IODINE: Verifying Constant-Time Execution of Hardware

Klaus v. Gleissenthall
University of California, San Diego

Rami Gökhan Kıcı
University of California, San Diego

Deian Stefan
University of California, San Diego

Ranjit Jhala
University of California, San Diego

Abstract. To be secure, cryptographic algorithms crucially rely on the underlying hardware to avoid inadvertent leakage of secrets through timing side channels. Unfortunately, such timing channels are ubiquitous in modern hardware, due to its labyrinthine fast-paths and optimizations. A promising way to avoid timing vulnerabilities is to devise—and verify—conditions under which a hardware design is free of timing variability, *i.e.*, executes in *constant-time*. In this paper, we present IODINE: a clock-precise, constant-time approach to eliminating timing side channels in hardware. IODINE succeeds in verifying various open source hardware designs in seconds and with little developer effort. IODINE also discovered two constant-time violations: one in a floating-point unit and another one in an RSA encryption module.

1 Introduction

Trust in software systems is always rooted in the underlying hardware. This trust is apparent when using hardware security features like enclaves (*e.g.*, SGX and TrustZone), crypto units (*e.g.*, AES-NI and the TPM), or MMUs. But our trust goes deeper. Even for simple `ADD` or `MUL` instructions, we expect the processor to avoid leaking any of the operands via *timing side channels*, *e.g.*, by varying the execution time of the operation according to the data. Indeed, even algorithms specifically designed to be resilient to such timing side-channel attacks crucially rely on these assumptions [23–25]. Alas, recently discovered vulnerabilities have shown that the labyrinthine fast-paths and optimizations ubiquitous in modern hardware expose a plethora of side channels that undermine many of our deeply held beliefs [34, 36, 42].

A promising way to ensure that trust in hardware is properly earned is to formally specify our expectations, and then, to *verify*—through mathematical proof—that the units used in security critical contexts do not exhibit any timing variability, *i.e.*, are *constant-time*. For instance, by verifying that certain parts of an arithmetic logic unit (ALU) are constant-time, we can provide a foundation for implementing secure crypto algorithms in software [16, 20, 22]. Dually, if timing variability is unavoidable, *e.g.*, in SIMD or floating-point units, making this variability *explicit* can better inform mechanisms that attempt to mitigate timing channels at the software level [18, 46, 54] in order to avoid vulnerabilities due to gaps in the hardware-software contract [17, 18].

In this paper, we introduce IODINE: a clock-precise, constant-time approach to eliminating timing side channels in hardware. Given a hardware circuit described in Verilog, a *specification* comprising a set of sources and sinks (*e.g.*, an FPU pipeline start and end) and a set of usage assumptions (*e.g.*, no division is performed), IODINE allows developers to automatically synthesize *proofs* which ensure that the hardware runs in constant-time, *i.e.*, under the given usage assumptions, the time taken to flow from source to sink, is independent of operands, processor flags and interference by concurrent computations.

Using IODINE, a crypto hardware designer can be certain that their encryption core does not leak secret keys or messages by taking a different number of cycles depending on the secret values. Similarly, a CPU designer can guarantee that programs (*e.g.*, cryptographic algorithms, SVG filters) will run in constant-time when properly structured (*e.g.*, when they do not branch or access memory depending on secrets [20]).

IODINE is *clock-precise* in that it enforces constant-time execution directly as a semantic property of the circuit rather than through indirect means like information flow control [55]. As a result, IODINE neither requires the constant-time property to hold unconditionally nor

demands the circuit be partitioned between different security levels (*e.g.*, as in SecVerilog [55]). This makes IODINE particularly suited for verifying existing hardware designs. For example, we envision IODINE to be useful in verifying ARM's recent set of *data independent timing (DIT)* instructions which should execute in constant-time, if the PSTATE.DIT processor state flag is set [2, 41].

While there have been significant strides in verifying the constant-time execution of software [14–16, 18, 20–22, 53], IODINE unfortunately cannot directly reuse these efforts. Constant time methods for software focus on straight-line, sequential—often cryptographic—code.

Hardware designs, however, are inherently *concurrent* and *long-lived*: circuits can be viewed as collections of processes that run forever, performing parallel computations that update registers and memory in every clock cycle. As a result, in hardware, even the definition of constant-time execution becomes problematic: how can we measure the timing of a hardware design that never stops and performs multiple concurrent computations that mutually influence each other?

In IODINE, we address these challenges through the following contributions.

1. Definition. First, we define a notion of constant-time execution for concurrent, long-lived computations. In order to reason about the timing of values flowing between sources and sinks, we introduce the notion of *influence set*. The influence set of a value contains all cycles t, such that an input (*i.e.*, a source value) at t was used in its computation. We say that a hardware design is constant time, if all its computation paths (that satisfy usage assumptions) produce the same sequence of influence sets for sinks.

2. Verification. To enable its efficient verification, we show how to reduce the problem of checking constant-time execution—as defined through influence sets—to the standard problem of checking assertion validity. For this, we first eschew the complexity of reasoning about several concurrent computations at once, by focusing on a *single* computation starting (*i.e.*, inputs issued) at some cycle t. We say that a value is *live* for cycle t (t-live), if it was influenced by the computation started at t, *i.e.*, t is in the value's influence set. This allows us to reduce the problem of checking equality of influence sets, to checking the equivalence of membership, for their elements. We say that a hardware design is *liveness equivalent*, if, for any two executions (that satisfy usage assumptions), and any t, t-live values are assigned to sinks in the same way, *i.e.*, whenever a t-live value is assigned to a sink in one execution, a t-live value must also be assigned to a sink in the other.

To check a hardware design for liveness equivalence, we *mark* source data as live in some *arbitrarily chosen* start cycle t, and track the flow of t-live values through the circuit using a simple standard taint tracking monitor [44]; the problem of checking liveness equivalence then reduces to checking a simple assertion stating that sinks are always tainted in the same way. Reducing constant-time execution to the standard problem of checking assertion validity allows us to rely on off-the-shelf, mature verification technology, which explains IODINE's effectiveness.

3. Evaluation. Our final contribution is an implementation and evaluation of IODINE on seven open source VERILOG projects—CPU cores, an ALU, crypto-cores, and floating-point units (FPUs). We find that IODINE succeeds in verifying different kinds of hardware designs in a matter of seconds, with modest developer effort (§ 6). Many of our benchmarks are constant-time for intricate reasons (§ 6.3), *e.g.*, whether or not a circuit is constant-time depends on its execution history, circuits are constant-time despite triggering different control flow paths depending on secrets, and require a carefully chosen set of assumptions to be shown constant-time. In our experience, these characteristics—combined with the circuit size—make determining whether a hardware design is constant-time by code inspection near impossible.

IODINE also revealed two constant-time violations: one in the division unit of an FPU designs, another in the modular exponentiation module of an RSA encryption module. The second violation—a classical timing side channel—can be abused to leak secret keys [27, 35].

In summary, this paper makes the following contributions.

- First, we give a definition for constant-time execution of hardware, based on the notion of *influence sets* (§ 2). We formalize the semantics of VERILOG programs with influence sets (§ 3), and use this formalization to define constant-time execution with respect to usage assumptions (§ 4).
- Our second contribution is a reduction of constant-time execution to the easy-to-verify problem of liveness equivalence. We formalize this property (§ 4), prove its equivalence to our original notion of constant-time execution (§ 4.3), and show how to verify it using standard methods (§ 5).

```
1  // source(x); source(y); sink(out);
2  // assume(ct = 1);
3
4  reg flp_res, x, y, ct, out, out_ready, ...;
5  wire iszero, isNaN, ...;
6
7  assign iszero = (x == 0) || (y == 0);
8
9  always @(posedge clk) begin
10   ...
11   flp_res <= ... // (2) compute x * y
12 end
13
14 always @(posedge clk) begin
15   if (ct)
16     ...; out <= flp_res; // (4)
17   else
18     if (iszero)
19       out <= 0; // (1)
20     else if (isNaN)
21       ...
22     else out <= flp_res; // (3)
23     end
24   end
25 end
```

Figure 1: Floating point multiplier (EX1).

- Our final contribution is an implementation and evaluation of IODINE on several challenging open source hardware designs (§ 6). Our evaluation shows that IODINE can be used to verify constant-time execution of existing hardware designs, rapidly, and with modest user effort.

2 Overview

In this section, we give an overview of IODINE and show how our tool can be used to verify that a piece of VERILOG code executes in constant-time. As a running example, we consider a simple implementation of a floating-point multiplication unit.

