Solver-Aided Constant-Time Hardware Verification

Klaus v. Gleissenthall
Vrije Universiteit Amsterdam
The Netherlands

Rami Gökhan Kıcı
University of California, San Diego
USA

Deian Stefan
University of California, San Diego
USA

Ranjit Jhala
University of California, San Diego
USA

ABSTRACT

We present Xenon, a solver-aided, interactive method for formally verifying that Verilog hardware executes in constant-time. Xenon scales to realistic hardware designs by drastically reducing the effort needed to localize the root cause of verification failures via a new notion of constant-time counterexamples, which Xenon uses to synthesize a minimal set of secrecy assumptions in an interactive verification loop. To reduce verification time Xenon exploits modularity in Verilog code via module summaries, thereby avoiding duplicate work across multiple module instantiations. We show how Xenon's assumption synthesis and summaries enable us to verify different kinds of circuits, including a highly modular AES-256 implementation where modularity cuts verification from six hours to under three seconds, and the ScarVside-channel hardened RISC-V micro-controller whose size exceeds previously verified designs by an order of magnitude. In a small study, we also find that Xenon helps non-expert users complete verification tasks correctly and faster than previous state-of-art tools.

KEYWORDS

constant-time; side-channels; hardware; verification;

1 INTRODUCTION

Timing side-channel attacks are no longer theoretical curiosities. Over the last two decades, they have been used to break implementations of cryptographic primitives ranging from public-key encryption algorithms [26, 62, 88], to block ciphers [23, 71], digital signature schemes [70], zero-knowledge proofs [35], and pseudo-random generators [33]. This, in turn, has allowed attackers to break systems that rely on these primitives for security—for example, to steal TLS keys used to encrypt web traffic [26, 33, 88], to snoop and forge virtual private network traffic [70], and to extract information from trusted execution environments [25, 33, 35, 87].

The gold standard for preventing timing side-channel attacks is to follow a discipline of *constant-time* or *data-oblivious* programming [1, 15, 22, 29, 36, 89]. At its core, this discipline ensures that **(1)** secret data is not used as an operand to variable-time instructions (*e.g.*, floating-point operations like division [19, 20, 63, 75]) and **(2)** the program's control flow and memory access patterns do not depend on secrets. But for the constant-time discipline to be effective, it is crucial that the constant-time property be preserved by the underlying hardware. For example, an instruction that is deemed constant-time needs to indeed produce its outputs after the same number of clock cycles, irrespective of operands or internal state. Similarly, given that control-flow and memory access patterns are *public*, *i.e.*, free of secrets, a CPU's timing must indeed be secret-independent.

Unfortunately, simply *assuming* that hardware preserves 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 [56]. Attempts to address these attacks (*e.g.*, [76]) were also foiled: they relied on yet other incorrect microarchitectural assumptions (*e.g.*, about the timing-variability of SIMD instructions) [63]. Yet more recently, hardware crypto co-processors (*e.g.*, Intel and STMicroelectronics's trusted platform modules) turned out to exhibit similar secret-dependent timing variability [70].

A promising path towards eliminating such attacks is to *formally verify* that our hardware preserves the constant-time property of the software it is executing. Such verification efforts, however, require tool support. Unfortunately, unlike software verification of constant-time, which has had a long history [21], constant-time hardware verification is still in its infancy [28, 50, 53, 89]. As a result, existing verification approaches fail to scale to realistic hardware. This is because of two fundamental reasons. First, existing tools do not help when verification fails—and inevitably it does fail: hardware circuits only preserve constant-time execution under very specific *secrecy assumptions* that describe which port and wire values are public or secret. In our experience, dealing with failures takes up most of the verification time. With tools like Iodine [50], you must manually determine whether the circuit is leaky (*i.e.*, variable-time), or whether it is missing additional secrecy assumptions that the tool needs to be made aware of. Second, 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.

In this paper, we present Xenon, a solver-aided, interactive method for formally verifying that Verilog hardware executes in constant-time. We develop Xenon via five contributions.

1. Counterexamples. To help users understand verification failures, we introduce the notion of constant-time *counterexamples* (§ 4.1). 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 verification failure (and

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484810

thereby a small fraction of the circuit). To compute counterexamples, Xenon leverages information extracted from the failed proof attempt. In particular, the solver communicates

(1) which variables (*i.e.*, registers and wires) remained constant-time during the failed proof attempt, and
(2) the *order* in which the remaining variables became non-constant time.

This allows Xenon to break cyclic data-dependencies which cause a chicken-and-egg problem that is hard to resolve when assigning blame manually.

2. Assumption Synthesis. To help the user resolve the verification failure, Xenon uses the counterexample to synthesize a suggested fix. For example, Xenon may find a constant-time counterexample for a processor pipeline where the two different runs may execute two different ISA instructions (say, addition and division) which take different numbers of clock cycles. Yet, the execution of each instruction (for any inputs) may be constant-time. Xenon uses the counterexample to synthesize a minimal candidate set of secrecy assumptions (*e.g.*, that any two executions have the same, publicly visible sequence of instructions) which address the root cause of the verification failure (§ 4.2). The user then decides either to accept the candidate assumption or, if they do not match their intuition for the intended usage of the circuit, reject them, in which case Xenon computes an alternative. Internally, Xenon computes candidate assumptions via a reduction to integer linear programming [73].

3. Modular Verification. Finally, to scale verification and counterexample generation to larger and more complex hardware, and to keep counterexamples and suggested assumptions local, we introduce a notion of module summaries (§ 3). Module summaries 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. Our modular verification approach also allows the user to focus attention on one module at a time, which keeps errors and assumptions local, and helps to bootstrap the verification of large circuits (§ 7).

4. Evaluation. We implement Xenon and evaluate the impact of counterexamples, assumption synthesis, and modularity on the verification of different kinds of hardware modules (§ 6). We find that Xenon's solver-aided interactive verification process drastically reduces verification effort (*e.g.*, verifying the largest benchmark of [50] took us several minutes instead of multiple days) and, together with module summaries, allows us to scales verification to realistic hardware (*e.g.*, we verify the SCARV side-channel hardened RISC-V core [4], which is order of magnitude larger than the RISC-V cores verified by previous state-of-the-art tools). From a small (ten person) user study, in which users were tasked with verifying three circuits (an ALU, an FPU, and a full RISC core), we find that Xenon has large ($d = 1.62$), statistically significant ($t(8) = 2.56, p = .016$) positive effect on correct completion: Participants using Xenon were able to correctly complete significantly more tasks in the allotted time (40 min), and their solution sizes were (on average) smaller. On the most challenging task—a full RISC-V processor with a complex assumption set—no participant in the control-group succeeded, whereas 60% of the participants using Xenon were able to successfully complete the verification task.

```
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 1: A simple, constant-time lookup-table in Verilog, taken from [10].

6. Secrecy Assumptions. As a side product of the verification of SCARV, we obtain a set of annotations (§ 7) detailing secrecy assumptions under which SCARV is guaranteed to execute in constant-time. These secrecy assumptions, together with Xenon's source code are open source and available on GitHub (https://xenon.programming.systems). We hope that these artifacts will facilitate further efforts to provide end-to-end constant-time guarantees across hardware and software.

2 OVERVIEW

We start by reviewing how to specify and verify the absence of timing channels in Verilog hardware designs (§ 2.1), show how existing techniques fail to scale on real-world hardware designs, as these designs are often only constant-time under additional secrecy assumptions which are tedious to derive by hand (§ 2.2), sketch how Xenon helps to find secrecy assumptions automatically (§ 2.3), and finally discuss how Xenon exploits modularity (§ 2.4).

2.1 Verifying Constant-Time Execution of Hardware

Lookup Circuit. Figure 1 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 input `in` contains a secret value, producing output `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 2 makes this intuition more precise, using a recent definition of constant-time execution for hardware [50]. Instead of tracking timing indirectly through information flow [66, 82, 91] the definition uses a direct notion of timing. The figure shows two runs 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 1 to pass through the circuit and produce their respective outputs. For this, we put a "tracer" on the inputs by assigning a *liveness-bit* to each register. For some register `x`, we set its liveness-bit $\texttt{x}^\bullet$ to ★, if `x` has been influenced by the input at initial cycle 1 (we say `x` is *1-live*) and •, otherwise. Figure 2 shows how liveness-bits are propagated through the circuit.

time	in		out		in•		out•	
	L	R	L	R	L	R	L	R
1	h00	hff	X	X	★	★	•	•
2	h00	hff	h63	h2c	•	•	★	★

Figure 2: Two runs of Figure 1 showing values and liveness-bits for input (in) and output (out). X represents an undefined value.

Initially, in both executions, the input is *1-live* and the output is not. In cycle 2, both outputs *1-live* due to the case-split on the value of in. Assuming that an attacker can observe the liveness-bits of all outputs, here, register out, the attacker cannot distinguish the two executions, and we can conclude that the pair of executions is indeed constant-time.

Verifying Constant-Time Execution. To show constant-time execution, not only for the two runs in Figure 2, but for the whole circuit, we have to prove that for *any* pair of runs, that is, for any pair of inputs, and any initial cycle, the constant-time property holds. This can be achieved by constructing a *product circuit* [50][1] 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 x_L and x_R that hold the values of x in the left and right runs, respectively. We can then use the product circuit to synthesize invariant properties of the circuit. For example, let's define that a variable x is constant time (and write $ct(x)$), if for any pair of executions, its liveness-bit in the left execution $x_L^\bullet$ is always the same as its liveness-bit in the right execution $x_R^\bullet$, *i.e.*, $x_L^\bullet = x_R^\bullet$ always holds, for all initial cycles t. Then, the following invariant on the module proves constant-time execution, under the condition, that module inputs are constant-time: $ct(\texttt{in}) \Rightarrow ct(\texttt{out})$.

2.2 Real-World Hardware is Not Constant-Time

Unfortunately, unlike the simple lookup table from Figure 1, real-world circuits are typically *not* constant-time, in an absolute sense. Instead, when carefully designed, they are constant-time under specific *secrecy assumptions* detailing which circuit inputs are supposed to be *public* (visible to the attacker) or *secret* (unknown to the attacker). Thus, verification requires the user to painstakingly discover secrecy assumptions through manual code inspection, which can be prohibitively difficult in real-world circuits.

A Pipelined MIPS Processor. We illustrate the importance of secrecy assumptions using the program in Figure 3 which shows a code-fragment taken from one of our benchmarks—a simple, pipelined MIPS processor [9]. If the reset bit rst is set (Line 18), the processor sets several registers to zero (Line 19). Otherwise, the processor checks whether the pipeline is stalled (Line 21) and either forwards the current instruction from the instruction-fetch stage to the instruction-decode stage (Line 28) and advances the program counter (Line 29), or stalls by reassigning the current values (Lines 22 to 24).

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.*, that it is constant-time. Unfortunately, our example pipeline is *not* constant-time without any

[1]In architecture, this is often referred to as miter circuit [43].

