge search space. The user can however manually construct a more efficient inversion function by exploiting the structure of the circuits at hand. This leads to a general paradigm of constructing simulators through inversion.

Example 3.7: Revisiting the adder (§ 2), we can construct an inversion function as follows:

```
inv_ss ss = if ss then Just 0 else Nothing
inv (ss, (Just True, True))  = (inv_ss ss, (Just 0, Just 0))
inv (ss, (Just False, True)) = (inv_ss ss, (Just 7, Just 0))
inv (ss, (Nothing, True))    = (inv_ss ss, (Nothing, Just 0))
inv (ss, (_, False))         = (inv_ss ss, (Just 7, Nothing))
sim = mapFst (snd . proj) . add_obs . inv
```

Function `inv` reconstructs a preimage state and input that would produce the given simulator state `ss` and leakage output `ol`. The specific non-zero integer values (like 7) are arbitrary since the leakage abstracts away the concrete values, only preserving the relevant information for simulation.

4 Case Study: Pipelined Processor

This section shows how to construct a simulation based-leakage proof for a minimalistic three stage pipelined processor with a single register. While we use a Haskell-like syntax for readability, the processor is fully expressible in our core language.

4.1 The Processor

Instruction Set. We encode instructions as the variant type below:

```
data Instr = Add Word8 | Clr | Out | Jmp Word8 | Bz Word8
```

`Add` adds a `Word8` value to the register, `Clr` resets it `0`, `Out` outputs its current value, `Jmp` jumps to a `Word8` address, and `Bz` branches if the register is `0`, adding its `Word8` offset to the current program counter.

Toplevel Circuit and State. We encode the processor as a circuit `proc` that takes a raw `Word16` instruction, and returns a pair: an optional `Maybe Word32` output and the new `Word8` program counter.

```
proc :: (State, Word16) -> (State, (Maybe Word32, Word8))
```

We encode the processor's state in the following record type:

```
data State = State
  { reg :: Word32 , fePC :: Word8, exPC :: Word8, exInstr :: Instr
  , wbRes :: Maybe Word32 , wbOut :: Maybe Word32 }
```

Here, `fePC` and `exPC` are the program counters in the fetch and execute stage; `reg` holds the register value; `exInstr` holds the instruction in the execute stage; we will explain the others as we go.

Pipeline. Each instruction passes through three pipeline stages. *Fetch* decodes a raw `Word16` instruction into an `Instr` (`exInstr`). *Execute* runs the instruction fetched last cycle, optionally producing either a register update (`wbRes`), an *output* value (`wbOut`), or the next program counter. *Writeback* commits the operations of the instruction fetched two cycles ago by updating the register state or outputting its value, if applicable. For brevity, we disuss only the execute stage in detail; the other stages and their composition are shown in Appendix C.

Execute Stage. The execute stage (Listing 1) executes the instruction fetched and decoded in the last cycle; this instruction is stored in register `exInstr`. For the `Add` and `Clr` instructions, we store the result of the computation in `wbRes`. This value is then commited in the writeback stage. For `Out`, we store the register value in `wbOut`, to be output by the writeback stage. For `Bz` and `Jmp`, we compute the appropriate jump target and output it to a wire connected to the fetch stage.

4.2 Observation, Leakage, and Simulator

Observation. The observation function projects out the program counter:

```
obs :: (Maybe Word32, Word8) -> Word8
obs (_, pc) = pc
```

This encodes an attacker that can observe the control flow of the program (and therefore, its timing).

Leakage. Since the attacker only sees the program counter, we only need to simulate the control flow. For `Jmp` we need to know the target address, and for `Bz` the offset. For all other instructions, the program counter increases by 1. This suggests the following leakage (i.e., input for the simulator):

```haskell
execute :: (State, ()) -> (State, Maybe Word8)
execute (state@State { exInstr, reg, exPC }, _) = do
  let wbRes = case exInstr of
        Add imm -> Just (reg + word8ToWord32 imm)
        Clr -> Just 0
        _ -> Nothing
  let wbOut = case exInstr of
        Out -> Just reg
        _ -> Nothing
  let jump = case exInstr of
        Bz off | reg == 0 -> Just (exPC + off)
        Jmp addr -> Just addr
        _ -> Nothing
  (state { wbRes, wbOut }, jump)
```