Floating Point Multiplier. Our running example, like most FPUs, is generally *not* constant-time—common operations (*e.g.*, multiplication by zero) are dramatically faster than rare ones (*e.g.*, multiplication by denormal numbers [17,36]). But, like the ARM's recent support for data independent timing instructions, our FPU contains a processor flag that can be set to ensure that all multiplications are constant-time, at the cost of performance. Fig. 1 gives a simplified fragment of VERILOG code that implements this FPU multiplier. While our benchmarks consist of hundreds of threads with shared variables, pipelining, and contain a myriad of branches and flags which cause dependencies on the execution history (see § 6.3), we have kept our running example as simple as possible: our multiplier takes two floating-point values—input registers x and y—and stores the computation result in output register out. Recall that VERILOG programs operate on two kinds of data-structures: *registers* which are assigned in always-blocks and store values across clock cycles and *wires* which are assigned in assign-blocks and hold values only within a cycle. Control register ct is used to configure the FPU to run in constant-time (or not). For simplicity, we omit most other control logic (*e.g.*, reset or output-ready bits and processing of inputs). Internally, the multiplier consists of several *fast* paths and a single *slow* path. For example, to implement multiplication by zero, one, NAN, and other special values we, inspect the input registers for these values and produce a result in a single cycle (see (1)). Multiplication by other numbers is more complex, however, and generally takes more than a single cycle. As shown in Fig. 1, this *slow* path consists of multiple intermediate steps, the final result of which is assigned to a temporary register flp_res (see (2)) before out (see (3)).[1] Importantly, when the constant-time configuration register ct is set, only this slow path is taken (see (4)).

Outline. In the rest of this section, we show how IODINE verifies that this hardware design runs in constant-time when the ct flag is set and violates the constant-time property otherwise. We present our definition of constant-time based on of influence sets in § 2.1, liveness equivalence in § 2.2, and finally show how IODINE formally verifies liveness equivalence by reducing it to a simple safety property that can be handled by standard verification methods § 2.3.

2.1 Constant-Time For Hardware

We start by defining *constant-time* execution for hardware.

Assumptions and Attacker Model. Like SecVerilog [55], we scope our work to synchronous circuits with a single, fixed-rate clock. We further assume an *external attacker* that can measure the execution time of a piece of hardware (given as influence sets) using a cycle-precise timer. In particular, an attacker can observe the timing of *all* inputs that influenced a given output. These assumptions afford us many benefits. (Though, as we describe in § 7, they are not for free.) For example, assuming a single

[1] For simplicity, we omit the intermediate steps and assume that they implement floating-point multiplication in constant-time. In practice, FPUs may also take different amounts of time depending on such values.

P	::=		**Program**
		$\lfloor s \rfloor_{id}$	*process*
		P ‖ P	*parallel composition*
		repeat P	*sync. iteration*
		›	*empty process*
s	::=		**Command**
		skip	*no-op*
		$v = e$	*blocking*
		$v \Leftarrow e$	*non-blocking*
		$v := e$	*continuous*
		ite(e, s, s)	*conditional*
		$s_1 ; \ldots ; s_k$	*sequence*
		a	*annotation*
e	::=		**Expression**
		v	*variables*
		n	*constants*
		$f(e_1, \ldots, e_k)$	*function literal*

Figure 2: Syntax for intermediate language VINTER.

```
    repeat  [iszero := (x == 0 || y == 0)]
||  repeat      [... ; flp_res ⇐ ...]
||  repeat  ⎡ ite(ct,                  ⎤
            ⎢    out ⇐ flp_res,        ⎥
            ⎢    ite(iszero,           ⎥
            ⎢       out ⇐ 0,           ⎥
            ⎣       out ⇐ flp_res))    ⎦
```

Figure 3: EX1 written in VINTER

fixed-rate clock, allows us to translate VERILOG programs, such as our FPU multiplier to a more concise representation shown in Figure 3.

Intermediate Language. In this language—called VINTER—VERILOG always- and assign-blocks are represented as concurrent *processes*, wrapped inside an infinite repeat-loop. As Fig. 2 shows, each process sequentially executes a series of VERILOG-like statements. (Each process also has a unique identifier $id \in PIDs$, which we sometimes omit, for brevity.) Most of these are standard; we only note that VINTER—like VERILOG—supports three types of assignment statements: blocking ($v = e$), non-blocking ($v \Leftarrow e$) and continuous ($v := e$). Blocking assignments take effect immediately, within the current cycle; non-blocking assignments are deferred until the next cycle. Finally, continuous assignments enforce directed equalities between registers or wires: whenever the right-hand side of an equality is changed, the left-hand side is updated by re-running the assignment. Note that VINTER focuses only on the synthesizable fragment of VERILOG, *i.e.*, does not model delays, etc., which are only relevant for simulation.

VINTER processes are composed in parallel using the (‖) operator. Unlike concurrent software processes, they are, however, synchronized using a single (implicit) fixed-rate clock: each process waits for all other (parallel) processes to finish executing before moving on to the next iteration, *i.e.*, next clock cycle. Moreover, unlike software, these programs are usually data-race free, in order to be synthesizable to hardware.

VINTER processes run forever; they perform computations and update registers (*e.g.*, out in our multiplier) on every clock cycle. For example, pipelined hardware units execute multiple, different computations simultaneously.

From Software to Hardware. This execution model, together with the fact that software operates at a higher level of abstraction than hardware, makes it difficult for us to use existing verification tools for constant-time software (*e.g.*, [16, 20]).

First, constant-time verification for software only considers straight-line, sequential code. This makes it ill-suited for the concurrent, long-lived execution model of hardware.

Second, software constant-time models are necessarily conservative. They deliberately abstract over hardware details—*i.e.*, they don't rely on a precise hardware models (*e.g.*, of caches or branch predictors)—and instead use *leakage models* that make control flow and memory access patterns observable to the attacker. This makes constant-time software portable across hardware. But, it also makes the programming model restrictive: the model disallows any branching to protect against hidden microarchitectural state (*e.g.*, the branch predictor).

Since we operate on VERILOG, where all state is explicit and visible, we can instead directly track the influence of secret values on the timing of attacker-observable outputs. This allows us to be more permissive than software constant-time models. For instance, if we can show that the execution of two branches of a hardware design takes the same amount of time, independent of secret inputs, we can safely allow branches on secrets. However, this still leaves the problem of pipelining: hardware ingests inputs and produce outputs at every clock cycle: how then do we know (if and) which secret inputs influenced a particular output?

Influence Sets. This motivates our definition for *influence sets*. In order to define a notion of constant-time execution that is suitable for hardware, we first add annotations marking inputs (*i.e.*, x and y in our example) as *sources* and outputs (*i.e.*, out) as *sinks*. For a given cycle, we then associate with each register x its *influence-*

Cycle #	x	y	ct	fr	out	θ(x)	θ(y)	θ(ct)	θ(fr)	θ(out)
0	0	1	**F**	**X**	**X**	{0}	{0}	∅	∅	∅
1	0	1	F	X	0	{1}	{1}	∅	∅	{0}
					⋮					
k-1	0	1	F	0	0	{k−1}	{k−1}	∅	{0}	{k−2}
k	0	1	F	0	0	{k}	{k}	∅	{1}	{**k−1**}

Figure 4: Execution of Ex1, where x = 0 and y = 1, and ct is unset. For each variable and cycle, we show its current value and influence set. We assume that it takes k cycles to compute the output along the slow path, and abbreviate flp_res as fr. **X** denotes an unknown/irrelevant value. Register out is only influenced by values from the last cycle. Highlighted cells are the difference with Figure 5. Values that stayed the same in the next cycle are shaded.

Cycle #	x	y	ct	fr	out	θ(x)	θ(y)	θ(ct)	θ(fr)	θ(out)
0	1	1	**F**	**X**	**X**	{0}	{0}	∅	∅	∅
1	1	1	F	X	**X**	{1}	{1}	∅	∅	{0}
					⋮					
k-1	1	1	F	1	X	{k−1}	{k−1}	∅	{0}	{k−2}
k	1	1	F	1	1	{k}	{k}	∅	{1}	{**0**,**k−1**}

Figure 5: Execution of Ex1, where both x = 1 and y = 1, and ct is unset. The execution produces the same influence sets as the execution in Fig. 4, except for cycle k, where out's influence set contains the additional value 0, thereby violating our definition of constant-time execution.

set θ(x). The influence set of a register x contains all cycles t, such that an input at t was used in the computation of x's current value. This allows us to define constant-time execution for hardware: we say that a hardware design is constant-time, if any two executions (that satisfy usage assumptions) produce the same sequence of influence sets for their sinks.

Example. We now illustrate this definition using our running example Ex1 by showing that Ex1 violates our definition of constant-time, if the ct flag is unset. For this, consider Fig. 4 and Fig. 5, which show the state of registers and wires as well as their respective influence sets, for two executions. In both executions, we let y = 1, but vary the value of the x register: in Fig. 4, we set x to 0 to trigger the fast path in Fig. 5 we set it to 1. In both executions, sources x and y are only influenced by the current cycle, constant-time flag ct is set independently of inputs, and temporary register flp_res is influenced by the inputs that were issued k − 1 cycles ago, as it takes k − 1 cycles to compute flp_res along the slow path.

The two executions differ in the influence sets of out. In Fig. 4, out is only influenced by the input issued in the last cycle, through a control dependency on iszero. In the execution in Fig. 5, its value at cycle k is however also influenced by the input at 0. This reflects the propagation of the computation result through the slow path. Crucially, it also shows that the multiplier is not constant-time—the sets θ(out) differing between two runs reflects the influence of data on the duration of the computation.

2.2 Liveness Equivalence

We now show how to reduce verifying whether a given hardware is constant-time to an easy-to-check, yet equivalent problem called liveness equivalence. Intuitively, liveness equivalence reduces the problem of checking equality of influence sets, to checking the equivalence of membership, for arbitrary elements.

