UNIVERSITY OF CALIFORNIA SAN DIEGO

Verifying Constant-Time Execution of Hardware

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Rami Gökhan Kıcı

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Sorin Lerner
Professor Deian Stefan

2020

The Dissertation of Rami Gökhan Kıcı is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2020

DEDICATION

For Mom, Dad, and Nergis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank everyone who made this PhD possible. I especially want to thank my parents and my sister, for all of their relentless love, support and advice throughout the last six years. A special thank you to my advisor, Ranjit Jhala; my co-authors Klaus v. Gleissenthall, Alexander Bakst, Dimitar Bounov, and Deian Stefan.

Chapter 1, in part, has been submitted for publication of the material as it may appear in the 26^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), 2021, Kıcı, Rami Gökhan; v. Gleissenthall, Klaus; Stefan, Deian; Jhala; Ranjit, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in full, is a reprint of the material as it appears in the 28^{th} USENIX Conference on Security Symposium. v. Gleissenthall, Klaus; Kıcı, Rami Gökhan; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in the 28^{th} USENIX Conference on Security Symposium. v. Gleissenthall, Klaus; Kıcı, Rami Gökhan; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been submitted for publication of the material as it may appear in the 26^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), 2021, Kıcı, Rami Gökhan; v. Gleissenthall, Klaus; Stefan, Deian; Jhala; Ranjit, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, has been submitted for publication of the material as it may appear in the 26^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), 2021, Kıcı, Rami Gökhan; v. Gleissenthall, Klaus; Stefan, Deian; Jhala; Ranjit, 2021. The dissertation author was the primary investigator and author of this paper.

VITA

2014	Bachelor of Science, Computer Engineering, Boğaziçi University
2017	Master of Science, Computer Science, University of California San Diego
2014–2020	Research Assistant, University of California San Diego
2020	Doctor of Philosophy, Computer Science, University of California San Diego

PUBLICATIONS

Rami Gökhan Kıcı, Klaus v. Gleissenthall, Deian Stefan, and Ranjit Jhala. Solver-Aided Constant-Time Hardware Verification. Submitted to *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21).*

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Conference on Security Symposium (SEC'19).* pages 1411–1428, 2019

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. In *Proc. ACM Program. Lang. 3, POPL, Article 59 (January 2019), 30 pages.*

Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *Proc. ACM Program. Lang. 1, OOPSLA, Article 110 (October 2017), 27 pages.*

Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*

ABSTRACT OF THE DISSERTATION

Verifying Constant-Time Execution of Hardware

by

Rami Gökhan Kıcı

Doctor of Philosophy in Computer Science

University of California San Diego, 2020

Professor Ranjit Jhala, Chair

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*. While there have been significant strides in verifying constant time execution for software, these efforts focus on sequential, cryptographic code. Unfortunately, this makes them unsuitable for hardware designs which are inherently *concurrent* and *long-lived*.

First, we present IODINE: a clock-precise, constant-time approach to eliminating timing

side channels in hardware. To realize IODINE, we first define a new notion of constant-time execution that is suitable for concurrent and long lived computations. Our definition is based on the notion of *influence sets* containing all cycles whose inputs influenced the current computation. We then show how to reduce the problem of verifying constant time execution to the standard problem of verifying assertion validity.

Second, we present XENON, which extends IODINE and scales to realistic hardware designs by exploiting modularity in VERILOG code via a notion of module summaries. XENON drastically reduces the effort needed to localize the causes of verification failure via a novel constant-time counterexamples which are used to automatically synthesize minimal secrecy assumptions that enable constant-time verification. We show how XENON's summaries and assumption synthesis enable the verification of a variety of circuits including a highly modular AES-256 implementation where modularity cuts verification from six hours to under three seconds, and ScarV, a timing channel hardened RISC-V micro-controller whose size exceeds previously verified designs by an order of magnitude.

We find that IODINE and XENON present a practical way to specify and verify the absence of timing channels in hardware. They succeed in verifying various open source hardware designs in seconds and with little developer effort thanks to generated secrecy assumptions. They also discovered two constant-time violations: one in a floating-point unit and another one in an RSA encryption module.

Chapter 1

Introduction

Timing side-channel attacks are no longer theoretical curiosities. Over the last two decades, timing side-channel attacks have been used to break implementations of cryptographic primitives ranging from public-key encryption algorithms [36, 67, 105], to block ciphers [32, 80], digital signature schemes [79], zero-knowledge proofs [44], and pseudorandom generators [43]. This, in turn, has allowed attackers to break systems that rely on these primitives for security – to, for example, steal TLS keys used to encrypt web traffic [36, 43, 105], snoop and forge virtual private network traffic [79], and extract information from trusted execution environments [35, 43, 44, 104]. Worse, these attacks are not limited to cryptography. Timing side-channel attacks have also been used to steal sensitive cross-site data in the browser [23, 69, 95], deanonymize differentially private databases [62], sniff user browsing history [34, 58, 92], and recover passwords from keystrokes [87, 93]. More recently, timing side-channels attacks have played a crucial role in making transient execution attacks practical, breaking fundamental software- and hardware-based isolation mechanisms [38, 66, 75, 89, 100].

The most robust approach to addressing timing side-channel attacks is to follow a discipline of *constant-time* or *data-oblivious* programming [2, 20, 28, 39, 45, 106]. This discipline comes down to ensuring that **(1)** the program control flow and memory access patterns do not depend on secret data and **(2)** secrets are not used as operands to variable-time instructions (*e.g.*, floating point operations like division [23, 25, 68, 83]).

Constant-Time Programs Need Constant-Time Hardware

The constant-time discipline critically requires that the underlying hardware preserves the constant-time property. In particular, we require that the circuit implementing a constant-time machine instruction yields outputs in the same number of clock cycles independent of its operands.

Unfortunately, simply assuming that hardware is constant-time doesn't work. Incorrect assumptions about the timing-variability of floating-point instructions, for example, allowed attackers to break the differentially private Fuzz database [62]. Attempts to address these attacks (*e.g.*, [84]) were also foiled: they relied on yet other incorrect microarchitectural assumptions (*e.g.*, the timing-variability of SIMD instructions) [68]. Yet more recently, hardware crypto co-processors (*e.g.*, Intel and STMicroelectronics's trusted platform modules) turned out to exhibit similar data-dependent timing variability [79].

Formal Verification of Constant-Time Hardware

One path towards eliminating timing side-channel attacks is to *formally verify* that our hardware preserves the constant-time property of the software it is executing. This is especially important as hardware designers are starting to design side-channel resistant processors [5, 53, 106] and mainstream instruction set architectures like ARM's AArch64 are promising constant-time instructions [26]. This verification requires new tools.

1.1 IODINE: Verifying Constant-Time Execution of Hardware

In this dissertation, we first introduce IODINE [99]: 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 [29]).

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 [108]. 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 [108]). 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 [1, 74].

While there have been significant strides in verifying the constant-time execution of software [18, 19, 21, 24, 29, 30, 31, 102], 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. Specification

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 [77]; 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 (§ 3.3). Many of our benchmarks are constant-time for intricate reasons (Section 3.3.3), *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 [36, 67].

In summary, Chapters 2 and 3 make the following contributions:

- First, we give a definition for constant-time execution of hardware, based on the notion of *influence sets* (Section 2.1). We formalize the semantics of VERILOG programs with influence sets (Section 2.2), and use this formalization to define constant-time execution with respect to usage assumptions (Section 2.3).

- Our second contribution is a reduction of constant-time execution to the easy-to-verify problem of liveness equivalence. We formalize this property (Section 2.3), prove its equivalence to our original notion of constant-time execution (Section 2.3.3), and show how to verify it using standard methods (Section 3.1).

- Our final contribution is an implementation and evaluation of IODINE on several challenging open source hardware designs (Section 3.3). Our evaluation shows that IODINE can be used to verify constant-time execution of existing hardware designs, rapidly, and with modest user effort.

1.2 XENON: Solver-Aided Constant-Time Hardware Verification

However, existing verification approaches including IODINE fail to scale to realistic hardware. This is because of two fundamental reasons. First, current methods fail to exploit the *modularity* that is already explicit at the register transfer level. Hence, they duplicate verification effort across replicated modules which leads to a blow up in verification time. Second, existing tools do not help when verification fails, as is inevitable, since hardware circuits are only constant-time under very specific secrecy assumptions that describe which port and wire values are public or secret. Currently, the user must undertake the tedious and time consuming task of determining and explicating these assumptions, which, in our experience, takes up the overwhelming majority of time spent on the verification effort.

In Chapters 4 and 5 , we present XENON, a solver-aided method for formally verifying that VERILOG hardware executes in constant-time. We develop XENON via four contributions.

1. Modular Verification

We introduce a notion of module summaries (Section 4.3) to succinctly capture the timing properties of a module's input and output ports at a given usage site. By abstracting inessential details about the exact computations performed by the module and focusing solely on its timing behavior, XENON produces fewer and more compact constraints. This, in turns, allows verification to scale to larger and more complex hardware.

2. Counterexamples

Constant-time verification is an interactive process that requires the user to intervene on failures – either to declare the circuit to be leaky (*i.e.*, variable-time) and stop, or specify additional assumptions under which the circuit should be constant-time. To help users understand failures, we introduce the notion of constant-time *counterexamples* (Section 5.2.1). A counterexample highlights the *earliest* point in the circuit where timing variability is introduced; this simplifies the task of understanding whether a circuit is variable-time by narrowing the user's

attention to the root cause of the failure (and a small fraction of the circuit).

3. Assumption Synthesis

Inevitably, hardware is only constant-time under certain assumptions. For example, XENON will find a constant-time counterexample for a processor pipeline where the two different runs may execute two different ISA instructions (*e.g.*, addition and division) which take different numbers of clock cycles. Yet the execution of each instruction (for any inputs) may be constant time. XENON thus uses counterexamples to synthesize a minimal set of *secrecy assumptions* (*e.g.*, that the two executions have the same, publicly visible sequence of instructions) that address the root cause of the verification failure (Section 5.2.2). This *solver-aided verification* approach takes advantage of both modern solvers' capabilities and users' understanding of a circuit to scale verification to realistic hardware, which neither the solver nor the user alone would have been able to verify.

4. Evaluation

We implement XENON (Section 4.5) and evaluate the impact of modularity, and counterexample and assumption synthesis on the verification of different kinds of hardware modules. We find that module summaries are key to reducing verification times for certain hardware designs (*e.g.*, for AES-256 crypto core summaries reduced the verification time from six hours to a three seconds). We similarly find that XENON's counterexample synthesis dramatically reduces the number of potential error locations users have to manually inspect (6% of its original size) and, for constant-time hardware, that most of XENON's assumption suggestions are useful (on average 81.67%). Overall, we find that XENON's solver-aided verification process drastically reduces overall verification effort (*e.g.*, manually verifying the largest benchmark of [99] took us several minutes instead of days [65]) and is key to scaling verification to realistic hardware (*e.g.*, the SCARV side-channel hardened RISC-V core [5] we verify is an order of magnitude larger than the RISC-V cores verified by previous state-of-the-art tools).