Listing 1. Execute Stage.

```haskell
leak :: (LState, Word16) -> (LState, LInstr)
leak (LState { reg, exInstr, wbRes }, rawInstr) = do
  let reg' = case wbRes of Just value -> value; Nothing -> reg
  let wbRes' = case exInstr of
        Add imm -> Just (reg' + word8ToWord32 imm)
        Clr -> Just 0
        _ -> Nothing
  let (exInstr', leakInstr) = case exInstr of
        Jmp addr -> (Add 0, LJmp addr)
        Bz off | reg' == 0 -> (Add 0, LBr off)
        -- Bz off | reg == 0 -> (Add 0, LBr off)
        -- ^ Wrong leakage produces counterexample
        _ -> (decode rawInstr, LOther)
  (LState { reg = reg', exInstr = exInstr', wbRes = wbRes' }, leakInstr)
```

Listing 2. Leakage.

```haskell
data LInstr = LJmp Word8 | LBr Word8 | LOther
```

The leakage circuit (Listing 2) must produce a leakage of type `LInstr` from raw `Word16` instructions. To determine when to branch, `leak` keeps track of the register value. Like the processor, it decodes and stores instructions for subsequent execution—this suggests the following state for `leak`:

```haskell
data LState = LState{reg :: Word32, exInstr :: Instr, wbRes :: Maybe Int}
```

To construct the leakage instructions, the leakage function first updates the current register file with the writeback result, if present. For the register operations `Add` and `Clr`, it computes the register update accordingly. Given a control flow instruction, it leaks the instruction, if the target is taken, placing a no-op instruction in the pipeline to stall (like `proc`). For the remaining instructions, it simply decodes the next instruction and leaks that this instruction does not branch via `LOther`. We do not need any code handling `Out`, as the attacker cannot observe outputs.

Simulator. The simulator circuit receives the `LInstr` as produced by the leakage function and uses it to faithfully reproduce the program counter. Its state is the part of the original state that was not used in the leakage, minus `wbOut`, as it does not influence the program counter.

```
data SState = SState { fePC :: Word8, exPC :: Word8 }
```

For brevity, we include the code of the simulator only in Appendix C. The circuit itself is quite straightforward: it executes the leakage instructions provided by `leak`, only computing the values of the program counter.

State Projection. To show the leakage correctly captures the processor, we must prove that the circuit `proc_obs` = `proc` ∘ (`lift obs`) reduces to the circuit `leak_sim` = `leak` ∘ `sim`, via a state-projection. The state projection is a straightforward function that divides the processor state into leakage and simulator states.

```
proj :: State -> (LState, SState)
proj State { reg, wbRes, exInstr, fePC, exPC } = do
  (LState { reg, wbRes, exInstr }, SState { fePC, exPC })
```

Counterexamples. Errors in the leakage or simulator can be quite subtle. Consider the comment in `leak`, where the code branches based on the current register instead of the new one. Pantomime rejects this leakage and returns the following counterexample to state projection equivalence.

```
counterexample = State
  { exInstr = Bz 8, exPC = 0, reg = 10, wbRes = Just 0, fePC = 1, ... }
```

Since both leakages and simulators are executable, Pantomime lets us run the counterexample.

```
proc_obs ↦ LState { exInstr = Add 0, fePC = 8, ... }
leak_sim ↦ LState { exInstr = Out,   fePC = 2, ... }
```

The execution shows that the original processor takes the branch and places a no-op in the pipeline, whereas the leakage does not take the branch and executes the next instruction.

5 Implementation

Pantomime is a full-path symbolic execution engine for GHC Core—the internal language of GHC, which is based on System F_C [41]. We implement Pantomime as a GHC Core plugin, which verifies that a circuit's leakage specification is correct, given an annotation, as shown below.

```
{-# ANN check (Theory axioms) #-}
check :: s -> i -> Bool
check = compose Pantomime
  {observation = obs, leakage = leak, simulator = sim, projection = proj}
```

To check the specification, Pantomime generates constraints, as described in Definition 3.9 and queries an SMT solver for their validity. Alternatively, the user can also omit the simulator, in which case Pantomime checks the conditions from Lemma 3.11. In case of a violation, Pantomime returns a counterexample, as described in § 4. As Pantomime symbolically executes GHC Core, it allows hardware designs, leakages and simulators to use the full range of functional programming techniques supported by Haskell. This includes monads, typeclasses, and GADTs—provided they can be synthesized to hardware. We implemented Pantomime in ~7600 lines of Haskell.

PCore. Pantomime represents Haskell programs in a core calculus called PCore. PCore closely matches GHC Core, but allows for symbolic variables which make SMT-solver types available within the calculus. Concretely, PCore uses symbolic variables in place of the concrete literals. For example, the expression `x & (y + 1) :: SymWordN 64` represents a 64-bit bitvector, where `x` and `y` are symbolic variables representing literals of type `SymWordN 64`. Symbolic values can contain case-splits over symbolic Booleans, which we capture via the `Union` data type, following Grisette [29].