Liveness Equivalence. Our reduction focuses on a single computation started at some cycle t. We say that register x is *live* for cycle t (t-live), if its current value is influenced by an input issued in cycle t, *i.e.*, if t ∈ θ(x). Two executions are t-liveness equivalent, if whenever a t-live value is assigned to a sink in one execution, a t-live value must also be assigned in the other. Finally, a hardware design is liveness equivalent, if any two executions that satisfy usage assumptions are t-liveness equivalent, for any t.

Live Value Propagation. To track t-liveness for a fixed t, IODINE internally transforms VINTER programs as follows. For each register or wire (*e.g.*, x in our multiplier), we introduce a new shadow variable (*e.g.*, $x^\bullet$) that represents its liveness; a shadow variable $x^\bullet$ is set to L if x is live and D (dead) otherwise.[2] We then propagate live-

[2] For liveness-bits $x^\bullet$ and $y^\bullet$, we define a join operator ∨, such that

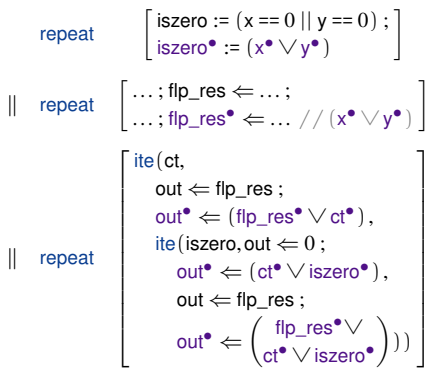

Figure 6: EX1, after we propagate liveness using a standard taint-tracking inline monitor.

	x	y	ct	fr	out	x•	y•	ct•	fr•	out•
0	0	1	**F**	**X**	**X**	L	L	D	D	D
1	0	1	F	X	0	D	D	D	D	L
					⋮					
k-1	0	1	F	0	0	D	D	D	L	D
k	0	1	F	0	0	D	D	D	D	D

Figure 7: Execution of EX1•, where x = 0 and y = 1. We show current value and liveness bit for each register and cycle. Register out is live in cycle one, due to the fast path and dead, otherwise. Highlights are the differences with Figure 8. Values that stayed the same in the next cycle are shaded.

ness using a standard taint-tracking inline monitor [44] shown in Figure 6. Intuitively, our monitor ensures that registers and wires that depend on a live value—directly or indirectly, via control flow—are marked live.

Example. By tracking liveness, we can again see that our floating-point multiplier is not constant-time when the ct flag is unset. To this end, we "inject" live values at sources (x and y) at time t = 0 for two runs; as before, we set y = 1, and vary the value of x: in one execution, we set x to 0 to trigger the fast path, in the other execution, we set it to 1. Fig. 7 and 8 show the state of the different registers and wires for these runs. In both runs, out is live at cycle 1—due to a control dependency in Fig. 7, due a direct assignment in Fig. 8. But, in the latter, out is *also* live at the kth cycle. This reflects the fact that the influence sets of out at cycle k differ in the membership of 0, and therefore witnesses the constant-time violation.

2.3 Verifying Liveness Equivalence

Using our reduction to liveness equivalence, we can *verify* that a VERILOG program executes in constant-time using standard methods. For this, we *mark* source data as

	x	y	ct	fr	out	x•	y•	ct•	fr•	out•
0	1	1	**F**	**X**	**X**	L	L	D	D	D
1	1	1	F	X	**X**	D	D	D	D	L
					⋮					
k-1	1	1	F	1	X	D	D	D	L	D
k	1	1	F	1	1	D	D	D	D	L

Figure 8: Execution of EX1•, where both x = 1 and y = 1. The liveness bits are the same as in 7, except for cycle k, where out is now live. This reflects the propagation of the output value through the slow path and shows the constant-time violation.

live in some *arbitrarily chosen* start cycle t. We then verify that any *two* executions that satisfy usage assumptions assign t-live values to sinks, in the same way.

Product Programs. Like previous work on verifying constant-time software [16], IODINE reduced the problem of verifying properties of *two* executions of some program P by proving a property about a *single* execution of a new program Q. This program—the so-called *product program* [22] – consists of two disjoint copies of the original program.

Race-Freedom. Our product construction exploits the fact that VERILOG programs are *race-free*, *i.e.*, the order in which always-blocks are scheduled within a cycle does not matter. While races in software often serve a purpose (*e.g.*, a task distribution service may allow races between equivalent worker threads to increase throughput), races in VERILOG are always artifacts of poorly designed code: any synthesized circuit is, by its nature, race-free, *i.e.*, the scheduling of processes *within* a cycle does not affect the computation outcome. Indeed, races in VERILOG represent an under-specification of the intended design.

Per-Process Product. We leverage this insight to compose the two copies of a program in *lock-step*. Specifically, we merge each process of the two program copies and execute the "left" (L) and "right" (R) copies together. For example, IODINE transforms the VINTER multiplier code from Figure 6 into the *per-process product program* shown in Figure 9.

Merging two copies of a program as such is sound: since the program is race-free—any ordering of process transitions *within* a cycle yields the same results—we are free to pick an arbitrary schedule.[3] Hence, IODINE takes a simple ordering approach and schedules the left and right copy of same process at the same time.

Constant-Time Assertion. Given such a product program, we can now frame the constant-time verification

x• ∨ y• is L, if x• or y• is L and D, otherwise.

[3] To ensure that hardware designs are indeed race-free, our implementation performs a light-weight static analysis to check for races.

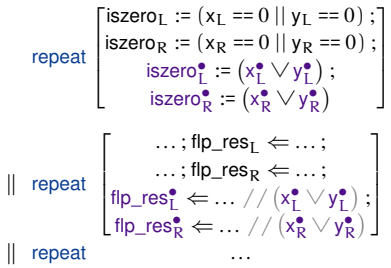

Figure 9: Per-process product form of EX1.

challenge as a simple *assertion*: the liveness of the left and right program sink-variables must be the same (regardless of when the computation started). In our example, this assertion is simply $\mathsf{out}^\bullet{}_L = \mathsf{out}^\bullet{}_R$. This assertion can be verified using standard methods. In particular, IODINE synthesize process-modular invariants [45] that imply the constant-time assertion (§ 5).

The following two sections formalize the material presented in this overview.

3 Syntax and Semantics

Since VERILOG's execution model can be subtle [12], we formally define syntax and semantics of the VERILOG fragment considered in this paper.

3.1 Preliminaries

For a function f, we write *dom* f to denote f's domain and *ran* f for its co-domain. For a set $S \subseteq$ *dom* f, we let $f[S \leftarrow b]$ denote the function that behaves the same as f except S, where it returns b, *i.e.*, $f[S \leftarrow b](x)$ evaluates to b if $x \in S$ and $f(x)$, otherwise. We use $f[a \leftarrow b]$ as a short hand for $f[\{a\} \leftarrow b]$. Sometimes, we want to update a function by setting the function values of some subset S of its domain to a non-deterministically chosen value. For $S \subseteq$ *dom* f, we write $f[S \leftarrow *](x)$ to denote the function that evaluates to some y with $y \in$ *ran* f, if $x \in S$ and $f(x)$ otherwise.

3.2 Syntax

We restrict ourselves to the *synthesizable* fragment of VERILOG, *i.e.*, we do not include commands like initial blocks that only affect simulation and implement a *normalization step* [32] in which the program is "flattened" by removing module instantiation through in-lining. We provide VERILOG syntax and a translation to VINTER in Appendix A.2, but define semantics in VINTER (Fig. 2).

Annotations. We define annotations in Figure 10. Let *Regs* denote the set of registers and *Wires* the set of wires and let VARS denote their disjoint union, *i.e.*,

$a ::=$	**In/Out**		**Assump.**
$\mid \mathsf{source}(v)$	*source*	$\mid \mathsf{init}(\varphi)$	*initially* φ
$\mid \mathsf{sink}(v)$	*sink*	$\mid \Box(\varphi)$	*always* φ

Figure 10: Annotation syntax.

Config	**Meaning**	**Trace**	**Meaning**
σ	*store*	Σ	*configuration*
τ	*liveness map*	l	*label*
θ	*influence map*	b	*liveness bit*
μ	*assign. buffer*	π	*trace*
ev	*event set*	$\mathit{store}(\pi, i)$	σ_i
P	*current program*	$\mathit{live}(\pi, i)$	τ_i
I	*initial program*	$\mathit{inf}(\pi, i)$	θ_i
c	*clock cycle*	$\mathit{clk}(\pi, i)$	c_i
		$\mathit{reset}(\pi, i)$	b_i

Figure 11: Configuration and trace syntax.

$\mathrm{VARS} \triangleq \mathit{Regs} \uplus \mathit{Wires}$. For a register $v \in \mathit{Regs}$, annotations $\mathsf{source}(v)$ and $\mathsf{sink}(v)$ designate v as source or sink, respectively.[4] We let $\mathrm{IO} \triangleq (\mathit{Src}, \mathit{Sink})$ denote the set of input/output assumptions, where *Src* denotes the set of all sources and *Sink* denote the set of all sinks. Let φ be a first-order formula over some background theory that refers to two disjoint sets of variables Vars_L and VARS_R. Then, annotations $\mathsf{init}(\varphi)$ and $\Box(\varphi)$ indicate that formula φ holds initially or throughout the execution. The assumptions are collected in $A \triangleq (\mathrm{INIT}, \mathrm{ALL})$, such that INIT contains all formulas under init and ALL all formulas under $\Box$.