Chapter 2

Specification of Constant-Time Hardware

2.1 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 [23, 68]). 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. Figure 2.1 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 section 3.3.3), 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

```
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 2.1. Floating point multiplier (EX1).

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 Figure 2.1, 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

[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**
	$\mid [\mathsf{s}]_{id}$	*process*
	$\mid P \parallel P$	*parallel composition*
	$\mid \mathsf{repeat}\ P$	*sync. iteration*
	$\mid \varepsilon$	*empty process*
s	::=	**Command**
	$\mid \mathsf{skip}$	*no-op*
	$\mid v = e$	*blocking*
	$\mid v \Leftarrow e$	*non-blocking*
	$\mid v := e$	*continuous*
	$\mid \mathsf{ite}(e, \mathsf{s}, \mathsf{s})$	*conditional*
	$\mid \mathsf{s}_1 ; \ldots ; \mathsf{s}_k$	*sequence*
	$\mid a$	*annotation*
e	::=	**Expression**
	$\mid v$	*variables*
	$\mid n$	*constants*
	$\mid f(e_1, \ldots, e_k)$	*function literal*

Figure 2.2. Syntax for intermediate language VINTER.

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 § 2.1.1, liveness equivalence in § 2.1.2, 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 in § 2.1.3.

2.1.1 Constant-Time For Hardware

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

Assumptions and Attacker Model

Like SecVerilog [108], 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 Section 3.4, they are not for free.) For example,

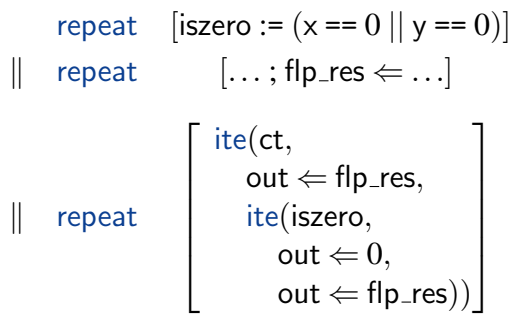


Figure 2.3. EX1 written in VINTER

assuming a single fixed-rate clock, allows us to translate VERILOG programs, such as our FPU multiplier to a more concise representation shown in Figure 2.3.

Intermediate Language

In this language—called VINTER—VERILOG always- and assign-blocks are represented as concurrent *processes*, wrapped inside an infinite repeat-loop. As Figure 2.2 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.*, [21, 29]).

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 micro-architectural 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`

Cycle #	x	y	ct	fr	out	θ(x)	θ(y)	θ(ct)	θ(fr)	θ(out)
0	0	1	**F**	**X**	**X**	$\{0\}$	$\{0\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	0	1	F	X	0	$\{1\}$	$\{1\}$	$\emptyset$	$\emptyset$	$\{0\}$
⋮										
k-1	0	1	F	0	0	$\{k-1\}$	$\{k-1\}$	$\emptyset$	$\{0\}$	$\{k-2\}$
k	0	1	F	0	0	$\{k\}$	$\{k\}$	$\emptyset$	$\{1\}$	$\{\mathbf{k-1}\}$

Figure 2.4. 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 2.5. 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**	$\{0\}$	$\{0\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	1	1	F	X	**X**	$\{1\}$	$\{1\}$	$\emptyset$	$\emptyset$	$\{0\}$
⋮										
k-1	1	1	F	1	X	$\{k-1\}$	$\{k-1\}$	$\emptyset$	$\{0\}$	$\{k-2\}$
k	1	1	F	1	1	$\{k\}$	$\{k\}$	$\emptyset$	$\{1\}$	$\{\mathbf{0},\mathbf{k-1}\}$

Figure 2.5. 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. 2.4, except for cycle k, where out's influence set contains the additional value 0, thereby violating our definition of constant-time execution.

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-set* $\theta(x)$. 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 Figure 2.4 and Figure 2.5, 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

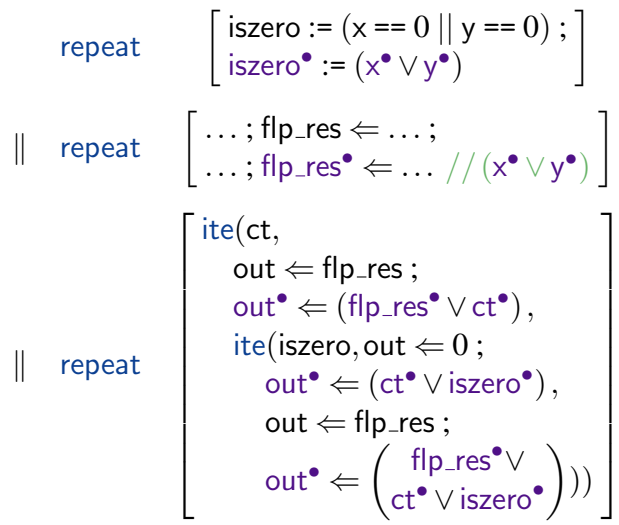


Figure 2.6. EX1, after we propagate liveness using a standard taint-tracking inline monitor.

Figure 2.4, we set x to 0 to trigger the fast path in Figure 2.5 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 Figure 2.4, out is only influenced by the input issued in the last cycle, through a control dependency on iszero. In the execution in Figure 2.5, 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 $\theta(\text{out})$ differing between two runs reflects the influence of data on the duration of the computation.

2.1.2 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

	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 2.7. 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 2.8. Values that stayed the same in the next cycle are shaded.

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 $t \in \theta(x)$. 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•) that represents its liveness; a shadow variable x• is set to L if x is live and D (dead) otherwise. [2] We then propagate liveness using a standard taint-tracking inline monitor [77] shown in Figure 2.6. 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. Figure 2.7 and 2.8 show the state

[2] For liveness-bits x• and y•, we define a join operator $\vee$, such that x• $\vee$ y• is L, if x• or y• is L and D, otherwise.

	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 2.8. Execution of Ex1•, where both x = 1 and y = 1. The liveness bits are the same as in 2.7, 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.

of the different registers and wires for these runs. In both runs, out is live at cycle 1—due to a control dependency in Figure 2.7, due a direct assignment in Figure 2.8. But, in the latter, out is *also* live at the *k*th 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.

2.1.3 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 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 [21], 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* [31] – consists of two disjoint copies of the original program.

Race-Freedom

Our product construction exploits the fact thatVERILOG 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

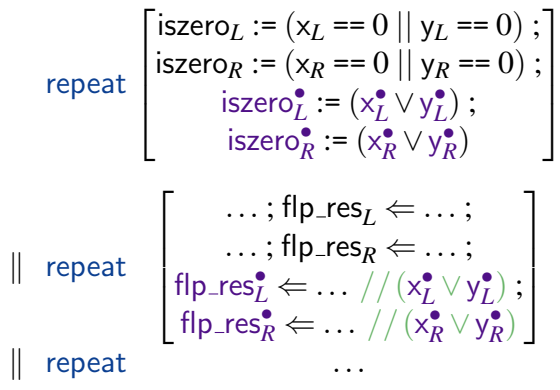


Figure 2.9. Per-process product form of EX1.

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 2.6 into the *per-process product program* shown in Figure 2.9.

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 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

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

synthesize process-modular invariants [81] that imply the constant-time assertion (Section 3.1).

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

2.2 Syntax and Semantics

Since VERILOG's execution model can be subtle [16], we formally define syntax and semantics of the VERILOG fragment considered in this paper.

2.2.1 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.

2.2.2 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* [59] in which the program is "flattened" by removing module instantiation through in-lining. We provide VERILOG syntax and a translation to VINTER in Section 2.5, but define semantics in VINTER (Figure 2.2).

Annotations

We define annotations in Figure 2.10. Let *Regs* denote the set of registers and *Wires* the set of wires and let VARS denote their disjoint union, *i.e.*, $\text{VARS} \triangleq Regs \uplus Wires$. For a register $v \in Regs$, annotations $\mathsf{source}(v)$ and $\mathsf{sink}(v)$ designate v as source or sink, respectively.[4]

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

$a ::=$		**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 2.10. 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*	$store(\pi, i)$	σ_i
P	*current program*	$live(\pi, i)$	τ_i
I	*initial program*	$inf(\pi, i)$	θ_i
c	*clock cycle*	$clk(\pi, i)$	c_i
		$reset(\pi, i)$	b_i

Figure 2.11. Configuration and trace syntax.

We let IO $\triangleq (Src, Sink)$ 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 $\text{A} \triangleq (\text{INIT}, \text{ALWAYS})$, such that INIT contains all formulas under init and ALWAYS all formulas under $\Box$.

2.2.3 Semantics

Values

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

Configurations

The program state is represented by a *configuration* $\Sigma \in Configs$. Figure 2.11 shows

$$\text{[VAR]}\quad \frac{}{v,\sigma,\tau,\theta \dashrightarrow \sigma(v),\tau(v),\theta(v)} \qquad \text{[CONST]}\quad \frac{}{n,\sigma,\tau,\theta \dashrightarrow n,D,\emptyset}$$

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

Figure 2.12. Expression evaluation.

the components of a configuration. A store $\sigma \in \textsc{Stores} \triangleq (\textsc{Vars} \mapsto \textsc{Vals})$ is a map from registers and wires to values. A *liveness map* $\tau \in \textsc{LiveMap} \triangleq (\textsc{Vars} \mapsto \{L,D\})$ is a map from registers and wires to liveness bits. A *influence map* $\theta \in \textsc{InfMaps} \triangleq (\textsc{Vars} \mapsto \mathscr{P}(\mathbb{Z}))$ 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 $\mu \in \mathit{PIDs} \mapsto (\textsc{Vars} \times \textsc{Vals} \times \{L,D\} \times \mathscr{P}(\mathbb{Z}))^*$ is a map from process identifier to a sequence of variable/value/liveness-bit/influence set tuples. An *event set* $ev \in \mathscr{P}(\textsc{Vars})$ 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 $\dashrightarrow \in (\textsc{Expr} \times \textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps}) \mapsto (\textsc{Vals} \times \{L,D\} \times \mathscr{P}(\mathbb{Z}))$ that computes value, liveness-bit, and influence map for an expression. We define the relation through the inference rules shown in Figure 2.12. 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 $\rightsquigarrow_{\mathrm{P}}$

The per-process transition relation $\rightsquigarrow_{\mathrm{P}}$ describes how to step along individual processes. It is defined in Figure 2.13. 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 $P \parallel Q$ and $Q \parallel P$.

Non-blocking Transition $\rightsquigarrow_{\mathrm{N}}$

Transition relation $\rightsquigarrow_{\mathrm{N}}$ applies deferred non-blocking assignments. It is defined by a single rule [NB-APP] shown in Figure 2.13. 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 $\rightsquigarrow_{\mathrm{C}}$

Relation $\rightsquigarrow_{\mathrm{C}}$ specifies how to execute continuous assignments. It is described by rule [C-ASN] in Figure 2.13, 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 $\rightsquigarrow_{\mathrm{G}}$

Finally, global transition relation $\rightsquigarrow_{\mathrm{G}}$ is defined by rules [NEWCYCLE] and [NEWCYCLE-ISSUE] shown in Figure 2.13. [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 $\textsc{Repeat}(P) \in \mathscr{P}(\mathit{Progs})$ denote the set of processes that occur under `repeat`. For a set of programs S, we let $\sqcap S$ 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* $l \in \mathit{Labels} \triangleq ((\textsc{Stores} \times \textsc{LiveMap} \times \textsc{InfMaps} \times \mathbb{N} \times \{L, D\}) \uplus \varepsilon)$ 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 $\rightsquigarrow_{\mathrm{C}}$) can be applied, that step is taken. Whenever no continuous assignment step can be applied, however, a *per-process* step (relation $\rightsquigarrow_{\mathrm{P}}$) can be applied, a $\rightsquigarrow_{\mathrm{P}}$ step is taken. If no continuous assignment and process local steps can be applied, however, an *non-blocking assignment* step (relation $\rightsquigarrow_{\mathrm{N}}$) is applicable, a $\rightsquigarrow_{\mathrm{N}}$ 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 $\rightsquigarrow_{\mathrm{G}}$). Our overall transition relation closely follows the Verilog simulation reference model from Section 11.4 of the standard [16].

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 ε, 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 $\textsc{Traces}(P) \in \mathscr{P}(\mathit{Labels}^*)$ to denote the set of its traces, *i.e.*, all traces with initial program P.

2.3 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.

2.3.1 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 $\mathrm{A} \triangleq (\textsc{Init}, \textsc{Always})$, 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 $\varphi_A \in \textsc{Always}$, 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 $0 \leq i \leq n-1$. Intuitively, pairs of traces that satisfy the assumptions are "low" or "input"

[SEQ-STEP]
$$\frac{\langle \sigma, \mu, \theta, ev, \tau, \mathsf{s}_1, I, c\rangle \rightsquigarrow_{\mathrm{P}} \langle \sigma', \mu', \theta', ev', \tau', \mathsf{s}_1', I, c\rangle}{\langle \sigma, \mu, \theta, ev, \tau, [\mathsf{s}_1; \mathsf{s}_2], I, c\rangle \rightsquigarrow_{\mathrm{P}} \langle \sigma', \mu', \theta', ev', \tau', [\mathsf{s}_1'; \mathsf{s}_2], I, c\rangle}$$

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

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

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

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

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

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

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

Figure 2.13. Per-thread transition relation $\rightsquigarrow_{\mathrm{P}}$, non-blocking transition relation $\rightsquigarrow_{\mathrm{N}}$, continuous transition relation $\rightsquigarrow_{\mathrm{C}}$, and global restart relation $\rightsquigarrow_{\mathrm{G}}$.

equivalent.

Constant Time Execution

For a program P, assumptions A and traces $\pi_L, \pi_R \in \text{TRACES}(P)$ 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.*, $inf(\pi_L, i)(v) = inf(\pi_R, i)(v)$, for $0 \leq i \leq n-1$ and all $v \in Sink$, 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.

2.3.2 Liveness Equivalence

t-Trace

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

Liveness Equivalence

For a program P, let $\pi_L, \pi_R \in \text{TRACES}(P)$, 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 $live(\pi_L, i)(v) = live(\pi_R, i)(v)$, for $0 \leq i \leq n-1$ and all $v \in Sink$. 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.

2.3.3 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 $0 \leq i \leq n-1$, and variable v, if v is $t-$live,* i.e.*, $live(\pi, i)(v) = L$, then t is in v's influence map,* i.e.*, $t \in inf(\pi, i)(v)$.*

We can now state our proof for theorem 1.