Constraint Generation. To create constraints for a function, Pantomime generates new symbolic expressions for its arguments and then evaluates the function on them, which yields a symbolic constraint of `Union` datatype. Evaluation follows the rules of GHC core. We illustrate this process using function `add`, from § 2. Pantomime first generates fresh symbolic arguments for the function, starting with the first argument of type `Maybe Int`.

```
let state = If s.isN Nothing (Just s.val) :: Maybe Int
```

This symbolic expression captures an arbitrary value of type `Maybe Int`. The symbolic Boolean variable `s.isN` decides whether the data constructor is `Nothing` or `Just`. The symbolic bitvector variable `s.val` captures an arbitrary bitvector value whose size matches the platform size of `Int`. Pantomime builds a similar expression for the second input of type `(Maybe Int, Maybe Int)` :

```
let input = ( If i.0.isN Nothing (Just i.0.val)
            , If i.1.isN Nothing (Just i.1.val))
```

Pantomime uses these expressions to evaluate `add`. For simplicity, we focus on the first output of `add`, which is the state for its next cycle. Symbolically evaluating `add` on the freshly constructed input yields a case-statement with three patterns. The first pattern requires that both parts of the `input` are `Just` and the `Int` value is equal to `0` — this represents the fast path.

```
let pat1 = If (!(i.0.isN || i.1.isN) && i.0.val == 0) Nothing pat2
```

The second pattern only requires both inputs to be `Just` – this captures the slow-pass.

```
let pat2 = If !(i.0.isN || i.1.isN) (Just (i.0.val + i.1.val)) pat3
```

The last pattern represents the default case, without further conditions `let pat3 = Nothing`.

Merging Constraints. To avoid a blowup in constraint size, Pantomime merges equivalent data constructors, and applies simplifications. For this, it first sorts patterns according into a fixed ordering. Let us for example order `Nothing` before `Just`. Then, the second pattern becomes:

```
let pat2' = If (i.0.isN || i.1.isN) Nothing (Just (i.0.val + i.1.val))
```

We can now merge `pat2'` into `pat1`:

```
let pat1' =
  If ((i.0.isN || i.1.isN) || (!(i.0.isN || i.1.isN) && i.0.val == 0))
  Nothing (Just (i.0.val + i.1.val))
```

Using the transformation `x || (!x && y) = x || y`. we can simplify the constraint into its final form.

```
> fst $ add state input
If (i.0.isN || i.1.isN || i.0.val = 0) Nothing (Just (i.0.val + i.1.val))
```

Solver Types and User Axioms. Pantomime uses CλaSH which offers types such as unsigned bitvector type `Unsigned n`, and functions such as addition `addU` and bitwise-or `orU`, to express hardware. When symbolically interpreting CλaSH code, we want the SMT solver to interpret `Unsigned` as bitvectors; similarly `addU` and `orU` should be interpreted as their respective bitvector operation. In Pantomime, the user can directly encode this mapping via an annotation. For example, the following annotation maps `Unsigned n` and its operations to their respective bitvector equivalents.

```
axioms = PluginAxioms { typeAxioms = [Unsigned ↦ BitVector]
                      , termAxioms = [addU ↦ addBV, orU ↦ orBV] }
```

6 Evaluation

We evaluate Pantomime by asking a series of research questions.

RQ1: Can we use Pantomime to write/verify a RISC-V processor? We answer RQ1 in the affirmative by reporting on implementing and verifying AIMCore using Pantomime. AIMCore is a 5-stage in-order pipelined processor that implements the RISC-V V2.1 RV32I Base Integer Instruction Set [48]. It has been thoroughly tested against the official RISC-V test suites [38] and is capable of running actual software, including cryptographic benchmarks testing libsodium's ChaCha20, BLAKE2b, and SHA-256 [17] and wolfSSL's ECDSA [34]. The core also supports essential syscalls for program termination, printing, and reading entropy from system randomness. AIMCore consists of around 800 lines of Haskell and extracts to around 2400 lines of Verilog. Its design builds on the simpler processor in § 4 and consists of five pipeline stages. The core receives values from memory and the register file as inputs and produces memory and register file accesses as outputs. Programs are stored alongside values in memory. The core forwards values from the `wb` and `mem` stages to the `exe` stage (see `proc` below); pipeline stalls arise only from memory operations and branches.