```
1  // source : IF_pc; sink : WB_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
9  assign ID_rs = ID_instr[25:21];
10 assign ID_rt = ID_instr[20:16];
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 always @(posedge clk)
18   if (rst)
19     ID_instr   <= 0; ...
20   else
21     if (Stall == 1) begin
22       ID_instr <= ID_instr;
23       IF_pc    <= IF_pc;
24       EX_rt    <= EX_rt;
25       ...
26       WB_reg <= WB_reg;
27     end else begin
28       ID_instr <= IF_instr;
29       IF_pc    <= IF_pcn;
30       EX_rt    <= ID_rt;
31       ...
32       WB_reg <= WB_wd;
33     end
34   end
```

Figure 3: MIPS Pipeline Fragment.

further restrictions on its usage. For example, the execution time for a given instruction depends on whether the pipeline is stalled before the instruction is retired. This is illustrated in Figure 4. We model an attacker that can measure how long an instruction takes to move through the pipeline, *i.e.*, from *source* IF_pc to *sink* WB_reg. Such an attacker can distinguish the two runs in Figure 4, as the liveness-bits of WB_reg differ in cycle 3. This timing difference lets the attacker make inferences about the control flow of the program which is executed on the processor, and therefore any attempt to verify constant-time execution results in a failure.

2.3 Automatically Finding 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 4 would only leak information that the attacker is already aware of. However, assuming that Stall is public may not be the best choice. Stall is defined deep inside the pipeline which makes it hard to translate this assumption into a restriction on the kind of *software* we are allowed to execute on the processor. Instead, we want to

time	stall		ID_jmp		IF_inst•		ID_inst•		EX_rt•		WB_reg•	
	L	R	L	R	L	R	L	R	L	R	L	R
1	0	1	1	0	★	★	•	•	•	•	•	•
2	0	0	0	1	•	★	★	•	•	•	•	•
3	0	0	0	0	★	•	★	★	★	•	★	•

Figure 4: Two runs of Figure 3, where the right run stalls in cycle 1. The liveness bits of sink `wb_reg` differ in cycle 3 and therefore the circuit is not constant-time.

pick assumptions as closely as possible to the sources, *i.e.*, the *externally visible* computation inputs. For example, restricting program counter `IF_pc` to be public directly translates into the obligation that the executed program's control flow be independent of secrets.

To discover this assumption using existing technology, the verification engineer first has to *manually* identify that the timing variability is first introduced in variable `ID_instr` (Lines 22 and 28) due to a control dependency on `Stall`. They then need to inspect how `Stall` is set (Line 14) and painstakingly trace the definitions which may involve complex combinatorial logic (excerpt starting in Line 5) and circular data-flows to identify a promising candidate register, such that marking the register as *public* will render the circuit constant-time. Counterexamples like Figure 4 are often of little help as they are hard to interpret and fail to focus attention on the relevant parts of the circuit.

Solver Aided Verification: Xenon's Interactive Loop. Xenon drastically simplifies this time-consuming process through an interactive, solver-aided workflow that helps to find an optimal set of 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 'rst' as PUBLIC? [Y/n]
```

The user either answers with `Y` indicating that `rst` 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 choose `Y`: this marks `rst` public and re-starts Xenon for another verification attempt.

Step 2. Next, Xenon suggests marking `M_PCSrc` as public. Flag `M_PCSrc` indicates whether the current instruction in the memory stage contains an indirect jump. But whether an indirect jump is executed depends on register values (*i.e.*, `M_PCSrc` is set depending on whether the output of the ALU is zero) and therefore, indirectly, on the *data memory*. Assuming that `M_PCSrc` is public would lead to assumptions about the memory which we wish to avoid. Hence, we tell Xenon to exclude it in future verification attempts and restart verification.

Step 3. Restarting verification causes Xenon to suggest candidate variable `IF_pc`, the program counter of the fetch stage. We mark `IF_pc` as public as this directly encodes the assumption that the program's control flow does not depend on secrets. Xenon restarts the solver which proves that the program—under the inferred secrecy assumptions—executes in constant-time. This concludes the verification process. In addition to the assumptions that `rst` and `IF_pc` are public, Xenon also infers a set of usage assumptions that detail the parts of the pipeline that have to be flushed on context switches. These assumptions would otherwise have to be supplied by the user as well.

Counterexamples. When synthesizing assumptions, Xenon internally computes a constant-time *counterexample* which contains the set of variables that have lost the constant-time property earliest. While the user can simply follow Xenon's suggestions without further investigating the root cause of the violation, we find that—if the user chooses to do so—the counterexample often helps to further understand *why* the circuit has become non constant-time. In our example, Xenon returns as counterexample, variable `ID_instr`, for all three interactions. Indeed, inspecting the parts of the circuit where `ID_instr` is assigned focuses our attention on the relevant parts of the circuit, that is, the conditional assignment of `ID_instr` under `rst` (Line 19) and under `Stall` (Lines 22 and 28). We discuss how Xenon computes counterexamples using artifacts extracted from the failed proof attempt in § 4.1, and how Xenon uses them to synthesize an optimal set of secrecy assumptions via a reduction to integer linear programming in § 4.2.

2.4 Real-World Circuits Are Not Small

While Xenon's solver-aided, interactive verification loop significantly reduces the time the *user* has to spend on verification efforts, large real-world circuits often also present a challenge for the *solver*. This is because computing invariants and synthesizing assumptions naively requires a *whole-program* analysis. Hence, efficiency crucially depends on the *size* of the circuit we are analyzing.

Consider, for example, the AES-256 benchmark from [10]. Fig. 5 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 both assumption synthesis and verification all but intractable. (In fact, Xenon does manage to verify the naive, inlined circuit, however, a single verification run 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 1.

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.

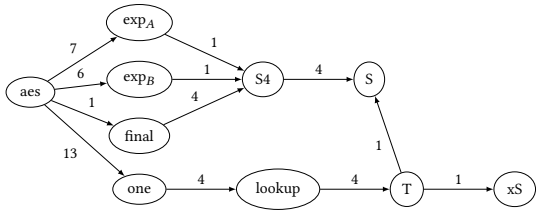


Figure 5: Module dependency graph of the AES-256 benchmark.

