VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties

Armaiti Ardeshiricham
University of California, San Diego
aardeshi@ucsd.edu

Yoshiki Takashima
University of California, San Diego
y1takash@ucsd.edu

Sicun Gao
University of California, San Diego
sicung@ucsd.edu

Ryan Kastner
University of California, San Diego
kastner@ucsd.edu

ABSTRACT

We present VeriSketch, a security-oriented program synthesis framework for developing hardware designs with formal guarantee of functional and security specifications. VeriSketch defines a synthesis language, a code instrumentation framework for specifying and inferring timing-sensitive information flow properties, and uses specialized constraint-based synthesis for generating HDL code that enforces the specifications. We show the power of VeriSketch through security-critical hardware design examples, including cache controllers, thread schedulers, and system-on-chip arbiters, with formal guarantee of security properties such as absence of timing side-channels, confidentiality, and isolation.

ACM Reference Format:
Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354246

1 INTRODUCTION

The prevalent way of designing digital circuits uses register-transfer level (RTL) hardware description languages (HDLs). It requires designers to fully specify micro-architectural features on a cycle-by-cycle basis. The verbosity and complexity of RTL HDLs opens the door for security vulnerabilities. With the growing number and severity of hardware security-related attacks [11, 16, 28, 31], we urgently need better tools for detecting and mitigating security vulnerabilities for hardware designs.

We propose the VeriSketch program synthesis framework for developing secure-by-construction hardware designs. VeriSketch frees hardware designers from exactly specifying cycle-by-cycle behaviors. Instead, the designer provides an RTL *sketch*, a set of security and functional specifications, and an optional set of soft constraints. VeriSketch outputs complete Verilog programs that satisfy all the specified functional and security properties, and heuristically favors designs that satisfy the soft constraints. The unique aspect of VeriSketch revolves around the use of program synthesis techniques and timing-sensitive hardware information flow analysis to enable the synthesis of hardware designs that are functionally correct and provably secure, as shown in the Fig. 1. VeriSketch employs Information Flow Tracking (IFT) methods to allow the definition and verification of security properties related to non-interference [46], timing invariance [3, 53], and confidentiality, and it extends counterexample-guided synthesis methods (CEGIS) [42] to hardware design.

VeriSketch uses CEGIS to complete the sketch by breaking the synthesis problem into separate verification and synthesis subproblems which can be solved by a SAT/SMT solver. In each verification round, the solver searches for a counterexample which falsifies the properties. During synthesis, the solver suggests a new design which adheres to the properties for the visited counterexamples. Iterating over these two stages, the algorithm either finds a design which has passed the verification round or the synthesis fails if the solver cannot propose a new design.

VeriSketch makes three extensions to CEGIS to enable synthesis of hardware designs with security objectives. First, VeriSketch runs CEGIS over a program which is automatically instrumented with IFT labels and inference logic. This enables reasoning about wider range of security properties based on the model of information flow. Second, VeriSketch extends CEGIS to synthesize sequential hardware designs with streams of inputs and outputs. This requires enforcing the properties over multiple clock cycles as outputs are continuously updated. This is done by expanding the formulation of SAT problems over multiple cycles bounded by the sequential depth of the circuit. Lastly, VeriSketch introduces heuristics to guide the search algorithm away from undesirable trivial designs, which is one of the major challenges of program synthesis frameworks. This is done by collecting and reasoning about both counterexamples and positive examples (i.e., input traces where properties fail and pass). Guided by the counterexamples, the synthesis algorithm finds a design which satisfies the properties, while positive examples are used to enforce soft constraints where properties are held. Soft constraints enable specification of design attributes which are preferable for improved quality but are not strictly necessary. The term soft constraint is used as opposed to hard constraints (i.e., our original properties) which should always hold. Through positive examples and iterative synthesis rounds, VeriSketch favors programs