Processor in Haskell. Since the core is written in Haskell, it can make use of expressive monad abstractions and elide explicit passing of inputs, outputs, and state; the complete processor pipeline is specified simply as a sequence of its individual stages:[1]

```
proc :: State -> Input -> (State, Output)
proc = execRWS (wb >> mem >> exe >> de >> fetch)
```

Since `proc` consists only of combinatorial logic, it can be directly compiled into an HDL using a tool like CλaSH and then synthesized.

Observation. We model an attacker that can observe the program counter, revealing control flow and timing. In a given cycle, the processor may access a memory value instead of fetching an instruction; our attacker observation model reflects this by exposing the program counter only during instruction fetches by wrapping the output type in the `Maybe` functor:

```
obs :: Output -> Maybe Address
```

Leakage. As in § 4, we write a leakage circuit that takes as input the processor's input (a memory read and two register file reads) and constructs the leakage:

```
leak :: LState -> Input -> (LState, Leak)
```

Since the register file and memory are external to the core, the core's pipeline structure affects when inputs are requested: for instance, `de` must request register operands one cycle before `exe` needs them. Similarly, the result of a load request issued by `mem` appears in `wb` in the following cycle. Since jump and branch addresses (and, by extension, the program counter) can result from arbitrary computation, `leak` must faithfully replicate the processor's architectural state, which requires matching the timing of requests to the register file and memory. Rather than distilling the core's timing behavior into separate logic, we capture it implicitly by structurally matching `leak` to `proc`—like in our case-study. `leak` therefore has a 5-stage pipeline structure. Our leakage shares this pipeline structure with other formal leakage descriptions, *e.g.*, [12, 16].

ISA Interpreter. To compute addresses, `leak` calls a black-box ISA interpreter that implements the ISA spec. So, `leak` must only explicitly replicate the timing behavior of `proc`—all actual computation is delegated to the black-box interpreter, which is reusable across different leakage models. As a

[1] `execRWS` runs the `RWS` monad, which automatically threads the processor's input, state, and output through the pipeline.

result, `leak` is easy to write for hardware designers: it looks just like `proc`, but with all computation abstracted away—retaining only the core's input timing, stalling, and value-forwarding.

Table 1. AIMCORE compared with the processors verified in LeaVe [47].

	AIMCore (Our work)		Processors verified by LeaVe [47]		
	Standard	Secure	Sodor	DarkRISCV-3	Ibex-small
Architecture					
ISA	RV32I		RV32I	RV32E	RV32IMC
Pipeline stages	5		2	3	2
Code size (loc)	800 Haskell / 2400 Verilog		400 Chisel / 2000 Verilog	620 Verilog	2500 Verilog
Forwarding	✓		✗	✗	✓
Security Properties & Proof					
Proof effort	180 loc simulator; 30 loc projection	1 loc simulator; 20 loc projection	16 manual invariants	13 manual invariants	59 manual invariants
Verification Time	38.8 sec (w/ simulator) 40.1 sec (w/o simulator)	3.5 sec (w/ simulator) 5.5 sec (w/o simulator)	97.8 min	11.1 min	118.7 min
Unconditional proof	✓	✓	✗	✗	✗

Leakage Datatype. The leakage data-type `Leak` that `leak` computes is a tuple defined as:

```
data LInstr = LJmp | LLoad RegId | LStore | LOther
type Leak = (LInstr, (Maybe RegId, Maybe RegId), Maybe Address)
```

It consists of three components: (1) the leakage instruction `LInstr`, (2) `(Maybe RegId, Maybe RegId)`, a pair of optional registers that the core instruction corresponding to the given leakage instruction may depend on, and (3) `Maybe Address`, the address of a jump or branch. `LJmp` has no `Address` payload because `leak` does not yet know the jump target when it needs to issue the leakage instruction. Instead, we leak jump addresses later—when they become available—using the third component of `Leak`. This approach also eliminates the need for a separate instruction for conditional branches: `leak` outputs jump target `Nothing` for conditional branches that end up not being taken. The simulator uses an instruction's register dependencies to stall when the core stalls. Instructions `LLoad` and `LStore` are needed to determine stalls. `LLoad` includes its destination register to determine load dependencies, and `LStore` has no payload. All other instructions act as a no-op, represented with `LOther`. As such, our leakage description captures the constant-time discipline.

Simulator. The simulator `sim` takes as input the leakage `Leak` and outputs the program counter:

```
sim :: SState -> Leak -> (SState, Maybe Address)
```

Like `leak`, `sim` focuses on timing: it must know when `leak` sends a jump address and when the core outputs the program counter (instead of the address for a memory access). So, `sim` too structurally mirrors `proc`: it's a 5-stage pipeline. `sim`'s pipeline is simple—it models only the core's stalling behavior (which also determines when `leak` sends a jump address).