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

Figure 6: Module from the AES benchmark.

We can formalize this in the following module summary, which Xenon computes automatically: $ct(\texttt{in}) \Rightarrow ct(\texttt{out})$. Instead of inlining the module, both assumption synthesis, and verification can now use its summary thereby eschewing the code explosion. The code in Figure 6 shows an instantiation of module S in module S4. 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 for the full AES circuit, which only takes 3 seconds.

3 MODULAR CONSTANT-TIME VERIFICATION

We now formalize the concepts introduced in the overview. We first review a formal definition of constant-time execution for hardware circuits (§ 3.1) and its translation to Horn-clause verification conditions, which we use to encode the verification task (§ 3.2). We then modularize this naive encoding via summaries (§ 3.3). After that, we discuss counterexamples and assumption synthesis (§ 4).

3.1 Defining Constant-Time Execution

Configurations. Configurations represent the state of a Verilog computation. A configuration $\Sigma \triangleq (P, \sigma, \theta, c, t, \textsc{Src})$ is made up of a Verilog program P (say, the processor in Figure 3), a store σ, a liveness map θ, current clock cycle $c \in \mathbb{N}$, initial clock cycle $t \in \mathbb{N}$ and, finally, a set of sources $\textsc{Src} \subseteq \textsc{Vars}$. Store $\sigma \in \textsc{Vars} \to \mathbb{Z}$ maps variables Vars (registers and wires) to their current values; map $\theta \in \textsc{Vars} \to \{\bigstar, \bullet\}$ maps variables to liveness-bits; cycle t marks the starting-cycle of the computation we want to track and finally, Src identifies the inputs of the computation we are interested in.

Transition relation. Transition relation $\rightsquigarrow \in (\Sigma \times \Sigma)$ encodes a standard Verilog semantics which defines how a configuration is updated from one clock cycle to the next. We omit its definition, as it is not needed for our purposes, but formal accounts can be found in [50, 51, 91]. In addition to updating the store and current cycle, 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 liveness-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 Src and let $store(\pi, i) = \sigma_i$ and $live(\pi, i) \triangleq \theta_i$, for $i \in \{0, \ldots, n-1\}$.

Example. Consider again Figure 2. The Figure depicts two runs π_L and π_R of length 3 with respect to initial cycle 1 and source $\{\texttt{in}\}$ of the program in Figure 1. Columns in and out show $store(\pi, i)(in)$ and $store(\pi, i)(out)$, for $\pi \in \{\pi_L, \pi_R\}$ and $i \in \{1, 2\}$. Similarly, columns $\texttt{in}^\bullet$ and $\texttt{out}^\bullet$ show $live(\pi, i)(\texttt{in})$ and $live(\pi, i)(\texttt{out})$, for $\pi \in \{\pi_L, \pi_R\}$, and $i \in \{1, 2\}$. The Figure omits the initial state at cycle 0, where all liveness-bits are set to $\bullet$.

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

Secrecy Assumptions. A set of secrecy assumptions

$$\mathcal{A} \triangleq (\textsc{Flush}, \textsc{Pub})$$

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 $\mathcal{A}$, if, for each $v \in \textsc{Flush}$, v is flushed, and for each v in Pub, v is public. We describe how Xenon synthesizes secrecy assumptions in § 4.

Constant-Time Execution. We now define constant-time execution with respect to a set of sinks $\textsc{Snk} \subseteq \textsc{Vars}$, sources Src, and assumptions $\mathcal{A}$. 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 $\mathcal{A}$, and any sink $o \in \textsc{Snk}$, o is constant-time.

Example. Consider again Figure 2. If we assume that variables in cycle 0 have the same value as in cycle 1, then out is flushed while in is not. Neither in, nor out are public, but both are constant-time. As out is constant-time in all runs, the program in Figure 1 is constant-time with respect to the empty set of assumptions and sink $\{out\}$. In Figure 4, none of the variables are public or constant-time, however, the program in Figure 3 can be shown to be constant-time with $\textsc{Pub} = \{\texttt{IF\_pc}, \texttt{rst}\}$.

3.2 Verifying Constant-Time Execution via Horn Constraints

To verify constant-time execution, we mirror the formal definition in a set of Horn clauses [24]—an intermediate language for verification. We start with the naive, monolithic encoding and discuss how to make it modular in § 3.3. At high level, the constraints

$$init(vs_L, vs_R) \wedge flush \wedge pub \Rightarrow inv(vs_L, vs_R, 0, t) \qquad (init)$$

$$\begin{pmatrix} inv(vs_L, vs_R, c, t) \wedge pub \\ \wedge\ next(vs_L, vs_R, vs'_L, vs'_R, t) \end{pmatrix} \Rightarrow inv(vs'_L, vs'_R, c+1, t) \quad (cons)$$

$$inv(vs_L, vs_R, c, t) \wedge pub \Rightarrow o^{\bullet}_L = o^{\bullet}_R,\ for\ o \in \textsc{Snk} \qquad (ct)$$

Figure 7: Horn clause encoding of the verification conditions for constant-time execution.

(1) issue a new live instruction at a non-deterministically chosen initial cycle t, and
(2) ensure constant-time execution by verifying that the liveness-bits for each sink are always the same, in any two runs.

The clauses—shown in Figure 7—encode verification conditions over an inductive invariant $inv(vs_L, vs_R, c, t)$ of the product circuit, where vs ranges over all variables in the circuit and their respective liveness bits, and c and t are the current and initial cycles, respectively.

Initial States and Transition Relation. Formula $init(vs_L, vs_R)$ describes the product circuit's initial states and requires all liveness-bits to be set to $\bullet$. To ensure that the proof holds for any initial cycle, *init* does not constrain t. Formula $next(vs_L, vs_R, vs'_L, vs'_R, t)$ encodes the transition relation of the product circuit, where un-primed variables represent state before, and primed variables represent state after the transition. Like $\rightsquigarrow$, *next* sets liveness-bits of all sources to $\bigstar$ 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 $\mathcal{A} \triangleq (\textsc{Flush}, \textsc{Pub})$, we construct formulas *flush* and *pub*, both of which require the variables in their respective sets to be equal in the two runs. We let

$$flush \triangleq (\wedge_{x \in \textsc{Flush}}\ x_L = x_R) \text{ and } pub \triangleq (\wedge_{x \in \textsc{Pub}}\ x_L = x_R)\,.$$

Horn Constraints & Solutions. We then require that the invariant holds initially (*init*), assuming all variables in Flush and Pub are equal in both runs; that the invariant is preserved under the transition relation of the product circuit, assuming that public variables are equal in both runs (*cons*), and finally, that the liveness-bits of any sink are the same in both runs (*ct*). These constraints can then be passed to any of a vast array of existing Horn constraint solvers [8, 34, 37, 44, 52, 55, 57] yielding a formula which, when substituted for *inv*, makes all implications valid and thus proves constant-time execution.

Proof Artifacts. To compute constant-time counterexamples and synthesize secrecy assumptions upon a failed proof attempt, as described in the next section, Xenon requires the solver to generate the following artifacts:

(1) the *set of variables* which remained constant-time and public, during the current failed proof attempt, and
(2) the *order* in which the remaining variables lost the respective properties.

These artifacts can, for example, be extracted from a concrete counterexample trace like Figure 4. However, even if the solver is unable to produce concrete counterexample traces, the necessary information can often be recovered from the internal solver state.

3.3 Finding Modular Invariants

Naively, constructing *next* requires all the code to be in a *single* module. However, this can yield gigantic circuits whose Horn clauses are too large to analyze efficiently. To avoid instantiating the entire module at each usage site, Xenon constructs *module summaries* that concisely describe the timing relevant properties of the module's input and output ports.

Per-Module Invariants and Summaries. Instead of a single whole program invariant *inv*, the modular analysis constructs a per-module invariant inv_m, and an additional summary sum_m, for each module m. The summary only ranges over module inputs and outputs, and respective liveness-bits (`io`) and needs to include all input/output behavior captured by the invariant, *i.e.*, we add a clause: $inv_m(vs_L, vs_R, t) \Rightarrow sum_m(io_L, io_R, t)$. 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 (*init*), and be preserved under the transition relation (*cons*), 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 a 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 any sink has the same liveness-bits in both runs (*ct*). The summaries are also used to modularize our assumption synthesis algorithm § 4.3, which is crucial for our modular verification approach, as we will discuss in § 7.