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354246

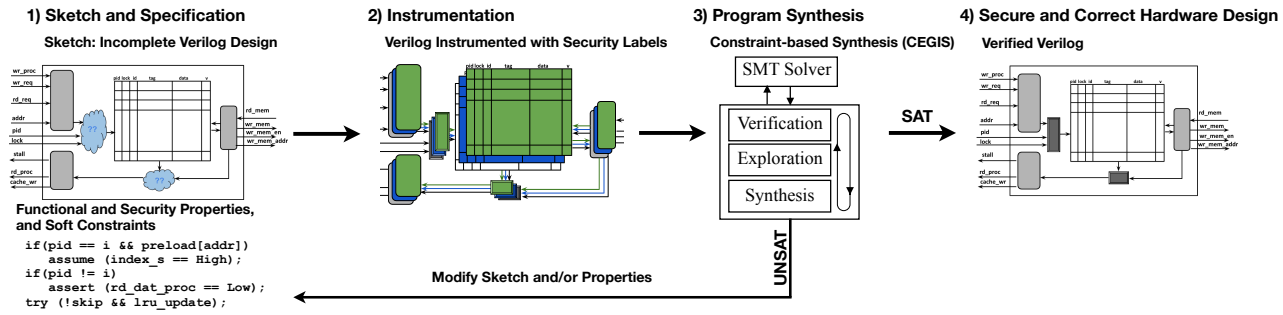


Figure 1: VeriSketch accepts as input an incomplete hardware design (i.e., a "sketch") and a set of functional and security properties and soft constraints. VeriSketch leverages hardware information flow tracking and program synthesis to build a Verilog design that satisfies the properties.

where soft constraints are held without changing the satisfiability of the synthesis problem.

We use VeriSketch to generate hardware units that adhere to various properties from sketches with different levels of details spelled out by the programmer. We synthesize a cache controller which is provably resilient against access-based timing side channel attacks. We design fixed point arithmetic units such that they are proven to run in constant time. Furthermore, we generate multiple SoC arbiters and hardware thread schedulers that enforce non-interference, timing predictability, and access control policies.

In all, we make the following contributions and organize the paper as follows. We introduce the VeriSketch framework for semi-automated synthesis of RTL hardware designs that enforce timing-sensitive information flow policies. Section 3 introduces the formal language definitions and main components of VeriSketch at a high-level. Next, we demonstrate how IFT analysis can be used to complete information flow constraints in Section 4. Section 5 focuses on introducing new program synthesis techniques that extend CEGIS for the synthesis based on information flow properties, sequential circuits with bounded depth, and soft constraints. We discuss the synthesized designs in Section 6.

2 BACKGROUND AND RELATED WORK

VeriSketch adopts and extends techniques from program synthesis and repair, as well as hardware information flow tracking systems. Here, we briefly review the related work in each of these domains.

2.1 Program Synthesis

Constraint-based synthesis is modeled as $\exists p \forall x.\ \phi(x, p)$ where ϕ denotes the design and specification, x is the design inputs and p is the synthesis parameter encoding the undefined portion of the design. The synthesizer's goal is to find parameter p such that the properties in ϕ are satisfied for all inputs x. CEGIS [2, 42, 43] introduces a method for breaking down the exists-forall quantification to iterations between *verification* and *synthesis* procedures that can be solved by SAT/SMT solvers. The *verification* phase at each round i fixes the parameter p to p_i and attempts to verify the universal conditions on all input combinations. The verification problem can be written as $\exists x.\ \neg\phi(x,\ p_i)$, which asks the solver to find a case where properties are violated for the program synthesized by parameter p_i. Unsatisfiability here indicates that properties holds for all input cases. Thus, p_i is a valid solution and the synthesis flow ends successfully. If satisfiable, the solver provides a counterexample x_i which falsifies the properties. The *synthesis* stage looks for a new parameter that satisfies the properties for all the previously visited counterexamples. This problem in round i can be modeled as: $\exists p.\ \bigwedge_{x_j \in CE} \phi(x_j,\ p)$, where CE is the set of visited counterexamples. If the solver fails to find a solution, the synthesis flow terminates unsuccessfully indicating that the properties are unsatisfiable for the given sketch.

Synthesis techniques are widely used to automate difficult software engineering tasks [17, 22, 27, 37, 43]. Program synthesis have been employed in different domains such as data processing [41, 51], data completion [18, 47], databases [52, 54], and more recently in security applications [24, 38]. In the HDL domain, Sketchilog [6, 7] translates partially written Verilog code to complete ones by directly solving the exists-forall problem employing a QBF solver. Sketchilog can only synthesize small combinatorial circuits, and is not scalable due to the limitations of QBF solvers. Furthermore, Sketchilog does not support expressive properties as high level specifications. VeriSketch extends CEGIS to enable synthesis of combinational and sequential circuits written in HDLs from high level specifications. Our problem statement is similar to that of program repair techniques for automatically generating patches for security-critical programs [20, 21, 23, 44]. Our work is unique from these previous works because we enforce security and functional properties while synthesizing incomplete hardware designs. Counterexample guided algorithms have been used to automatically synthesize device drivers [39, 40] and generate abstraction models for SoCs [45] and ISAs [25]. Similar techniques have been used at the gate level to automatically modify a netlist when errors are detected late in the design flow [10, 50]. VeriSketch uses CEGIS at a higher level of abstraction to complete partial HDLs with respect to security properties and acquaint the traditional hardware design flow with automated policy enforcement.

