Pantomime: Simulation-Based Leakage Proofs for Hardware Side-Channel Security

Robin Webbers
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
r.m.a.webbers@vu.nl

Robert Schenck
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
r@bert.lol

Alp Adnan Basar
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
a.a.basar@vu.nl

Kristina Sojakova
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
k.sojakova@gmail.com

Klaus v. Gleissenthall
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
k.freiherrvongleissenthal@vu.nl

Abstract

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, current techniques either don't allow for leakage descriptions expressive enough to capture real-world software countermeasures like constant time programming, or they rely on expensive solvers and handwritten invariants, making them difficult to apply to larger designs—especially for hardware designers who are not experts in formal verification. In this paper, we present a new approach to leakage verification: *simulation-based leakage proofs*, where proofs are developed alongside the design.

Inspired by techniques in cryptography, simulation-based leakage proofs show that a leakage description correctly captures a hardware design by constructing a simulator—another hardware design that must faithfully replicate all attacker-observable behavior from explicitly leaked secrets. Simulation-based leakage proofs are easy to write and to debug: writing a simulator just means writing another hardware module, which is what hardware designers are already best at. They also capture all common software defenses and make proof checking orders of magnitude faster.

We implemented 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. Unlike previous leakage proofs, which are conditional on the unproven functional correctness of the CPU, our proofs hold unconditionally.

Keywords

Hardware, Side-Channels, Leakage

1 Introduction

Context. Side-channel attacks via cache and timing can leak secrets used in private computations [22, 25, 30, 31, 45]. To prevent side-channels in software, programmers use countermeasures like branch balancing [1, 6, 44], and constant-time (or data-oblivious) programming [2, 9, 12, 43]. 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 [13], and speculative execution [21, 28, 33], and their various micro-architectural buffers [39, 40]. 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 aims to formally verify existing open-source hardware designs against descriptions of their intended leakage [4, 5, 16–18, 20, 36–38, 41]. While there has been tremendous progress in the area and these techniques have now managed to verify leakage descriptions for a number of open-source processors, their focus on verifying existing designs comes with a number of drawbacks. Their leakage descriptions are either not expressive enough to capture real-world defenses like the constant-time programming discipline [4, 5, 16–18], or they heavily rely on expensive solvers—either to find inductive invariants [37, 38, 41], or to exhaustively explore the design's state space [20, 36]—which limits their applicability to larger designs. 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 [18, 41]. While this helps with scaling, it places a heavy burden on the user: inductive invariants are difficult to come up with by hand and even harder to debug when they are wrong [27]. This is especially true for hardware designers, unless they also happen to be experts in formal verification.

Our Solution. In this paper, we propose simulation-based leakage proofs—a co-design approach to verifying hardware against explicit leakage descriptions, where proofs are developed alongside the code. Based on insights borrowed from cryptography [10, 24], 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. As simulators are just programs, they can be written, debugged, and executed like any other hardware design. This makes simulation-based leakage proofs a good candidate for proofs that are written alongside the design by the hardware developers themselves. Indeed, the main proof effort lies in doing what hardware designers are already best at: writing hardware designs. Simulation-based leakage proofs are expressive and capture all common software defenses against cache and timing side-channels.

They are also fast to check, as we only need to show single-step equivalence between the simulator and the original design.

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.

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 [8, 34], 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. We build on the insight that functions describing hardware behavior *within* a clock tick are *loop-free* and therefore allow for easy equivalence proofs. Interestingly, our proofs generally require us to prove an equivalence between functions of *different types*. To enable these proofs, we propose 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 [23]. Except for state projection functions—which translate between the states of different types and have to be provided by the user—equivalence proofs can be fully automated via an SMT solver.

Implementation and Evaluation. We implemented our method in Pantomime, a tool for writing processors and proofs in unrestricted Haskell (including support for higher-order functions, type classes, abstract data types, and monads). Hardware descriptions written with Pantomime can be extracted to hardware description languages using CλaSH [7]. To prove equivalence between simulator and implementation, we developed a new symbolic execution engine based on Grisette [26]. 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 verify it against a leakage description. Unlike [36, 41], where leakage proofs are conditional on the (unproven) functional correctness of the CPU, our proof of AIMCore is unconditional. Moreover, proof checking is orders of magnitude faster than previous methods, making Pantomime suitable for interactive use.

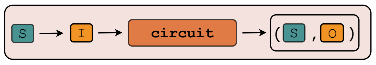

Figure 1: Circuits take state of type `S`, an input of type `I`, and return a pair of type `(S,O)`.

Contributions. In summary, we make the following contributions.

- **Simulation Based Proofs:** A co-design approach for verifying leakage descriptions of hardware circuits.
- **The State Projection Rule:** A proof rule which allows us to prove the equivalence of circuits with different types.
- **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 and its leakage description, resulting in the first processor with an unconditional leakage proof against the constant-time discipline.

Availability. The full source code for Pantomime and AIMCore is available in the supplementary material and will be open-sourced.

2 Overview

We illustrate our technique on a simple adder with a fast path (§ 2.1) and discuss how 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 function of type `S -> I -> (S, O)`, where `S` represents the circuit's state (i.e., the current values of all its registers), `I` and `O` represent its inputs and outputs, and `(S, O)` represents a pair combining the circuit's state with its output of type `O`—see Figure 1.

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. We show the adder's code in Listing 1 using Haskell syntax. 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.

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

Listing 1: Running Example: A simple adder with a fast-path.

Computing the Output. When add's first component is Just 0, we take the fast path (line 3): 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 4): 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 5), 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. mealy, shown below, transforms a circuit into a *transducer*—a function of type [I] -> [O], which maps streams of inputs to streams of outputs. mealy takes a single-clock-tick circuit description and an initial state and returns the corresponding transducer: on input, it applies the circuit to update its state and produce an output, which is prepended onto the outputs produced by recursing on the remaining inputs.

```
mealy :: (S -> I -> (S, O)) -> S -> [I] -> [O]
mealy _ _ [] = []
mealy circuit s (i:is) =
  let (s', o) = circuit s i in
  o : mealy circuit s' is
```

Running the Adder. On add, initial state Nothing, and inputs

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

mealy produces the following output:

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

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. In particular, this circuit outputs True in a given clock cycle if and only if an output was produced by add. To construct add_obs, we must first define circuit composition.

Sequential Composition. Two circuits c1 :: S1 -> I1 -> (S1, O1) and c2 :: S2 -> I2 -> (S2, O2) may be composed to form a new circuit c1 ∘ c2 that first executes c1 and then c2. Figure 2 illustrates circuit composition, and we define ∘ below; in short, circuit composition uses circuit c1's output as input to circuit c2, and combines their individual states.

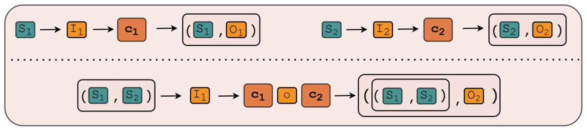

Figure 2: For circuits c1, and c2, we write c1 ∘ c2 for their sequential composition.

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

Listing 2: Leakage function leak takes two optional integers, like add, and returns leaked values for each component.

```
c1 ∘ c2 :: (S1, S2) -> I1 -> ((S1,S2), O2)
c1 ∘ c2 (s1,s2) i1 =
  let (s1', o1) = c1 s1 i1 in
  let (s2', o2) = c2 s2 o1 in
  ((s1',s2'), o2)
```

Lifting. We cannot directly apply circuit composition to obs since it's 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 :: () -> I -> ((), O).

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

```
add_obs :: (Maybe Int, ())
        -> (Maybe Int, Maybe Int)
        -> ((Maybe Int, ()), Bool)
add_obs = add ∘ (lift obs)
```

The obs observation function describes an attacker that can observe the transducer corresponding to circuit add_obs. 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 using the initial state (Nothing, ()) on the inputs is produces the following output list:

```
mealy add_obs (Nothing, ()) 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, shown in Listing 2. The function leak 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. Applying lift leak on inputs is using mealy produces the following outputs:

```haskell
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)
```

Listing 3: Simulator `sim` which replicates the attacker-observable behavior of `add` using the outputs of `leak`.

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

While the leakage and observation are pure, stateless functions here, 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. The simulator circuit `sim` is shown in Listing 3 and closely follows circuit `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.

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:

```haskell
leak_sim :: ((), Bool)
         -> (Maybe Int, Maybe Int)
         -> (((), Bool), Bool)
leak_sim = (lift leak) ∘ sim
```

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 (Nothing,())` and `mealy leak_sim ((),Nothing)` 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.

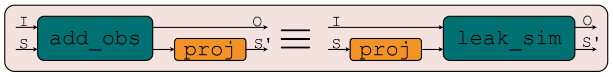

Figure 3: Proof obligation for the correctness of `leak` with respect to `add`. `S = (Maybe Int, ())`, `S' = ((), Bool)`, `I = (Maybe Int, Maybe Int)`, and `O = Bool`. ≡ is input/output equivalence.

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` in Listing 1, we can see that the integer value `v` stored in the state of form `Just v` is *irrelevant* in circuit `add_obs`. Lines (3) – (4) do not depend on the state at all. Line (5) does output `v`, but we discard the result in `add_obs` due to the output projection with `obs`.

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 keeping the information whether there is a value or not, and finally adds `leak_sim`'s unit state.