Solving Modularity Constraints. To solve the modular Horn constraints, the solver first computes an invariant for each module, and then uses quantifier elimination [58] to project the module's behavior onto its inputs and outputs, which yields the summary. Since a module's summary may show up in another module's transition relation and thereby influence its invariant, this yields an interdependent constraint system, which we solve via a fix-point iteration loop [24, 52].

4 COUNTEREXAMPLES & ASSUMPTION SYNTHESIS

We now explain how Xenon uses the proof artifacts to help the user understand and explicate secrecy assumptions when verification fails. We first describe how Xenon analyzes the artifacts from the failed proof attempt in order to compute a *counterexample* consisting of the set of variables that—according to the information communicated by the prover—lost the constant-time property first (§ 4.1). Next, we discuss how Xenon uses the counterexample to synthesize a set of *secrecy assumptions* that eliminate the root cause of the verification failure (§ 4.2). This is done by computing a *blame-set* that contains the variables that likely caused the loss of constant-time for the variables in the counterexample via a control dependency. This blame set is then used to encode an optimization problem whose solution determines a minimal set of assumptions required to remove the timing violation. Finally, we briefly discuss

how Xenon uses module summaries to speed up counterexample generation and secrecy assumption synthesis (§ 4.3).

4.1 Computing Counterexamples

Dependency Graph. To compute the counterexample from a failed proof attempt, Xenon first creates a *dependency graph* $G \triangleq (V, D \cup C)$ which encodes data- and control-dependencies between program variables. G consists of

- *variables* $V \subseteq$ Vars,
- *data-dependencies* $D \subseteq$ (Vars×Vars), where $(v, w) \in D$ if v's value is used to compute w directly through an assignment, and
- *control-dependencies* $C \subseteq$ (Vars × Vars) where $(v, w) \in C$, if v's value is used indirectly, *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. Importantly, if for some v $varTime$ is undefined ($varTime(v) = \bot$), v was constant-time *throughout* the failed proof attempt. For any other variables v and w, if $varTime(v) < varTime(w)$, then v started to exhibit timing variability *before* w.[2] Xenon uses this map to break cyclic data-flow dependencies.

Computing the Counterexample. Using the data-flow graph, and $varTime$, Xenon computes a *reduced graph*. 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, it cannot be the cause for v losing the constant-time property. Finally, Xenon removes all nodes that cannot reach a sink node using the remaining edges. This leaves us with a set of variables Cex ⊆ Vars without incoming edges, which we present—as counterexample—to the user. We now define the reduced graph in more detail.

Reachability. For dependency 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 $i \in \{0, \ldots, n-2\}$. Finally, we 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) No node is constant-time, *i.e.*, for all $v \in V'$, $varTime(v) \neq \bot$.
(2) All edges respect the causal order given by $varTime$, *i.e.*, for all $(v, w) \in (D' \cup C')$, we have $varTime(v) \leq varTime(w)$.
(3) All nodes can reach a sink, *i.e.*, for all $v \in V'$, there is $o \in$ Snk such that o is reachable from v.

[2] More formally, for variables v, w if $varTime(v) < varTime(w)$, then there exist two runs π_L and π_R of some 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 $i < j$. This information can *e.g.*, be extracted from a concrete counterexample trace, like the one shown in Figure 4.

```
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
```

Figure 8: Simplified MIPS Pipeline Fragment in Verilog.

For variable v, and graph $G \triangleq (V, D \cup C)$, let $pre(v, G)$ be the set of its immediate predecessors in G, that is

$$pre(v, G) \triangleq \{w \mid (w, v) \in (D \cup C)\}.$$

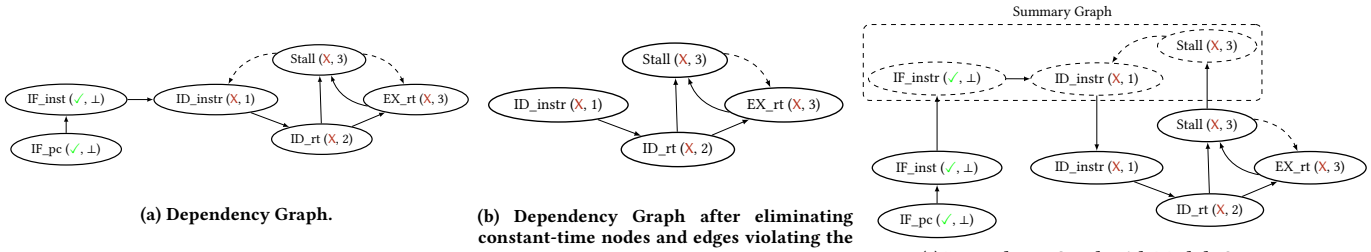
We define the counterexample Cex of a graph G with map $varTime$ as the set of nodes in the reduced graph G' (wrt. $varTime$), that have no predecessors, *i.e.*, $\text{Cex} \triangleq \{v \mid pre(v, G') = \emptyset\}$.

Example: Simplified Pipeline. The code in Figure 8 shows a simplified version of the pipelined processor from Figure 3. Like in Figure 3, the pipeline either stalls (Lines 10 and 11) if flag `Stall` is set (Line 9), or else forwards values to the next stage (Lines 13 and 14). 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. To check if the pipeline fragment executes in constant-time, we mark `IF_pc` as source, and `ID_instr` as sink and run Xenon. Since the pipeline is variable-time, the verification fails. To compute a minimal counterexample, Xenon creates the dependency graph shown in Figure 9a. 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 the variable remained constant-time throughout the proof attempt and (X) otherwise. Solid edges represent data- and dashed edges represent control-dependencies.

Reduced Dependency Graph. Figure 9b shows the dependency graph after removing all constant-time nodes and edges that violate the causal ordering. Xenon erases all nodes that cannot reach sink `ID_instr`. This only leaves `ID_instr` which we return as counterexample. The ordering induced by $varTime$ allowed us to break the cyclic dependency between variables `Stall` and `ID_instr`, thereby resolving the chicken-and-egg problem.

Remark. In case the proof artifact only partially resolves the cyclic dependencies, that is, $varTime$ only defines a partial order over non-constant-time variables, the reduced graph may still contain cycles, and therefore there may be no nodes without predecessor.



(a) Dependency Graph.

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

(c) Dependency Graph with Module Summary.

Figure 9: Figure 9a shows the dependency graph for Figure 8. Data-dependencies are shown as solid edges, and control-dependencies are shown dashed. Each node is labeled with its *varTime*-value and marked (✓) if the variable remained constant-time throughout the proof attempt and (X) otherwise. Figure 9b shows the dependency graph after eliminating constant-time nodes from Figure 9a, 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 9c shows the variable dependency graph with a summary graph extracted from the module summary.

We can however still apply our technique by computing the graph's strongly connected components and including all nodes in the respective component in the counterexample.

4.2 Assumption Synthesis

The previous step leaves us with a set of nodes Cex, which lost the constant-time property first. Since these nodes must have lost the constant-time property through a control dependency on a secret value, we can compute a set of variables Blame that are directly responsible: the immediate predecessors of Cex in the dependency graph with respect to a control dependency. Formally, for dependency graph $G = (V, D \cup C)$, we let

$$\textsc{Blame} \triangleq \{w \mid v \in \textsc{Cex} \wedge (w, v) \in C\} .$$

To synthesize secrecy assumptions that remove the constant-time violation, we could directly assume that all nodes in Blame are public. But 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 Integer Linear Programming (ILP). 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. 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 new set of 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.

$$m_v \geq p_v, \qquad \textit{if } v \in V',\ pre(v, G') = \emptyset \quad (1)$$

$$m_v + \left(\frac{\sum_{w \in pre(v,G')} p_w}{\#pre(v,G')}\right) \geq p_v \qquad \textit{if } v \in V',\ pre(v, G') \neq \emptyset \quad (2)$$

$$p_v = 1 \qquad \textit{if } v \in (\textsc{Blame} \setminus \text{No}) \quad (3)$$

$$m_v = 0 \qquad \textit{if } v \in \text{No} \quad (4)$$

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:

$$\sum_{v \in V'} w_v m_v, \qquad \text{(objective)}$$

and we let $w_v = (\min_{in \in \text{Src}} d(in, v))$. A solution to the constraints defines a set of assumptions $\mathcal{A} = (\textsc{Flush}, \textsc{Pub})$, with

$$\textsc{Flush} \triangleq \{v \in V' \mid m_v = 0 \wedge p_v = 0\}$$

and $\textsc{Pub} \triangleq \{v \in V' \mid m_v = 1\}$. The constraints can be solved efficiently by an off-the-shelf ILP solver.