2.2 Information Flow Control

VeriSketch leverages hardware-level information flow analysis to reason about security properties. Hardware IFT tools can be broadly

divided into two categories based on whether they introduce new HDLs enabling definition of security labels [29, 30, 53] or rely on automated label inference rules [3, 4, 46]. Here, we take the latter approach in order to enable integration of flow tracking with sketching and synthesis. The structure of common HDLs facilitate precise analysis of information flow policies and detection of timing leakage (refer to Remarks 4.8, 4.11 and 4.13). VeriSketch adopts the approach from Clepsydra [3] which provides a sound labeling system for precisely capturing timing flows in RTL designs and verifying timing invariance properties. We extend and formalize Clepsydra's label inference rules and integrate them with program synthesis techniques to automatically enforce timing-sensitive information flow policies.

2.3 Motivating Example

To illustrate the challenges of secure hardware design, we take design of a cache that is resilient to timing side channel attack as an example, and show how it is done via the traditional hardware design flow versus by using VeriSketch. Unfortunately, modifying hardware designs according to security requirements is often not trivial; even the foremost hardware security experts can make errors as we discuss in the following.

2.3.1 Threat Model. We consider the Percival attack model [36] where the adversary runs concurrently with the victim process on a Simultaneous Multi-Threading processor. The adversary is an unprivileged user process which is isolated from the victim process, i.e., it does not share the address space with the victim. The attacker aims to learn information about the addresses which the victim uses to access the cache. The attack relies on the fact that in certain RSA implementations parts of the encryption key is used to look up a precomputed table in the cache. Hence, by observing the cache access pattern of the victim process, the adversary could gain knowledge about the key. While the Percival attack originally targeted the OpenSSL implementation of the RSA algorithm, similar attacks can target different applications where the cache index is driven from secret data [28]. In order to launch the attack, the adversary repeatedly fills the cache with its own data and measures each access time. Once the victim accesses some cache line, it evicts the attacker's data from that line. This eviction increases the attacker's access time in the following round.

2.3.2 Traditional Secure Hardware Design Flow. Assume that the designers decide to implement the partition locked cache (PLCache) mitigation technique [53] to secure the cache against the described attack model. PLCache enables processes to preload and lock sensitive data in the cache to avoid eviction and timing variations. It extends a "normal" cache controller with logic that arbitrates access to the cache based on the security requirements. As a proof of concept, we created a Verilog design of the PLCache based upon the details in their paper. We instrumented it and verified it against the IFT properties modeling cache timing leakage (described in Example 4.17), and *the security verification failed.* Analyzing the counterexample trace given by the verification tool, we discovered that the side channel manifests itself through the cache metadata related to the cache replacement policy. PLCache uses a least recently used (LRU) policy even for the locked data: *in case of a cache hit, normal cache access is performed.* This introduces a subtle timing side channel that can be exploited by extending the Percival attack (described in Section 6.3.1). This shows that even the foremost security experts can create mitigation strategies that have flaws that go undiscovered for more than a decade. And even worse, the designer is now stuck with developing a new strategy to fix this flaw. In this work, we show how we use VeriSketch to synthesize a cache which is provably resilient against the described attack model.

3 THE VERISKETCH FRAMEWORK

VeriSketch synthesizes incomplete hardware designs that adhere to the specified security and functional properties. It targets designs at the register transfer level (RTL) abstraction. RTL remains the prevalent way of specifying hardware designs and it has the required information to precisely analyze timing-sensitive information flow properties and identify timing side channels. We first give an overview of the main components of VeriSketch and then describe the details of the language design.