```
// source(in_low); source(in_high);
// sink(out_low); sink(out_high);
module test(input  {L} clk,
            input  {L} in_low,
            input  {H} in_high,
            output {L} out_low,
            output {H} out_high);
  reg {H} flp_res;
  reg {H} slow;
  reg {L} out_low;
  reg {H} out_high;
  always @(posedge clk) begin
    out_low <= in_low;
    flp_res <= in_hi;
    if (slow)
      out_hi <= flp_res;
    else
      out_hi <= in_hi;
  end
endmodule
```

Figure 2.14. EX2: Non-constant time but info-flow safe.

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 $\pi_L^*, \pi_R^* \in \text{TRACES}(P)$ that satisfy A, and some index i^* such that $inf(\pi_L^*, i^*)(v^*) \neq inf(\pi_R^*, i^*)(v^*)$, and therefore without loss of generality, there is a cycle t^*, such that $t^* \in inf(\pi_L^*, i^*)(v^*)$ and $t^* \notin inf(\pi_R^*, i^*)(v^*)$. 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.*, $live(\hat{\pi}_L, i^*)(v^*) = live(\hat{\pi}_R, i^*)(v^*) = L$ and, by lemma 1, $t^* \in inf(\hat{\pi}_R, i^*)(v^*)$. Since $\hat{\pi}_R$ and π_R^* only differ in their liveness map, this implies $t^* \in inf(\pi_R^*, i^*)(v^*)$, from which the contradiction follows. □

```
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 2.15. EX3: Constant time but not info-flow safe.

2.4 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 2.14 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 [108] 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 2.15 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

```
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 2.16. Example diverging computation in [3]

under the assumption that + does not contain asynchronous assignments.

2.5 Translation

In Figure 2.17, 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

$$\frac{P \Rightarrow P' \qquad Q \Rightarrow Q'}{P \cdot Q \Rightarrow P' \parallel Q'} \qquad \frac{\mathsf{s}_1 \Rightarrow \mathsf{s}_1' \qquad \ldots \qquad \mathsf{s}_n \Rightarrow \mathsf{s}_n'}{\mathsf{begin}\ \mathsf{s}_1; \ldots; \mathsf{s}_n;\ \mathsf{end} \Rightarrow \mathsf{s}_1'; \ldots; \mathsf{s}_n'} \qquad \frac{\mathsf{s} \Rightarrow \mathsf{s}' \qquad id \text{ fresh}}{\mathsf{always}\ @(\_)\ \mathsf{s} \Rightarrow \mathsf{repeat}\ [\mathsf{s}']_{id}}$$

$$\frac{id \text{ fresh}}{\mathsf{assign}\ v = e \Rightarrow \mathsf{repeat}\ [v := e]_{id}} \qquad \frac{\mathsf{s}_1 \Rightarrow \mathsf{s}_1' \qquad \mathsf{s}_2 \Rightarrow \mathsf{s}_2'}{\mathsf{if}\ (e)\ \ \mathsf{s}_1\ \mathsf{else}\ \mathsf{s}_2\ \mathsf{end} \ \Rightarrow\ \mathsf{ite}(e, \mathsf{s}_1', \mathsf{s}_2')}$$

Figure 2.17. Translation from VERILOG to VINTER.

order. assign blocks are transformed into threads executing a continuous assignment. Blocking and non-blocking assignments remain unchanged.

2.6 Acknowledgements

This chapter, in full, is a reprint of the material as it appears in the 28^{th} USENIX Conference on Security Symposium. v. Gleissenthall, Klaus; Kıcı, Rami Gökhan; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Verification of Constant-Time Hardware

3.1 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* [33, 60] 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 [11], 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 section 3.3 which predicates were used in our benchmarks.

3.2 Generating Horn Clause Constraints

Now, we will describe how the algorithm works by looking at the five types of Horn clause constraints that it generates for every process p_i. Algorithm 1 below contains the IODINE algorithm described in the previous section in pseudocode format.

Algorithm 1. Algorithm IODINE

Input: VINTER program P, taints T, and assumptions A
Output: SAFE if P is constant-time, or UNSAFE otherwise

```
1:  procedure IODINE(P, T, A)
2:      ps ← processes(P)
3:      inits ← {INIT(p, A) | p ∈ ps}
4:      rs ← {RESET(p, T) | p ∈ ps}
5:      trs ← {STEP(p, A) | p ∈ ps}
6:      is ← {INTER(p1, p2, A) | p1, p2 ∈ ps ∧ p1 ≠ p2}
7:      cs ← {CT-CONSTRS(p, T) | p ∈ ps}
8:      hs ← inits ∪ rs ∪ trs ∪ is ∪ cs
9:      if SOLVE(hs) then
10:         return SAFE
11:     else
12:         return UNSAFE
13:     end if
14: end procedure
```

Initial State

In the initial state, we set all the taint bits to D, and assume that the variables x used in

$\mathsf{init}(x)$ and $\Box(x)$ assumptions are equal:

$$\textsc{Init}(p_i, A) \triangleq \left(\bigwedge_{\mathsf{init}(x)\in A} x_L = x_R\right) \wedge \left(\bigwedge_{\Box(x)\in A} x_L = x_R\right) \wedge {v^\bullet}_L = D \wedge {v^\bullet}_R = D \rightarrow \mathsf{inv}_i(v_L, {v^\bullet}_L, v_R, {v^\bullet}_R)$$

Taint Reset

To implement the taint reset logic for the process p_i, we generate an Horn clause that assumes the existence of inv_i and either (1) sets the taint bits of the source variables to L while resetting the rest to D, or (2) resetting only the taint bits of the source variables while keeping the rest the same:

$$\textsc{Reset}(p_i, T) \triangleq \mathsf{inv}_i(v_L, {v^\bullet}_L, v_R, {v^\bullet}_R) \wedge \left(\begin{array}{l}\left(\begin{array}{l}\bigwedge_{x\in srcs} {x^\bullet}'_L = {x^\bullet}'_R = L \wedge \\ \bigwedge_{x\in rest} {x^\bullet}'_L = {x^\bullet}'_R = D\end{array}\right) \vee \\ \left(\bigwedge_{x\in srcs} {x^\bullet}'_L = {x^\bullet}'_R = D\right)\end{array}\right) \rightarrow \mathsf{inv}_i(v_L, {v^\bullet}'_L, v_R, {v^\bullet}'_R)$$

where $srcs = \{x \mid \mathsf{source}(x) \in T\}$ and *rest* are the variables that are not in *srcs*.

Transition Relation

Taking a step in the transition relation is simply done by updating the variables of inv_i according to next_i, while considering only the assumptions of the form $\Box(x)$:

$$\begin{aligned}\textsc{Step}(p_i, A) \triangleq \mathsf{inv}_i(v_L, {v^\bullet}_L, v_R, {v^\bullet}_R) \wedge \left(\bigwedge_{\Box(x)\in A} x_L = x_R\right) \wedge \\ \mathsf{next}_i(v_L, {v^\bullet}_L, v'_L, {v^\bullet}'_L) \wedge \mathsf{next}_i(v_R, {v^\bullet}_R, v'_R, {v^\bullet}'_R) \rightarrow \\ \mathsf{inv}_i(v'_L, {v^\bullet}'_L, v'_R, {v^\bullet}'_R)\end{aligned}$$

Interference

Since the processes share variables between each other, we also have to check whether for a given pair (p_i, p_j) of processes, the invariant inv_j of the process p_j is preserved when p_i executes.

Let $writes(p_i)$ be the set of variables that are updated by the process p_i, and $vars(p_i)$ be the set of variables that are read or written by p_i. Then, we can define $\textsc{Inter}(p_i, p_j, A)$ as the following:

$$\textsc{Inter}(p_i, p_j, A) \triangleq \begin{cases} s \neq \emptyset & check \\ \text{otherwise} & true \end{cases}$$

where $s = writes(p_i) \cap vars(p_j)$, and $check$ is defined as the following:

$$\begin{aligned} &\mathsf{inv}_i(v_L, v^\bullet{}_L, v_R, v^\bullet{}_R) \wedge \mathsf{inv}_j(w_L, w^\bullet{}_L, w_R, w^\bullet{}_R) \wedge \\ &\mathsf{next}_i(v_L, v^\bullet{}_L, v'_L, v^{\bullet\prime}{}_L) \wedge \mathsf{next}_i(v_R, v^\bullet{}_R, v'_R, v^{\bullet\prime}{}_R) \rightarrow \\ &\mathsf{inv}_j(w'_L, w^{\bullet\prime}{}_L, w'_R, w^{\bullet\prime}{}_R) \end{aligned}$$

where the variables $x \in w_L$ and $x^\bullet \in w^\bullet{}_L$ in inv_j are updated with $v'_L[x]$ and $v^{\bullet\prime}{}_L[x^\bullet]$ respectively when $x \in s$, or stay the same if $x \notin s$ (similarly for w_R and $w^\bullet{}_R$).

Taint Check

Finally, we check whether the taint bits of the sink variables are equal to each other in any two runs using the following Horn clause for every $\mathsf{sink}(x) \in T$:

$$\text{CT-}\textsc{Constrs}(p_i, T) \triangleq \mathsf{inv}_i(v_L, v^\bullet{}_L, v_R, v^\bullet{}_R) \rightarrow x^\bullet{}_L = x^\bullet{}_R$$

3.3 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.[1]

3.3.1 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 [27]).

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 Section 3.1, 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 [46] 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 [46] are part of our trusted computing base. This is similar to other constant-time and information flow tools (*e.g.*, SecVerilog [108] and ct-verif [21]). 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

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

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.

3.3.2 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 [6] that implements ($+, -, \times, \ll, \ldots$).

In our benchmarks, following our attacker model from Section 2.1.1, 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

Table 3.1. #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.

Name	**#LOC**	**#Assum**		**CT**	**Check** (s)
		#flush	**#always**		
MIPS [10]	434	31	2	✓	1.329
RISC-V [7]	745	50	19	✓	1.787
SHA-256 [13]	651	5	3	✓	2.739
FPU [3]	1182	0	0	✓	12.013
ALU [6]	913	1	5	✓	1.595
FPU2 [4]	272	3	4	✗	0.705
RSA [14]	870	4	0	✗	1.061
Total	5067	94	33	-	21.163

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 3.1, 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.

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 Section 2.1, 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 [36, 67].

(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 3.1 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.*, $\Box(x_L = x_R)$. 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 Section 3.3.3).
- **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.

3.3.3 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 examples from our benchmarks.

History Dependencies

```
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 3.1. Stalling in MIPS [10].

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 Figure 3.1 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 [21]. In hardware, this assumption is too restrictive. Consider the code snippet in Figure 3.2 taken from our constant time FPU benchmark (the full

```
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 3.2. Diverging control flow in FPU [3].

logic is shown in Figure 3.2 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 executions 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 3.3, 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

```
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 3.3. Update of CSRs in RISC-V [7].

an obligation for software mitigation mechanisms to ensure proper use of CSR registers.

3.4 Limitations

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 potentially 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.

3.5 Acknowledgements

This chapter, in full, is a reprint of the material as it appears in the 28^{th} USENIX Conference on Security Symposium. v. Gleissenthall, Klaus; Kıcı, Rami Gökhan; Stefan, Deian; Jhala, Ranjit. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Modular Verification

4.1 Verifying Constant Time Execution of Hardware

Lookup Circuit

Figure 4.2 shows the code for a VERILOG module, which implements a lookup table by case-splitting over the 8-bit input value. This module executes in constant time: even if `in` contains a secret value, producing `out` takes the same amount of time (one clock cycle), irrespective of the value of `in`, and therefore, an attacker cannot make any inference about the value of `in` by observing the timing of the computation.