Example: Simplified Pipeline. Consider again the simplified pipeline in Figure 8. As we identified `ID_instr` as counterexample in the previous step, we need to ensure that its blame-set consisting of all indirect influences is public. `ID_instr` only depends on `Stall`, and therefore we add constraint $p_{\texttt{Stall}} = 1$. Since all variables are secret (*i.e.*, we didn't make any public-assumptions yet), the reduced graph is equal to the original graph. For variables `IF_instr` and `ID_instr`, we get: $m_{\texttt{IF\_instr}} + p_{\texttt{IF\_pc}} \geq p_{\texttt{IF\_instr}}$ and

$$m_{\texttt{ID\_Instr}} + \frac{p_{\texttt{IF\_instr}} + p_{\texttt{Stall}}}{2} \geq p_{\texttt{ID\_Instr}}.$$

We obtain the following objective function:

$$m_{\texttt{IF\_pc}} + 2m_{\texttt{IF\_instr}} + 3m_{\texttt{ID\_instr}} + \ldots .$$

Sending the constraints to an ILP solver produces a solution, where $m_{\texttt{IF\_pc}} = 1$, and $m_v = 0$, for all variables $v \neq \texttt{IF\_PC}$, and $p_v = 1$, for all v. This corresponds to the following assumption set $\mathcal{A} \triangleq (\mathit{Flush}, \{\texttt{IF\_PC}\})$, where *Flush* includes all variables except `IF_PC`. This is exactly our 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 in Appendix A.

4.3 Modular Assumption Synthesis

To avoid a blowup in constraint size and to keep counterexamples and assumptions local, we want to avoid inlining 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` 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 Figure 8 to factor out the updates to `ID_instr` into a separate module. Xenon computes the following summary invariant, from which we create the graph in Figure 9c

$$ct(\texttt{IF\_instr}) \wedge pub(\texttt{Stall}) \Rightarrow pub(\texttt{ID\_instr})\ .$$

Since connecting the instantiated variables to the summary graph is equivalent to (Figure 9a), our analysis returns the same result.

5 IMPLEMENTATION

Xenon is split into front-end and back-end. Our front-end translates Verilog to an intermediate representation (IR) and associates secrecy assumptions with input and output wires. Our back-end translates this annotated IR into verification conditions (Horn clauses); when verification fails, we generate counterexamples 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) [8] and Z3 (4.8.1) [37] libraries for verification, and the GLPK (4.65) [7] library for synthesizing assumptions by solving the ILP problem of § 4. Our tool and evaluation data sets, including the secrecy assumptions discovered for SCARV (§ 7) are open source and available on GitHub at https://xenon.programming.systems.

6 EVALUATION

We evaluate Xenon by asking the following questions:

- **Q1:** Are constant-time counterexamples effective at localizing the cause of verification failures?
- **Q2:** Are the secrecy assumptions suggested by Xenon useful?
- **Q3:** What is the combined effect of counterexamples and secrecy assumption generation on the verification effort?
- **Q4:** Do module summaries improve scalability?
- **Q5:** Does Xenon reduce verification time by helping users find secrecy assumptions?
- **Q6:** How does using Xenon affect assumption quality?

To answer questions **Q1** and **Q2**, we use Xenon to recover the assumptions for the benchmark suite from [50]. These benchmarks include a MIPS and RISC-V core, ALU and FPU modules, and RSA and SHA-256 crypto modules. To answer **Q3** and **Q4**, we evaluate Xenon on two challenging new benchmarks, the SCARV "side-channel hardened RISC-V" processor [4] whose size exceeds the largest benchmark from [50] by a factor of 10, and a highly modular AES-256 implementation [10]. Finally, we conduct a user study to answer **Q5** and **Q6**, in which participants were asked to find assumptions for three benchmarks from [50]: two benchmarks with relatively simple assumptions (ALU and FPU) and RISC-V core with a more complex assumption set.[3]

Summary. Xenon's counterexample synthesis dramatically reduces the number of potential error locations users have to manually inspect (6% of its original size) and most of Xenon's assumption suggestions are accepted by the user (on average 81.67%). Module summaries are key to reducing verification times for certain designs (*e.g.*, for AES-256 summaries reduced the verification time from six hours to three seconds). 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. Our user-study findings indicate that—using Xenon—participants were able to correctly complete significantly more tasks, showing a very large ($d = 1.62$), statistically significant ($t(8) = 2.56, p = .016$) positive effect on correct completion. Participants in the test group produced fewer ($d = 1.03$) incorrect solutions ($t(5.5) = 1.63, p = .07$), and solution sizes were smaller on average.

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 used to answer **Q1**–**Q4**, we start with an empty set of secrecy assumptions and run Xenon repeatedly to recover the missing assumptions needed to verify the benchmark. We collect the following information after every invocation of the tool: the total number of variables that are variable-time and secret; the size of the counterexamples measured by the number of variables they contain; the number of assumptions Xenon suggests, and how many of these assumptions we reject; finally, we record the number of times we invoke Xenon to complete each verification task. 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—due to its size—we report the median of three runs.

User-Study Design. For our user study, we recruited ten participants who had some familiarity with software constant-time execution, but had never used Xenon or Iodine. The participants were randomly split into two equally sized groups: **Test** who were given Xenon and **Control** who were given Iodine. Participants using Iodine were given access to Iodine's counterexample outputs. After reading the instructions, both groups were given 40 minutes to complete the three tasks, *i.e.*, find assumptions for three Iodine benchmarks. For each task, we recorded the time taken to complete the task in minutes (**Time**), the number of annotations in the solution set (**Size**), and whether the solution was correct (**Crt**). We rejected solutions for ALU and FPU if they contained assumptions about the operands, and for RISC-V, if they contained assumptions about memory or the register-file.

Q1: Error Localization. To understand whether our counterexample generation is effective at localizing the cause of verification failures, we compare the number of variables in the counterexample to the total number of non-constant-time variables. The **CEX Ratio** column Table 1 reports the average ratio per iteration. We observe

[3] We include the larger RISC-V core to evaluate the hypothesis that Xenon benefits users even if many of Xenon's suggestions are eventually rejected. We chose this benchmark because Xenon achieves the lowest accept-ratio over all benchmarks.

Name	#LOC	#Assum		CT	Check (H:M:S)		# Iter	CEX	Sugg	Accept
		#flush	#public		Inlined	Modular		Ratio	Ratio	Ratio
MIPS [9]	447	28	3	✓	2.42	3.13	3	2.50%	1.73%	83.33%
RISC-V [6]	514	10	11	✓	13.21	10.23	5	16.24%	3.98%	46.90%
SHA-256 [11]	563	4	3	✓	7.21	8.90	2	4.28%	3.57%	100.00%
FPU [2]	1108	3	1	✓	9.10	11.54	1	0.33%	0.26%	100.00%
ALU [5]	1327	1	3	✓	2.01	2.29	2	0.88%	1.38%	75.00%
FPU2 [3]	272	24	4	✗	1.31	3.65	-	-	-	-
RSA	855	29	4	✗	2.87	1.51	-	-	-	-
AES-256 [10]	800	0	0	✓	6:05:01.82	2.74	-	-	-	-
SCARV [4]	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%*

Table 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 ratio of user-accepted assumptions to all Xenon suggestions. In the Total row, we use * to denote averages instead of sums. We do not run error localization on FPU2 and RSA because they are variable-time; AES-256 does not need any assumptions.

that fewer than 6% of non-constant-time variables are included in the counterexample. Since the total number of non-constant-time variables is typically on the order of hundreds (*e.g.*, the median (and geomean) number of non-constant-time variables across all benchmarks and iteration is 97 (94)), this dramatically reduces the number of variables the developer has to inspect in order to understand the violation. For the benchmarks that were variable time, the counterexamples also precisely pinpointed where in the circuit the constant-time property was violated. For example, in the FPU2 benchmark Xenon included the `state` register in its third iteration counterexample. This register indicates when the FPU's output is ready. Inspecting the register's blame-set (similar to the process described in § 2.3) revealed that its value is set depending on whether one of the operands to the division operation is NaN and thus the FPU clearly leaks information about its operands.

Q2: Identifying Secrecy Assumptions. To assess the quality of secrecy assumptions suggested by Xenon, we record the number of suggestions that the user accepts (*useful* suggestions) and the ratio of suggestions to the total number of secret variables the user would otherwise have to inspect manually. We find that most (on average 81.67%) of Xenon's suggestions are useful, reported in the **Accept Ratio** column of Table 1. Moreover, we observe that the number of variables included in the counterexamples is relatively small (**Sugg Ratio** column); on average, we only had to inspect 2.77% of the secret variables.