```haskell
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 3. As the resulting functions have the same type, we can now 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 leakage `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.

3 Simulation-Based Leakage Proofs

3.1 Language

We formalize our proof method in a core language (Figure 4) based on the 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 [32], slightly adapted to our purposes. Importantly, our language satisfies the *preservation* property that says 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 integers and a number of binary and unary operators (functions), which are used in the translation of expressions in the language to a hardware description language (see § 3.4). 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. 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 $\mathtt{let}\ \mathtt{p}\ =\ \mathtt{a}\ \mathtt{in}\ \mathtt{e}$ according to the choice of pattern $\mathtt{p}$: $\mathtt{let}\ \mathtt{x}\ =\ \mathtt{a}\ \mathtt{in}\ \mathtt{e}$ is elaborated to $(\lambda\,\mathtt{x}.\ \mathtt{e})\ \mathtt{a}$, and $\mathtt{let}\ \{\mathtt{l}_1 = \mathtt{p}_1, \ldots, \mathtt{l}_n = \mathtt{p}_n\}\ =\ \mathtt{a}\ \mathtt{in}\ \mathtt{e}$ elaborates to $\mathtt{let}\ \mathtt{p}_1\ =\ \mathtt{a.l}_1$ $\mathtt{in}\ \ldots\ \mathtt{in}\ \mathtt{let}\ \mathtt{p}_n\ =\ \mathtt{a.l}_n\ \mathtt{in}\ \mathtt{e}$. We write $\lambda\,\mathtt{x}_1\ \ldots\ \mathtt{x}_n.\ \mathtt{e}$ for $\lambda \mathtt{x}_1. \ldots \lambda\,\mathtt{x}_n.\ \mathtt{e}$, and $\lambda\,\mathtt{p}.\ \mathtt{e}$ for $\lambda\,\mathtt{x}.\ \mathtt{let}\ \mathtt{p}\ =\ \mathtt{x}\ \mathtt{in}\ \mathtt{e}$. If $\mathtt{e}$ is a term of a record type $\{\overrightarrow{\mathtt{l_i} : \mathtt{s_i}}\}$, we write $\mathtt{e}\{\mathtt{l_i} := \mathtt{e_i}\}$ for the single-field update of the record $\mathtt{e}$ to $\mathtt{e_i}$ at field $\mathtt{l_i}$, and $\mathtt{e}\{\mathtt{l}_{\mathtt{i}_1} := \mathtt{e}_{\mathtt{i}_1}\}\ \ldots \{\mathtt{l}_{\mathtt{i}_k} := \mathtt{e}_{\mathtt{i}_k}\}$ for the sequence of updates $\mathtt{e}\{\mathtt{l}_{\mathtt{i}_1} := \mathtt{e}_{\mathtt{i}_1}\}\ \ldots\ \{\mathtt{l}_{\mathtt{i}_k} := \mathtt{e}_{\mathtt{i}_k}\}$. Finally, infix applications aren't supported: when unambiguous, $\mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2$ should be understood as syntactic sugar for $\mathtt{bop}\ \mathtt{e}_1\ \mathtt{e}_2$.

Evaluation. Figure 5 shows the evaluation rules for the core language. Evaluation is based on the reduction relation $\rightsquigarrow$, which is defined on closed, well-typed terms of the same type. The substitution operator $[\mathtt{x} = \mathtt{e}]$ performs a capture-avoiding substitution of free occurrences of the variable $\mathtt{x}$ with term $\mathtt{e}$. Beta reduction and the step rules for evaluating applications, records, variants, and case-expressions are standard. Rule [Record Reduce] reduces a record to the expression associated with the applied label. Rule [Case Reduce] evaluates a case expression $\mathtt{case}\ \mathtt{l_k}\ \mathtt{e}\ \mathtt{of}\ \overrightarrow{\mathtt{l_i}\ \mathtt{x_i} \longrightarrow \mathtt{e_i}}$ over a variant by matching $\mathtt{l_k}\ \mathtt{e}$ with case $\mathtt{l_k}\ \mathtt{x_k}$, returning $\mathtt{e_k}[\mathtt{x_k} = \mathtt{e}]$. Since the STLC is both confluent and strongly normalizing, the order in which we apply the evaluation rules does not matter and evaluation always terminates. We write $[\![\mathtt{e}]\!]$ to represent the unique term obtained by exhaustive application of the evaluation steps to $\mathtt{e}$. We write $\rightsquigarrow_*$ for the reflexive-transitive closure of $\rightsquigarrow$.

Unit, Optional, and Tuple Types. We encode the unit type $()$ as the empty record type with no fields. For any type $\mathtt{t}$, we encode $\mathtt{Maybe\ t}$ as the variant type $\langle \mathtt{Just} : \mathtt{t},\ \mathtt{Nothing} : ()\rangle$. Conditionals $\mathtt{if}\ \mathtt{e}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$ stand for the corresponding $\mathtt{case}$ expressions over the variant type $\mathtt{Bool} := \langle \mathtt{true} : (),\ \mathtt{false} : ()\rangle$. We encode the tuple type $(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ as the the record type $\{1 : \mathtt{t}_1, \ldots, \mathtt{n} : \mathtt{t}_n\}$.

Base Types	$\mathtt{b}$	$::=$	$\mathtt{Int}, \mathtt{Word8}, \mathtt{Word16}, \ldots$	
Operators	$\mathtt{bop}$	$::=$	$+, -, \ldots$	*binary operators*
	$\mathtt{uop}$	$::=$	$\sim, !, \ldots$	*unary operators*
Constants	$\mathtt{c}$	$::=$	$0, 1, \ldots$	*numbers*
		$\mid$	$\mathtt{bop}\ \mid\ \mathtt{uop}$	*operators*
Types	$\mathtt{t}$	$::=$	$\mathtt{b}$	*base types*
		$\mid$	$\{\overrightarrow{\mathtt{l_i} : \mathtt{t_i}}\}$	*record types*
		$\mid$	$\langle\overrightarrow{\mathtt{l_i} : \mathtt{t_i}}\rangle$	*variant types*
		$\mid$	$\mathtt{t}_1 \rightarrow \mathtt{t}_2$	*function types*
Terms	$\mathtt{e}$	$::=$	$\mathtt{c}$	*constants*
		$\mid$	$\mathtt{x}$	*variables*
		$\mid$	$\lambda \mathtt{x}.\ \mathtt{e}$	*abstraction*
		$\mid$	$\mathtt{e}_1\ \mathtt{e}_2$	*application*
		$\mid$	$\{\overrightarrow{\mathtt{l_i} = \mathtt{e_i}}\}$	*records*
		$\mid$	$\mathtt{e.l}$	*record fields*
		$\mid$	$\mathtt{l\ e}$	*variants*
		$\mid$	$\mathtt{case}\ \mathtt{e}\ \mathtt{of}\ \overrightarrow{\mathtt{l_i x_i} \longrightarrow \mathtt{e_i}}$	*pattern match*
Extractable	$\mathtt{s}$	$::=$	$\mathtt{b}$	*base types*
Types		$\mid$	$\{\overrightarrow{\mathtt{l_i} : \mathtt{s_i}}\}$	*record types*
		$\mid$	$\langle\overrightarrow{\mathtt{l_i} : \mathtt{s_i}}\rangle$	*variant types*
Patterns	$\mathtt{p}$	$::=$	$\mathtt{x}$	*variables*
		$\mid$	$\{\overrightarrow{\mathtt{l_i} = \mathtt{p_i}}\}$	*records*

Figure 4: Syntax of types and terms in our core language.

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e}'}{\lambda\,\mathtt{x}.\ \mathtt{e} \rightsquigarrow \lambda\,\mathtt{x}.\ \mathtt{e}'}\ [\lambda]$$

$$\frac{\mathtt{e}_1 \rightsquigarrow \mathtt{e}_1' \qquad \mathtt{e}_2 \rightsquigarrow \mathtt{e}_2'}{\mathtt{e}_1\ \mathtt{e}_2 \rightsquigarrow \mathtt{e}_1'\ \mathtt{e}_2'}\ [\text{App}]$$

$$\frac{}{(\lambda \mathtt{x}.\ \mathtt{e}_1)\ \mathtt{e}_2 \rightsquigarrow \mathtt{e}_1[\mathtt{x} = \mathtt{e}_2]}\ [\text{Beta Reduce}]$$

$$\frac{\mathtt{e_i} \rightsquigarrow \mathtt{e_i}'}{\overrightarrow{\mathtt{l_i} = \mathtt{e_i}} \rightsquigarrow \overrightarrow{\mathtt{l_i} = \mathtt{e_i}'}}\ [\text{Record Field}]$$

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e}'}{\mathtt{e.l} \rightsquigarrow \mathtt{e}'.\mathtt{l}}\ [\text{Record Label}]$$

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e}'}{\mathtt{l\ e} \rightsquigarrow \mathtt{l\ e}'}\ [\text{Variant}]$$

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e}'}{\mathtt{case}\ \mathtt{e}\ \mathtt{of}\ \overrightarrow{\mathtt{l_i x_i} \longrightarrow \mathtt{e_i}} \rightsquigarrow \mathtt{case}\ \mathtt{e}'\ \mathtt{of}\ \overrightarrow{\mathtt{l_i x_i} \longrightarrow \mathtt{e_i}}}\ [\text{Case}]$$

$$\frac{}{\{\overrightarrow{\mathtt{l_i} = \mathtt{e_i}}\}.\mathtt{l_j} \rightsquigarrow \mathtt{e_j}}\ [\text{Record Reduce}]$$

$$\frac{}{\mathtt{case}\ \mathtt{l_k}\ \mathtt{e}\ \mathtt{of}\ \overrightarrow{\mathtt{l_i x_i} \longrightarrow \mathtt{e_i}'} \rightsquigarrow \mathtt{e_k}'[\mathtt{x_k} = \mathtt{e}]}\ [\text{Case Reduce}]$$

Figure 5: Evaluation rules for our core language.