Specifying Constant Time Execution

Figure 4.1 makes this intuition more precise. The Figure shows two executions of module *S*: one for input `8'h00` and one for input `8'hff`. We want to track how long it takes for the two inputs, issued at cycle 0 to pass through the circuit and produce their respective output. For this, we assign a color-bit to each register such that for some register `x`, its color-bit $\texttt{x}^\bullet$ is set to ★, if `x` has been influenced by the input at cycle 0 (in that case, we say that `x` is 0-live), or •, otherwise. Figure 4.1 shows the flow of colors through the circuit. Initially, in both runs, the input is 0-live and the output is not. In cycle 1, both outputs are affected by the input of cycle 0, due to the case-split on the value of `in`. Assuming that an attacker can observe the colors of all outputs, here, register `out`, the attacker cannot distinguish the two runs, and we can therefore conclude that the pair of runs is constant time.

c	i		o		$\mathtt{i}^\bullet$		$\mathtt{o}^\bullet$	
	L	R	L	R	L	R	L	R
0	h00	hff	**X**	**X**	★	★	•	•
1	h00	hff	h63	h2c	•	•	★	★

Figure 4.1. Two runs of Figure 4.2 showing values and color-bits input (i) and output (o) and clock cycle c. **X** represents an undefined value.

```
1  module S (clk, in, out);
2     input clk; [7:0] in;
3     output reg [7:0] out;
4     always @ (posedge clk)
5       case (in)
6         8'h00: out <= 8'h63;
7         ...
8         8'hff: out <= 8'h2c;
9       endcase
10 endmodule
```

Figure 4.2. A simple, constant-time lookup table in VERILOG.

Verifying Constant Time Execution

In order to prove constant time execution for the whole circuit, XENON proves that for *any* pair of execution, for any inputs (9'h00, 9'h01, ...), and any initial cycles (0, 2, 20 or $2 \cdot 10^9$), the constant time property holds. For this, XENON constructs a *product circuit* whose runs correspond to *pairs* of runs – called *left* and *right* – of the original circuit. In this product, each original variable x has two copies $\mathtt{x}_L$ and $\mathtt{x}_R$ that hold the values of x in the left and right runs, respectively. This allows us to express certain invariant properties of the circuit. For example, we say that a variable x is constant time (and write $ct(\mathtt{x})$), if for any pair of executions, its color in the left execution $\mathtt{x}_L^\bullet$ is always the same as its color in the right execution $\mathtt{x}_L^\bullet$, *i.e.*, $x_L^\bullet = x_R^\bullet$ always holds, for all initial cycles t. This allows us to express the following invariant on the module which proves constant time execution under the condition that module inputs are constant time: $ct(\mathtt{in}) \Rightarrow ct(\mathtt{out})$.

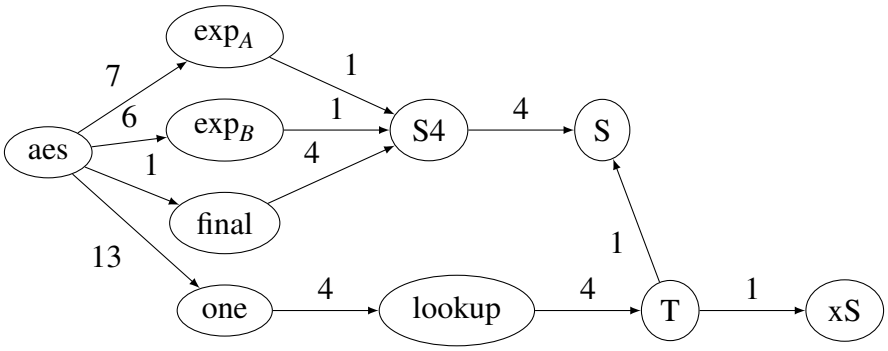


Figure 4.3. Module dependency graph of the AES-256 benchmark.

4.2 Real World Hardware is Not Small

Unfortunately, applying the above approach to larger, real world hardware designs poses two problems. The first problem is that computing invariants requires a *whole program* analysis. Hence, the efficiency of our prover crucially depends of the *size* of the circuit we are analyzing, and therefore verification might become prohibitively slow on large designs.

Consider, for example, the AES-256 benchmark from [12]. Figure 4.3 depicts the dependency graph of its modules, where each node m represents a Verilog module, and we draw an edge between modules m and n, if m instantiates n. Each edge is annotated with the number of instantiations. Even though there are only ten modules, the total number of module instantiations is 789. This, in turn, causes a blowup in the size of code XENON has to verify. Even though the sum of #LOC of the modules is only 856, inlining module instances causes this number to skyrocket to 135194 rendering verification all but intractable. (In fact, XENON does manage to verify the naive, inlined circuit, however, verification takes over 6 hours to complete). Fortunately, we can avoid this blowup by exploiting the modularity that is already apparent at the VERILOG level. We illustrate this process using module `S` from Figure 4.2.

Module Summaries

Since the value of `out` only depends on `in`, we can characterize its timing behavior as follows: the module output `out` is constant time, if module input `in` is constant-time. We can formalize this in the following module summary, which we XENON computes automatically:

$ct(\texttt{in}) \Rightarrow ct(\texttt{out})$. Instead of inlining the module, we can now use its summary thereby eschewing the code explosion and enabling efficient verification. The following code shows an instantiation of module `S` in module `S4`.

```
1  module S4 (clk, in, out);
2     input clk;
3     input [31:0] in;
4     output [31:0] out;
5       wire [7:0]    out_0, out_1, out_2, out_3;
6     S S_0 (clk, in[31:24], out_3),
7       S_1 (clk, in[23:16], out_2),
8       S_2 (clk, in[15:8],  out_1),
9       S_3 (clk, in[7:0],   out_0);
10      assign out = {out_3, out_2, out_1, out_0};
11 endmodule
```

Instead of inlining `S` at its four instantiation sites `S_0` to `S_3`, XENON uses the single module summary to compute a correctness proof, which only takes 3 seconds.

4.3 Modular Invariants

Let's see how XENON verifies constant time execution through modular summaries. We start by formally defining constant time execution (Section 4.3.1). Then, we show how to verify constant time execution by generating and solving a set of Horn constraints (Section 4.3.2), and finally, we explain how to scale the analysis up to large designs via module summaries (Section 4.3.3).

4.3.1 Defining Constant-Time Execution

Configurations

Configurations represent the sate of a VERILOG computation. A configuration $\Sigma \triangleq (P, \sigma, \theta, c, t, \text{SRC})$ is made up of a VERILOG program P, a store σ, a liveness map θ, a clock

cycle $c \in \mathbb{N}$, an initial clock cycle $t \in \mathbb{N}$ identifying the starting-cycle of the we want to track and, finally, a set of sources $\textsc{Src} \subseteq \textsc{Vars}$, identifying the inputs of the computation we are interested in. Store $\sigma \in \textsc{Vars} \to \mathbb{Z}$ maps variables $\textsc{Vars}$ (registers and wires) to their current values. Map $\theta \in \textsc{Vars} \to \{\bigstar, \bullet\}$ maps variables to liveness color-bits.

Transition relation

Transition relation $\rightsquigarrow \in (\Sigma \times \Sigma)$ defines how a configuration is updated from one clock-cycle to the next. We will omit its definition, as it is not needed for our purposes, but formal accounts can be found in [59, 99, 108]. In addition to updating the configuration according to VERILOG's semantics, the transition relation updates the liveness map θ by tracking which variables are currently influenced by the computation started in t. At initial cycle t, our transition relation starts a new computation by setting the color-bits of all variables in *Src* to $\bigstar$, and those of all others variables to $\bullet$.

Runs

We call a sequence of configurations $\pi \triangleq \Sigma_0 \Sigma_1 \ldots \Sigma_{n-1}$ a *run*, if each consecutive pair of configurations is related by the transition relation, *i.e.*, if $\Sigma_i \rightsquigarrow \Sigma_{i+1}$, for $i \in \{0, \ldots, n-2\}$. We call $\Sigma_0 \triangleq (P, \sigma_0, \theta_0, 0, t, \textsc{Src})$ initial state, and require that θ_0 maps all variables to $\bullet$. Finally, for a run $\pi \triangleq (P, \sigma_0, \theta_0, c_0, t, \textsc{Src}) \ldots (P, \sigma_{n-1}, \theta_{n-1}, c_{n-1}, t, \textsc{Src})$, we say that π is a run of P of length n with respect to t and $\textsc{Src}$ and let $\mathit{store}(\pi, i) = \sigma_i$ and $\mathit{live}(\pi, i) \triangleq \theta_i$, for $i \in \{0, \ldots, n-1\}$.

Flushed, Constant Time, Public

For two runs π_L and π_R of length n, we say that variable v is *flushed*, if $\mathit{store}(\pi_L, 0) = \mathit{store}(\pi_L, 0)$, we call v *public*, if $\mathit{store}(\pi_L, i) = \mathit{store}(\pi_R, i)$, for $i \in \{0, \ldots, n-1\}$ and finally, we call v *constant time*, if $\mathit{live}(\pi_L, i) = \mathit{live}(\pi_R, i)$, for $i \in \{0, \ldots, n-1\}$.

Assumptions

A set of assumptions $\mathscr{A} \triangleq (\textsc{Flush}, \textsc{Pub})$ consists of a set of variables $\textsc{Flush} \subseteq \textsc{Vars}$ that are assumed to be flushed in the initial state, and a set of variables $\textsc{Pub} \subseteq \textsc{Vars}$, that are assumed equal throughout. A pair of runs π_L and π_R of length n, satisfy a set of assumptions $\mathscr{A}$,

if, for each $v \in$ FLUSH, v is flushed, and for each v in PUB, v is public.

Constant Time Execution

We now define constant time execution with respect to a set of sinks SNK $\subseteq$ VARS, sources SRC, and assumptions $\mathscr{A}$. We say that a program P is constant time, if for any initial cycle t and any pair of runs π_L and π_R of P with respect to t and SRC of length n that satisfy $\mathscr{A}$ and any sink $o \in$ SNK, o is constant time.

4.3.2 Verifying Constant Time Execution via Constraints

To verify constant time execution, XENON mirrors our formal definition of constant time using Horn clauses [33], which declaratively describe a proof in the form of an invariant property of the product circuit. At high level, the constraints **(1)** issue a new live instruction at a non-deterministically chosen cycle t, and **(2)** ensure constant time execution by verifying that color-bits of all sinks are always the same. We construct the proof of constant time by finding an invariant property $inv(v_L, v_R, c, t)$, where v ranges over variables and liveness bits and c, and t are the current and initial cycles, respectively.

Initial States and Transition Relation

We first construct a logical formula $init(v_L, v_R, t)$ that requires all color-bits to be set to $\bullet$. To ensure that the proof holds for any initial cycle, *init* does not constrain t. Transition relation $next(v, v', c, t)$ describes how variables and color-bits v in the current clock cycle relate to variables and color-bits v' in the next clock cycle, *i.e.*, *next* is a logical formula describing $\rightsquigarrow$. Like $\rightsquigarrow$, *next* sets color-bits of all sources to $\bigstar$ and those of all other variables to $\bullet$, at clock cycle t. Importantly, constructing *next* requires inlining all modules and therefore can lead to large constraints that are beyond the abilities of the solver.

Assumptions

For a set of assumptions $\mathscr{A} \triangleq (\text{FLUSH}, \text{PUB})$, we construct two formulas $flush(v_L, v_R)$ and $pub(v_L, v_R)$, both of which require the variables in their respective sets to be equal in the two runs. That is, we let $flush \triangleq (\wedge_{x \in \text{FLUSH}}\, x_L = x_R)$ and $pub \triangleq (\wedge_{x \in \text{PUB}}\, x_L = x_R)$.