3.1 Main Components

Fig. 1 describes the VeriSketch framework, which converts an RTL sketch and a set of hard and soft constraints into a complete Verilog design. All inputs are written in the VeriSketch language, which extends Verilog with sketch and IFT specification syntax (see Table 1). As we show in the rest of this section, the VeriSketch language facilitates the modeling of security properties and a partial description of the hardware. The sketch is first translated to a Verilog design which contains synthesis parameters. The Verilog design is then instrumented with IFT analysis logic. This step (discussed in Section 4) enables reasoning about security properties alongside the functional ones. The instrumented design is given to the synthesis engine (described in Section 5) which uses constraint-based synthesis to resolve the parameters. If the synthesis succeeds (i.e., a parameter is found), the post-processor fills out the initial sketch based on the parameters values and discards the IFT instrumentation. Otherwise, the programmer has to repeat the process after relaxing the specifications or modifying the sketch.

3.2 VeriSketch Language

VeriSketch extends the standard Verilog language [13] with sketch constructs and security property specifications. The formal syntax is shown in Table 1.

3.2.1 Sketch Syntax. Sketches are language constructs that facilitate writing partial programs [43]. VeriSketch enables users to describe a partial hardware design by combining low-level and high-level sketch constructs with the original Verilog syntax. With low-level sketches, the designer can define unknown n-bit constants ($n(??)$), operation select (e_1 ($bop_1, \ldots,$ bop_m) e_2), operand select ($sel(e_1, \ldots, e_m)$), or choose the value of a variable from any of n previous cycles ($step?(v, n)$). To facilitate higher level sketching, VeriSketch introduces hardware-specific sketch constructs for describing arbitrary combinational ($y = comb(x_1, \ldots, x_m)$) and sequential circuits ($y = seq(x_1, \ldots,$ $x_m)$) with inputs $x_1, \ldots, x_m$, and procedural statements with unknown control flow (v ?= e_0). The original Verilog language supports two types of assignments: continuous and procedural assignments. Continuous assignments

are specified by the keyword `assign` and are used to specify combinational logic. Procedural assignments are only activated when they are triggered (e.g., by each rising edge of the clock signal) and are used to describe complex timing behaviour. Procedural assignments can be either blocking (=) or non-blocking (<=) which indicates if the statements are executed sequentially or in parallel. VeriSketch allows sketches of procedural assignments with unspecified control logic using the v ?= e_0 $[e_1, ..., e_m]$ syntax. This is synthesized to a blocking or non-blocking assignment where a function of $[e_1, ..., e_m]$ signals is used as the control logic. The list of control variables $[e_1, ..., e_m]$ can be defined separately for each statement or for the whole design.

3.2.2 Pre-Processing. Sketch constructs are compiled to synthesis parameters in the pre-processing round. The unknown constants are directly replaced by parameters. Operand and operation selects are modeled as multiplexers where control lines are parameters. $step?(v, n)$ is mapped to a shift-register where one of its n slots is selected by a synthesis parameter. v ?= e_0 $[e_1, \ldots, e_m]$ is translated into a block where assignment of e_0 to v is guarded with an unknown control signal defined by $comb(e_1, \ldots, e_m)$. The *comb* construct is compiled to a Binary Decision Diagram (BDD) template where the nodes are the inputs to the *comb* function. The leaves of the tree are replaced by synthesis parameters. Hence, $y = comb\ (x_1, \ldots,\ x_m)$ is translated to $y = (p_1 \wedge x_1 \wedge \ldots x_m) \vee \ldots \vee (p_{2^m} \wedge \neg x_1 \wedge \ldots \neg x_m)$ where $\{p_1, \ldots, p_{2^m}\}$ are synthesis parameters. The *seq* construct generates a finite state machine with binary encoded states where all state transitions are parameters driven by the inputs. Thus, $seq\ (x_1,\ \ldots,\ x_m)$ is mapped to an FSM where transitions from any state s_i to unknown state p_{ij} are conditioned on "$\{x_1,\ \ldots,\ x_m\} = q_j$" where p_{ij} and q_j are synthesis parameters. The template FSM receives its caller *reset* and *clock* signals.

While high-level constructs (i.e., *comb*, *seq* and ?=) greatly simplify sketching by providing generic templates for combinational and sequential circuits and procedural statements, they adversely affect synthesis time since the parameter size grows exponentially according to the number of data and control inputs. Consequently, these templates should be used sparingly if possible, e.g., to synthesize small but critical parts of the design.