Example 3.1. Consider the term

$$\begin{aligned}&\texttt{ex1} : \texttt{Int} \to \texttt{Maybe Word32} \to (\texttt{Word32}, \texttt{Word32})\\ &\texttt{ex1} = \lambda\, \texttt{s i}.\ (\texttt{s} + 1, \texttt{case i of Just a} \to \texttt{s} + \texttt{a},\ \texttt{Nothing} \to \texttt{s}).\end{aligned}$$

Then we have

$$\begin{aligned}[\![\texttt{ex1 0 Nothing}]\!] &= (1, 0),\\ [\![\texttt{ex1 0 (Just 1)}]\!] &= (1, 1)\ .\end{aligned}$$

3.2 Circuits, Transducers, and Circuit Reduction

Definition 3.2 (circuit). Let S, I, O be types in the core language. A circuit is a term of type $\texttt{S} \to \texttt{I} \to (\texttt{S}, \texttt{O})$.

For circuits to be hardware-extractable, S, I, O must be extractable types (*i.e.*, they can't contain function types). In practice, this means we can use higher-order functions freely whenever we don't need to synthesize the circuit into hardware—for example, for the leakage, observation or simulator circuits.

Example 3.3. ex1 is an extractable circuit that increments its state 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. Circuits describe the behavior of a hardware design in a single clock tick. The *Mealy* function M transforms a circuit into a *transducer* that applies one input per clock tick (from a list of inputs) to the circuit and produces a corresponding list of outputs. Transducers therefore represent the actual input/output behavior of sequential hardware.

Definition 3.4 (Mealy). Let $c : S \to I \to (S, O)$ be a circuit. The Mealy function M is defined as follows, where $(s', o) = [\![c\ s\ i]\!]$, : is list concatenation, and [] is the empty list.

$$M\ :\ (S \to I \to (S, O), S, [I])\ \to\ [O]$$
$$M(c, s, is) = \begin{cases}[] & \text{if } is = [],\\ o : M(c, s', is') & \text{if } is = \texttt{i} : is'\end{cases}$$

M takes a circuit c and initial state s along with a list of inputs is. It then applies c to the first input of is to compute a new state s' and output o, which it prepends to the list of outputs and then recursively computes the remaining outputs using the new state s' and remaining inputs is'.

Example 3.5. Revisiting circuit ex1, let $s = \texttt{0}$ be an initial state and $is = [\texttt{Just 1}, \texttt{Nothing}, \texttt{Just 2}, \texttt{Just 3}]$ a list of inputs. Applying the Mealy function to ex1 with these inputs yields

$$\begin{aligned}M(\texttt{ex1}, s, is) &= M(\texttt{ex1}, \texttt{0}, [\texttt{Just 1}, \texttt{Nothing}, \texttt{Just 2}, \texttt{Just 3}])\\ &= 1\ :\ M(\texttt{ex1}, 1, [\texttt{Nothing}, \texttt{Just 2}, \texttt{Just 3}])\\ &= [1, 1, 4, 6].\end{aligned}$$

Observational Equivalence. As discussed in § 2, our proof method requires proving equivalence of functions. In particular, we are interested in *observable* function equivalence (rather than syntactic equivalence). For example, the functions $\lambda\,\texttt{i}.\ \texttt{i} + \texttt{0}$ and $\lambda\,\texttt{i}.\ \texttt{0} + \texttt{i}$ should be equivalent.

Definition 3.6 (Observational Equivalence). Two closed terms e, e′ of the same type t are *observationally equivalent*, written $e \equiv e'$ when one of the following holds:

(1) t is a base type and $[\![\texttt{e}]\!] = [\![\texttt{e}']\!]$.
(2) $\texttt{t} = \texttt{t}_1 \to \texttt{t}_2$ and if for any two closed terms $\texttt{e}_1, \texttt{e}'_1 : \texttt{t}_1$ with $e_1 \equiv e'_1$, then $\texttt{e e}_1 \equiv \texttt{e}'\ \texttt{e}'_1$.
(3) $\texttt{t} = \{\overrightarrow{\texttt{l}_\texttt{i} : \texttt{t}_\texttt{i}}\}$, $\texttt{e} \rightsquigarrow_* \{\overrightarrow{\texttt{l}_\texttt{i} = \texttt{e}_\texttt{i}}\}$, $\texttt{e}' \rightsquigarrow_* \{\overrightarrow{\texttt{l}_\texttt{i} = \texttt{e}'_\texttt{i}}\}$, and for all i, $\texttt{e}_\texttt{i} \equiv \texttt{e}'_\texttt{i}$.
(4) $\texttt{t} = \langle\overrightarrow{\texttt{l}_\texttt{i} : \texttt{t}_\texttt{i}}\rangle$, $\texttt{e} \rightsquigarrow_* \texttt{l}_\texttt{k}\ \texttt{e}_1$, $\texttt{e}' \rightsquigarrow_* \texttt{l}_\texttt{k}\ \texttt{e}'_1$, and $\texttt{e}_1 \equiv \texttt{e}'_1$.

We also lift observational equivalence to lists of terms: two lists of equal length are observationally equivalent if their elements are component-wise observationally equivalent.

Example 3.7. The three functions $\lambda\,\texttt{i}.\ \texttt{i} + \texttt{0}$, $\lambda\,\texttt{i}.\ \texttt{0} + \texttt{i}$, and $\lambda\,\texttt{i}.\ \texttt{i}$ are observationally equivalent because they return the same result on any input. Similarly, $\lambda\,\texttt{b}.\ \texttt{b}\ \equiv\ \lambda\,\texttt{b}.\ \texttt{if b then true else false}$ as each function returns the same result on both true and false.

Definition 3.8 (State Projection Rule). Given circuits $c_1 : S_1 \to I \to (S_1, O)$ and $c_2 : S_2 \to I \to (S_2, O)$, and a function $p : S_1 \to S_2$, define

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

We say that c_1 *reduces* to c_2, written $c_1 \hookrightarrow c_2$, if $c_1p \equiv pc_2$, and call p the *state projection function* witnessing this reduction.

Example 3.9. In the adder example from § 2, we showed that add_obs ↪ leak_sim, i.e., that add_obs reduces to leak_sim using state projection function proj, by the observational equivalence between `c1p s i = let (s',o) = add_obs s i in (proj s', o)`, and `pc2 s i = leak_sim (proj s) i`. We wanted to show that the transducers of add_obs and leak_sim are equal, which confirms that `lift leak` is a suitable leakage for add under obs. In fact, circuit reduction implies transducer equivalence:

Theorem 3.10 (Circuit Reduction implies Mealy Equivalence). *Given two circuits $c_1 : S_1 \to I \to (S_1, O)$ and $c_2 : S_2 \to I \to (S_2, O)$, if $c_1 \hookrightarrow c_2$ with state projection function $p : S_1 \to S_2$, then for all initial states $s_1 : S_1$ there exists an initial state $s_2 : S_2$ such that for all input sequences is of type [I], we have $M(c_1, s_1, is) \equiv M(c_2, s_2, is)$.*

We prove Theorem 3.10 in § A.