3.3 Semantics

Values. The set of values $\mathrm{VALS} \triangleq \mathbb{Z} \uplus \{\mathbf{X}\}$ consists of the disjoint union of the integers and special value $\mathbf{X}$ which represents an irrelevant value. A function application that contains $\mathbf{X}$ as an argument evaluates to $\mathbf{X}$.

Configurations. The program state is represented by a *configuration* $\Sigma \in \mathit{Configs}$. Figure 11 shows the components of a configuration. A store $\sigma \in \mathrm{STORES} \triangleq (\mathrm{VARS} \mapsto \mathrm{VALS})$ is a map from registers and wires to values. A *liveness map* $\tau \in \mathrm{LIVEMAP} \triangleq (\mathrm{VARS} \mapsto \{L, D\})$ is a map from registers and wires to liveness bits. A *influence map* $\theta \in \mathrm{INFMAPS} \triangleq (\mathrm{VARS} \mapsto \mathcal{P}(\mathbb{Z}))$ is a map from registers and wires to influence sets. *Assignment buffers* serve to model non-blocking assignments. Let *PIDs* denote a set of process identifiers. An assignment buffer $\mu \in \mathit{PIDs} \mapsto (\mathrm{VARS} \times \mathrm{VALS} \times \{L, D\} \times \mathcal{P}(\mathbb{Z}))^*$ is a map from pro-

[4] To use wires as source/sink, one has to define an auxiliary register.

$$\frac{}{v,\sigma,\tau,\theta \dashrightarrow \sigma(v),\tau(v),\theta(v)}\ [\textsc{Var}] \qquad \frac{}{n,\sigma,\tau,\theta \dashrightarrow n,\mathsf{D},\emptyset}\ [\textsc{Const}]$$

$$\frac{e_1,\sigma,\tau,\theta \dashrightarrow v_1,t_1,i_1 \quad \ldots \quad e_k,\sigma,\tau,\theta \dashrightarrow v_k,t_k,i_k \qquad t = (t_1 \vee \cdots \vee t_k) \qquad i = (i_1 \cup \cdots \cup i_k)}{f(e_1,\ldots,e_k),\sigma,\tau,\theta \dashrightarrow f(v_1,\ldots,v_k),t,i}\ [\textsc{Fun}]$$

Figure 12: Expression evaluation.

cess identifier to a sequence of variable/value/liveness-bit/influence set tuples. An *event set* $ev \in \mathcal{P}(\textsc{Vars})$ is a set of variables, where we use $v \in ev$ to indicate that variable v has been changed in the current cycle. Finally, $I \in \mathit{Progs}$ contains the initial program. Intuitively, the initial program is used to activate all processes when a new clock cycle begins.

Evaluating Expressions. We define an evaluation relation $\dashrightarrow \in (\textsc{Expr} \times \textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps}) \mapsto (\textsc{Vals} \times \{\mathsf{L},\mathsf{D}\} \times \mathcal{P}(\mathbb{Z}))$ that computes value, liveness-bit, and influence map for an expression. We define the relation through the inference rules shown in Fig. 12. An evaluation step (below the line) can be taken, if the preconditions (above the line) are met. Rule [Var] evaluates a variable to its current value under the store, its current liveness-bit and influence set. A numerical constant evaluates to itself, is dead and not influenced by any cycle. To evaluate a function literal, we evaluate its arguments and apply the function on the resulting values. A function value is live if any of its arguments are, and its influence set is the union of its influences.

Transition Relations. We define our semantics in terms of four separate transition relations of type $(\mathit{Configs} \times \mathit{Labels} \times \mathit{Configs})$. We now discuss the individual relations and then describe how to combine them into an overall transition relation $\rightsquigarrow$.

Per-process transition $\rightsquigarrow_{\mathsf{P}}$**.** The per-process transition relation $\rightsquigarrow_{\mathsf{P}}$ describes how to step along individual processes. It is defined in Fig. 13. Rules [Seq-Step] and [Par-Step] are standard and describe sequential and parallel composition. Rule [B-Asn] reduces a blocking update $x = e$ to skip, by first evaluating e to yield a value v, liveness bit t and influence set i, updating store σ, liveness map τ and influence map θ, and finally adding x to the set of modified variables. Rule [NB-Asn] defers a non-blocking assignment. In order to reduce an assignment $(x \Leftarrow e)_{id}$ for process *id* to skip, the rule evaluates expression e to value v, liveness bit t and influence set i, and defers the assignment by appending the tuple (x,v,t,i) to the back of *id*'s buffer. We omit rules for conditionals and structural equivalence. Structural equivalence allows transitions between trivially equivalent programs such as $\mathsf{P} \parallel \mathsf{Q}$ and $\mathsf{Q} \parallel \mathsf{P}$.

Non-blocking Transition $\rightsquigarrow_{\mathsf{N}}$**.** Transition relation $\rightsquigarrow_{\mathsf{N}}$ applies deferred non-blocking assignments. It is defined by a single rule [NB-App] shown in Fig. 13. The rule first picks a tuple (x,v,t,i) from the front of the buffer of some process *id*, and, like [B-Asn], updates store σ, liveness map τ and influence map θ, and finally adds x to the set of updated variables.

Continuous Transition $\rightsquigarrow_{\mathsf{C}}$**.** Relation $\rightsquigarrow_{\mathsf{C}}$ specifies how to execute continuous assignments. It is described by rule [C-Asn] in Fig. 13, which reduces a continuous assignment $x := e$ to skip under the condition that some variable y occurring in e has changed, *i.e.*, $y \in ev$. To apply the assignment, it evaluates e to value, liveness bit and influence set, and updates store and liveness map and influence map. Importantly, variable y is not removed from the set of events, *i.e.*, a single assignment can enable several continuous assignments.

Global Transition $\rightsquigarrow_{\mathsf{G}}$**.** Finally, global transition relation $\rightsquigarrow_{\mathsf{G}}$ is defined by rules [NewCycle] and [NewCycle-Issue] shown in Fig. 13. [NewCycle] starts a new clock cycle by discarding the current program and event set, emptying the assignment buffer, resetting the wires to some non-deterministically chosen state (as wires only hold their value *within* a cycle), and rescheduling and activating a new set of processes, extracted from initial program I. For a program P, let $\textsc{Repeat}(\mathsf{P}) \in \mathcal{P}(\mathit{Progs})$ denote the set of processes that occur under repeat. For a set of programs S, we let $\sqcap\, \mathsf{S}$ denote their parallel composition. [NewCycle] uses these constructs to reschedule all processes that appear under repeat in I. Both sources and wires are set to D. The influence map is updated by mapping all wires to the empty set, and each source to the set containing only the current cycle.

[NewCycle-Issue] performs the same step, but additionally updates the liveness map by issuing new live bits for the source variables. Both rules increment the cycle counter c. The rules issue a *label* $l \in \mathit{Labels} \triangleq ((\textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps} \times \mathbb{N} \times \{\mathsf{L},\mathsf{D}\}) \uplus \epsilon)$ which is written above the arrow (all previous rules issue the empty label ϵ). The label contains the current store, liveness map, influence map, clock cycle, and a bit indicating whether new live-bits have been issued. Labels are used to construct the *trace* of an execution, as

we will discuss later.

Overall Transition $\rightsquigarrow$**.** We define the overall transition relation $\rightsquigarrow \in \mathit{Configs} \times \mathit{Labels} \times \mathit{Configs}$ by fixing an order in which to apply the relations. Whenever a *continuous assignment* step (relation $\rightsquigarrow_C$) can be applied, that step is taken. Whenever no continuous assignment step can be applied, however, a *per-process* step (relation $\rightsquigarrow_P$) can be applied, a $\rightsquigarrow_P$ step is taken. If no continuous assignment and process local steps can be applied, however, an *non-blocking assignment* step (relation $\rightsquigarrow_N$) is applicable, a $\rightsquigarrow_N$ step is taken. Finally, if neither continuous assignment, per-process, or non-blocking steps can be applied, the program moves to a new clock cycle by applying a *global step* (relation $\rightsquigarrow_G$). Our overall transition relation closely follows the Verilog simulation reference model from Section 11.4 of the standard [12].

Executions and Traces. An *execution* is a finite sequence of configurations and transition labels $r \triangleq \Sigma_0 l_0 \Sigma_1 \ldots \Sigma_{m-1} l_{m-1} \Sigma_m$ such that $\Sigma_i \overset{l_i}{\rightsquigarrow} \Sigma_{i+1}$ for $i \in \{1,\ldots,m-1\}$. We call Σ_0 *initial state* and require that all taint bits are set to D, the influence map maps each variable to the empty set, the assignment buffer is empty, the current program is the empty program $\rangle$, and the clock is set to 0. The *trace* of an execution is the sequence of its (non-empty) labels. For a trace $\pi \triangleq (\sigma_0,\tau_0,\theta_0,c_0,b_0)\ldots(\sigma_{n-1},\tau_{n-1},\theta_{n-1}c_{n-1},b_{n-1}) \in \mathit{Labels}^*$ and for $i \in \{0,\ldots,n-1\}$ we let $\mathit{store}(\pi,i) \triangleq \sigma_i$, $\mathit{live}(\pi,i) \triangleq \tau_i$, $\mathit{inf}(\pi,i) \triangleq \theta_i$, $\mathit{clk}(\pi,i) \triangleq c_i$ and $\mathit{reset}(\pi,i) = b_i$, and say the trace has length n. For a program P we use $\textsc{Traces}(P) \in \mathcal{P}(\mathit{Labels}^*)$ to denote the set of its traces, *i.e.*, all traces with initial program P.