RQ2: Do processors with more defenses have simpler leakage? In addition to the standard core, we also support a version of the core with a general security policy for hardware-based taint-tracking, inspired by [16]. This policy introduces a custom syscall that allows marking values in a specified memory region as secret. Any subsequent values that were computed using these secrets become tainted. To enforce constant-time software execution, the CPU immediately halts if a program attempts to branch on a tainted value. This mechanism provides strong security guarantees against timing side-channels by preventing secret-dependent control flow.

The core is parameterized by an applicative functor that wraps values; different instantiations implement different security policies. The `PubSec` applicative tags values as `Public` or `Secret` and propagates secrecy through computations, while the `Identity` applicative does not track secrecy. We instantiate the core with `PubSec` for the secure variant and `Identity` for the standard variant.

For the secure version of the core, the leakage definition is extremely simple: it's just the original input with any secret values censored (by setting them to a default value, like 0). The simulator is then just the original core composed with the projection function, which censors all secret values. This is an instance of the inversion paradigm from Lemma 3.11 with the inversion function being the identity. This short leakage definition demonstrates that with more secure processors, the leakage becomes simpler to define and reason about, as the core's inherent security guarantees can withstand a more permissive leakage model. As a consequence of having a simpler leakage, the time taken to verify the secure core is also significantly reduced.

RQ3: What's the proof effort of verification with Pantomime? Table 1 reports proof statistics for AIMCore and processors verified in LeaVe [47]. Writing a simulator is optional—Pantomime can directly check for the existence of a simulator—but simulators are executable programs that the user can step through, making them especially useful for debugging hardware, leakages, and the simulator itself (see § 4). This also means our proofs benefit from a broad ecosystem of programming aids—from static type checking to property-based testing frameworks like QuickCheck [15] (which we used extensively during development). For the normal version of AIMCore, the proof effort totals around 200 lines of Haskell: 180 lines for the simulator and only around 30 lines for the state projection function. The secure version of AIMCore requires only around 20 lines of Haskell for the state projection function.[2] Because Pantomime can simply check for the existence of a simulator, the actual required proof effort in both cases is just writing the state projection functions.

State projection functions are straightforward field-by-field mappings from the full processor state to the leakage and simulator states, censoring secret data which the simulator cannot reproduce. The projection logic is intuitive for hardware designers since it directly corresponds to selecting which parts of the processor state are visible to the attacker; LeaVe proofs are invariant-based assertions about the state space and inter-variable relations that the solver can't automatically discover. These kinds of assertions are notoriously challenging to derive and debug, often requiring expertise beyond hardware design.

RQ4: How does AIMCore compare to other verified processors? Table 1 shows a comparison between AIMCore and the processors verified in LeaVe. AIMCore is similar in complexity to other verified cores. While the other cores implement 2-, and 3-stage pipelines, AIMCore implements a more complex 5-stage pipeline. The largest processor in LeaVe, Ibex, implements further instructions outside the integer base-set, e.g., instructions for manipulating CSR registers, multiplication and division, and support for compressed instructions. We note that these extensions (except for multiplication and division), have been disabled during verification. Ibex and AIMCore are comparable in size when measured in lines of code. Finally, we note that LeaVe's proofs are conditional on functional correctness, and often make other assumptions *e.g.*, that instructions are memory aligned, or that certain instructions are not used. By contrast, our proofs are unconditional.

RQ5: How long does Pantomime take to verify correctness? Table 1 reports the verification time for AIMCore using Pantomime, alongside results for processors verified by LeaVe. LeaVe doesn't report on the hardware used for benchmarking; for AIMCore, we benchmarked using a consumer-grade AMD Ryzen 7 9700X CPU. While the verification results are not directly comparable as they concern different designs, Pantomime's verification time for AIMCore is two orders of magnitude lower than LeaVe's time for Ibex-small. We attribute this to the fact that LeaVe has to rely on expensive solvers to compute inductive invariants, while Pantomime only has to perform one-step equivalence checking between implementation and simulator. As a result, Pantomime allows users to check proofs interactively—as they write the core—simplifying development.

[2]The leakage circuits (which aren't parts of the proof) for the normal and secure core are about 300 and 10 lines of Haskell respectively, and both rely on an ISA interpreter that's a further 70 lines.

RQ6: Can leakages detect software bugs? As our leakages are executable, AIMCore can generate detailed leakage logs during execution. When running test software, users can enable a debug mode that executes multiple instances of the same program in lockstep and verifies all instances produce identical leakage traces. Upon detecting a divergence in the leakage traces, AIMCore pinpoints the instruction where the divergence occurs, along with the differing leakage values.