3.3 Leakage Description

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

Definition 3.11 (Sequential Composition). Let $c_1 : S_1 \to I_1 \to (S_1, O_1)$ and $c_2 : S_2 \to O_1 \to (S_2, O_2)$ be circuits. Their *sequential composition*, denoted $c_1 \circ c_2$, is defined as

$$\begin{aligned}&c_1 \circ c_2 : (S_1, S_2) \to I_1 \to ((S_1, S_2), O_2)\\ &c_1 \circ c_2 \triangleq \lambda\,(\texttt{s}_1, \texttt{s}_2)\ \texttt{i}_1.\\ &\qquad \texttt{let}\ (\texttt{s}'_1, \texttt{o}_1)\ =\ c_1\ \texttt{s}_1\ \texttt{i}_1\ \texttt{in}\\ &\qquad \texttt{let}\ (\texttt{s}'_2, \texttt{o}_2)\ =\ c_2\ \texttt{s}_2\ \texttt{o}_1\ \texttt{in}\ ((\texttt{s}'_1, \texttt{s}'_2), \texttt{o}_2).\end{aligned}$$

For transducers, sequential composition of circuits becomes function composition (which follows by induction on the list of inputs):

Basic Types

r	$\in$	*Regs*	*registers*
v	$\in$	*Vars*	*variables*
i	$\in$	*Regs* $\cup$ *Vars*	*identifiers*
n	$\in$	$\{0,1\}^*$	*values*

Syntax

e	$:=$	$n \mid i \mid \ominus e \mid e_1 \otimes e_2 \mid \{e1, e2\}$	*expressions*
	$\mid$	$\textbf{if } e_1 \textbf{ th } e_2 \textbf{ el } e_3 \mid e_1[e_2 : e_3]$	
w	$:=$	$v = e$	*wires*
W	$:=$	$\{w_1, w_2, \ldots\}$	*wire sets*
a	$:=$	$r \leftarrow v$	*assignments*
A	$:=$	$\{a_1, a_2, \ldots\}$	*assignments sets*
I	$:=$	$\{v_1, v_2, \ldots\}$	*inputs*
O	$:=$	$\{v_1, v_2, \ldots\}$	*outputs*
C	$:=$	$I : W : A : O$	*circuits*

Figure 6: μ-Verilog syntax.

Lemma 3.12. *Given two circuits $c_1 : S_1 \to I_1 \to (S_1, O_1)$ and $c_2 : S_2 \to O_1 \to (S_2, O_2)$ be circuits, states $s_1 : S_1$ and $s_2 : S_2$, and an input sequence $is : [I]$, we have*

$$M(c_1 \circ c_2, (s_1, s_2), is) = M(c_2, s_2, M(c_1, s_1, is)).$$

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

Definition 3.13 (Leakage). Let $c : S \to I \to (S, O)$ be a circuit and $o : S_o \to O \to (S_o, O_o)$ an observation. A circuit $l : S_l \to I \to (S_l, O_l)$ is a *leakage description* for c under o if there exists a circuit $s : S_s \to O_l \to (S_s, O_o)$ such that $c \circ o \hookrightarrow l \circ s$.

Example 3.14. 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`.

Next, we adapt contract equivalence [41]—a prior notion of leakage—to our setting and show it's implied by our definition.

Definition 3.15 (Contract Equivalence). Let $c : S \to I \to (S, O)$ be a circuit and $o : S_o \to O \to (S_o, O_o)$ its observation. A circuit $l : S_l \to I \to (S_l, O_l)$ is called a *contract* for c with respect to the observation o if, for all states $s_l : S_l$ and all input sequences $is, is' : [I]$, whenever $M(l, s_l, is) = M(l, s_l, is')$, then for all states $s : S$ and $s_o : S_o$, we have $M(c \circ o, (s, s_o), is) = M(c \circ o, (s, s_o), is')$.

Theorem 3.16 (Leakage implies Contract Equivalence). *Let $c : S \to I \to (S, O)$ be a circuit and $o : S_o \to O \to (S_o, O_o)$ be its observation. If circuit $l : S_l \to I \to (S_l, O_l)$ is a leakage for c with respect to observation o, then l is also a contract.*

Theorem 3.16 is proved in § A.

3.4 From Circuits to Hardware

If its types are extractable, we can compile a circuit into a hardware design that matches the behavior of its associated transducer. We show this process with μ-Verilog, a simplified version of Verilog.[1]

μ-Verilog Syntax. Figure 6 shows μ-Verilog's syntax, based on [41]. μ-Verilog circuits consist of a set of input variables I, a set of assignments A, and a set of output wires O. Input variables are assigned an external input value at the start of each clock cycle. A wire assignment of the form $v = e$ immediately updates the value of variable v. An assignment of the form $r \leftarrow v$ updates the value of register r in the next clock cycle. We assume that assignments are disjoint—no two assignments update the same register or variable. Expressions are formed from bitvectors, register identifiers, unary and binary operations ($\ominus e$ and $e_1 \otimes e_2$), conditionals, the bit-selection operator $e_1[e_2 : e_3]$ (which selects a subset of bits from a value), and the wire concatenation operator $\{e_1, e_2\}$ (which forms a new wire consisting of both e_1 and e_2). Output wires specify the circuit's outputs.

[1]Verilog is one of the main hardware description languages; the other, VHDL, is similar.

μ-Verilog Semantics. § B describes the semantics. In short, wire assignments take immediate effect, while register assignments occur concurrently, synchronized by the clock. Circuits first assign inputs to input wires, evaluate all wire expressions, and compute the outputs. Finally, they update registers for the next clock cycle. The circuit semantics $(\!|C|\!)(\mu, i) = (\mu', o)$ captures this behavior: given a valuation μ that maps registers to values and a tuple of inputs i, it computes an output valuation μ' and a tuple of outputs o. Variable values don't need to be recorded, as they aren't preserved across cycles. The transducer semantics $M(C, \mu, is)$ is similarly defined.

Example 3.17. Consider the following μ-Verilog circuit

$$C_{\text{ex2}} = \{i\} : W : \{s \leftarrow s'\} : \{o\},$$

where $W = \{s' = s + 1, \quad o = \textbf{if } i[32:32] \textbf{ th } s + i[32:1] \textbf{ el } s\}$, $size(i) = 33$, and $size(s) = size(o) = 32$. C_{ex2} increments register s in each clock cycle. It then assigns to output register o: if i's most significant bit is 1, o is assigned $s + i[32:1]$ (i.,e., the sum of s and i's remaining 32 bits), otherwise it's assigned s.

From Circuits to μ-Verilog. This section sketches the translation from the core language (Figure 4) to Verilog; full translations rules appear in § C. A term λ `s i. e` of type $S \to I \to (S, O)$ (where types S, I, and O are extractable) is translated as follows, where v_i is an input variable, v_o an output variable, s is a state register, and $v_{\mathtt{e}}$ is the result of translating `e`.

$$\{v_i\} : \{v_o = \mathit{field}(v_{\mathtt{e}}, 2), s' = \mathit{field}(v_{\mathtt{e}}, 1), \ldots\} : \{s \leftarrow s'\} : \{v_o\}$$

The field extraction function $\mathit{field}(v, l)$ accesses the field l of a record encoded into variable v by appropriately slicing the variable. For example, for the record type $\{\mathtt{a} : \mathtt{Word8}, \mathtt{b} : \mathtt{Word4}\}$, $\mathit{field}(v, \mathtt{a}) = v[0:8]$.

The translation creates a variable $v_{\mathtt{e}'}$ for each sub-term `e'` of `e`. It encodes terms of record types by creating a variable with enough bits to fit all components of the record and using the bit-slice operator to extract individual component (via *field*). To encode terms of variant types, the translation creates a variable with enough bits to encode all possible data-types for the variant, as well as a label indicating which choice the term represents. It encodes case expressions of the form `case e of` $\overrightarrow{\mathtt{l_i x_i} \to \mathtt{e_i}}$ into conditional expressions, using bit-slicing to assign the relevant parts of expression `e` to variable $\mathtt{x_i}$ for each case.

Example 3.18. Applying the translation to the circuit `ex1` from Example 3.1 yields the μ–Verilog circuit C_{ex2} from Example 3.17. The optional type `Maybe Word32` is encoded into a 33-bit wire by conjoining a single bit to encode the choice between labels `Nothing` and `Just`, with 32 bits encoding the value of type `Word32`.

```
fe :: State -> (Word16, Maybe Word8) -> (State,())
fe state (rawInstr, jmp) =
  let curPC = pc state in
  let instr = decode rawInstr in
  case jmp of
    Just newPC -> (state {exInstr = Add 0,
                          exPC = curPC,
                          pc = newPC}, ())
    Nothing -> (state {exInstr = instr,
                       exPC = curPC,
                       pc = curPC + 1}, ())
```

Listing 4: Fetch Stage.

4 Case Study: Pipelined Processor

This section shows how to construct a simulation based-leakage proof for a simple three stage pipelined processor with a single register. While we use Haskell 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 | Beq 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 `Beq` branches if the register is `0`, adding its `Word8` offset to the current program counter.

Pipeline Stages. Each instruction passes through three processor stages. *Fetch* decodes a raw `Word16` instruction into an `Instr`. *Execute* runs the instruction, producing a register update (the *writeback*), an optional *output* value, and the next program counter. *Writeback* applies the register update and returns any outputs.

Toplevel Circuit and State. We encode the processor as a circuit `proc` (Listing 7) that takes a raw `Word16` instruction, and returns a pair: an optional `Maybe Word32` output and the new `Word8` program counter. The processor's state is encoded with the record type

```
data State = State
  {pc :: Word8, reg :: Word32,
  exInstr :: Instr, exPC :: Word8,
  wbOut :: (Maybe Word32, Maybe Word32)}
```

Here, `pc` is the current program counter and `reg` holds the register value. The other fields will be explained as we go.