4 Constant-Time Execution

We now first define constant-time execution with respect to a set of assumptions. We then define liveness equivalence and show that the two notions are equivalent.

4.1 Constant-Time Execution

Assumptions. For a formula φ that ranges over two disjoint sets of variables $\textsc{Vars}_L$ and $\textsc{Vars}_R$ and stores σ_L and σ_R such that $\mathit{dom}\ \sigma_L = \textsc{Vars}_L$ and $\mathit{dom}\ \sigma_R = \textsc{Vars}_R$, we write $\sigma_L,\sigma_R \models \varphi$ to denote that formula φ holds when evaluated on σ_L and σ_R. For some program P and a set of assumptions $A \triangleq (\textsc{Init},\textsc{All})$, we say that two traces $\pi_L,\pi_R \in \textsc{Traces}(P)$ of length n *satisfy* A if *i)* for each formula $\varphi_I \in \textsc{Init}$, φ_I holds initially, and *ii)* for each formula $\varphi_A \in \textsc{All}$, φ_A hold throughout, *i.e.*, $\mathit{store}(\pi_L,0),\mathit{store}(\pi_R,0) \models \varphi_I$ and $\mathit{store}(\pi_L,i),\mathit{store}(\pi_R,i) \models \varphi_A$, for $0 \leqslant i \leqslant n-1$. Intuitively, pairs of traces that satisfy the assumptions are "low" or "input" equivalent.

Constant Time Execution. For a program P, assumptions A and traces $\pi_L,\pi_R \in \textsc{Traces}(P)$ of length n that satisfy A, π_L and π_R are *constant time* with respect to A, if they produce the same influence sets for all sinks, *i.e.*, $\mathit{inf}(\pi_L,i)(v) = \mathit{inf}(\pi_R,i)(v)$, for $0 \leqslant i \leqslant n-1$ and all $v \in \mathit{Sink}$, and where two sets are equal if they contain the same elements. A program is constant time with respect to A, if all pairs of its traces that satisfy A are constant time.

4.2 Liveness Equivalence

t-Trace. For a trace π, we say that π is a t-trace, if $\mathit{reset}(\pi,t) = L$ and $\mathit{reset}(\pi,i) = D$, for $i \neq t$.

Liveness Equivalence. For a program P, let $\pi_L,\pi_R \in \textsc{Traces}(P)$, such that both π_L and π_R are of length n. We say that π_L and π_R are t-*liveness equivalent*, if both are t-traces, and $\mathit{live}(\pi_L,i)(v) = \mathit{live}(\pi_R,i)(v)$, for $0 \leqslant i \leqslant n-1$ and all $v \in \mathit{Sink}$. A program is t-liveness equivalent, with respect to a set of assumptions A, if all pairs of t-traces that satisfy A are t−liveness equivalent. Finally, a program is liveness equivalent with respect to A, if it is t-liveness equivalent with respect to A, for all t.

4.3 Equivalence

We can now state our equivalence theorem.

Theorem 1. *For all programs* P *and assumptions* A*,* P *executes in constant-time with respect to* A *if and only if it is liveness equivalent with respect to* A*.*

We first give a lemma which states that, if a register is t-live, then t is in its influence set.

Lemma 1. *For any* t*-trace* π *of length* n*, index* $0 \leqslant i \leqslant n-1$*, and variable* v*, if* v *is* t*−live,* i.e.*,* $\mathit{live}(\pi,i)(v) = L$*, then* t *is in* v*'s influence map,* i.e.*,* $t \in \mathit{inf}(\pi,i)(v)$*.*

We can now state our proof for Theorem 1.

Proof Theorem 1. The interesting direction is "right-to-left", *i.e.*, we want to show that a liveness equivalent program is also constant-time. We prove the contrapositive, *i.e.*, if a program violates constant-time, it must also violate liveness equivalence. For a proof by contradiction, we assume that P violates constant time execution, but satisfies liveness equivalence. If P violates constant-time execution, then there must be a sink v^*, two trace $\pi_L^*,\pi_R^* \in \textsc{Traces}(P)$ that satisfy A, and some

$$\text{[SEQ-STEP]}\quad \frac{\langle \sigma,\mu,\theta,ev,\tau,s_1,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',s_1',I,c\rangle}{\langle \sigma,\mu,\theta,ev,\tau,[s_1;s_2],I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',[s_1';s_2],I,c\rangle}$$

$$\text{[PAR-STEP]}\quad \frac{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',P',I,c\rangle}{\langle \sigma,\mu,\theta,ev,\tau,P \parallel Q,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu',\theta',ev',\tau',P' \parallel Q,I,c\rangle}$$

$$\text{[B-ASN]}\quad \frac{e,\sigma,\tau,\theta \dashrightarrow v,t,i \qquad \sigma' = \sigma[x \leftarrow v] \qquad \tau' = \tau[x \leftarrow t] \qquad \theta' = \theta[x \leftarrow i]}{\langle \sigma,\mu,\theta,ev,\tau,x = e,I,c\rangle \rightsquigarrow_P \langle \sigma',\mu,\theta',ev \cup \{x\},\tau',\mathsf{skip},I,c\rangle}$$

$$\text{[NB-ASN]}\quad \frac{e,\sigma,\tau,\theta \dashrightarrow v,t,i \qquad \mu' = \mu[id \leftarrow (x,v,t,i)\cdot q]}{\langle \sigma,\mu[id \leftarrow q],\theta,ev,\tau,(x \Leftarrow e)_{id},I,c\rangle \rightsquigarrow_P \langle \sigma,\mu',\theta,ev,\tau,\mathsf{skip},I,c\rangle}$$

$$\text{[NB-APP]}\quad \frac{\sigma' = \sigma[x \leftarrow v] \qquad \mu' = \mu[id \leftarrow q] \qquad \theta' = \theta[x \leftarrow i] \qquad \tau' = \tau[x \leftarrow t] \qquad ev' = ev \cup \{x\}}{\langle \sigma,\mu[id \leftarrow q\cdot(x,v,t,i)],\theta,ev,\tau,P,I,c\rangle \rightsquigarrow_N \langle \sigma',\mu',\theta',ev',\tau',P,I,c\rangle}$$

$$\text{[C-ASN]}\quad \frac{e,\sigma,\tau,i \dashrightarrow v,t,i \qquad y \in \textsc{Vars}(e) \qquad \sigma' = \sigma[x \leftarrow v] \qquad \tau' = \tau[x \leftarrow t] \qquad \theta' = \theta[x \leftarrow i]}{\langle \sigma,\mu,\theta,ev \cup \{y\},\tau,x := e,I,c\rangle \rightsquigarrow_C \langle \sigma',\mu,\theta',ev \cup \{x,y\},\tau',\mathsf{skip},I,c\rangle}$$

$$\text{[NEWCYCLE]}\quad \frac{\sigma' \triangleq \sigma[\mathit{Wires} \leftarrow *] \qquad \tau' \triangleq \tau[\mathit{Src} \leftarrow D][\mathit{Wires} \leftarrow D] \qquad \theta' \triangleq \theta[\mathit{Wires} \leftarrow \emptyset][\mathit{Src} \leftarrow \{c+1\}] \qquad \mu' \triangleq \mu[\mathit{PIDs} \leftarrow \epsilon]}{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \overset{(\sigma,\tau,\theta,c,D)}{\rightsquigarrow_G} \langle \sigma',\mu',\theta',\emptyset,\tau,\sqcap\,\textsc{Repeat}(I),I,c+1\rangle}$$

$$\text{[NEWCYCLE-ISSUE]}\quad \frac{\sigma' \triangleq \sigma[\mathit{Wires} \leftarrow *] \qquad \tau' \triangleq \tau[\mathit{Src} \leftarrow L][(\textsc{Vars} - \mathit{Src}) \leftarrow D] \qquad \theta' \triangleq \theta[\mathit{Wires} \leftarrow \emptyset][\mathit{Src} \leftarrow \{c+1\}] \qquad \mu' \triangleq \mu[\mathit{PIDs} \leftarrow \epsilon]}{\langle \sigma,\mu,\theta,ev,\tau,P,I,c\rangle \overset{(\sigma,\tau,\theta,c,L)}{\rightsquigarrow_G} \langle \sigma',\mu',\theta',\emptyset,\tau',\sqcap\,\textsc{Repeat}(I),I,c+1\rangle}$$

Figure 13: Per-thread transition relation $\rightsquigarrow_P$, non-blocking transition relation $\rightsquigarrow_N$, continuous transition relation $\rightsquigarrow_C$, and global restart relation $\rightsquigarrow_G$.

index i^* such that $\mathit{inf}(\pi_L^*,i^*)(v^*) \neq \mathit{inf}(\pi_R^*,i^*)(v^*)$, and therefore without loss of generality, there is a cycle t^*, such that $t^* \in \mathit{inf}(\pi_L^*,i^*)(v^*)$ and $t^* \notin \mathit{inf}(\pi_R^*,i^*)(v^*)$. We can find two traces t^*-traces $\hat{\pi}_L$ and $\hat{\pi}_R$ that only differ from π_L^* and π_R^* in their liveness maps. But then, since the traces are t^*-liveness equivalent, by definition, at index i^* both $\hat{\pi}_L$ and $\hat{\pi}_R$ are t^*—live, *i.e.*, $\mathit{live}(\hat{\pi}_L,i^*)(v^*) = \mathit{live}(\hat{\pi}_R,i^*)(v^*) = L$ and, by lemma 1, $t^* \in \mathit{inf}(\hat{\pi}_R,i^*)(v^*)$. Since $\hat{\pi}_R$ and π_R^* only differ in their liveness map, this implies $t^* \in \mathit{inf}(\pi_R^*,i^*)(v^*)$, from which the contradiction follows. □

5 Verifying Constant Time Execution

In this section, we describe how IODINE verifies liveness equivalence by using standard techniques.