To demonstrate this capability, we used AIMCore to check the constant-time behavior of cryptographic algorithms in the latest versions of libsodium (v1.0.20), and wolfSSL (v5.8.2) on AIMCore. When executing an ECDSA key generation program using wolfSSL (with compiler optimizations enabled), we discover that the program exhibits leakage divergences during execution.

By tracing the PC values at the point of divergence, we identified that the leak occurs in an inlined `sp_256_get_entry_256_9` function. Although the source code employs constant-time defense techniques to remove secret-dependent branches, the compiler's optimizations can undo them, causing timing leaks [39]. The program also exhibits leakage divergences when compiled without optimizations. In this case, GCC introduces a variant-time routine for multiplication as RV32I lacks native multiplication instructions. To the best of our knowledge, this leakage is previously unknown and not specific to AIMCore. We suspect this would also apply on other cores with variable-time multiplication opcodes. While we simulate Pantomime in software for our experiments, its leakage trace can be used to check leakages (*e.g.*, via fuzzing) at native speed in hardware.

7 Discussion and Limitations

Executable Leakages. Pantomime focuses on executable leakages that operate over the same input/output sequence as the hardware. This limits the level of abstraction a leakage can achieve, which may make it harder for programmers to understand how to program the hardware securely. We show that executable leakages are useful: they can be run together with the hardware to monitor leakage (even in silicone); more secure processors have easy-to-understand leakage descriptions; beyond processors, they are applicable to other circuits like specialized accelerators that don't offer an instruction set architecture. Unlike other methods [19, 42, 47], they do not rely on functional correctness assumptions, which increases our confidence in their correctness. In the future, we want to use them as targets for symbolic execution of cryptographic software [6], and explore refinement proofs for relating them to leakage at the level of an instruction set architecture.

State Projection Proofs and Inductive Invariants. Pantomime only supports proofs via the state projection rule. Intuitively, the state projection rule supports proofs where the equivalence is established by erasing information in implementation and observation state to derive leakage and simulator state. Leakages that do not fit this pattern cannot be proven via our rule. We give an example in Appendix D. The state projection rule establishes an invariance between leakage and simulator, and implementation and observation – the projected parts of their state need to be equal. This is similar to inductive invariants establishing equality between different copies of the same program, as used by non-interference based methods for leakage verification. Pantomime allows programmers to establish equivalence computationally, via the projection function. We believe that this is easier for hardware designers without experience in formal methods.

Leakage Description and Observation. Users specify attacker observations (via a circuit `obs`), similar to other approaches [19, 42, 47]. This description serves as ground-truth specification of the attacker capabilities. Pantomime verifies that the leakage description (circuit `leak`) correctly captures everything that's leaked via the attacker observation. Insufficient leakage descriptions result in a verification failure that yields a counterexample (see § 4). Leakage descriptions can be arbitrary stateful circuits. In practice, their output type reflects the intended high-level leakage of the processor. E.g., in AIMCore, this type specifies the reason for leakage (control-flow or memory).

Translation to Verilog. We trust CλaSH to not introduce further side-channels via compilation to hardware. We have manually inspected the generated Verilog code to ensure that the compiled code closely matches its description in Haskell. Indeed, CλaSH already provides cycle-level descriptions of the circuit that are very close to the extracted hardware description. We leave equivalence proofs between the Haskell description and the Verilog target as future work.

8 Related Work

Verifying Security Properties of RTL Designs. UPEC [21] detects transient execution vulnerabilities in RTL designs (or proves their absence), but is restricted to fixed properties, while Pantomime can express arbitrary leakage properties via (stateful) observation functions and leakages. ConjunCT [19] and UPEC-DIT [18] identify subsets of a processor's instruction set that are data-oblivious in the sense that their operands do not affect the timing of the computation. H-Houdini [20] checks for the same property, but scales better by proposing a new invariant synthesis approach that exploits locality, e.g., due to pipelining. While these techniques scale well, they offer limited guarantees: since branch instructions trivially affect timing, these instruction are not data-oblivious. Hence, these techniques cannot guarantee safety of code involving branches, as is used, e.g., in constant-time code in cryptographic libraries. SecVerilog [53] extends Verilog with a type system to statically check timing-sensitive information flow. SpecVerilog [52] builds upon SecVerilog's type system to express information-flow safety under speculative execution. Unlike Pantomime, these techniques all use classical non-interference reasoning and do not provide executable leakages and proof artifacts.