Fetch Stage. The fetch stage (Listing 4) decodes the raw instruction and updates the program counter based on the previous instruction's execution. It stores the decoded instruction in the state's `exInstr` field and the current program counter (needed when executing a `Beq` instruction) in `exPC`; these become state inputs to execute stage in the next cycle. If a jump occured in the previous cycle (i.e., `jmp` is a `Just`-value), the instruction is discarded and is replaced with a no-op (`Add 0`). Otherwise, the program counter is incremented by 1.

Execute Stage. The execute stage (Listing 5) executes the instruction fetched last cycle, storing the result—a register update and the computed output—as `wbOut`. In the next cycle, the writeback stage will update the register and perform the processor's output according to the stored `wbOut` values. For an `Add imm` instruction, we add `imm` to the register value and store the update in `wbOut`. For `Out`, we store the register value in `wbOut`, to be output by the writeback stage. For `Beq off`, if the current register value is `0`, we jump by outputting `Just newPC` on a wire connected to the fetch stage.

```
ex :: State -> () -> (State, Maybe Word8)
ex state _ =
  let curReg = reg state in
  case exInstr state of
    Add imm -> let newReg = curReg + imm in
      (state {wbOut = (Just newReg, Nothing)},
        Nothing)
    Beq off ->
      if curReg == 0
      then let newPC = (exPC state) + off in
           (state {wbOut = (Nothing, Nothing)},
             Just newPC)
      else (state {wbOut = (Nothing, Nothing)},
             Nothing)
    ...
```

Listing 5: Execute Stage.

```
wb :: State -> () -> (State, Maybe Word32)
wb state _ =
  let (wb, out) = wbOut state in
  case wb of
    Just newReg -> (state {reg = newReg}, out)
    Nothing -> (state, out)
```

Listing 6: Writeback Stage.

```
proc :: State -> Word16
     -> (State, (Maybe Word32, Word8))
proc state rawInstr =
  let (state', out) = wb state () in
  let (state'', jmp) = ex state' () in
  let (state''', _) = fe state'' (rawInstr,jmp) in
  (state''', (out, pc state'''))
```

Listing 7: The Pipelined Processor.

Writeback Stage. The writeback stage (Listing 6) updates the register value in the processor state and returns the output.

Pipelining. In one clock cycle, `proc` executes all three pipeline stages concurrently, processing three instructions in parallel: it *fetches* the instruction at the current program counter, *executes* the instruction fetched last cycle, and performs the *writeback* for the instruction fetched two cycles ago. The pipeline executes stages in reverse order—writeback, then execute, then fetch—to prevent overwriting stage state inputs before they are used.

```haskell
leak :: LState -> Word16 -> (LState, LInstr)
leak state rawInstr =
  let curReg = reg state in
  let instr = decode rawInstr in
  case exInstr state of
    Add imm -> let newReg = curReg + imm in
      (state {reg = newReg,
              exInstr = instr}, LOther)
    Beq off ->
      if curReg == 0
      then (state {exInstr = Add 0}, LBeq off)
      else (state {exInstr = instr}, LOther)
    ...
```

Listing 8: Leakage.

4.2 Observation, Leakage, and Simulator

Observation. The observation function discards the processor output, only keeping the program counter:

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

This observation 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. To do so, for `Jmp` we need to know the target address, and for `Beq` the offset. For all other instructions, the program counter increases by 1. This suggests the following leakage (i.e., input for the simulator):

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

The leakage circuit (Listing 8) must produce a leakage of type `LInstr` from raw `Word16` instructions. To determine when to branch, `leak` must track 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}
```

For `Add imm`, we add `imm` to the register value, update the register, and decode the next instruction, storing it in `exInstr` for the next cycle. For `Beq off`, if the register value is 0, we jump: we discard the raw instruction, schedule a no-op (`Add 0`) for the next cycle, and—to allow the simulator to compute the address of the next instruction to fetch—record that a jump has occurred by leaking the offset as `LBeq off`.

Simulator. The simulator circuit must take `LInstr`s as inputs and produce the `Word8` program counter. The simulator state is the remainder of the original processor state (except for `wbOut`):

```haskell
data SState = SState {pc :: Word8, exPC :: Word8}
```

Listing 9 shows the code. On `LJmp addr` instructions, the next program counter is the specified address. For `LBeq off`, it's the original instruction address `exPC` plus the offset. Finally, on `LOther`, it's just the current counter incremented by 1.

State Projection. To show that the simulator matches the leakage, we need to prove that the circuit `proc` ∘ (`lift obs`) reduces to

```haskell
sim :: SState -> LInstr -> (SState, Word8)
sim state leakInstr =
 let curPC = pc state in
 case leakInstr of
   LNonJmp ->
     (state {exPC = curPC, pc = curPC + 1},
       curPC + 1)
   LJmp addr ->
     (state {exPC = curPC, pc = addr}, addr)
   LBeq off ->
     let newPC = (exPC state) + off in
     (state {exPC = curPC, pc = newPC}, newPC)
```

Listing 9: Simulator.

```haskell
proj : State -> (LState, SState)
proj (State pc reg exInstr exPC wbOut) =
  let (wb, _) = wbOut in
  case wb of
    Just newReg ->
      (LState newReg exInstr, SState pc exPC)
    Nothing ->
      (LState reg exInstr, SState pc exPC)
```

Listing 10: State Projection.

the circuit `leak` ∘ `sim`. To do so, we construct the state projection function shown in Listing 10. The state projection function divides the processor state into leakage and simulator states. The exception is `wbOut`, which lacks an analogue in the leakage and simulator states. Instead, we discard the output part of `wbOut` and perform an in-flight register update according to the writeback part of `wbOut`.

5 Implementation

PANTOMIME. PANTOMIME is a full-path symbolic execution engine for GHC Core—the internal language of GHC (the Glasgow Haskell Compiler), which is based on System F_C.[2] Implemented as a GHC Core plugin, PANTOMIME verifies that a circuit's leakage specification is correct. To use it, the user provides the leakage specification as an annotation, like the one shown below.

```haskell
{-# ANN circuit Pantomime
  { observation = 'obs, leakage    = 'leak,
    simulator   = 'sim, projection = 'proj } #-}
circuit :: s -> i -> (s, o)
circuit = ...
```

Given a specification, PANTOMIME generates constraints that entail its correctness. After constraint generation, PANTOMIME queries an SMT solver to check validity of the constraints. In case of a violation, PANTOMIME returns a pair of diverging observations along with their corresponding inputs. As PANTOMIME symbolically executes GHC Core, it supports all of System F_C. This allows hardware designs in PANTOMIME (and their corresponding leakage descriptions and simulators) to use the full range of functional programming techniques supported by Haskell—including monads, typeclasses

[2]System F_C [35] extends System F with non-syntactic type equality support.

and GADTs—provided they can be synthesized to hardware. Pantomime is written in approximately 7600 lines of Haskell code.

Grisette. We implemented Pantomime using the Grisette [26] symbolic execution library. Grisette provides an *Ordered Guards* representation, which allows for efficient merging of symbolic values. Additionally, Grisette folds constants and deduplicates expressions within terms where possible, further reducing constraint sizes.

In Grisette, a value of type `Maybe Word` is represented as

```
SymMaybeWord { tag = "tag" :: SymInt64
             , word = "word" :: SymWord64 }
```

where `"tag"` and `"word"` are *symbolic constants*.[3] Since `Maybe Word64` has only two variants (`Just` or `Nothing`), `SymMaybeWord`'s `tag` field needs to be restricted to be within range; Grisette allows us to encode this assumption directly in the expression:[4]

```
tag = If !(0 <= "tag" && "tag" < 2)
  (Left Invalid) (Right "tag")
```

Any value outside of the range of variants is `Invalid` and should not be considered by the constraint solver. More generally, a symbolic expression is represented by a chain of if-then-else expressions.

Merging Expressions. Instead of exposing the primitive constructor `If` directly, Grisette wraps symbolic branching within the `mrgIte` function. Directly using `If` hides opportunities for optimizations due to possible `Invalid` branches. We illustrate this in the following example, where we branch on conditional `condB` with both branches `x` and `y` containing `Invalid` in their body.