3.2.3 Specification Syntax. Property specifications are logical formulas which express an implementation-agnostic relationship between design variables and describe a desired invariant in the design's behavior. VeriSketch introduces syntax for specifying properties using an information flow model and also supports properties written in the *System Verilog Assertion* language. VeriSketch uses two labels (s and t) corresponding to logical and timing flows for specifying information flow properties. These labels are binary values similar to design variables (i.e., $L \in \{Low,\ High\}$). Security properties are expressed by initializing labels of the input variables and constraining the labels of the output or intermediate variables. Alternatively, information flow properties can be more abstractly stated by $\nrightarrow$ and $\nrightarrow_t$ operators. These operators indicate absence of logical and timing flows from left hand-side to right hand-side. Properties written over the security labels or the design variables form the specification using `assume`, `assert`, or `try` keywords. `assume` restricts the analysis to cases where the inner expression is true

Table 1: VeriSketch Syntax.

	$v \in Vars$	Variable
	$n \in Nums$	Constant
e	::= $v \mid n \mid$ **uop** $e \mid e_1$ **bop** $e_2 \mid$ n (??) $\mid$ **step**? $(v,\ n) \mid$ **sel** $(e_1, \ldots, e_m) \mid$ e_1 (**bop**$_1, \ldots,$ **bop**$_m$) $e_2 \mid$ (**uop**$_1, \ldots,$ **uop**$_m$) $e \mid$ **comb** $(e_1, \ldots, e_m) \mid$ **seq** $(e_1, \ldots, e_m)$	Expression
a	::= **assign** $v = e$;	Continuous Assignment
s	::=$v = e; \mid v \Leftarrow e; \mid$ **if** $(e)\ s_1$ **else** $s_2 \mid$ **begin** $s_1 \ldots s_m$ **end** $\mid$ **for** $(v = n_1 : n_2)\ s; \mid$ v ?= e_0 $[e_1, \ldots, e_m]$	Procedural Assignment
γ	::= **posedge** clk $\mid$ **negedge** clk $\mid$ * $\mid \vec{v}$	Trigger
B	::= **always** @ $(\gamma)\ s \mid a$	Block
M	::= $B_1 \ldots B_m$	Module
S	::= $M_1 \ldots M_m$	Sketch
L	::= $v_s \mid v_t$	Label
p	::= $v \mid$ **uop** $v \mid v_1$ **bop** $v_2 \mid$ $L \mid$ **uop** $L \mid L_1$ **bop** $L_2 \mid$ $v \nrightarrow v \mid v \nrightarrow_t v$	Property
C	::= **assume** $(p) \mid$ **assert** $(p) \mid$ **try** (p)	Spec.

while `assert` causes the verification to fail once the inner expression is false. `try` is unique to VeriSketch and is used to model soft constraints.

```
module Sketch_Cache(…);
 assign skip =
 comb(rd_rq,wr_rq,hit,lru_block[m]);
 assign lru_update =
 comb(rd_rq,wr_rq,lock,stall,waiting);
 always @ (posedge clk)
  if(!skip)
    //cache rd/wr
    if(lru_update)
      //update LRU
  else
    //direct memory access
endmodule
```

(a)

```
module Sketch_Cache(…);
 assign skip = !hit &
 comb(rd_rq,wr_rq,lru_block[m]);
 assign lru_update = (c_rd | c_wr)
&& lock == ?? && stall ==??;
 always @ (posedge clk)
  if(!skip)
   //cache rd/wr
   if(lru_update)
    //update LRU
  else
   //direct memory access
endmodule
```

(b)

Figure 2: Sketching the control logic for a modified and secure version of PLCache. (a) A high-level sketch written in VeriSketch. *comb* denotes a combinational circuit where the implementation is totally unspecified. (b) Another sketch for the same design with more provided details.