Horn Constraints & Solutions

We then require that the invariant holds, initially, where variables in FLUSH are assumed to be equal in both runs.

$$init(v_L, v_R, t) \wedge flush(v_L, v_R) \rightarrow inv(v_L, v_R, 0, t) \tag{init}$$

Next, we require that the invariant is preserved under the transition relation *next*, assuming that public variables are equal in both runs.

$$inv(v_L, v_R, c, t) \wedge \begin{pmatrix} next(v_L, v'_L, c', t) \\ \wedge\, next(v_R, v'_R, c', t) \\ \wedge\, pub(v_L, v_R) \end{pmatrix} \rightarrow inv(v'_L, v'_R, c', t) \tag{ind}$$

This constraints ensures that property *inv* is indeed an invariant of the circuit: if *inv* holds, then *inv* must also hold after both the left, and right copy in the product circuit execute a single cycle in lockstep. Together with eq. (init), this ensures that *inv* holds after *any* number of cycles. Finally, we require that the invariant implies that the color-bits of all sinks are the same, *i.e.*, we add the following constraint, for each $o \in$ SNK.

$$inv(v'_L, v'_R, c', t) \rightarrow o^{\bullet}_L = o^{\bullet}_R \tag{ct}$$

These constraints can be solved by an off-the-shelf solver yielding a formula which, when substituted for *inv*, makes all implications valid and thus proving constant time execution.

4.3.3 Finding Modular Invariants

Note that, constructing *next* requires that all the code be in a *single* module. Ultimately, large VERILOG designs *are* compiled down to a single module by *inlining* or *instantiating* all the modules in a bottom-up fashion. However, this can yield gigantic circuits, whose Horn clauses

are too large to analyze efficiently. A key insight of XENON is that instead of instantiating the *entire* module definition at a usage site, we need only instantiate a *summary* that describes the constant-time properties of the module's input and output ports and signals at that usage site. Crucially, by abstracting inessential details about the exact computations performed by the module, and focusing attention solely on the timing relationships, the summaries yield compact constraints and enable scalable verification. Next, we formalize this intuition and show how it can be implemented using Horn clauses.

Per-Module Invariants and Summaries

Instead of a single whole program invariant *inv*, the modular analysis requires a per-module invariant inv_m, and an additional module summary sum_m, for each module m. The summary only ranges over module inputs and outputs `io`, and we require that the invariant implies the summary, *i.e.*, we add a clause stating that $inv_m(v_L, v_R) \Rightarrow sum_m(io_L, io_R)$. The analysis produces the same constraints as before, but now on a per-module basis, that is, we require module invariants to hold on initial states (eq. (init)), and be preserved under the transition relation (eq. (ind)), but, instead of using the overall transition relation *next* we use a per-module transition relation $next_m$. It may now happen that $next_m$ makes use of another module n, but, instead of inlining the transition relation of n as before, we substitute it by its module summary sum_n, thereby avoiding the blowup in constraint size. Finally, we restrict sources and sinks to occur at the top-level module, and add a clause requiring that all sinks have the same colors (eq. (ct)). The resulting clauses can be exponentially smaller than the naive version (*e.g.*, for our AES benchmark) thereby enabling fast solving and allowing our analysis to scale.

4.4 Implementation

XENON is made up of two parts: a front-end and a back-end. Our front-end translates VERILOG to the IODINE intermediate representation (IR) and associates secrecy assumptions

with input and output wires. Our back-end then translates this annotated IR into a set of verification conditions (Horn clauses); when verification fails, we generate counter examples and secrecy assumptions and present them to the user for feedback. We implement the back-end in roughly 9KLOC Haskell, using the `liquid-fixpoint` (0.8.0.2) [11] and Z3 (4.8.1) [47] libraries for verification, and the GLPK (4.65) [8] library for synthesizing assumptions (by solving the ILP problem of Section 5.2). Our tool and evaluation data sets (described next) are open source and available on GitHub.[1]

4.5 Evaluation

We evaluate XENON by asking four questions:

- **Q1:** Do module summaries improve scalability?
- **Q2:** Are constant-time counterexamples effective at localizing the cause of verification failures?
- **Q3:** Does XENON suggest useful secrecy assumptions?
- **Q4:** What is the combined effect of counterexamples and secrecy assumption generation on the verification effort?

To answer these questions, we evaluate XENON on various VERILOG programs: a modular AES-256 implementation [12], the SCARV "side-channel hardened RISC-V" processor [5], and the benchmark suite from [99] (which includes a MIPS and RISC-V core, ALU and FPU modules, and RSA and SHA-256 crypto modules).

Summary

XENON takes a couple of seconds to verify all but the SCARV core (which takes under nine minutes). While in most cases our approach performs on-par with IODINE's inlining approach, our modular summary approach is key to scaling verification to some benchmarks

[1] We omit the link for the double-blind review process.

Table 4.1. #LOC is the number of lines of Verilog code (without comments or empty lines), **#Assum** is the number of assumptions; **flush** and **public** are sizes of the sets FLUSH and PUB respectively, **CT** shows if the program is constant-time, **Check** is the time XENON took to check the program; **Inlined** and **Modular** represent inlining module instances and using module summaries respectively. **# Iter** is the number of times the user has to invoke XENON to verify the benchmark starting with an empty set of assumptions, **CEX Ratio** is the average ratio of the number of identifiers in the counterexample to all variable-time identifiers in a given iteration, **Sugg Ratio** is the average ratio of the number of secrecy assumptions that XENON suggests to all secret variables in a given iteration, and **Accept Ratio** is the average ratio of the suggested assumptions accepted by the user. In the **Total** row, we use * to denote averages instead of sums. We do not run our error localization experiments on FPU2 and RSA because they are variable-time, and on AES-256 because it do not need any assumptions.

Name	**#LOC**	**#Assum**		**CT**	**Check (H:M:S)**		**# Iter**	**CEX**	**Sugg**	**Accept**
		#flush	**#public**		**Inlined**	**Modular**		**Ratio**	**Ratio**	**Ratio**
MIPS [10]	447	28	3	✓	2.42	3.13	3	2.50%	1.73%	83.33%
RISC-V [7]	514	10	11	✓	13.21	10.23	5	16.24%	3.98%	46.90%
SHA-256 [13]	563	4	3	✓	7.21	8.90	2	4.28%	3.57%	100.00%
FPU [3]	1108	3	1	✓	9.10	11.54	1	0.33%	0.26%	100.00%
ALU [6]	1327	1	3	✓	2.01	2.29	2	0.88%	1.38%	75.00%
FPU2 [4]	272	24	4	✗	1.31	3.65	-	-	-	-
RSA [14]	855	29	4	✗	2.87	1.51	-	-	-	-
AES-256 [12]	800	0	0	✓	6:05:01.82	2.74	-	-	-	-
SCARV [5]	8468	73	54	✓	14:20.93	8:35.46	34	9.08%	5.68%	84.80%
Total	14354	159	89	-	6:20:00.88	9:19.45	47	5.55%*	2.77%*	81.67%*

(*e.g.*, AES-256 and SCARV). Finally, we find the counterexamples and secrecy assumptions suggested by XENON to be crucial to reducing the human-in-the-loop time from days to (at worst) hours.

Experimental Setup

We run all experiments on a 1.9GHz Intel Core i7-8650U machine with 16 GB of RAM, running Ubuntu 20.04 with Linux kernel 5.4.

Methodology

For every benchmark, we start with an empty set of secrecy assumptions, and run XENON repeatedly to recover the missing assumptions and verify the benchmark. We collect the following information after every invocation of the tool: the total number of the variable-time and secret identifiers, the size of the counterexamples, the number of assumptions XENON suggests, and how many assumptions we reject during each iteration. We also record the number of times we invoke XENON to complete each verification task. Finally, with all the assumptions in place, we measure the time it takes for the tool to verify each benchmark; we report the median of thirty runs for all but the non-modular (inlined) AES benchmark, for which we report the median of three runs.

In this chapter, we will only discuss the first research question. The rest will be discussed in Section 5.3.

Q1: Scalability

To evaluate how module summaries affect the scalability, we compare the time it takes to verify (or show variable-time) a program with and without module summaries. Columns **Inlined** and **Modular** of Table 4.1 gives the run times of XENON with inlining (no summaries) and module summaries, respectively. On the IODINE benchmarks (the first seven benchmarks), we observe that module summaries don't meaningfully speed up verification. Indeed, on average, module summaries only reduce the size of the query sent to our solver by roughly 5% on these benchmarks. On the more complex AES-256 and SCARV benchmarks, however, we see the

benefit of module summaries. For AES-256, using module summaries reduces the query size by 99.7%, from 391.3MB to 1.2MB, which, in turn, reduces the verification time by three orders of magnitudes – from six hours to three seconds. Module summaries allow XENON to exploit the core's modular design, i.e., AES-256's multiple and nested instantiations of the same modules (see Figure 4.3). For SCARV, summaries reduce the query size by 41%, and speed up the verification time by 40%. Though this reduction is not as dramatic as the AES-256 case, the speedup did improve XENON's interactivity.

4.6 Acknowledgements

This chapter, in part, has been submitted for publication of the material as it may appear in the 26^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), 2021, Kıcı, Rami Gökhan; v. Gleissenthall, Klaus; Stefan, Deian; Jhala; Ranjit, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Solver-Aided Verification

5.1 Real World Hardware is Not Constant Time

Real world circuits are also typically *not* constant time, in an absolute sense. Instead, when carefully designed, they are constant time under very particular *secrecy assumptions* detailing which circuit inputs are supposed to be *public* (*i.e.*, visible to the attacker) or *secret* (*i.e.*, unknown to the attacker). Thus, verification requires the user to painstakingly discover these assumptions through manual code inspection, which can be prohibitively difficult in real world circuits.

A Pipelined MIPS Processor

We illustrate the importance of these assumptions using the code fragment in Figure 5.1 which shows a simplified version of one of our benchmarks – a pipelined MIPS processor. If the reset bit `rst` is set, the processor sets a number of registers to zero (line 19). The processor then checks whether the pipeline is stalled (line 22) and either forwards the current instruction from the instruction-fetch stage to the instruction-decode stage (line 27) and advances the program counter, or stalls by reassigning the current values.

The Pipeline is not Constant Time

When using the processor in a security critical context, we want to make sure that it avoids leaking secrets through timing, *i.e.*, is constant time. Unfortunately, this is not true, without any further restrictions on its usage. For example, the execution time for a given instruction depends

```
1  // source(IF_instr); sink(WB_wd_reg);
2  module mips_pipeline(clk,rst);
3  input clk, rst;
4
5  assign IF_pc4 = IF_pc + 32'd4;
6  assign IF_pcj = ID_Jmp  ? ID_jaddr : IF_pc4;
7  assign IF_pcn = M_PCSrc ? M_btgt   : IF_pcj;
8  assign ID_rs = ID_instr[25:21];
9  assign ID_rt = ID_instr[20:16];
10
11 rom32 IMEM(IF_pc, IF_instr);
12
13 always @(*)
14   Stall = EX_MemRead &&
15           ( EX_rt == ID_rs ||
16             EX_rt == ID_rt );
17
18 always @(posedge clk)
19   if (rst)
20     ID_instr   <= 0;
21   else
22     if (Stall == 1) begin
23       ID_instr <= ID_instr;
24       IF_pc    <= IF_pc;
25       EX_rt    <= EX_rt;
26     end else begin
27       ID_instr <= IF_instr;
28       IF_pc    <= IF_pcn;
29       EX_rt    <= ID_rt;
30     end
31
32 always @(posedge clk)
33   WB_wd_reg <= WB_wd;
34 endmodule
```

Figure 5.1. MIPS Pipeline Fragment in Verilog.

on whether or not the pipeline is stalled before the instruction is retired.

This is illustrated in Figure 5.2. We model an attacker that can measure how long an instruction takes to move through the pipeline, *i.e.*, from *source* `IF_instr` to *sink* `WB_wd_reg` (this is specified through the annotation in line 1).

Such an attacker can distinguish the two runs in Figure 5.2, as the colors of `WB_wd_reg` differ in cycle k. This timing difference lets the attacker make inferences about the control flow of the program which is executed on the processor, so any attempt to verify constant time

time	stall		IF_i•		ID_i•		WB_w•	
	L	R	L	R	L	R	L	R
0	1	0	★	★	•	•	•	•
1	0	0	★	•	★	★	•	•
2	0	0	•	•	★	•	•	•
⋮								
$k-1$	0	0	•	•	•	•	★	★
k	0	0	•	•	•	•	★	•

Figure 5.2. Two runs of our pipeline example, where the left run stalls in cycle 0.

execution results in a failure.