```
x = If condX (Left Invalid) (Right "x")
y = If condY (Left Invalid) (Right "y")

mrgIte condB x y
> If (condX || condY)
    (Left Invalid)
    (Right (If condB "x" "y"))
```

Notice that merging opens up the possibility to simplify `condX || condY`. In comparison, this reduction is non-obvious when using the naive branch operation `If x y`.

Constraint Generation. The constraint generation algorithm parallels concrete evaluation in GHC Core. In fact, Pantomime doubles as an interpreter when given constant inputs as Grisette will perform constant folding on expressions. The symbolic evaluator takes an environment and a Core expression as inputs and converts it into a symbolic expression. For example, a Core variable expression triggers a lookup in the environment, and a Core lambda converts into a function that receives a symbolic argument.

Pantomime follows Haskell's evaluation semantics. It supports type and coercion reductions and tracks typecasts to ensure correct and well-typed reductions. Abstract data types are implemented as records containing a tag—that symbolically represents the data constructor—along with symbolic versions of the fields for every possible data constructor. Pantomime supports abstract data types generically, such that it can accepts any new user definitions as is.

[3]`"tag"` and `"word"` are *not* strings and are indeed of type `SymInt64` and `SymWord64`; the `OverloadedStrings` GHC extension enables us to represent these symbolic expressions as strings.

[4]`If` constructs *symbolic* if-then-else expressions.

The symbolic valuation produced by Pantomime follows lazy semantics—expression constraints appear in the resulting expression only if the expression is used. This keeps constraint sizes small by including only what is strictly necessary.

CλaSH Support. We implemented AIMCore, in CλaSH [7], a hardware description language embedded in Haskell. CλaSH exposes hardware primitives, such as the sized bit vector type `BitVec n` and operations on them (e.g., bit vector concatenation), which it can compile to synthesisable HDLs, like Verilog. In Pantomime, we implemented interpretations for these hardware primitives and operations, enabling verification of specifications for CλaSH circuits. For example, we encode `BitVec n` as the appropriately sized bit vector in the corresponding first-order theory. This requires Pantomime to perform type-level reductions to statically determine vector bit-sizes. We take a similar approach for primitive operations.

Function Equivalence. Given an annotation, Pantomime constructs two circuits whose equivalence determines the validity of the leakage specification, illustrated below.

```
real s i =
  let (s', o) = circuit ∘ obs $ s i in
  (projection s', o)

synth s i = leakage ∘ simulator $ (projection s) i
```

The first circuit, `real`, composes `circuit` with `observation`, and then aplies `projection` to mask the output state. The second circuit, `synth` masks the input state with `projection` before passing it to `leakage` composed with `simulator`. The leakage specification is valid iff `real s i == synth s i` for all `s`, `i`—i.e., there is no pair of input state for which the circuits diverge.

6 Evaluation

We evaluate Pantomime by asking the following research questions.
RQ1: Can we use Pantomime to write/verify a RISC-V processor?
RQ2: What's the proof effort of verification with Pantomime?
RQ3: How does AIMCore compare to other verified processors?
RQ4: How do Pantomime's leakage descriptions and guarantees compare to other methods?
RQ5: How long does Pantomime take to verify correctness?

RQ1: Implementing AIMCore and its Proof. 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 [42].[5] 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 (load dependencies or a memory access blocking instruction fetch) 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

[5]Excluding the `ecall` and `ebreak` instructions.

is specified simply as a sequence of its individual stages, where `runCPUM` extracts the function payload from the `CPUM` monad[6] that the pipeline is written in.

```
proc :: State -> Input -> (State, Output)
proc = runCPUM (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:

```
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 [11].

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

Leakage Datatype. The actual leakage that `leak` computes is a tuple defined as:

```
type Leak = (LInstr, (Maybe RegId, Maybe RegId)
            , Maybe Address)
```

It consists of three components: (1) the leakage instruction `LInstr`:

```
data LInstr = LJmp | LLoad RegId | LStore | LOther
```

(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 beacause `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 (e.g., for load dependencies). 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`.

[6]Which is just a `newtype` wrapper around the `RWS` monad from the `mtl` package.

Simulator. The simulator `sim` takes as input the leakage `Leak` from `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: Proof Effort. Table 1 shows proof statistics for AIMCore and processors verified in LeaVe [41]. Pantomime proofs involve writing a simulator and state projection function, totalling around 200 lines of Haskell for AIMCore.[7] LeaVe proofs are invariant-based, consisting of assertions about the processor's state space and relations between variables the solver can't automatically find.

While writing 200 lines of simulator code might superficially seem like more work than writing a modest number of invaraints, recall that in Pantomime the simulator—and hence leakage proof—is a simplified version of the core and is therefore easy to write. Moreover, simulators are just programs: they can be interactively executed and benefit from a broad ecosystem of programming aids—from static type checking to property-based testing frameworks like QuickCheck [14] (which we used extensively during development). Debugging incorrect or missing invariants, however, is notoriously challenging and requires expertise beyond hardware design.

RQ3: 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.

RQ4: Leakage guarantees vs. other verified cores. Our leakage description for AIMCore leaks the class of instruction, the register dependencies, and the address of branch targets. As such, our leakage description verifies that both the constant-time discipline, as well as branch balancing (as long as it respects register dependencies) are effective against timing side-channels on this architecture. Table 1 summarizes the leakage descriptions verified by LeaVe. Notably, the leakage descriptions computed by LeaVe are less precise than that of Pantomime. For instance, every processor in Table 1 leaks the program counter value along with the executed

[7]The leakage circuit (which isn't part of the proof) is about 300 lines of Haskell and relies on an ISA interpreter that's a further 70 lines.

Table 1: AIMCore compared with the processors verified in LeaVe [41].

	AIMCore (Our work)	Processors verified by LeaVe [41] 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				
Leakage	Inst types, Reg deps, Jump Addr	PC, Inst, Br taken	PC, Inst	PC, Inst, Br taken, Jump Addr, Div Op
Proof effort	180 loc simulator; 30 loc projection	16 manual invariants	13 manual invariants	59 manual invariants
Verification Time	18.4 seconds	97.8 min	11.1 min	118.7 min
Unconditional proof	✓	✗	✗	✗

instruction; in our framework, this would result in a trivial simulator that directly outputs the leaked program counter. We conjecture that invariants restrict the class of leakages for which proofs are tractible. Invariant-based approaches also assume an immutable program, disallowing processors from modifying their own code (as done in e.g., boot loaders and JIT compilers); Pantomime does not have this restriction. Pantomime also enables us to group instructions that have the same leakage behavior using the `LOther` instruction. Finally, we note that LeaVe's proofs are conditional on functional correctness of the processor pipeline, which itself was not formally proven. By contrast, our proofs are unconditional.

RQ5: Verification Time. 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, we can see that Pantomime's verification time for AIMCore is two orders of magnitude lower than LeaVe's verification 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.

7 Related Work

Verifying Security Properties of RTL Designs. UPEC [18] 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 [16] and UPEC-DIT [15] 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 [17] 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 [47] extends Verilog with a type system to statically check timing-sensitive information flow. SpecVerilog [46] builds upon SecVerilog's type system to express information-flow safety under speculative execution. Unlike Pantomime, they are too restrictive to capture software defenses like the constant-time discipline.

Verifying Leakage Descriptions. Iodine [37] proves secret-independent timing in hardware, given usage assumptions expressed on inputs and internal wires. Xenon [38] synthesizes such usage assumptions semi-automatically. While their ability to represent arbitrary assumptions makes Iodine and Xenon expressive, they can only do so on internal wires, making translation to assumptions on software difficult. By contrast, Pantomime can express assumptions directly on the instruction stream of the program, as described in § 4 and 6. LeaVe [41] verifies leakage contracts (Definition 3.15) of RTL designs. However, it relies on expensive solvers, and requires hand-written invariants to carry out proofs. By contrast, Pantomime uses simulation-based proofs written in the same language as the processor, making them easier to write and fast to check. Contract Shadow Logic [36] 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 assume functional correctness of the design against an ISA-level specification to prove contract satisfaction. Pantomime makes no such assumption and therefore delivers unconditional proofs.

Synthesizing Leakage Descriptions. [29] synthesizes leakage contracts using a user-supplied lists of "contract atoms," which represent potential leakage sources, along with a set of test cases. However, [29] does not prove the correctness of the leakage contracts it synthesizes. RTL2MμPATH [20] 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. Their notion of leakage as dependencies between instructions makes it hard to translate assumptions back to software, limiting generality.

Simulation-based Proofs in Crypto. In designing our approach, we were inspired by simulation-based proofs in cryptography [24] (as used e.g., in Universal Composability [10]). 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.

8 Conclusion

This paper introduces simulation-based leakage proofs, an approach to leakage verification that integrates directly into the hardware design process, enabling rapid, interactive verification with unconditional leakage guarantees. We realized this approach in the PANTOMIME verification tool and used it to verify the AIMCORE RISC-V CPU. Unlike previous work, where proofs depend on the functional correctness of a CPU against an ISA spec, our proofs are unconditional.

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] 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.

[7] 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.

[8] 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

[9] 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

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

[11] 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

[12] 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

[13] 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*.

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] Mike Gordon. 1995. The Semantic Challenge of Verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95)*. IEEE Computer Society, USA, 136.

[20] 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

[21] 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.

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

[23] 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

[24] 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.

[25] 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.

[26] 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

[27] 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

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

[29] 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

[30] 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.

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

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

[33] 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

[34] 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

[35] 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.

[36] 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

[37] 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

[38] 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

[39] 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)*.

[40] 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.

[41] 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

[42] 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.

[43] 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

[44] 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

[45] 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.

[46] 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

[47] 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

A Proofs for Section 3

Proof of Theorem 3.10. Let s be a state of type S and $\texttt{is}$ be an input sequence of Is. We proceed by induction on $\texttt{is}$.

- **Base Case:** $\texttt{is} = []$. By definition.
- **Induction hypothesis:** $M(c_1, s, \texttt{is}') = M(c_2, p\ s, \texttt{is}')$.
- **Inductive Step:** $\texttt{is} = i : \texttt{is}'$. We show the equivalence between $M(c_1, s, \texttt{is})$ and $M(c_2, p\ s, \texttt{is})$ as follows:

$$\begin{aligned}
&M(c_1, s, \texttt{is})\\
&\equiv^{(1)}\ o : M(c_1, s', \texttt{is}')\\
&\equiv^{(2)}\ o : M(c_2, p\ s', \texttt{is}')\\
&\equiv^{(3)}\ o : M(c_2, [\![c_1 p\ s\ i]\!].1, \texttt{is}')\\
&\equiv^{(4)}\ o : M(c_2, [\![p c_2\ s\ i]\!].1, \texttt{is}')\\
&\equiv^{(5)}\ o : M(c_2, (c_2\ (p\ s)\ i).1, \texttt{is}')\\
&\equiv^{(6)}\ M(c_2, p\ s, \texttt{is}).
\end{aligned}$$

Step (1) follows by definition of M, where $(s', o) = [\![c_1\ s\ i]\!]$; step (2) from our induction hypothesis; and step (3) from the definition of $c_1 p$ in the reduction proof, where 1 is the label for the first element of a tuple. Step (4) follows from the observational equivalence between $c_1 p$ and $p c_2$, which we get from the assumption $c_1 \hookrightarrow c_2$. Step (5) follows from the definition of $p c_2$; and step (6) from the definition of M. □

Proof of Theorem 3.16. Let $\texttt{sim} : S_s \to O_l \to (S_s, O_o)$ be the simulator, with respect to which we have the mealy reduction $c \circ o \hookrightarrow \texttt{leak} \circ \texttt{sim}$. Given an initial state $(s, s_o) : (S, O)$ of the circuit $c \circ o$, from Theorem 3.10 we get an initial state $(s_l, s_s) : (S_l, S_s)$ of the circuit $\texttt{leak} \circ \texttt{sim}$ such that

$$M(c \circ o, (s, s_o), is) = M(\texttt{leak} \circ \texttt{sim}, (s_l, s_s), is) \qquad (\star)$$

for all input sequences $\texttt{is} : [I]$. Fix input sequences $\texttt{is}, \texttt{is'} : [I]$. Then we have

$$\begin{aligned}
&M(c \circ o, (s, s_o), \texttt{is})\\
&=^{(1)}\ M(\texttt{leak} \circ \texttt{sim}, (s_l, s_s), \texttt{is})\\
&=^{(2)}\ M(\texttt{sim}, s_s, M(\texttt{leak}, s_l, \texttt{is}))\\
&=^{(3)}\ M(\texttt{sim}, s_s, M(\texttt{leak}, s_l, \texttt{is'}))\\
&=^{(4)}\ M(\texttt{leak} \circ \texttt{sim}, (s_l, s_s), \texttt{is'})\\
&=^{(5)}\ M(c \circ o, (s, s_o), \texttt{is'})
\end{aligned}$$

Steps (1) and (5) follow from $(\star)$, steps (2) and (4) from Lemma 3.12, and step (3) from the assumption. □

B Verilog Semantics