***Algorithm* IODINE.** Given a VINTER program P, a set of input/output specifications IO and a set of assumptions A, IODINE checks that P executes in constant time with respect to A. For this, IODINE first checks for race-freedom. If a race is detected, IODINE returns a witness describing the violation. If no race is detected, IODINE takes the following four steps: **(1)** It builds a set of Horn clause constraints hs [26, 33] whose solution characterizes the set of all configurations that are reachable by the per-process product and satisfy A. **(2)** Next, it builds a set of constraints cs whose solutions characterize the set of liveness equivalent states. **(3)** It then computes a solution Sol to hs and checks whether the solution satisfies cs. To find a more precise solution, the user can supply additional hints in the form of a set of predicates which we describe later. **(4)** If the check succeeds, P executes in constant time with respect to A, otherwise, P can potentially exhibit timing variations.

Constraint Solving. IODINE solves the reachability constraints by using Liquid Fixpoint [10], which computes the *strongest solution* that can be expressed as a conjunction of elements of a set of logical formulas. These formulas are composed of a set of *base predicates*. We use base predicates that track equalities between the liveness bits and values of each variable between the two runs. In addition to these base predicates, we use hints that are defined by the user. We discuss in § 6 which predicates were used in our benchmarks.

6 Implementation and Evaluation

In this section, we describe our implementation and evaluate IODINE on several open source VERILOG projects, spanning from RISC processors, to floating-point units and crypto cores. We find that IODINE is able to show that a piece of code is not constant-time and otherwise verify that the hardware is constant-time in a matter of seconds. Except our processor use cases, we found the annotation burden to be light weight—often less than 10 lines of code. All the source code and data are available on GitHub, under an open source license.[5]

6.1 Implementation

IODINE consists of a front-end pass, which takes annotated hardware descriptions and compiles them to VINTER, and a back-end that verifies the constant-time execution of these VINTER programs. We think this modular designs will make it easy for IODINE to be extended to support different hardware description languages beyond VERILOG (*e.g.*, VHDL or Chisel [19]).

Our front-end extends the Icarus Verilog parser [9] and consists of 2000 lines of C++. Since VINTER shares many similarities with VERILOG, this pass is relatively straightforward, however, IODINE does not distinguish between clock edges (positive or negative) and, thus, removes them during compilation. Moreover, our prototype does not support the whole VERILOG language (*e.g.*, we do not support assignments to multiple variables).

IODINE's back-end takes a VINTER program and, following § 5, generates and checks a set of verification conditions. We implement the back-end in 4000 lines of Haskell. Internally, this Haskell back-end generates Horn clauses and solves them using the liquid-fixpoint library that wraps the Z3 [29] SMT solver. Our back-end outputs the generated invariants, which **(1)** serve as the proof of correctness when the verification succeeds, or **(2)** helps pinpoint why verification fails.

Tool Correctness. The IODINE implementation and Z3 SMT solver [29] are part of our trusted computing base. This is similar to other constant-time and information flow tools (*e.g.*, SecVerilog [55] and ct-verif [16]). As such, the formal guarantees of IODINE can be undermined by implementation bugs. We perform several tests to catch such bugs early—in particular, we validate: (1) our translation into VINTER against the original VERILOG code; (2) our translation from VINTER into Horn clauses against our semantics; and, (3) the generated invariants against both the VINTER and VERILOG code.

[5] https://iodine.programming.systems

6.2 Evaluation

Our evaluation seeks to answer three questions: (Q1) Can IODINE be easily applied to existing hardware designs? (Q2) How efficient is IODINE? (Q3) What is the annotation burden on developers?

(Q1) Applicability. To evaluate its applicability, we run IODINE on several open source hardware modules from GitHub and OpenCores. We chose VERILOG programs that fit into three categories—processors, crypto-cores, and floating-point units (FPUs)—these have previously been shown to expose timing side channels. In particular, our benchmarks consist of:

- MIPS- and RISCV-32I-based pipe-lined CPU cores with a single level memory hierarchy.
- Crypto cores implementing the SHA 256 hash function and RSA 4096-bit encryption.
- Two FPUs that implement core operations $(+, -, \times, \div)$ according to the IEEE-754 standard.
- An ALU [1] that implements $(+, -, \times, \ll, \ldots)$.

In our benchmarks, following our attacker model from § 2.1, we annotated all the inputs to the computation. For example, this includes the sequence of instructions for the benchmarks with a pipeline (*i.e.*, MIPS, RISC-V, FPU and FPU2) in addition to other control inputs, and all the top level VERILOG inputs for the rest (*i.e.*, SHA-256, ALU and RSA). Similarly, we annotated as sinks, all the outputs of the computation. In the case of benchmarks with a pipeline, this includes the output from the last stage and other results (*e.g.*, whether the result is NaN in FPU), and all the top level VERILOG outputs for the rest. The modifications we had to perform to run IODINE on these benchmarks were minimal and due to parser restrictions (*e.g.*, desugaring assignments to multiple variables into individual assignments, unrolling the code generated by the loop inside the generate blocks).

(Q2) Efficiency. To evaluate its efficiency, we run IODINE on the annotated programs. As highlighted in Table 1, IODINE can successfully verify different VERILOG programs of modest size (up to 1.1K lines of code) relatively quickly (<20s). All but the constant-time FPU finished in under 3 seconds. Verifying the constant-time FPU took 12 seconds, despite the complexity of IEEE-754 standard which manifests as a series of case splits in VERILOG. We find these measurements encouraging, especially relative to the time it takes to synthesize VERILOG—verification is orders of magnitude smaller.

Name	#LOC	#Assum		CT	Check (s)
		#flush	#always		
MIPS [5]	434	31	2	✓	1.329
RISC-V [7]	745	50	19	✓	1.787
SHA-256 [8]	651	5	3	✓	2.739
FPU [6]	1182	0	0	✓	12.013
ALU [1]	913	1	5	✓	1.595
FPU2 [3]	272	3	4	✗	0.705
RSA [4]	870	4	0	✗	1.061
Total	5067	94	33	-	21.163

Table 1: #LOC is the number of lines of Verilog code, **#Assum** is the number of assumptions (excluding source and sink); **flush** and **always** are annotations of the form init and □ respectively, **CT** shows if the program is constant-time, and **Check** is the time IODINE took to check the program. All experiments were run on a Intel Core i7 processor with 16 GB RAM.

Discovered Timing Variability. Running IODINE revealed that two of our use cases are not constant-time: one of the FPU implementations and the RSA crypto-core. The division module of the FPU exhibits timing variability depending on the value of the operands. In particular, similar to the example from § 2, the module triggers a fast path if the operands are special values.

The RSA encryption core similarly exhibited time variability. In particular, the internal modular exponentiation algorithm performs a Montgomery multiplication depending on the value of a source bit e_i: if $e_i = 1$ then $\overline{c} := \mathsf{ModPro}(\overline{c}, \overline{m})$. Since e is a secret, this timing variability can be exploited to reveal the secret key [27, 35].

(Q3) Annotation burden. While IODINE automatically discovers proofs, the user has to provide a set of assumptions A under which the hardware design executes in constant time. To evaluate the burden this places on developers, we count the number and kinds of assumptions we had to add to each of our use cases. Table 1 summarizes our results: except for the CPU cores, most of our other benchmarks required only a handful of assumptions. Beyond declaring sinks and sources, we rely on two other kinds of annotations. First, we find it useful to specify that the initial state of an input variable x is equal in any pair of runs, *i.e.*, $\mathsf{init}(x_L = x_R)$. This assumption essentially specifies that register x is flushed, *i.e.*, is set to a constant value, to remove any effects of a previous execution from our initial state. Second, we find it useful to specify that the state of an input variable x is equal, throughout any pair of runs, *i.e.*, $\square(x_L = x_R)$. This assumption is important when certain behavior is expected to be the same in both runs. We now describe these assumptions for our benchmarks.