Verifying Leakage Descriptions. Iodine [43] proves secret-independent timing in hardware, given usage assumptions expressed on inputs and internal wires. Xenon [44] synthesizes such usage assumptions semi-automatically. LeaVe [47] verifies a restricted form of leakage contracts (Definition 3.6) of RTL designs. Like Iodine and Xenon, LeaVe expresses leakage via stateless leakage monitors that can observe internal processor state. Unlike Iodine, assumptions in LeaVe can only be expressed on wires that represent final computation results at instruction retirement. Under the assumption that the processor is functionally correct (*i.e.*, implements its ISA), this allows LeaVe to link leakage at the hardware level to leakage at the ISA level. However, there are no (executable) artifacts that represent ISA level leakage and the functional correctness assumption is often left unverified, making end-to-end correctness arguments difficult. Contract Shadow Logic [42] follows a similar approach to LeaVe, but uses exhaustive state space exploration via model checking instead of inductive invariants to validate contracts, which limits its scalability. Both Contract Shadow Logic and LeaVe use classic non-interference reasoning and rely on expensive solvers or handwritten invariants. By contrast, Pantomime uses constructive simulation-based proofs which yield executable artifacts, and make debugging easier.

Synthesizing Leakage Descriptions. [32] synthesizes leakage contracts using a user-supplied lists of "contract atoms," which represent potential leakage sources, along with a set of test cases. However, [32] does not prove the correctness of the leakage contracts it synthesizes. RTL2MμPATH [23] uses model checking to enumerate microarchitectural paths of an instruction through the processor and records which instructions may influence the path choice. It requires designers to annotate with μFSMs—microarchitectural finite state machines that govern updates of the processor state and relies on exhaustive state space exploration which limits scalability. As the technique builds on classic non-interference it doesn't yield executable leakages and proof artifacts.

Simulation-based Proofs in Crypto. In designing our approach, we were inspired by simulation-based proofs in cryptography [27] (as used e.g., in Universal Composability [11]). However, beyond the common idea of using simulators, the two proof methods diverge significantly. In cryptography, the aim is often to show that a (potentially active) adversary—with bounded computational resources—can learn nothing of interest about the secrets processed in a cryptographic protocol. By contrast, we use simulators to precisely characterize the side-channel leakage of a processors against a passive attacker—one that can observe side-channels (e.g., timing), but cannot actively influence the computation.

9 Conclusion

We introduced simulation-based leakage proofs, a new approach to leakage verification of hardware, where security is proved by constructing a simulator—another hardware design that must faithfully replicate all attacker-observable behavior from explicitly leaked secrets. Simulation-based proofs offer a constructive alternative to classic non-interference proofs, exposing a proof object—the simulator, witnessing the correctness claim. We realized this approach in the PANTOMIME verification tool and used it to verify the AIMCORE RISC-V CPU, as well as a side-channel hardened version of the processor. Proof checking with PANTOMIME takes seconds, rather than hours, and leakages and their proofs are computational, making them easy to run and debug.

References

[1] Johan Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. Association for Computing Machinery, New York, NY, USA, 40–53. doi:10.1145/325694.325702

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying {Constant-Time} Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.

[3] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 623–639.

[4] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 147–154. doi:10.1109/ICCAD.2017.8203772

[5] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 503–519. https://www.usenix.org/conference/osdi22/presentation/athalye

[6] Anish Athalye, M Frans Kaashoek, Nickolai Zeldovich, and Joseph Tassarotti. 2023. Leakage models are a leaky abstraction: the case for cycle-level verification of constant-time cryptography. In *Workshop on Programming Languages and Computer Architecture*.

[7] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCarthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2018. SideTrail: Verifying Time-Balancing of Cryptosystems. In *Verified Software. Theories, Tools, and Experiments*, Ruzica Piskac and Philipp Rümmer (Eds.). Springer International Publishing, Cham, 215–228.

[8] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. PhD Thesis - Research UT, graduation UT. University of Twente, Netherlands. doi:10.3990/1.9789036538039 eemcs-eprint-23939.

[9] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. *SIGPLAN Not.* 34, 1 (Sept. 1998), 174–184. doi:10.1145/291251.289440

[10] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 715–733. doi:10.1145/3460120.3484583

[11] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Paper 2000/067. https://eprint.iacr.org/2000/067

[12] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. doi:10.1145/3385412.3385970

[13] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. FaCT: A Flexible, Constant-Time Programming Language. In *2017 IEEE Cybersecurity Development (SecDev)*. 69–76. doi:10.1109/SecDev.2017.24

[14] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security*.