Secrecy Assumptions

We may, however, still be able to use this processor safely, if we can find a suitable set of secrecy assumptions. For example, we could assume that register `stall` is *public* (*i.e.*, free of secrets, which can be formally expressed by the assumption that $\texttt{stall}_L = \texttt{stall}_R$ always holds). In this case, the timing difference in Figure 5.2 would only leak information that the attacker is already aware of. However, `stall` is defined deep in the pipeline and therefore, it is hard to translate this assumption into a restrictions on the kind of *software i.e.*, code we are allowed to execute on the processor. Instead, we want to phrase our assumptions in terms of *externally visible* computation inputs, in our case the instruction `IF_instr`.

To achieve this using existing technology, the engineer first has to *manually* identify `stall` as the root cause of the issue; they then needs to inspect how `stall` is set (line 13) and painstakingly trace the definitions which may involve complex combinatorial logic (excerpt starting in line line 5) and circular data-flows in order to identify a promising candidate register, such that marking the register as *public* will render the circuit constant time.

Solver Aided Verification: XENON's Interactive Loop

XENON drastically simplifies this time consuming process by finding secrecy assumptions automatically.

Step 1

First, we start with an *empty* set of secrecy assumptions and run XENON on the pipeline. The verification fails, as the pipeline is not constant time, however, XENON displays the following prompt to guide the user towards a solution.

```
> Mark 'reset' as PUBLIC? [Y/n]
```

The user either says `Y` indicating that `reset` should indeed be considered public, or else responds `N` which tells XENON to *exclude* the variable from future consideration (*i.e.*, not suggest it in future). Suppose that we follow XENON's advice, and click `Y`: this marks `reset` public and re-starts XENON for another verification attempt.

Step 2

Next, XENON suggests marking `M_PCSrc` as public. `M_PCSrc` acts as a flag that indicates whether the current instruction in the memory stage contains an indirect jump. But since `M_PCSrc`'s value depends on register values (*i.e.*,, `M_PCSrc` is set depending on whether the output of the execute phase is zero) assuming that `M_PCSrc` is public would lead to assumptions about the main memory which we wish to avoid. We therefore tell XENON to exclude it in future verification attempts and restart verification.

Step 3

Restarting verification causes XENON to suggest candidate variables `IF_pc` and `IF_pcn`, the program counter of the fetch stage and its value in the next cycle, respectively. We mark `IF_pc` as public as this directly encodes the assumption that the program's control flow does not depend on secrets. XENON then proves that the resulting program executes in constant time and therefore concludes the verification process.

Counterexamples

In case, the user wishes to further inspect the root cause of the problem, XENON also computes a counterexample which pinpoints the (set of) variables that have lost the constant time property first. For our example, in all three interactions, XENON blames variable `ID_instr`,

which is indeed the root cause of the problem. We discuss how XENON computes counterexamples using an artifact extracted from the failed proof attempt in Section 5.2.1, and how XENON synthesizes secrecy assumptions via a reduction to integer linear programming in Section 5.2.2.

5.2 Counterexamples & Assumption Synthesis

In this section, we explain how XENON helps the user understand and explicate secrecy assumptions when verification fails. We describe how XENON computes a minimal counterexample which *localizes* the error (Section 5.2.1), how XENON *synthesizes* secrecy assumptions that eliminate the root cause of the verification failure (Section 5.2.2) and, finally, describe how counterexample generation and secrecy assumption synthesis interact with module summaries (Section 5.2.3).

5.2.1 Computing Minimal Counterexamples

How Circuits become Variable-Time

When verification fails, we want to present a minimal counterexample that localizes the error. To do so, we must explain to the user how a circuit can fail to be constant-time. This can happen when the following two conditions are met: (1) a *variable* needs to exhibit timing-variability (*i.e.*, *stop* being constant-time), and (2) there needs to be a data-flow from the variable with timing-variability to an output (sink). But how does a variable come to exhibit timing-variability in the first place? This can happen either transitively through an assignment from another variable with timing variability, or ultimately, from a control (*i.e.*, branch) dependency on a secret. XENON exploits this insight to compute a counterexample which localizes the root cause of the error. For this, XENON computes the *earliest* variable that exhibits timing variability through a control dependency.

Dependency Graph

XENON first creates a dependency graph $G \triangleq (V, D \cup C)$, consisting of a set of vari-

ables $V \subseteq \text{VARS}$, a set of directed edges $D \subseteq (\text{VARS} \times \text{VARS})$ such that $(v,w) \in D$ if there is a data-flow from v to w, *i.e.*, v's value is used to compute w through an assignment, and a set of directed edges $C \subseteq (\text{VARS} \times \text{VARS})$ such that $(v,w) \in C$, if v's value is used indirectly, when computing w's value, *i.e.*, w's value is computed under a branch whose condition depends on v.

Variable-Time Map

Next, XENON extracts an artifact from the failed proof attempt: a partial map $varTime \in (\text{VARS} \rightharpoonup \mathbb{N})$ which records the temporal *order* in which variables started to exhibit timing variability. That is, if for two variables v and w, $varTime(v) < varTime(w)$, then the verifier determined that v started to exhibit timing variability *before* w, or, more formally, there exist two runs π_L and π_R of length n, and two numbers $0 \leq i, j < n$ such that i is the smallest number such that $live(\pi_L, i)(v) \neq live(\pi_R, i)(v)$ and similarly j is the smallest number such that $live(\pi_L, j)(w) \neq live(\pi_R, j)(w)$ and $i < j$. XENON uses this map to break cyclic data-flow dependencies.

Computing the Root Cause

Using the data-flow graph, and map *varTime*, XENON computes a *reduced graph* which helps pinpoint the root cause that renders the circuit variable-time. To this end, XENON removes from the dependency graph, all nodes that *are* constant-time, and all edges (w,v) such that $varTime(w) > varTime(v)$. Intuitively, if variable w has started to exhibit timing variability *after* variable v, then w cannot be the cause of v losing the constant-time property. Finally, we compute a *slice* of the graph with respect to the sinks SNK, *i.e.*, we remove all nodes that cannot reach a sink node using the remaining edges. This leaves us with a set of variables $\text{CEX} \subseteq \text{VARS}$ without incoming edges, which we identify as the root cause of the violation to present to the user.

Reachability

For graph $G \triangleq (V, D \cup C)$ and nodes $v, w \in V$ we write $v \rightarrow w$, if $(v,w) \in (D \cup C)$, $v \rightarrow^n w$, if there is a sequence $v_0 v_1 \ldots v_{n-1}$, such that $v_0 = v$ and $v_{n-1} = w$, and $v_i \rightarrow v_{i+1}$ for

$i \in \{0, \ldots, n-2\}$, and finally, say w is reachable from v, if there exists n such that $v \rightarrow^n w$.

Reduced Graph

For a data-flow graph $G \triangleq (V, D \cup C)$, and map *varTime*, we define the reduced graph with respect to *varTime* as the largest subgraph $G' \triangleq (V', D' \cup C')$ such that $V' \subseteq V$, $D' \subseteq D$, $C' \subseteq C$ and

1. All nodes are variable-time, *i.e.*, for all $v \in V'$, $\mathit{varTime}(v) \neq \bot$.

2. All edges respect the variable-time order, *i.e.*, for all $(v, w) \in (D' \cup C')$, we have $\mathit{varTime}(v) < \mathit{varTime}(w)$.

3. All nodes can reach a sink, *i.e.*, for all $v \in V'$, there is $o \in \textsc{Snk}$ such that o is reachable from v.

For variable v, and graph $G \triangleq (V, D \cup C)$, let $\mathit{pre}(v, G)$ be the set of its immediate predecessors in G, that is $\mathit{pre}(v) \triangleq \{w \mid (w, v) \in (D \cup C)\}$. We define the counterexample $\textsc{Cex}$ of a graph G with map *varTime* as the set of nodes in the reduced graph G' (wrt *varTime*), that have no predecessors, *i.e.*, $\textsc{Cex} \triangleq \{v \mid \mathit{pre}(v, G') = \emptyset\}$.

Example: Simplified Pipeline

The code in Listing 5.1 shows a simplified version of the pipelined processor from Figure 5.1. Like in Figure 5.1, the pipeline either stalls (Line 10) if flag `Stall` is set (Line 9), or else forwards values to the next stage (Line 13). To avoid a write-after-write data-hazard, the `Stall` flag is set, if the instructions in the execute and decode stage have the same target registers (Line 6). The target register is calculated from the current instruction (Line 1), and the instruction is, in turn, fetched from memory using the current program counter (Line 3). Note the cyclic dependency between `ID_instr` and `Stall` that turns comprehending the root cause into a "chicken-and-egg" problem.

Dependency Graph

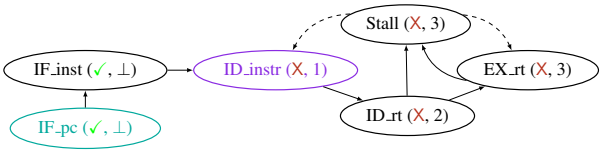
```
1  assign ID_rt = ID_instr[20:16];
2
3  rom32 IMEM(IF_pc, IF_instr);
4
5  always @(*)
6    stall = (ID_rt == EX_rt)
7
8  always @(posedge clk) begin
9    if (Stall == 1) begin
10       ID_instr  <= ID_instr;
11       EX_rt     <= EX_rt;
12   end else begin
13       ID_instr  <= IF_instr;
14       EX_rt     <= ID_rt;
15   end
16 end
```

Listing 5.1. Example 1: Simplified Pipeline.

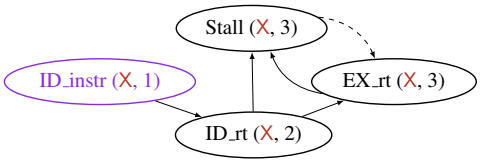
To check if the pipeline fragment executes in constant-time, we mark `IF_pc` as source, and `ID_instr` as sink and run XENON. But, since the pipeline is variable-time, the verification fails. To compute a minimal counterexample, XENON creates the dependency graph shown in Figure 5.3a. Each node is annotated with information extracted from the failed proof attempt: the node is labeled with its value under *varTime*, and is marked with (✓) if XENON was able to verify that the variable is constant-time and (X) otherwise. Edges represent data or control dependencies between variables. Dashed edges indicate that the verifier determines the source becomes variable-time *after* the target. Hence, the ordering induced by the *varTime* allows us to break cyclic dependencies, thereby resolving the chicken-and-egg problem.

Reduced Dependency Graph

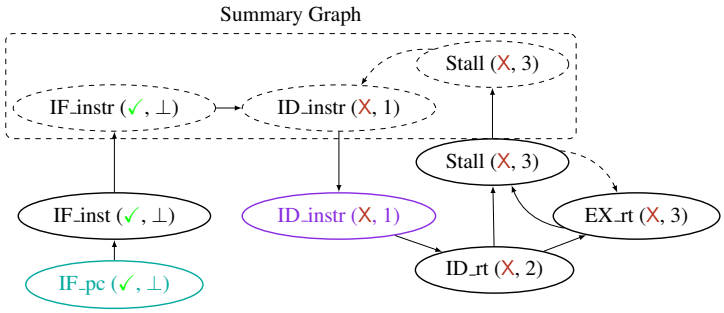
Figure 5.3b shows the dependency graph after removing all constant time nodes and edges that violate the precedence ordering. In a final step, XENON erases all nodes that cannot reach sink `ID_instr`. This only leaves `ID_instr`, the root cause of constant-time violation.



(a) Dependency Graph



(b) Dependency Graph after eliminating constant-time nodes and edges violating *varTime*.



(c) Dependency Graph with Module Summary

Figure 5.3. Figure 5.3a shows the dependency graph for Listing 5.1. Each node is labeled with its *varTime*-value and marked (✓) if XENON was able to prove the variable constant-time and (X) otherwise. Figure 5.3b shows the dependency Graph after eliminating constant-time nodes from Figure 5.3a, and removing edges that violate the variable-time map. Removing the edge between `Stall` and `ID_instr` breaks the cyclic dependency in the original graph. Figure 5.3c shows the variable dependency graph with a module summary.