We now sketch the semantics of μ-Verilog. For a more thorough description of Verilog's semantics, see [19, 37, 41, 47]. The circuit state consists of a pair (ρ, μ), where ρ maps variables to values, and μ maps registers to values. Values are fixed length bit-strings, where we write $size(v)$ to denote the number of bits in v. At the beginning of each clock cycle, we update ρ with the valuation of the inputs in the current cycle. That is, if the value for input i is value b in the current cycle, we set $\rho(i) = b$. Next, we execute wire assignments updating ρ accordingly, until we reach a fix-point, i.e., until re-executing any wires no longer changes ρ. For this, we make use of an evaluation function $(\!|\cdot|\!)^{\rho,\mu}$, where $(\!|e|\!)^{\rho,\mu}$ denotes the result of evaluating expression e under wire valuation ρ, and registers valuation μ. Finally, we pass the values of output wires on as outputs of the circuit. After assigning variables, we perform updates to registers stemming from assignments of the form $r \longleftarrow v$.

For this, we construct μ', the register valuation in the next clock cycle by setting $\mu' = \mu[r \leftarrow \rho(v)]$, for each $r \leftarrow v \in A$, where we write $f[x \leftarrow v]$ to denote the function that acts like f everywhere, except for x, where it returns v. Note that since we computed ρ using μ thereby using the values of registers in the last clock-cycle, effectively all register updates are performed at once. This is crucial, and reflects Verilog's synchronous semantics.

C Translation from STLC to Verilog

We now describe a translation from STLC circuits to μ-Verilog programs. We first change the names of the variables in the circuit to make sure each lambda abstraction and case expression variable has a unique name. We exhaustively apply β-reductions and η-expansions. β-reduction is defined in the evaluation rules of the core language. η-expansion expands function terms, that is, for each function term f that is not a lambda abstraction and does not occur in the function position of an application, we replace it with the abstraction $\lambda x.\ f\ x$, where x is fresh. Exhaustively applying these two rules ensures that there are no partially applied functions left, and all remaining applied functions are primitive functions that can be represented in μ-Verilog.

Representing Records and Variants. Our translation needs to encode record and variant types into variables. For records, it encodes the whole record into a variable of appropriate size, and then uses the bit-slicing operator to extract the parts. For variants, it encodes a representation of the label together with the actual data. We now define utility functions that help with this encoding.

Representation Size. The function $\mathit{sizeof}(t)$ computes the number of bits required to represent a type t in μ-Verilog. A base-type of size $\mathtt{Word}n$ requires n bits. For a record type $\mathtt{t} = \{\mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n\}$, we require the sum of the sizes of the record types, i.e., $\mathit{sizeof}(\mathtt{t}) = \sum_i \mathit{sizeof}(t_i)$. For a variant type $\mathtt{t} = \langle \mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n \rangle$, the size is defined as $\mathit{sizeof}(\mathtt{t}) = \max_i(\mathit{sizeof}(\mathtt{t}_i)) + \lceil \log_2(n) \rceil$, to account for both the size of the largest field and the size of the label.

Field Extraction. Field extraction function $\mathit{field}(v, l)$ lets us access field l of a record type encoded into variable v. For a record of type $\{\mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n\}$, the expression $\mathit{field}(v, \mathtt{l}_m)$ is defined as $v[\sum_{i=0}^{m-1} \mathit{sizeof}(\mathtt{t}_i) : \sum_{i=0}^{m} \mathit{sizeof}(\mathtt{t}_i)]$, meaning the field is accessed by slicing the appropriate bit range based on the total size of the preceding fields.

Label Encoding. The function $\mathit{tolabel}(l_i)$ converts the label l_i of a variant type $\mathtt{t} = \langle \mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n \rangle$ to a μ-Verilog bit-vector of size $\lceil \log_2(n) \rceil$ representing the integer i, which is a tag for the variant case corresponding to l_i. For example, for the variant type $\langle \mathtt{Just} : \mathtt{Word8}, \mathtt{Nothing} : () \rangle$, $\mathit{tolabel}(\mathtt{Just}) = 0$ and $\mathit{tolabel}(\mathtt{Nothing}) = 1$.

Label Extraction. Similar to field, the label extraction function $\mathit{label}(v, l)$ lets us access the label l of a record type encoded into variable v. For a record of type $\{\mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n\}$, the expression $\mathit{label}(v, \mathtt{l}_m)$ is defined as $v[\sum_{i=0}^{m} \mathit{sizeof}(\mathtt{t}_i) : \sum_{i=0}^{m} \mathit{sizeof}(\mathtt{t}_i) + \lceil \log_2(n) \rceil]$.

Casting. The function $\mathit{cast}(v, l)$ casts a variable by assuming its size has label ls size where l is a label of a variant type $\langle \mathtt{l}_1 : \mathtt{t}_1, \ldots, \mathtt{l}_n : \mathtt{t}_n \rangle$. For example, for the variant type $\langle \mathtt{First} : \mathtt{Word8}, \mathtt{Second} : \mathtt{Word9} \rangle$, $\mathit{cast}(v, \mathtt{First}) = v[0 : 8]$ and $\mathit{cast}(v, \mathtt{Second}) = v[0 : 9]$. We use the casting function while transforming a case-expression. The translated μ-Verilog circuit includes a wire for each $\mathtt{e}_\mathtt{i}$ in $\mathtt{case}\ \mathtt{e}_\mathtt{c}\ \mathtt{of}\ \overrightarrow{\mathtt{l}_\mathtt{i}\mathtt{x}_\mathtt{i} \rightarrow \mathtt{e}_\mathtt{i}}$.

Mapping STLC Operators to μ-Verilog Operators. We assume there exist corresponding unary and binary μ-Verilog operators for each unary and binary STLC operator. For each unary STLC operator uop, we represent the corresponding μ-Verilog operator as $\ominus_{\mathsf{uop}}$. Similarly, for each binary STLC operator bop, we represent the corresponding μ-Verilog operator as $\otimes_{\mathsf{bop}}$. Note that in STLC, there are no built-in operators. Operators are represented as primitive functions.

Translation Function. We translate an STLC circuit $\lambda \mathtt{s}\ \mathtt{i}.\ \mathtt{e}$ into the μ-Verilog circuit

$$C = \{v_\mathtt{i}\} : T(\mathtt{e}, V_0) \cup \{o = \mathit{field}(v_\mathtt{e}, 2), s' = \mathit{field}(v_\mathtt{e}, 1)\} : \{s \leftarrow s'\} : \{o\}\ .$$

The initial valuation is $V_0 = \{s \mapsto v_s,\ i \mapsto v_i\}$. The translation function T, defined in Figure 7, recursively translates each STLC subexpression into a corresponding μ-Verilog wire. It takes a valuation V as an additional argument, which maps μ-Verilog variables and registers to μ-Verilog expressions. This valuation tracks bindings introduced by case expressions and bookkeeps assumptions made during translation.

For example, when translating the expression

$$\begin{aligned} \mathtt{e} = \mathtt{case}\ &\mathtt{e}_\mathtt{c}\ \mathtt{of} \\ &\mathtt{Just}\ \mathtt{x} \longrightarrow \mathtt{e}_\mathtt{j} \\ &\mathtt{Nothing} \longrightarrow \mathtt{e}_\mathtt{n} \end{aligned}$$

we must translate $\mathtt{e}_\mathtt{j}$ under the assumption that x refers to a Just-tagged value carried by $\mathtt{e}_\mathtt{c}$. If Just is of type Word4, we extend the valuation as $V' = V[x \mapsto v_{\mathtt{e}_\mathtt{c}}[0 : 4]]$ while evaluating $T(\mathtt{e}_\mathtt{j}, V')$.

Example C.1. We now demonstrate our STLC to μ-Verilog translation with a complete example. Consider the STLC circuit

$$\begin{aligned} &\mathtt{e} : \mathtt{Word8} \longrightarrow \mathtt{Maybe\ Word8} \longrightarrow (\mathtt{Word8}, \mathtt{Word8}) \\ &\mathtt{e} = \lambda\, s\ i.\ (\lambda\, f.\ (f s, f s)) \\ &\qquad (\mathtt{case}\ \mathtt{i}\ \mathtt{of} \\ &\qquad\quad \mathtt{Just}\ \mathtt{x} \longrightarrow (-)\ \mathtt{x} \\ &\qquad\quad \mathtt{Nothing} \longrightarrow (+)\ 1) \end{aligned}$$

To translate e to a μ-Verilog circuit, we first exhaustively apply η-expansions and β reductions to get

$$\begin{aligned} &\mathtt{e}' : \mathtt{Word8} \longrightarrow \mathtt{Maybe\ Word8} \longrightarrow (\mathtt{Word8}, \mathtt{Word8}) \\ &\mathtt{e}' = \lambda\, \mathtt{s}\ \mathtt{i}. \\ &\qquad (\mathtt{case}\ \mathtt{i}\ \mathtt{of} \\ &\qquad\quad \mathtt{Just}\ \mathtt{x} \longrightarrow \mathtt{x} - \mathtt{s} \\ &\qquad\quad \mathtt{Nothing} \longrightarrow 1 + \mathtt{s}, \\ &\qquad \mathtt{case}\ \mathtt{i}\ \mathtt{of} \\ &\qquad\quad \mathtt{Just}\ \mathtt{x} \longrightarrow \mathtt{x} - \mathtt{s} \\ &\qquad\quad \mathtt{Nothing} \longrightarrow 1 + \mathtt{s}) \end{aligned}$$

$$
\begin{array}{ll}
T(\mathsf{x}, V) & = \{v_{\mathsf{x}} = V(\mathsf{x})\} \\
T(\mathsf{c}, V) & = \{v_{\mathsf{c}} = \mathsf{c}\} \\
T(\mathsf{e}_1 \ \mathsf{bop} \ \mathsf{e}_2, V) & = \{v_{\mathsf{e}} = v_{\mathsf{e}_1} \otimes_{\mathsf{bop}} v_{\mathsf{e}_2}\} \cup T(\mathsf{e}_1, V) \cup T(\mathsf{e}_2, V) \\
T(\mathsf{uop} \ \mathsf{e}_1, V) & = \{v_{\mathsf{e}} = \ominus_{\mathsf{uop}} v_{\mathsf{e}_1}\} \cup T(\mathsf{e}_1, V) \\
T(\mathsf{e}_1.\mathtt{l}, V) & = \{v_{\mathsf{e}} = \mathit{field}(v_{\mathsf{e}_1}, \mathtt{l})\} \cup T(\mathsf{e}_1, V) \\
T(\{\overrightarrow{\mathtt{l_i} = \mathsf{e_i}}\}, V) & = \{v_{\mathsf{e}} = \{v_{\mathsf{e}_1}, \ldots, v_{\mathsf{e}_n}\}\} \cup \bigcup_i T(\mathsf{e_i}, V) \\
T(\mathtt{l} \ \mathsf{e}, V) & = \{v_{\mathsf{e}} = \{\mathit{tolabel}(\mathtt{l}), v_{\mathsf{e}}\}\} \cup T(\mathsf{e}, V) \\
T(\mathtt{case} \ \mathsf{e_c} \ \mathtt{of} \ \overrightarrow{\mathtt{l_i x_i} \to \mathsf{e_i}}, V) & = T(\mathsf{e_c}, V) \cup \bigcup_i T(\mathsf{e_i}, V[\mathtt{x_i} \mapsto \mathit{cast}(\mathsf{e_c}, \mathtt{l_i})]) \\
& \quad \cup\{v_{\mathsf{e}} = (\textbf{if } \mathit{label}(v_{\mathsf{e_c}}) == 0 \textbf{ th } v_{\mathsf{e}_0} \textbf{ el} \\
& \qquad\quad \textbf{if } \mathit{label}(v_{\mathsf{e_c}}) == 1 \textbf{ th } v_{\mathsf{e}_1} \textbf{ el } \ldots \\
& \qquad\qquad \textbf{if } \mathit{label}(v_{\mathsf{e_c}}) == n-1 \textbf{ th } v_{\mathsf{e}_{n-1}} \textbf{ el } v_{\mathsf{e}_n})\}
\end{array}
$$

Figure 7: The $T(\mathsf{e}, V)$ transformation from Circuits to μ-Verilog.

We then apply the wire calculation function on the body of the circuit expression e' as

$$
\begin{array}{rcl}
T(\mathsf{e}_{\mathsf{body}}, V_0) & = & T(\mathsf{e}_{\mathsf{case}}, V_0) \cup \{v_{\mathsf{e}'} = \{v_{case}, v_{case}\}\} \\
& = & T(\mathsf{x} - \mathsf{s}, V_0[\mathsf{x} \mapsto v_i[0:8]]) \cup T(1 + \mathsf{s}, V_0) \\
& & \quad \cup\{v_{\mathsf{case}} = \textbf{if } v_i[8:9] == 0 \textbf{ th } v_{\mathsf{x-s}} \textbf{ el } v_{1+\mathsf{s}}\} \\
& = & T(x, V_0[x \mapsto v_i[0:8]]) \cup \{v_{\mathsf{x-s}} = v_{\mathsf{x}} - v_{\mathsf{s}}\} \\
& & \quad \cup\{v_{1-\mathsf{s}} = v_1 - v_{\mathsf{s}}\} \cup T(1, V_0[\mathsf{x} \mapsto v_i[0:8]]) \\
& & \quad \cup\{v_{\mathsf{case}} = \textbf{if } v_i[8:9] == 0 \textbf{ th } v_{\mathsf{x-s}} \textbf{ el } v_{1+\mathsf{s}}\} \\
& = & \{v_{\mathsf{x}} = v_i[0:8]\} \cup \{v_{\mathsf{x-s}} = v_x - v_s\} \\
& & \quad \cup\{v_{1-\mathsf{s}} = v_1 - v_s\} \cup \{v_1 = 1\} \\
& & \quad \cup\{v_{\mathsf{case}} = \textbf{if } v_i[8:9] == 0 \textbf{ th } v_{\mathsf{x-s}} \textbf{ el } v_{1+\mathsf{s}}\} \\
& = & \{v_x = v_i[0:8], v_{\mathsf{x-s}} = v_x - v_s, v_{1-\mathsf{s}} = v_1 - v_s, \\
& & \quad v_1 = 1, v_{\mathsf{case}} = \textbf{if } v_i[8:9] == 0 \textbf{ th } v_{\mathsf{x-s}} \textbf{ el } v_{1+\mathsf{s}}\}.
\end{array}
$$

Which finally gives us the circuit

$$C = \{v_i\} : T(e, V_0) \cup \{o = \mathit{field}(v_{body}, 2), s' = \mathit{field}(v_{body}, 1)\} : \{s \leftarrow s'\} : \{o\}\,.$$