- **MIPS:** We specify that the values of the fetched instructions, and the reset bit are the same.
- **RISC-V:** In addition to the assumptions required by the MIPS core, we also specify that both runs take the same conditional branch, and that the type of memory access (read or write) is the same in both runs (however, the actual values remain unrestricted). This corresponds to the assumption that programs running on the CPU do not branch or access memory based on secret values. Finally, CSR registers must not be accessed illegally (see § 6.3).
- **ALU:** Both runs execute the same type of operations (*e.g.*, bitwise, arithmetic), operands have the same bit width, instructions are valid, reset pins are the same.
- **SHA-256 and FPU (division):** We specify that the reset and input-ready bits are the same.

In all cases, we start with no assumptions and add the assumptions incrementally by manually investigating the constant-time "violation" flagged by IODINE.

Identifying Assumptions. From our experience, the assumptions that a user needs to specify fall into three categories. The first are straightforward assumptions—*e.g.*, that any two runs execute the same code. The second class of assumptions specify that certain registers need to be flushed, *i.e.*, they need to initially be the same (flushed) for any two runs. To identify these, we first flush large parts of circuits, and then, in a minimization step, we remove all unnecessary assumptions. The last, and most challenging, are implicit invariants on data and control—*e.g.*, the constraints on CSR registers. IODINE performs delta debugging to help pinpoint violations but, ultimately, these assumptions require user intervention to be resolved. Indeed, specifying these assumptions require a deep understanding of the circuit and its intended usage. In our experience, though, only a small fraction of assumptions fall into this third category.

User Hints. For one of our benchmarks (FPU), we needed to supply a small number of user hints (<5) to the solver. These hints come in the form of predicates that track additional equalities between liveness bits of the *same* run. This is required, when the two executions can take different control paths, yet execute in constant time. We hope to remove those hints in the future.

6.3 Case Studies

We now illustrate how IODINE verifies benchmarks with challenging features and helps explicate conditions under which a hardware design is constant-time, using exam-

```
1  always @(*) begin
2    if (...)
3      Stall = 1; else Stall = 0;
4  end
5  always @(posedge clk) begin
6    if (Stall)
7      ID_instr <= ID_instr;
8    else
9      ID_instr <= IF_instr;
10 end
```

Figure 14: Stalling in MIPS [5].

ples from our benchmarks.

History Dependencies. In hardware, the result of a computation often depends on inputs from previous cycles, *i.e.*, the computation depends on execution history. For example, when a hardware unit is in use by a previous instruction, the CPU stalls until the unit becomes free.

The code snippet in Fig. 14 contains a simplified version of the stalling logic from our MIPS processor benchmark. On line 3, register Stall is set to 1 if instructions in the *execute* and *instruction decode* stages conflict. Its value is then used to update the state of each pipeline stage. In this example, if the pipeline is stalled, the value of the register ID_instr, which corresponds to the instruction currently executing in the *instruction decode* stage, stays the same. Otherwise, it is updated with IF_instr—the value coming from the *instruction fetch* stage.

Without further assumptions, IODINE flags this behavior as non-constant time, as an instruction can take different times to process, depending on which other instructions are before it in the pipeline. However, after adding the assumption that any two runs execute the *same sequence of instructions*, IODINE is able to prove that Stall has the same value in any pair of traces, from which the constant time behavior follows. Importantly, however, we have no assumption on the state of the registers and memory elements that the instructions use.

Diverging Control Flow. Methods for enforcing constant time execution of software often require that any two executions take the same control flow path [16]. In hardware, this assumption is too restrictive. Consider the code snippet in Fig. 15 taken from our constant time FPU benchmark (the full logic is shown in Fig. 20 of the Appendix). The first always block calculates the sign bit of the multiplication result (`sign_mul_r`), using inputs opa and opb. The FPU uses this bit in line 17 (through `sign_mul_final`), to calculate output `out` in line 12. Even though we cannot assume that all exe-

```
1  always @(*)
2     case({opa[31], opb[31]})
3          2'b0_0: sign_mul_r <= 0;
4          2'b0_1: sign_mul_r <= 1;
5          ...
6     endcase
7  ...
8  assign sign_mul_final = (sign_exe_r & ...) ?
9              !sign_mul_r : sign_mul_r;
10 ...
11 always @(posedge clk)
12 out <= { ( ... ?
13             (f2i_out_sign &
14             !(qnan_d | snan_d) ) :
15             (((fpu_op_r3 == 3'b010)
16               & ... ?
17              sign_mul_final : ...))) };
```

Figure 15: Diverging control flow in FPU [6].

cutions select the same branches, IODINE can infer that every branch produces the same *influence sets* for the variables assigned under them. Using this information, IODINE can prove that the FPU operates in constant-time, despite diverging control flow paths.

Assumptions. IODINE can be used to inform software mechanisms for mitigating timing side-channels by explicating—and verifying—conditions under which a circuit executes in constant time. Consider Figure 16, which shows the logic for updating *Control and Status Registers (CSR)* in our RISC-V benchmark. The wire `de_illegal_csr_access`, defined on line 1 is set by checking whether a CSR instruction is executed in non-privileged mode. For this, the circuit compares the machine status register `csr_mstatus` to the instructions status bit. When `de_illegal_csr_access` is set, the branch instruction on line 8 traps the error and jumps to a predefined handler code. In order to prove that the cycle executes in constant-time, we add an assumption stating that CSR registers are not accessed illegally. This assumption translates into an obligation for software mitigation mechanisms to ensure proper use of CSR registers.

7 Limitations and Future Work

We discuss some of IODINE's limitations.

Clocks and Assumptions. For example, IODINE presupposes a single fixed-cycle clock and thus does not allow for checking arbitrary VERILOG programs. We leave an extension to multiple clocks as future work. Similarly, IODINE requires users to add assumptions by hand in somewhat ad-hoc trial-and-error fashion. For large circuits this could prove extremely difficult and poten-

```
wire de_illegal_csr_access =
     de_valid &&
     de_inst‘opcode == ‘SYSTEM &&
     de_inst‘funct3 != ‘PRIV &&
     ( csr_mstatus‘PRV < de_inst[29:28] ||
       ... );
always @(posedge clk) begin
   if (de_illegal_csr_access) begin
      ex_restart <= 1;
      ex_next_pc <= ...;
   end
end
```

Figure 16: Update of CSRs in RISC-V [7].

tially lead to errors where erroneous assumptions may lead IODINE to falsely mark a variable time circuit as constant-time. We leave the inference and validation of assumptions to future work.

Scale. We evaluate IODINE on relatively small sized (500-1000 lines) hardware designs. We did not (yet) evaluate the tool on larger circuits, such as modern processors with advanced features like a memory hierarchy, and out-of-order and transient-execution. In principle, these features boil down to the same primitives (always blocks and assignments) that IODINE already handles. But, we anticipate that scaling will require further changes to IODINE, for instance, finding per-module invariants rather than the naive in-lining currently performed by IODINE. We leave the evaluation to larger systems to future work.

8 Related Work

Constant-Time Software. Almeida et al. [16] verify constant-time execution of cryptographic libraries for LLVM. Their notion of constant-time execution is based on a *leakage model*. This choice allows them to be flexible enough to capture various properties like (timing) variability in memory access patterns and improper use of timing sensitive instructions like `DIV`. Unfortunately, their notion of constant-time is too restrictive for our setting, as it requires the control flow path of any two runs to be the same. This would, for example, incorrectly flag our FPU multiplier as variable-time. Like IODINE, their tool ct-verif employs a product construction that use the fact that loops can often be completely unrolled in cryptographic code, whereas we rely on race freedom.

Barthe et al. [20] build on the CompCert compiler [39] to enforce constant time execution through an information flow type system.

Reparaz et al. [47] present a method for discovering timing variability in existing systems through a black-box approach, based on statistical measurements.

All of these approaches address constant-time execution in software and do not translate to the hardware setting (see § 2).

Self-Composition and Product Programs. Barthe et al. [22] introduce the notion of self composition to verify information flow. Terauchi and Aiken generalize this construction to arbitrary 2-safety properties [50], *i.e.*, properties that relate two runs, and Clarkson and Schneider [28] generalize to multiple runs. Barthe et al. [21] introduce product programs that, instead of conjoining copies sequentially, compose copies in lock-step; this was later used in other tools like ct-verif. This technique is further developed in [49], which presents an extension of Hoare logic to hyper-properties that computes lock-step compositions on demand, per Hoare-triple.

Information Flow Safety and Side Channels. There are many techniques for proving information flow safety (*e.g.*, non-interference) in both hardware and software. Kwon et al. [37] prove information flow safety of hardware for policies that allow explicit declassification and are expressed over streams of input data. They construct relational invariants by using propositional interpolation and implicitly build a full self-composition; by contrast, we leverage race-freedom to create a per-thread product which contains only a subset of behaviors.

SecVerilog [55] proves timing-sensitive non-interference for circuits implemented in an extension of VERILOG that uses value-dependent information flow types. Caisson [40] is a hardware description language that uses information flow types to ensure that generated circuits are secure. GLIF [51, 52] tracks the flow of information at the gate level to eliminate explicit and covert channels. All these approaches have been used to implement information flow secure hardware that do not suffer from (timing) side-channels.