Example 3.1 (Sketching a Secure Cache)**.** Fig. 2 shows two example sketches for designing the locking strategy similar to PLCache but eliminating the metadata timing side channel (and any other security flaws). We define the structural connections between the elements of the secure cache similar to a "normal cache" and leave the tricky control and arbitration logic for VeriSketch to decide. One major aspect of the partitioning mitigation technique is specifying the logic for the `skip` signal which we leave as undefined. `skip`

makes the decision about whether to follow a normal cache access or perform a direct memory access. We also add sketch constructs to decide when the cache LRU bits are updated. We manually extend the cache blocks to store the lock status of the stored data similar to PLCache. The difference between the sketches in Fig. 2 is the amount of detail provided by the designer and conversely that which is left to be determined by VeriSketch. Fig. 2(a) is a high-level sketch; it states that the `skip` signal should be some combinational function of the signals `rd_rq`, `wr_rq`, `hit` and `lru_block[m]`. Here, `lru_block` is the cache block selected for eviction according to the replacement policy and `m` is the index of the bit which stores the lock status of the block. The sketch for determining how the LRU bits are updated is a combinational function depending on the signals `rd_rq`, `wr_rq`, `lock`, `stall` and `waiting`. Here, `lock` is the incoming lock request and `waiting` shows if the cache is accessing the memory. The sketch in Fig. 2(b) has more detail; here the designer provided additional information that the `skip` signal is low when there is a cache hit and the structure of the logic driving the `lru_update` signal is given. The `??` syntax assumes one bit if not specified.

4 INFORMATION FLOW TRACKING

Traditional HDLs like Verilog and VHDL lack a framework for capturing security traits. Information flow models enable the analysis of a wide range of hardware security properties such as confidentiality, integrity, isolation, and timing side channels. IFT tools define labels which convey security attributes of design variables (e.g., whether or not that variable contains sensitive or untrusted information). IFT models capture how data moves through the system, enabling an analysis of security behaviors of the hardware design. For instance, in order to assess unintended data leakage in a design, secret inputs are initialized with a *High* label. Next, the design is analyzed to ensure that public outputs maintain a *Low* label, which indicates that no secret data has reached these ports.

4.1 VeriSketch IFT Framework

VeriSketch tracks information flow by annotating each design variable (wire or register) v with two different security labels, v_s and v_t, where the s-labels track logical flows and t-labels track timing flows. Inference rules for propagating these labels are formalized in Table 2. VeriSketch defines the propagation rule for each assignment within the same block by using the same syntax as of the original assignment. For instance, label of a register which is updated in a non-blocking procedural assignments is defined via a non-blocking procedural assignments while label of a wire which is driven by combinational logic is defined using combinational logic as well. This ensures that variables and their labels are updated simultaneously. VeriSketch performs precise label propagation, i.e., all label updates take into account the exact Boolean values of the design at the given time. This is enabled by modeling labels and inference rules with standard Verilog syntax and leveraging EDA tools to reason about the IFT labels and design variables at once.

Example 4.1. Fig. 3 shows the IFT instrumentation for a snippet of Verilog code implementing a cache unit. Lines $1-3$ and $7-16$ show how instrumentation for combinational and sequential blocks are done within the same block following the syntax of the original code. Note that all assignments and nonblocking statements are executed simultaneously in Verilog. Hence, all variables (e.g., `stall`) and their labels (e.g., `stall_s` and `stall_t`) are updated at the same time. We will discuss the detail of the right hand-side logic in the following subsections.

Table 2: VeriSketch Label Inference Rules.

$$\frac{\Gamma \vdash e :: (s,t)}{\Gamma \vdash \text{uop } e :: (\text{uop}_{\text{ift}}\ (e,s),t)}\ \text{T-uop}$$

$$\frac{\Gamma \vdash e_1 :: (s_1,t_1),\quad e_2 :: (s_2,t_2)}{\Gamma \vdash e_1 \text{ bop } e_2 :: (\ \text{bop}_{\text{ift}}\ (e_1,s_1,e_2,s_2), t_1 \sqcup t_2)}\ \text{T-bop}$$

$$\frac{\Gamma \vdash e :: (s,t),\quad \text{assign } v = e}{\Gamma \vdash v :: (s,t)}\ \text{T-assign}$$

$$\frac{\Gamma \vdash e :: (s,t),\quad v =_\eta e,\quad \text{c}_i \in Ctrl(v) :: (s_i,t_i)}{\Gamma \vdash v :: (s \sqcup s_i, t \sqcup t_i \sqcup (\neg Bal(v) \sqcap s_i))}\ \text{T-blocking}$$

$$\frac{\Gamma \vdash e :: (s,t),\quad v \Leftarrow_\eta e,\quad \text{c}_i \in Ctrl(v) :: (s_i,t_i)}{\Gamma \vdash v :: (s \sqcup s_i, t \sqcup t_i \sqcup (\neg Bal(v) \sqcap s_i))}\ \text{T-nonblocking}$$

Remark 4.2. The blocking ($=_\eta$) and nonblocking ($<=_\eta$) assignments for statement η differ in that blocking assignments are performed sequentially while the nonblocking ones run in parallel. They have the same inference logic according to Table 2, to ensure that variables and their labels are updated simultaneously.