Q3: Verification Effort. Finally, as a rough measure of the overall verification effort, we count the number of user interactions, *i.e.*, the number of times we invoked Xenon after modifying our set of secrecy assumptions. Verifying the largest benchmark from [50], the Yarvi RISC-V core [6] took five invocations over several minutes. The final assumptions we arrived at were the same as the assumptions manually identified by the authors of Iodine in [50]; they, however, took multiple days to identify these assumptions and verify this core [61]. Verifying the SCARV core took thirty-four iterations and roughly three hours; this core is considerably larger (roughly 10×) 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 the need to understand complex implementation details irrelevant to the analysis.

Q4: 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 1 give 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, the benefits of module summaries become apparent. For AES-256, using module summaries reduces the query size by 99.7%, from 391.3 MB to 1.2 MB, which, in turn, reduces the verification time by three orders of magnitude—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 5). 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.

Q5: Reducing Verification Time. To determine whether Xenon helps users find annotations more quickly, we recorded the number of tasks that participants were able to correctly complete within the 40 minutes timeframe. Column **#Crt** of Figure 13 summarizes our results. Participants in the test group completed 2.6 tasks on average, while participants in the control group were only able to solve 1.4. Figure 10 shows the percentage of participants that were able to find a correct set of assumptions, split by task. A little over 50% of control group participants were able to complete the first task, while 75% were able to complete the second. In contrast, all

Figure 10: Percentage of participants who were able to correctly complete task.

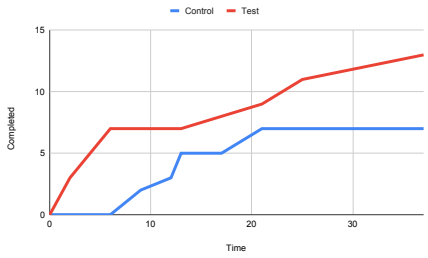


Figure 11: Correctly completed tasks over time.

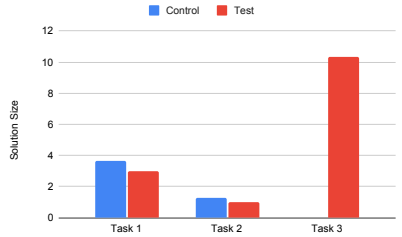


Figure 12: Solutions size for correctly completed tasks.

test-group users were able to correctly complete Task 1 and Task 2. None of the control group users correctly finished the significantly more difficult Task 3—verifying a full processor with a complex assumption set—whereas 60% of test-group participants were able to do so. We find these results encouraging, especially since our participants were non-expert users—some only with cursory exposure to hardware and RTL design.

Figure 11 shows the number of correctly completed tasks over time, split by group. Test group users were able to finish simple tasks more quickly and correctly solved more tasks over time.

Q6: Assumption Quality. To determine the quality of assumptions, we recorded how often a participant was able to finish a task within time, but found an incorrect set of assumptions. While the control group produced five incorrect solutions, the test group only produced one. Figure 12 shows the average size of correct assumption sets, per group. For tasks completed by both, the test-group sizes were on average smaller. Though the sample size is small, it's clear that Xenon helps guide the user's attention and avoid incorrect assumptions.

7 CASE STUDY: VERIFYING SCARV

We now describe our experience verifying SCARV and discuss the set of secrecy assumptions Xenon synthesized.

SCARV: Overview. SCARV is a 5-stage single-issue in-order CPU, implementing the RISC-V 32-bit integer base architecture. SCARV is side-channel hardened and explicitly designed to run cryptographic code. It supports an external hardware random number generator and implements fine-grained per-stage flushing of its processor pipeline via an instruction set extension.

Finding Assumptions Modularly. To verify SCARV, we follow Xenon's modular philosophy: we start with modules that occur at leaf-level in the instantiation-tree, that is, modules that have no sub-modules of their own, and iteratively work our way up such that in each stage, we already determined the assumptions for all sub-modules. At each step, we prove that the current module is unconditionally constant-time, where we set all module inputs as sources and outputs as sinks. This keeps errors and assumptions local: At every stage of the verification process, we only have to think about the current module. But this approach has a downside. We might end up with a set of assumptions that is unnecessarily restrictive. Our modular verification process ensures that *all* input/output paths of *all* submodules are constant-time. But, to ensure constant-time execution of the entire circuit, constant-time execution of only a *subset* of modules and their respective input/output paths might be required. Fortunately, we can use the assumptions found via our modular verification process to bootstrap a search for a minimal assumption set. As Xenon's module summaries can express that a module is constant-time only under certain conditions, and only for a subset of input/paths, we can safely erase assumptions, as long as Xenon can still prove the circuit to be constant time. Repeating this process yields a minimal assumption set, which we now discuss.

Sources and Sinks. Xenon represents assumptions as `yaml` files that are iteratively populated during verification. Figure 14 shows assumptions for the top-level module of SCARV. Annotations `src` and `snk` define sources and sinks, respectively. We choose all module inputs as sources, and all module outputs as sinks. This captures all relevant externally observable timing behaviors, including:

- The timing of signals interacting with both *instruction* and *data memory*, including requests (Lines 5 and 8), acknowledgments (Line 6), and strobe signals (Line 7),
- the timing of *flush signals* to external resources, such as caches (Line 9), and
- the timing of requests to the *external random number generator*, such as request ready bit (Line 10) and accept response bit (Line 11).

Secrecy Assumptions: External Devices. Annotation `pub` shows the secrecy assumptions synthesized by Xenon. At top-level, these assumptions concern external signals, hardware and interrupts. They require, for example, the external reset signal (Line 14), control inputs from external devices like memory (Lines 16 to 19), memory-mapped devices (Line 25), and the external random generator (Lines 21 to 23) to be public. The assumptions on memory are justified as they do not concern caches. Finally, the assumptions require that traps (Line 27), interrupts from external devices (Lines 29 to 31), and software (Lines 37 and 38) do not depend on secrets.

Importantly, neither are all inputs public nor are assumptions only placed on inputs. For example, Xenon doesn't synthesize assumptions requiring values read from data-memory (Line 2) or the external random source (Line 3) to be public. Conversely, the

	Task 1 (ALU)			Task 2 (FPU)			Task 3 (RISC-V)			#Crt
	Time	Size	Crt	Time	Size	Crt	Time	Size	Crt	
	2	3	✓	3	1	✓	20	10	✓	3
	2	3	✓	1	1	✓	14	11	✓	3
	5	3	✓	18	1	✓	14	10	✓	3
	21	3	✓	10	1	✓	-	-	-	2
	9	3	✓	4	1	✓	27	10	✗	2
μ	7.80	3.00	-	7.20	1.00	-	18.75	10.25	-	2.60
σ	7.92	0.00	-	6.91	0.00	-	6.18	0.50	-	0.55
μ^*	7.80	3.00	-	7.20	1.00	-	16.00	10.33	-	2.60
σ^*	7.92	0.00	-	6.91	0.00	-	3.46	0.58	-	0.55

(a) Test Group: Using Xenon

	Task 1 (ALU)			Task 2 (FPU)			Task 3 (RISC-V)			#Crt
	Time	Size	Crt	Time	Size	Crt	Time	Size	Crt	
	19	2	✗	2	1	✓	18	17	✗	1
	10	2	✗	5	3	✗	-	-	-	0
	9	3	✓	4	1	✓	12	1	✗	2
	9	5	✓	3	2	✓	-	-	-	2
	13	3	✓	21	1	✓	-	-	-	2
μ	12.00	3.00	-	7.00	1.60	-	15.00	9.00	-	1.40
σ	4.24	1.22	-	7.91	0.89	-	4.24	11.31	-	0.89
μ^*	10.33	3.67	-	7.50	1.25	-	-	-	-	1.40
σ^*	2.31	1.15	-	9.04	0.50	-	-	-	-	0.89

(b) Control Group: Using Iodine

Figure 13: Results of the user study. The participants were split into two equally sized groups: Test (Fig. 13a) using Xenon and Control (Fig. 13b) using Iodine. The participants were asked to find assumptions for three of the Iodine benchmarks: ALU, FPU, and RISC-V. Both groups were given 40 minutes to complete the three tasks. For each task, we record the time taken to complete the task in minutes (Time), the number of annotations in the solution set (Size), and whether the solution was correct (Crt). We reject solutions for ALU and FPU if they contain assumptions about operands, and for RISC-V, if they contain assumptions about memory or the register file. We report the average (μ) and standard deviation (σ) of all completed runs, *i.e.*, including those that yielded wrong solutions. μ^* and σ^* show average and standard deviation for correct runs, only. Finally, we report the overall number of correctly completed tasks #Crt. On average, participants in the test group were able to complete 2.6 tasks, while participants in the control group were only able to solve 1.4 within the 40 minutes trial. This indicates that using Xenon has a very large ($d = 1.62$), statistically significant ($t(8) = 2.56, p = .016$) positive effect on correct completion.

assumptions on traps, memory-mapped IO, and timer interrupts do not concern external inputs, but internal pipeline state.