5.2.2 Assumption Synthesis

The previous step leaves us with a set of nodes CEX that are the root cause of the constant-time violation. Since these nodes can only become variable-time through a control dependency on a secret value, we can compute a set of variables BLAME that are directly responsible: the immediate predecessors of nodes in CEX in the dependency graph with respect to a control dependency. Formally, for dependency graph $G = (V, D \cup C)$, we let $\text{BLAME} \triangleq \{w \mid v \in \text{CEX} \wedge (w, v) \in C\}$. To synthesize secrecy assumptions that remove the constant-time violation, we could directly assume that all nodes in BLAME are public (*i.e.*, their value is the same in any two runs). Unfortunately, this is often a poor choice: variables in BLAME can be defined deep inside the circuit, whereas we would like to phrase our assumptions in terms of

externally visible *input sources*.

Finding Secrecy Assumptions via ILP

Instead, we compute a minimal set of assumptions close to the input sources via a reduction to an Integer Linear Programming (ILP) optimization problem. To this end, we use a second proof artifact, a map *secret* that, similar to *varTime*, describes the temporal order in which the verifier determines variables have become *secret*, *i.e.*, ceased being public: there exists two runs along which the values of the variable differs. Let $G' = (V', D' \cup C')$ be the reduced dependency graph with respect to *secret*, and let $\text{NO} \subseteq V'$ be a set of variables that the user chose to exclude from consideration. XENON produces constraints on a set of new variables: two constraint variables $m_v \in \{0,1\}$ and $p_v \in \{0,1\}$, for each program variable v, such that $m_v = 1$, if program variable v is *marked* public by an assumption, and $p_v = 1$, if v can be *shown to be* public, that is, it is either marked public, or all its predecessors are public. Then, XENON produces the following set of constraints.

$$m_{\mathtt{v}} \geq p_{\mathtt{v}}, \qquad \textit{if } v \in V' \text{ , } \textit{pre}(v, G') = \emptyset \quad (1)$$

$$m_{\mathtt{v}} + \left(\frac{\sum_{w \in \textit{pre}(v)} p_{\mathtt{w}}}{\#\textit{pre}(v)} \right) \geq p_{\mathtt{v}} \quad \textit{if } v \in V' \text{ , } \textit{pre}(v, G') \neq \emptyset \quad (2)$$

$$p_{\mathtt{v}} = 1 \qquad \textit{if } v \in (\text{BLAME} \setminus \text{NO}) \quad (3)$$

$$m_{\mathtt{v}} = 0 \qquad \textit{if } v \in \text{NO} \quad (4)$$

Constraints (1) and (2) ensure that a variable is public, if either it is marked public, or all its predecessors in G' are public. Constraint (3) ensures that all blamed variables that have not been excluded can be shown to be public, and, finally, constraint (4) ensures that all excluded constraints are not marked. Let $d(v,w)$ be a distance metric, *i.e.*, a function that maps pairs of nodes to the natural numbers. Then we want to solve the constraints using the following objective function that we wish to minimize, where for $v \in V'$, we define as weight the minimal

distance from one of the source nodes $w_v = (\arg\min_{in \in \text{SRC}} d(in, v))$:

$$\sum_{v \in V'} w_v m_{\mathtt{v}}. \qquad \text{(objective)}$$

A solution to the constraints defines a set of assumptions $\mathscr{A} = (\text{FLUSH}, \text{PUB})$, where we let $\text{FLUSH} \triangleq \{v \in V' \mid m_{\mathtt{v}} = 0, p_{\mathtt{v}} = 0\}$ and $\text{PUB} \triangleq \{v \in V' \mid m_{\mathtt{v}} = 1\}$. The constraints can be solved efficiently by an ILP solver.

Example: Simplified Pipeline

We show how XENON computes a set of secrecy assumptions. In our example, there are several valid choices for such a set, however, not all are equally desirable. For example, XENON could just return variable `Stall`. But `Stall` is set deep in the circuit, and therefore it is hard to translate the obligation that `Stall` be public into proof obligations on the inputs. Another choice would be to choose `ID_rt` and `EX_rt`, but we would prefer a solution that uses fewer assumptions. The optimal solution is `IF_pc`, as it is closest to the inputs and requires only a single variable. Since, all variables are secret (*i.e.*, we didn't make any assumptions yet), the reduced graph is equal to the original graph. As we identified `ID_instr` as the root cause in the previous step, we need to make all its indirect influences public. `ID_instr` only depends on `Stall`, and therefore we add constraint $p_{\texttt{Stall}} = 1$. Next, we add constraints that ensure that a variable is public, if either it is marked public, or all its predecessors in the dependency graph are public. For example, for variable `IF_instr` and `ID_instr`, we get: $m_{\texttt{IF\_instr}} + p_{\texttt{IF\_pc}} \geq p_{\texttt{IF\_instr}}$ and $m_{\texttt{ID\_Instr}} + \frac{p_{\texttt{IF\_instr}} + p_{\texttt{Stall}}}{2} \geq p_{\texttt{ID\_Instr}}$. Finally, we want to minimize the following objective function in order to mark a minimal number of variables public, where we weigh variables by their distance from the source: $m_{\texttt{IF\_pc}} + 2m_{\texttt{IF\_instr}} + 3m_{\texttt{ID\_instr}} + \ldots$. Sending the constraints to an off-the-shelf ILP solver produces solution, where $m_{\texttt{IF\_pc}} = 1$ and $m_{\texttt{IF\_v}} = 0$, for all variables $v \neq \texttt{IF\_PC}$, and $p_{\mathtt{v}} = 1$, for all v. This corresponds to the following assumption set $\mathscr{A} \triangleq (Flush, \{\texttt{IF\_PC}\})$, where $Flush$ includes all variables except `IF_PC`. This corresponds exactly to the desired minimal solution where we only mark `IF_pc` as public. Note that our

method does not necessarily result in all variables becoming public. We give an example of such a program in Section 5.5.

5.2.3 Modules

To avoid a blowup in constraint size, we want to avoid inlining the code of instantiated modules. We therefore extract a dependency graph from the module summary: whenever the summary requires an input `in` to be public for an output `out` to be constant-time, we draw a control dependency between `in` and `out`. Whenever the summary requires an input `in` needs to be constant-time for an output `out` to be constant-time, we draw a data dependency. Finally, we insert the computed summary graph into the top level dependency graph, and connect the instantiation parameters to the graph's inputs and outputs.

Example

We modify Listing 5.1 to factor out the updates to `ID_instr` into a separate module. XENON computes the following summary invariant, from which we create the summary graph shown in Figure 5.3c: $ct(\texttt{IF\_instr}) \wedge pub(\texttt{Stall}) \Rightarrow pub(\texttt{ID\_instr})$. Since the graph obtained by connecting the instantiated variables to the summary graph is equivalent to the original (Figure 5.3a), our analysis returns the same result.

5.3 Evaluation

In this section, we will continue discussing the research questions presented in Section 4.5.

Q2: Error Localization

To understand if our counterexample generation is effective at localizing the cause of verification failures, we compare the number of identifiers in the counterexample to the total number of variable-time identifiers. The **CEX Ratio** column Table 4.1 reports the average ratio per iteration. On average, we observe that less than 6% of the variable-time identifiers

are included in the counterexamples. Since the total number of variable-time identifiers are typically on the order of hundreds (*e.g.*, the median (and geomean) number of variable-time identifiers across all benchmarks and iteration is 97 (94)), this dramatically reduces the number of variables the developer has to inspect manually. The counterexamples also precisely pinpointed the variable-time parts of the variable-time benchmarks – the FPU2 and RSA cores. For example, in the FPU2 benchmark XENON included a `state` register, used to dictate when the output of division is ready, in the third iteration counterexample. Inspecting this register revealed that the its value depends on whether one of operands to the division operation is NaN (and thus clearly leaks information about the operands).

Q3: Identifying Secrecy Assumptions

To understand if XENON suggests useful secrecy assumptions, we not only record the number of *useful* suggestions (suggestions accepted by the user) but also the ratio of suggestions to the total number of secret variables the user would otherwise have to manually inspect. We find that most (on average 81.67%) of the tool's suggestions are useful, reported in the **Accept Ratio** column of Table 4.1. Moreover, we observe that the number of suggestions is also relatively small (**Sugg Ratio** column); on average, we only had to inspect 2.77% of the secret variables.

Q4: Verification Effort

Finally, as a rough measure of the overall verification effort, we count the number of iteration loops, *i.e.*, the number of times we invoked XENON after modifying our set of secrecy assumptions. Verifying the largest benchmark from [99], the YARVI RISC-V core [7], for example, took five invocations over several minutes. The final assumption we arrived at were the same as the assumptions manually identified by the authors of IODINE in [99]; they, however, took multiple days to identify these assumptions and verify this core [65]. Verifying the SCARV core took thirty four iterations and roughly three hours; this core is considerably larger (roughly 20×) than the YARVI RISC-V core and, we think, beyond what would possible with tools like IODINE, which rely on manual annotations and error localization. Indeed, we found the

error localization and assumption inference to be especially useful in narrowing our focus and understanding to small parts of the core and avoid complex implementation details irrelevant to the analysis. We describe our experience verifying SCARV in more detail in Section 5.4.

5.4 Case Study: SCARV

In this section, we briefly describe our most complex benchmark, and the assumptions we made about its execution to verify that it is constant-time. SCARV [5] is a processor core implementation with a 5-stage, single issue, in-order pipeline. In addition to the baseline RISC-V 32-bit integer base architecture (RV32I), it extends its ISA with the standard Compressed (RV32C) and Multiply (RV32M) extensions, and the non-standard XCrypto [6] extension. It's a micro-controller, with no cache, branch prediction or virtual memory. We verified the SCARV processor not the SoC which so also has other features like RAM, UART, and a randomness generator among other things.

Assumptions

We assume that the sequence of instructions, the memory traces, arrival of interrupts are public. Similar to the YARVI core, we assume that control/status registers are not accessed illegally. Unique to this benchmark was the use of extra leakage fence instructions and the interaction between the core and the random number generator. We assumed that the leakage fence instructions cause the pipeline to stall at the same time, and the external random number generator generates values in constant time.

5.5 Example: Not All Variables Become Public

One might think that XENON requires all variables occurring in branch conditions to be annotated as public, however, this is not the case. Figure 5.4 shows an example of such a program. Running XENON produces the dependency graph shown in Figure 5.5. XENON computes root-cause candidates by eliminating constant-time nodes and edges violating the

```
1  module test(clk, in, cond, bubble, out);
2    input wire clk, in, cond, bubble;
3    output reg out;
4    reg tmp1, tmp2, r2, r3;
5
6    always @(posedge clk) begin
7      tmp1 <= in | r3;
8      tmp2 <= in & r3;
9
10     if (cond)
11       r2 <= tmp1;
12     else
13       r2 <= tmp2;
14
15     if (stall)
16       r3 <= r3;
17     else
18       r3 <= r2;
19
20     out <= r3;
21   end
22 endmodule
```

Figure 5.4. Example 3.

precedence order. The result is shown in Figure 5.6. Removing all nodes that cannot reach source `out` leaves only nodes `r3` and `out`, and since `r3` has no predecessors, we identify it as the earliest node that became non-constant time, and therefore the root cause of the problem. Solving the ILP constraints yields `stall` as candidate assumption, and marking `stall` as public and restarting XENON verifies constant time execution without the need to mark `cond` as public. This is possible because XENON is able to prove that `tmp1` and `tmp2` have the same colors, irrespective of the value of `cond`, *i.e.*, that $\texttt{tmp1}^{\bullet}{=}\texttt{tmp2}^{\bullet}$ holds irrespective of `cond`.

5.6 Acknowledgements

This chapter, in part, has been submitted for publication of the material as it may appear in the 26^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), 2021, Kıcı, Rami Gökhan; v. Gleissenthall, Klaus; Stefan, Deian; Jhala; Ranjit, 2021. The dissertation author was the primary investigator and author of

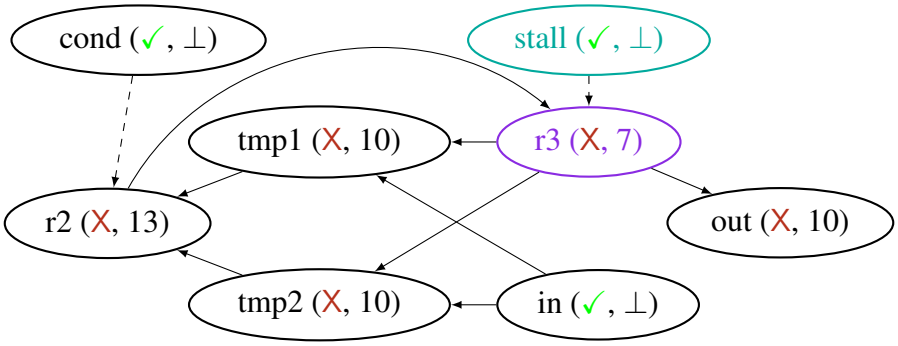


Figure 5.5. Example 3: Variable dependency graph.