IODINE focuses on clock-precise constant-time execution, not information flow. The two properties are related, but information flow safety does not imply constant-time execution nor the converse (see Appendix A.1 for details). Moreover, SecVerilog, Caisson, and GLIF take a language-design approach whereas we take an analysis-centric view that is more suitable for verifying *existing* hardware designs. Thus, we see our work as largely complementary. Indeed, it may be useful to use IODINE alongside these HDLs to verify constant-time execution for parts of the hardware that handle secret data only, and are thus not checked for timing variability, thereby extending their attacker model.

Combining Hardware & Software Mitigations. HyperFlow [31] and GhostRider [43], take hardware/software co-design approach to eliminating timing channels. Zhang et al. [54] present a method for mitigating timing side-channels in software and give conditions on hardware that ensure the validity of mitigations is preserved. Instead of eliminating timing flows all together, they specify quantitative bounds on leakage and offers primitives to mitigate timing leaks through padding. Many other tools [11, 13, 30, 38, 48] automatically quantify leakage through timing and cache side-channels. Our approach is complementary and focuses on clock-precise analysis of existing hardware. However, the explicit assumptions that IODINE needs to verify constant-time behavior can be used to inform software mitigation techniques.

References

[1] https://github.com/scarv/xcrypto-ref.

[2] ARM A64 instruction set architecture. https://static.docs.arm.com.

[3] https://github.com/dawsonjon/fpu.

[4] https://github.com/fatestudio/RSA4096.

[5] https://github.com/gokhankici/iodine.

[6] https://github.com/monajalal/fpga_-mc/tree/master/fpu.

[7] https://github.com/tommythorn/yarvi.

[8] https://opencores.org/project/sha_core.

[9] Icarus verilog. http://iverilog.icarus.com/.

[10] Liquid fixpoint. https://github.com/ucsd-progsys.

[11] TIS-CT. http://trust-in-soft.com/tis-ct/.

[12] *IEEE Standard for Verilog Hardware Description Language.* IEEE Std 1364-2005, 2005.

[13] J Bacelar Almeida, Manuel Barbosa, Jorge S Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. In *Science of Computer Programming*, 2013.

[14] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, 2017.

[15] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time mee-cbc. In *FSE*, 2016.

[16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, 2016.

[17] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[18] Marc Andrysco, Andres Noetzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *CCS*, 2018.

[19] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In *DAC*, 2012.

[20] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, 2014.

[21] Gilles Barthe, Juan Manuel Crespo, and Cesar Kunz. Relational verification using product programs. In *FM*, 2011.

[22] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSF*, 2004.

[23] Daniel J. Bernstein. The poly1305-aes message-authentication code. In *Fast Software Encryption*, 2005.

[24] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography*, 2006.

[25] Daniel J Bernstein. The salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 2008.

[26] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation*. 2015.

[27] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[28] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.

[29] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[30] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *USENIX Security*, 2013.

[31] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security. In *SIGSAC*, 2018.

[32] Michael J. C. Gordon. The semantic challenge of verilog hdl. In *LICS*, 1995.

[33] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[34] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, 2018.

[35] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[36] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, 2017.

[37] Hyoukjun Kwon, William Harris, and Hadi Esameilzadeh. Proving flow security of sequential logic via automatically-synthesized relational invariants. In *CSF*, 2017.

[38] Adam Langley. ctgrind: Checking that functions are constant time with valgrind. https://github.com/agl/ctgrind/.

[39] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006.

[40] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *PLDI*, 2011.

[41] Linux on ARM. ARM64 prepping ARM v8.4 features, KPTI improvements for Linux 4.17. https://www.linux-arm.info/.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[43] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Notices*, 2015.

[44] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. In *IFIP*, 2010.

[45] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communicationsof the ACM*, 1976.

[46] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *USENIX Security*, 2016.

[47] Oscar Reparaz, Joseph Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, 2017.

[48] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CCC*, 2016.

[49] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, 2016.

[50] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[51] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ISCA*, 2011.

[52] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Sigplan Notices*, 2009.

[53] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. 2019.

[54] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[55] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.

A Appendix

A.1 Comparison to Information Flow

In this section, we discuss the relationship between constant time execution and information flow checking. Information flow safety (IFS) and constant time execution (CTE) are *incomparable*, *i.e.*, IFS does not imply CTE, and vice versa. We illustrate this using two examples: one is information flow safe but does not execute in constant time and one executes in constant time but is not information flow safe.

Figure 17 contains example program Ex2 which is information flow safe but not constant time. The example contains three registers that are typed high as indicated by the annotation H, and one register that is typed low as indicated by the annotation L. The program is information flow safe, as there are no flows from high to low. Indeed, SecVerilog [55] type checks this program.

This program, however, is not constant time when $\mathsf{slow}_L \neq \mathsf{slow}_R$. This does not mean that the program leaks high data to low sinks—indeed it does not. Instead, what this means is that the high computation takes a variable amount of time dependent on the secret input values. In cases like crypto cores where the attacker has a stop watch and can measure the duration of the sensitive computation, it's not enough to be information flow safe: we must ensure the core is constant-time.

Next, consider Figure 18 that contains program Ex3 which executes in constant time but is not information flow safe. Ex3 violates information flow safety by assigning high input sec to low output out. The example however executes constant time with source in and sink out under the assumption that + does not contain asynchronous assignments.

```
1  // source(in_low); source(in_high);
2  // sink(out_low); sink(out_high);
3  module test(input  {L} clk,
4              input  {L} in_low,
5              input  {H} in_high,
6              output {L} out_low,
7              output {H} out_high);
8    reg {H} flp_res;
9    reg {H} slow;
10   reg {L} out_low;
11   reg {H} out_high;
12   always @(posedge clk) begin
13     out_low <= in_low;
14     flp_res <= in_hi;
15     if (slow)
16       out_hi <= flp_res;
17     else
18       out_hi <= in_hi;
19   end
20 endmodule
```

Figure 17: Ex2: Non-constant time but info-flow safe.

```
1  // source(in); sink(out);
2  // □ (slow_L = slow_R);
3  reg {L} in;
4  reg {L} out;
5  reg {H} sec;
6  always @(posedge clk) begin
7    out <= in + sec;
8  end
```

Figure 18: Ex3: Constant time but not info-flow safe.

A.2 Translation

In Figure 19, we define a relation $\Rightarrow$ that translates Verilog programs into VInter programs. The relation is given in terms of inference rules where a transition step in the rule's conclusion (below the line) is applicable only if all its preconditions (above the line) are met. Both always- and assign-blocks are translated into threads that are executed at every clock tick using withclock. Each process is given a unique id. Our translation does not distinguish between posedge and negedge events thereby relaxing the semantics by allowing them to occur in any order. assign blocks are transformed into threads executing a continuous assignment. Blocking and non-blocking assignments remain unchanged.

$$\frac{P \Rightarrow P' \qquad Q \Rightarrow Q'}{P \cdot Q \Rightarrow P' \parallel Q'} \qquad \frac{s_1 \Rightarrow s_1' \qquad \ldots \qquad s_n \Rightarrow s_n'}{\textsf{begin } s_1; \ldots; s_n; \textsf{ end} \Rightarrow s_1'; \ldots; s_n'}$$

$$\frac{s \Rightarrow s' \qquad id \text{ fresh}}{\textsf{always @}(\_)\ s \Rightarrow \textsf{repeat } [s']_{id}}$$

$$\frac{id \text{ fresh}}{\textsf{assign } v = e \Rightarrow \textsf{repeat } [v := e]_{id}}$$

$$\frac{s_1 \Rightarrow s_1' \qquad s_2 \Rightarrow s_2'}{\textsf{if } (e)\ s_1 \textsf{ else } s_2 \textsf{ end} \ \Rightarrow\ \textsf{ite}(e, s_1', s_2')}$$

Figure 19: Translation from VERILOG to VINTER.

```
always @(*)
   case({opa[31], opb[31]})
        2'b0_0:  sign_mul_r <= 0;
        2'b0_1:  sign_mul_r <= 1;
        ...
   endcase
assign sign_mul_final =
  (sign_exe_r &
   ((opa_00 & opb_inf) |
    (opb_00 & opa_inf))) ?
   !sign_mul_r : sign_mul_r;
always @(posedge clk)
  out <= {
   (((fpu_op_r3 == 3'b101) & out_d_00) ?
    (f2i_out_sign & !(qnan_d | snan_d)) :
    (((fpu_op_r3 == 3'b010) &
      !(snan_d | qnan_d)) ?
     sign_mul_final :
     (((fpu_op_r3 == 3'b011) &
       !(snan_d | qnan_d)) ? sign_div_final :
      ((snan_d | qnan_d | ind_d) ?
       nan_sign_d :
       (output_zero_fasu ?
        result_zero_sign_d :
        sign_fasu_r))))) ,
   ((mul_inf | div_inf |
     (inf_d & (fpu_op_r3 != 3'b011) &
      (fpu_op_r3 != 3'b101)) |
     snan_d | qnan_d) &
    fpu_op_r3 != 3'b100 ? out_fixed :
out_d) };
```

Figure 20: Example diverging computation in [6]