Secrecy Assumptions: Internal Processor State. While, in our experience, top-level assumptions about IO behavior are relatively easy to find, proving constant-time execution also requires harder to find assumptions about processor internals. These assumptions encode constraints on the kind of programs that can safely be executed on the processor. Figure 15 shows the assumptions for SCARV's pipeline module. Xenon discovers the classic constant-time assumptions stating that control-flow (Lines 4 and 5) and memory-trace (Line 7) are secret independent. Similarly, memory stalls (Line 9), instruction validity (Line 11), and the computed μop (Line 13) must not depend on secrets.

But Xenon also discovers systems-level assumptions that are not commonly associated with constant-time programming. For example, access errors for control and status (CSR) registers (Line 15) must not depend on secrets, and the timing of when to return from machine-mode (Line 17) and traps (Lines 21 and 22), must be public. Finally, we may not set the configuration register of SCARV's leakage fence instruction (Line 19) depending on secrets. This is necessary as different configurations flush different parts of the pipeline and might incur different delays.

Constant-Time Subset of Instructions. The above assumptions are satisfied, when the instructions run on SCAR-V are limited to the arithmetic, and bitwise-logic subset of RISC-V; instructions must be valid, *i.e.*, properly encoded; even division is constant-time.

8 LIMITATIONS AND FUTURE WORK

We discuss some of Xenon's limitations.

Assumptions about Data. Xenon currently only discovers secrecy assumptions, *i.e.*, whether a given value is public or private. It may be beneficial to also discover assumptions about data (*e.g.*, that a certain flag is always set). In future work, we would like to explore how to combine Xenon's assumption synthesis method with techniques for inferring data preconditions [38, 74].

Minimality of Assumptions. Xenon inherits the limitations of the underlying Horn solver (§ 3). In particular, an assumption set could be sufficient to ensure constant-time execution of the circuit, but the solver may be unable to prove it. The minimality of our assumption set (§ 7) is therefore relative to the solver, and could potentially be improved with more precise solving methods—at the cost of reduced interactivity and scalability. As future work, one could use fast over-approximating solvers in the assumption discovery phase of our verification method (§ 7), and then slower, more precise solvers to minimize the assumption set after bootstrapping.

Mapping Back to Software. Xenon discovers an assumption set that ensures constant-time execution of the verified design. But, it leaves open the question of how to map assumptions back into proof obligations on software. The assumption set Xenon discovered for SCARV (§ 7) suggests that this might require a whole system effort that goes beyond current practices of constant-time programming. We hope that open-sourcing assumptions for SCARV will help future research efforts in this direction.

Guarantees on Synthesized Circuits. Finally, we prove constant-time at the Verilog level. This is convenient for error-localization, but it doesn't ensure that guarantees carry over to the generated circuits. Proofs are guaranteed to carry over if the synthesizer produces behavior within the Verilog standard [13], as, for example, formalized in [50, 51]. In particular, we make no further assumptions apart from semantics preservation. However, we leave the problem of verifying semantics preservation in synthesis to future work.

9 RELATED WORK

Verifying Leakage Freedom. There are various techniques, such as `ct-verif` [16], [22], and CT-Wasm [85], that verify constant-time execution of software, and quantify leakage through timing

```
src:
    dmem_rdata
    rng_rsp_data
snk:
    dmem_req
    dmem_ack
    dmem_strb
    imem_req
    leak_fence_unc0
    rng_req_valid
    rng_rsp_ready
pub:
    #external reset
    g_resetn
    #data&instr. mem
    dmem_error
    dmem_gnt
    dmem_recv
    imem_error
    #random generator
    rng_req_ready
    rng_rsp_valid
    rng_rsp_status
    #memory-mapped io
    mmio_error
    #internal trap
    int_trap_req
    #external interrupts
    int_extern_cause
    int_external
    mie_meie
    #global interrupts
    mstatus_mie
    #non-mask. interrupts
    int_nmi
    #software interrupts
    int_software
    mie_msie
    #timer interrupts
    ti_pending
    mie_mtie
```

Figure 14: Secrecy Assumptions Synthesized by Xenon for the Toplevel Module of SCARV.

```
pub:

    #control-flow
    cf_req
    cf_target
    #dest. address
    lsu_addr
    hold_memory_requests
    hold_lsu_req
    #decoded instr valid
    s1_valid
    #mu-op
    s2_uop
    #contr-stat regs
    csr_error
    ret_from_mach_mode
    exec_mret
    #fence
    leak_lkgcfg
    #traps
    trap_cpu
    trap_int
```

Figure 15: Secrecy Assumptions Synthesized by Xenon for the Main-Pipeline module of SCARV.

and cache side-channels [12, 17, 41, 65, 79, 90]. However, their analyses do not directly apply to our setting: They consider straight-line, sequential code, unlike the highly parallel nature of hardware. There are many techniques for verifying *information flow* properties of hardware. Kwon et al. [64] prove information flow safety of hardware for policies that allow explicit declassification and are expressed over streams of input data. SecVerilog [91] and Caisson [66] use information flow types to ensure that generated circuits are secure. GLIFT [82, 83] tracks the flow of information at the gate level to eliminate timing channels. Other techniques such as HyperFlow [45], GhostRider [67] and Zhang et al. [90] take the hardware and software co-design approach to obtain end-to-end guarantees. `dudect` [77], detects end-to-end timing variability across the stack via a black-box technique based on statistical measurements. Iodine [50], like Xenon, focuses on clock-precise constant-time execution, not information flow. Unlike Xenon, none of these methods provides help in elucidating secrecy assumptions, in case the verification fails—a feature we found essential in scaling our analysis to larger benchmarks. We see the techniques presented in this paper as complementary and would like to explore their potential for scaling existing verification methods for hardware and software.

Fault Localization. There are several approaches to help developers localize the root causes of software bugs [86]. Logic-based fault localization techniques [31, 42, 59, 60] are the closest line of work to ours. For example, BugAssist [59] 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, Xenon focuses on constant-time, which is a relational property, and hardware which has a substantially different execution model.

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 [47, 48, 72, 80, 81], 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 triage analysis reports by allowing the user to interactively determine the preconditions under which a program is safe or unsafe [38] or to identify the most general assumptions or context under which a given module can be verified safe [14, 27, 39, 40, 49]. Livshits et al. [68] infer information-flow specifications for web-applications using probabilistic inference. Unlike these efforts, our abduction strategy is tailored to 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.

Modular Verification of Software and Hardware. Xenon exploits modularity to verify large circuits by composing *summaries* of the behaviors of smaller sub-components. This is a well-known idea in verification; [78] shows how to perform dataflow analysis of large programs by computing procedure summaries, and Houdini [46] shows how to verify programs by automatically synthesizing pre- and post- conditions summarizing the behaviors of individual procedures. For hardware, model checkers like Mocha [18] and SMV [69] use rely-guarantee reasoning for modular verification. Kami [30] and [84] 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 a counterexample guided fashion [54, 92]. All focus on functional verification of properties of a *single* run, and do not support abstractions needed for timing-channels which require relational hyper-properties [32].

ACKNOWLEDGMENTS

We thank the reviewers for their suggestions and insightful comments, and our our user study participants for their effort and feedback on Xenon. This work was supported in part by the NSF under Grant Number CNS-1514435, CCF-1918573, and CAREER CNS-2048262; by ONR Grant N000141512750; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] [n.d.]. BearSSL - Constant-Time Crypto. https://www.bearssl.org/constanttime.html. (Accessed on 08/19/2020).

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

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

[4] [n.d.]. GitHub - scarv/scarv-cpu: SCARV: a side-channel hardened RISC-V platform. https://github.com/scarv/scarv-cpu. (Accessed on 08/16/2020).

[5] [n.d.]. GitHub - scarv/xcrypto: XCrypto: a cryptographic ISE for RISC-V. https://github.com/scarv/xcrypto. (Accessed on 08/19/2020).