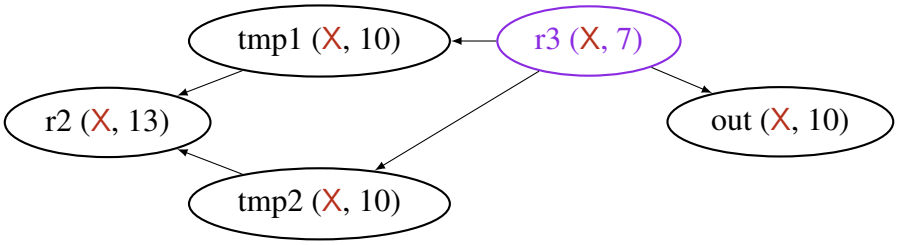


Figure 5.6. Example 3: Variable dependency graph after eliminating non-ct nodes and edges that violate the precedence relation.

this paper.

Chapter 6

Related Work

6.1 Constant-Time Software

Almeida et al. [21] 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. [29] build on the CompCert compiler [72] to enforce constant time execution through an information flow type system. Reparaz et al. [85] 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.

6.2 Self-Composition and Product Programs

Barthe et al. [31] introduce the notion of self composition to verify information flow. Terauchi and Aiken generalize this construction to arbitrary 2-safety properties [96], *i.e.*, proper-

ties that relate two runs, and Clarkson and Schneider [42] generalize to multiple runs. Barthe et al. [30] 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 [94], which presents an extension of Hoare logic to hyper-properties that computes lock-step compositions on demand, per Hoare-triple.

6.3 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. [70] 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 [108] proves timing-sensitive non-interference for circuits implemented in an extension of VERILOG that uses value-dependent information flow types. Caisson [73] is a hardware description language that uses information flow types to ensure that generated circuits are secure. GLIFT [97, 98] 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 Section 2.4 for details). Moreover, SecVerilog, Caisson, and GLIFT 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.

6.4 Combining Hardware & Software Mitigations

HyperFlow [53] and GhostRider [76], take hardware/software co-design approach to eliminating timing channels. Zhang et al. [107] 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 [15, 17, 51, 71, 88] 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.

6.5 Modular Verification of Software and Hardware

XENON exploits modularity to verify large circuits by composing *summaries* of the behaviors of smaller sub-components of those circuits. This is an old idea; for example, [86] shows how to perform dataflow analysis of large programs by computing procedure summaries, and Houdini [54] shows how to verify programs by automatically synthesizing pre- and post-conditions summarizing the behaviors of individual procedures. On the hardware side, model checkers like Mocha [22] and SMV [78] use rely-guarantee reasoning to perform modular verification. Kami [40] and [101] develops a compositional hardware verification methodology using the Coq proof assistant. However, the above require the user to provide module interface abstractions. There are some approaches that synthesize such abstractions in an counterexample guided fashion [61, 109]. All focus on functional verification of properties of a *single* run, and do not support abstractions needed to reason about timing-channels which require relational hyperproperties [42].

6.6 Fault Localization

There are several approaches to help developers localize the root causes of software bugs [103]. Logic-based fault localization techniques [41, 52, 63, 64] are the closest line of work to ours. For example, BugAssist [63] uses a MAXSAT solver to compute the maximal set of statements that may cause the failure given a failed error trace of a `C` program. XENON is similar in that we phrase localization as an optimization problem, allowing the use of ILP to locate the possible cause of a non-constant-time variable. However, as constant-time is a relational property, and we verify hardware, we cannot directly reuse the previous approaches directly.

6.7 Synthesizing Assumptions

Our approach to synthesizing secrecy assumptions is related to work on precondition synthesis for memory safety. Data-driven precondition inference techniques such as [55, 56, 82, 90, 91], unlike XENON, require positive and negative examples to infer preconditions. XENON's synthesis technique is an instance of *abductive* inference, which has been previously used to explain verification failures, triage analysis reports by allowing the user to interactively determine the preconditions under which a program is safe or unsafe [48] or to identify the most general assumptions or context under which a given module can be verified safe [37, 49, 50, 57]. Unlike the above, our abduction strategy is tailored for the relational constant-time property. Furthermore, XENON uses information from the verifier to ensure that the user interaction loop only invokes the ILP solver (not the slower Horn-clause verifier), yielding a rapid cycle that pinpoints the assumptions under which a circuit is constant-time. In future work, we would like to see, if ideas introduced in XENON can be applied to localization, explanation and verification of other classes of correctness or security properties.

Chapter 7

Future Work

While we were able to use IODINE and XENON to analyze realistic open-source hardware, the systems we have looked at are still much less complicated than the ones that we use daily. In the future I would like to continue using XENON to analyze such systems and tackle the challenges that they present.

Empirical Validation

One area of research I would like to pursue is using existing hardware simulators to empirically validate the results of our technique. While we proved the correctness of our analysis, we have not done the same for our implementation. When XENON verifies that a given hardware runs in constant-time, validating the result would bring more confidence into using the system in the real world. On the other hand, if XENON fails the verification, we could complement the counter-examples that it generates with the output of the simulator to better understand the system.

Almost all hardware systems that we have verified required the use of assumptions about their inputs. In order to use the existing simulators, these assumptions should be taken into account while creating the test inputs. We can handle the assumptions about the initial states of the registers by generating an `initial` block (Section 9.9.1 of [16]) for every such register, and ask the user to choose an initial value. Then, we can filter out the executions that do not satisfy every "always equals" assumption (which states that a variable is equal throughout any pair of

runs) during the simulation.

Extended Evaluation

Second, I would like to test the limits of XENON on even larger benchmarks. For example, the pipelined processors that we have looked at did not exhibit advanced features such as multi-level cache designs, branch predictors, out-of-order execution, or speculative execution. All of these features are the ubiquitous in modern hardware systems, and they are a barrier to following discipline of constant-time programming. I also would like to analyze other security related hardware modules such as enclaves, and ARM's recent set of data independent timing (DIT) instructions.

End-to-End Verification

Finally, I would like to achieve end-to-end verification using XENON. In addition to the infrastructure described in the previous section that allows the user to simulate the hardware, we need a RISC-V compiler which checks whether the generated assembly instructions satisfy the assumptions used to verify the hardware system using XENON. The challenging part here would be the error reporting messages shown to the user when an assumption is not satisfied, and choosing the language level statement or expression to blame.

Bibliography

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

[2] BearSSL - constant-time crypto. https://www.bearssl.org/constanttime.html. (Accessed on 08/19/2020).

[3] fpga_mc/fpu at master · monajalal/fpga_mc · github. https://github.com/monajalal/fpga_mc/tree/master/fpu. (Accessed on 08/16/2020).

[4] Github - dawsonjon/fpu: synthesiseable ieee 754 floating point library in verilog. https://github.com/dawsonjon/fpu. (Accessed on 08/16/2020).

[5] Github - scarv/scarv-cpu: Scarv: a side-channel hardened risc-v platform. https://github.com/scarv/scarv-cpu. (Accessed on 08/16/2020).

[6] Github - scarv/xcrypto: Xcrypto: a cryptographic ise for risc-v. https://github.com/scarv/xcrypto. (Accessed on 08/19/2020).

[7] Github - tommythorn/yarvi: Yet another risc-v implementation. https://github.com/tommythorn/yarvi. (Accessed on 08/16/2020).

[8] Glpk - gnu project - free software foundation (fsf). https://www.gnu.org/software/glpk/. (Accessed on 08/10/2020).

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

[10] iodine/benchmarks/472-mips-pipelined at master · gokhankici/iodine · github. https://github.com/gokhankici/iodine/tree/master/benchmarks/472-mips-pipelined. (Accessed on 08/16/2020).

[11] liquid-fixpoint: Horn clause constraint solving for liquid types. https://github.com/ucsd-progsys/liquid-fixpoint. Accessed: 2018-08-29.

[12] Overview :: AES :: OpenCores. https://opencores.org/projects/tiny_aes. (Accessed on 08/05/2020).

[13] Overview :: Sha cores :: Opencores. https://opencores.org/projects/sha_core. (Accessed on 08/16/2020).

[14] Rsa4096/modexp 2.0 at master · fatestudio/rsa4096 · github. https://github.com/fatestudio/RSA4096/tree/master/ModExp%202.0. (Accessed on 08/16/2020).

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

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

[17] 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.

[18] 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.

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

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

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

[22] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods Syst. Des.*, 15(1), 1999.

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

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

[25] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[26] Arm Ltd. DIT, data independent timing. https://developer.arm.com/docs/ddi0595/h/aarch64-system-registers/dit.

[27] 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.

[28] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

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

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

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

[32] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

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

[34] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *World Wide Web*, 2007.

[35] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *Workshop on Offensive Technologies*, 2017.

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

[37] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. 58(6).

[38] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[39] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A dsl for timing-sensitive computation. In *Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN, 2019.

[40] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2017.

[41] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation*, 2013.

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

[43] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[44] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2), May 2018.

[45] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. BINSEC/REL: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE Symposium on Security and Privacy*, 2020.

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

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

[48] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12. Association for Computing Machinery.

[49] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *SIGPLAN Not.*, 48(10), October 2013.

[50] Isil Dillig, Thomas Dillig, Boyang Li, Ken McMillan, and Mooly Sagiv. Synthesis of circular compositional program proofs via abduction. 19(5).

[51] 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.

[52] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. *FM 2012: Formal*

Methods Lecture Notes in Computer Science, Aug 2012.

[53] 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.

[54] Cormac Flanagan and K Rustan M Leino. Houdini, an Annotation Assistant for ESC/Java. Springer, Berlin, Heidelberg.

[55] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. *Computer Aided Verification Lecture Notes in Computer Science*, 2014.

[56] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Principles of Programming Languages*, POPL '16. ACM.

[57] Roberto Giacobazzi. Abductive analysis of modular logic programs. In *Proceedings of the 1994 International Symposium on Logic Programming*, ILPS '94. MIT Press.

[58] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections. In *USENIX Security Symposium*, 2020.

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

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

[61] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods Syst. Des.*, 32(3), 2008.

[62] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. Differential privacy under fire. In David Wagner, editor, *USENIX Security*, 2011.

[63] Manu Jose and Rupak Majumdar. Bug-assist: Assisting fault localization in ANSI-c programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, Lecture Notes in Computer Science. Springer.

[64] Manu Jose and Rupak Majumdar. Cause clue clauses. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI 11*, 2011.

[65] Rami Gökhan Kici. Personal communication, August 2020.

[66] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

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

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

[69] Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. Cross-origin pixel stealing: timing attacks using CSS filters. In *ACM CCS*, 2013.

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

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

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

[73] 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.

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

[75] 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.

[76] 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.

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

[78] Kenneth L. McMillan. A compositional rule for hardware design refinement. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Com-*

puter Science. Springer, 1997.

[79] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL:TPM meets timing and lattice attacks. In *USENIX Security*, 2020.

[80] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 2006.

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

[82] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Programming Language Design and Implementation*. ACM SIGPLAN, 2016.

[83] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

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

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

[86] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages*. ACM, 1995.

[87] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

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

[89] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[90] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc.

[91] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A Deep

Learning Framework for Program Verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, Lecture Notes in Computer Science. Springer International Publishing.

[92] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re:visited. In *Workshop on Offensive Technologies (WOOT)*. USENIX, August 2018.

[93] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security Symposium*, 2001.

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

[95] Paul Stone. Pixel perfect timing attacks with HTML5. *Context Information Security (White Paper)*, 2013.

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

[97] 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.

[98] 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.

[99] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. IODINE: Verifying constant-time execution of hardware. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1411–1428, Santa Clara, CA, August 2019. USENIX Association.

[100] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[101] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular Deductive Verification of Multiprocessor Hardware Designs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, Lecture Notes in Computer Science. Springer International Publishing.

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

[103] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8), 2016.

[104] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[105] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2), 2017.

[106] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

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

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

[109] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. Synthesizing Environment Invariants for Modular Hardware Verification. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science. Springer International Publishing.