[15] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 46, 4 (May 2011), 53–64. doi:10.1145/1988042.1988046

[16] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. {ProSpeCT}: Provably Secure Speculation for the {Constant-Time} Policy. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7161–7178.

[17] Frank Denis. [n. d.]. *libsodium*. https://github.com/jedisct1/libsodium

[18] Lucas Deutschmann, Johannes Müller, Mohammad Rahmani Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2024. A Scalable Formal Verification Methodology for Data-Oblivious Hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 9 (2024), 2551–2564. doi:10.1109/TCAD.2024.3374249

[19] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W. Fletcher. 2024. ConjunCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety Against Microarchitectural Timing Attacks . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3735–3753. doi:10.1109/SP54263.2024.00180

[20] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 603–618. doi:10.1145/3669940.3707263

[21] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Muller, Jorg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2023. An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors . *IEEE Trans. Comput.* 72, 01 (Jan. 2023), 222–235. doi:10.1109/TC.2022.3152666

[22] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. doi:10.1109/SP.1982.10014

[23] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P. Mulligan, Gustavo Petri, Christopher W. Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 507–524. doi:10.1109/MICRO61859.2024.00045

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[25] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[26] Leslie Lamport and Stephan Merz. 2022. Prophecy Made Simple. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 6 (April 2022), 27 pages. doi:10.1145/3492545

[27] Yehuda Lindell. 2016. How To Simulate It – A Tutorial on the Simulation Proof Technique. Cryptology ePrint Archive, Report 2016/046. https://eprint.iacr.org/2016/046 https://eprint.iacr.org/2016/046.pdf.

[28] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 605–622.

[29] Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (Jan. 2023), 33 pages. doi:10.1145/3571209

[30] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 370–384. doi:10.1145/3341301.3359651

[31] Daniel Moghimi. 2023. Downfall: Exploiting Speculative Data Gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*.

[32] Gideon Mohr, Marco Guarnieri, and Jan Reineke. 2024. Synthesizing Hardware-Software Leakage Contracts for RISC-V Open-Source Processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546681

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006*. Springer, 1–20.

[34] Todd Ouska. [n. d.]. *wolfSSL*. github.com/wolfssl/wolfssl

[35] Colin Percival. 2005. *Cache missing for fun and profit*. Technical Report. BSDCan.

[36] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[37] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *30th USENIX*

Security Symposium (USENIX Security 21). USENIX Association, 1451–1468. https://www.usenix.org/conference/usenixsecurity21/presentation/ragab

[38] RISC-V Foundation. 2025. riscv-tests: RISC-V Architectural Test Suite. GitHub repository. https://github.com/riscv/riscv-tests Accessed: 5 November 2025.

[39] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. 2025. Breaking Bad: How Compilers Break Constant-Time Implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*. Association for Computing Machinery, New York, NY, USA, 1690–1706. doi:10.1145/3708821.3733909

[40] Mary Sheeran. 2005. Hardware Design and Functional Programming: a Perfect Match. *JUCS - Journal of Universal Computer Science* 11, 7 (2005), 1135–1158. arXiv:https://doi.org/10.3217/jucs-011-07-1135 doi:10.3217/jucs-011-07-1135

[41] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. 53–66.

[42] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2024. RTL Verification for Secure Speculation Using Contract Shadow Logic. arXiv:2407.12232 [cs.AR] https://arxiv.org/abs/2407.12232

[43] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1411–1428. https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall

[44] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2021. Solver-Aided Constant-Time Hardware Verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 429–444. doi:10.1145/3460120.3484810

[45] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*. Paper=https://mdsattacks.com/files/ridl.pdfSlides=https://mdsattacks.com/slides/slides.htmlWeb=https://mdsattacks.comCode=https://github.com/vusec/ridlPress=http://mdsattacks.com Intel Bounty Reward (Highest To Date), Pwnie Award Nomination for Most Innovative Research, CSAW Best Paper Award Runner-up, DCSR Paper Award.

[47] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2128–2142. doi:10.1145/3576915.3623192

[48] Andrew Waterman and Krste Asanović. 2024. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. RISC-V Foundation. https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications Document Version 20240411.

[49] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (Jan. 2019), 29 pages. doi:10.1145/3290390

[50] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 19–33. doi:10.1145/3658644.3690319

[51] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium*. 719–732.

[52] Drew Zagieboylo, Charles Sherk, Andrew C. Myers, and G. Edward Suh. 2023. SpecVerilog: Adapting Information Flow Control for Secure Speculation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2068–2082. doi:10.1145/3576915.3623074

[53] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 503–516. doi:10.1145/2786763.2694372