[6] [n.d.]. GitHub - tommythorn/yarvi: Yet Another RISC-V Implementation. https://github.com/tommythorn/yarvi. (Accessed on 08/16/2020).

[7] [n.d.]. GLPK - GNU Project - Free Software Foundation (FSF). https://www.gnu.org/software/glpk/. (Accessed on 08/10/2020).

[8] [n.d.]. liquid-fixpoint: Horn Clause Constraint Solving for Liquid Types. https://github.com/ucsd-progsys/liquid-fixpoint. Accessed: 2018-08-29.

[9] [n.d.]. MIPS. https://github.com/gokhankici/iodine/tree/master/benchmarks/472-mips-pipelined. (Accessed on 08/16/2020).

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

[11] [n.d.]. Overview :: SHA cores :: OpenCores. https://opencores.org/projects/sha_core. (Accessed on 08/16/2020).

[12] [n.d.]. TIS-CT. http://trust-in-soft.com/tis-ct/.

[13] 2005. *IEEE Standard for Verilog Hardware Description Language.* IEEE Std 1364-2005.

[14] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 789–801. https://doi.org/10.1145/2837614.2837628

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

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

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

[18] Rajeev Alur and Thomas A. Henzinger. 1999. Reactive Modules. *Formal Methods Syst. Des.* 15, 1 (1999), 7–48. https://doi.org/10.1023/A:1008739929481

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

[20] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. 2018. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[21] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *IEEE Symposium on Security and Privacy*.

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

[23] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[24] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation*.

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

[26] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* (2005).

[27] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. [n.d.]. Compositional Shape Analysis by Means of Bi-Abduction. 58, 6 ([n. d.]), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[28] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rzk, and Gilles Barthe. 2020. Constant-time foundations for the new Spectre era. In *Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN.

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

[30] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN. http://plv.csail.mit.edu/kami/papers/icfp17.pdf

[31] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. 2013. Flow-Sensitive Fault Localization. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation* (2013), 189–208. https://doi.org/10.1007/978-3-642-35873-9_13

[32] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* (2010).

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

[34] CHC competition (CHC-COMP). [n.d.]. .

[35] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 2 (May 2018).

[36] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. BINSEC/REL: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *IEEE Symposium on Security and Privacy*.

[37] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

[38] Isil Dillig, Thomas Dillig, and Alex Aiken. [n.d.]. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012-06-11) *(PLDI '12)*. Association for Computing Machinery, 181–192. https://doi.org/10.1145/2254064.2254087

[39] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. *SIGPLAN Not.* 48, 10 (Oct. 2013), 443–456. https://doi.org/10.1145/2544173.2509511

[40] Isil Dillig, Thomas Dillig, Boyang Li, Ken McMillan, and Mooly Sagiv. [n.d.]. Synthesis of Circular Compositional Program Proofs via Abduction. 19, 5 ([n. d.]), 535–547. https://doi.org/10.1007/s10009-015-0397-7

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

[42] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error Invariants. *FM 2012: Formal Methods Lecture Notes in Computer Science* (Aug 2012), 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[43] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. In *DAC*.

[44] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2018. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*.

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

[46] Cormac Flanagan and K Rustan M Leino. [n.d.]. Houdini, an Annotation Assistant for ESC/Java. Springer, Berlin, Heidelberg, 500–517. https://doi.org/10.1007/3-540-45251-6_29

[47] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. *Computer Aided Verification Lecture Notes in Computer Science* (2014), 69–87. https://doi.org/10.1007/978-3-319-08867-9_5

[48] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. [n.d.]. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Principles of Programming Languages* (New York, NY, USA, 2016-01-11) *(POPL '16)*. ACM. https://doi.org/10.1145/2837614.2837664

[49] Roberto Giacobazzi. [n.d.]. Abductive Analysis of Modular Logic Programs. In *Proceedings of the 1994 International Symposium on Logic Programming* (Cambridge, MA, USA, 1994-11-01) *(ILPS '94)*. MIT Press, 377–391.

[50] Klaus V. Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *USENIX Conference on Security Symposium*.

[51] Michael J. C. Gordon. 1995. The semantic challenge of Verilog HDL. In *LICS*.

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

[53] Marco Guarnieri, Boris Koepf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *S&P*.

[54] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. 2008. Automated assumption generation for compositional verification. *Formal Methods Syst. Des.* 32, 3 (2008), 285–301. https://doi.org/10.1007/s10703-008-0050-0

[55] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV*.

[56] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy under Fire. In *USENIX Security*, David Wagner (Ed.).

[57] Hossein Hojjat and Philipp Rümmer. 2018. The Eldarica Horn Solver. In *FMCAD*.

[58] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. 2018. *Predicate Abstraction for Program Verification*. Springer International Publishing, Cham, 447–491. https://doi.org/10.1007/978-3-319-10575-8_15

[59] Manu Jose and Rupak Majumdar. [n.d.]. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *Computer Aided Verification* (Berlin, Heidelberg, 2011) *(Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 504–509. https://doi.org/10.1007/978-3-642-22110-1_40

[60] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI 11* (2011). https://doi.org/10.1145/1993498.1993550

[61] Rami Gökhan Kici. 2020. Personal communication.

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

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

[64] Hyoukjun Kwon, William Harris, and Hadi Esameilzadeh. 2017. Proving Flow Security of Sequential Logic via Automatically-Synthesized Relational Invariants. In *CSF*.

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

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

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

[68] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 75–86. https://doi.org/10.1145/1542476.1542485

[69] Kenneth L. McMillan. 1997. A Compositional Rule for Hardware Design Refinement. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings (Lecture Notes in Computer Science)*, Orna Grumberg (Ed.), Vol. 1254. Springer, 24–35. https://doi.org/10.1007/3-540-63166-6_6

[70] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL:TPM meets Timing and Lattice Attacks. In *USENIX Security*.

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

[72] Saswat Padhi, Rahul Sharma, and Todd Millstein. [n.d.]. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016-06-02) *(PLDI '16)*. Association for Computing Machinery, 42–56. https://doi.org/10.1145/2908080.2908099

[73] Christos H. Papadimitriou and Kenneth Steiglitz. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., USA.

[74] Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification Synthesis with Constrained Horn Clauses. In *PLDI*.

[75] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution.. In *USENIX Security*.

[76] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2016. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. In *USENIX Security*.

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

[78] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61. https://doi.org/10.1145/199448.199462

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

[80] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. [n.d.]. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7751–7762. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification.pdf

[81] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. [n.d.]. Code2Inv: A Deep Learning Framework for Program Verification. In *Computer Aided Verification* (Cham, 2020) *(Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, 151–164. https://doi.org/10.1007/978-3-030-53291-8_9

[82] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *ISCA*.

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

[84] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. [n.d.]. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification* (Cham, 2015) *(Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, 109–127. https://doi.org/10.1007/978-3-319-21668-3_7

[85] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-driven Secure Cryptography for the Web Ecosystem. *POPL*.

[86] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/tse.2016.2521368

[87] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. 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*. 859–874.

[88] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.

[89] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing.. In *NDSS*.

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

[91] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*.

[92] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. [n.d.]. Synthesizing Environment Invariants for Modular Hardware Verification. In *Verification, Model Checking, and Abstract Interpretation* (Cham, 2020) *(Lecture Notes in Computer Science)*, Dirk Beyer and Damien Zufferey (Eds.). Springer International Publishing, 202–225. https://doi.org/10.1007/978-3-030-39322-9_10

A EXAMPLE: NOT ALL VARIABLES BECOME PUBLIC

Example 3. One might think that Xenon requires all variables occurring in branch conditions to be annotated as public, however, this is not the case. Appendix A shows an example of such a program. Running Xenon produces the dependency graph shown in Figure 16. Xenon computes root-cause candidates by eliminating constant-time nodes and edges violating the precedence order. The result is shown in Figure 17. 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 liveness-bits, irrespective of the value of `cond`, *i.e.*, that $\texttt{tmp1}^\bullet = \texttt{tmp2}^\bullet$ holds irrespective of `cond`.

```
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
```

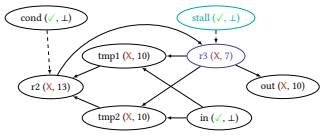


Figure 16: Example 3: Variable dependency graph.

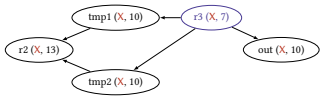


Figure 17: Example 3: Variable dependency graph after eliminating non-ct nodes and edges that violate the precedence relation.