Remark 4.3. Labels of variables defined via procedural assignments are triggered by the same event as the original statement and are defined in the same block. This ensures synchronous updates to variables and their labels.

In the following, we go over the details of the label inference rules. Note that since sketch constructs are pre-processed before the instrumentation, the inference rules are only defined for the original Verilog syntax.

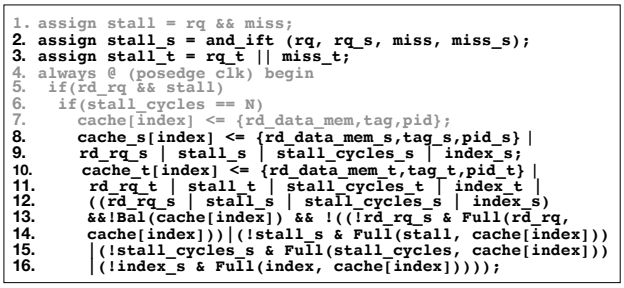
4.2 Tracking Logical Flows

Logical flows are tracked via label v_s defined for each variable v. VeriSketch tracks both explicit and implicit flows (i.e., flow of information via the data path and the control path). Explicit flows are tracked by instrumenting each operation.

Definition 4.4 (IFT Operator)**.** Let op be a valid binary/unary operator in Verilog RTL. IFT operator op_{ift} computes the label of op's output based on its inputs' values and labels.

For instance, explicit flows of assignment $z = x\ bop\ y$ are tracked via $z_s = bop_{\text{ift}}(x, x_s, y, y_s)$. In the simplest case, z_s is the join ($\sqcup$) of x_s and y_s. In a more precise analysis (i.e., lower number of false positives), z_s also depends on the Boolean values (i.e., x and y) and the operator's functionality [4, 26]. IFT operators are pre-defined and stored in VeriSketch IFT library where label tracking precision level is controllable by the user.

Implicit flows for each statement are tracked by upgrading the label of the left hand-side variable according to the labels of variables which control the statement's execution.



```
1. assign stall = rq && miss;
2. assign stall_s = and_ift (rq, rq_s, miss, miss_s);
3. assign stall_t = rq_t || miss_t;
4. always @ (posedge clk) begin
5.  if(rd_rq && stall)
6.   if(stall_cycles == N)
7.     cache[index] <= {rd_data_mem,tag,pid};
8.     cache_s[index] <= {rd_data_mem_s,tag_s,pid_s} |
9.     rd_rq_s | stall_s | stall_cycles_s | index_s;
10.    cache_t[index] <= {rd_data_mem_t,tag_t,pid_t} |
11.     rd_rq_t | stall_t | stall_cycles_t | index_t |
12.     ((rd_rq_s | stall_s | stall_cycles_s | index_s)
13.     &&!Bal(cache[index]) && !((!rd_rq_s & Full(rd_rq,
14.     cache[index]))|(!stall_s & Full(stall, cache[index]))
15.     |(!stall_cycles_s & Full(stall_cycles, cache[index]))
16.     |(!index_s & Full(index, cache[index])));
```

Figure 3: VeriSketch IFT framework automatically extends Verilog code with IFT labels and inference rules. The example is a portion of a cache. The gray lines here are the original code and the instrumentation is shown in black. Logical and timing flows are captured via s-labels and t-labels.

Definition 4.5 ($Ctrl(v)$)**.** Let η be a procedural assignment. $Ctrl(\eta)$ is the set of all variables which control the execution of η. $Ctrl(v)$ is the union of all $Ctrl(\eta_i)$ where η_i is a procedural assignment where v is the l-value variable. $Ctrl(v)$ is determined statically by analyzing the program control flow graph.

It immediately follows that:

Proposition 4.6 (c.f. [34])**.** Implicit flows via each procedural statement η with l-value variable v can be conservatively estimated by:

$$\bigsqcup \{c_{i_s} : c_i \in Ctrl(v)\} \tag{1}$$

Notation 4.7. We use join ($\sqcup$) and meet ($\sqcap$) to describe the inference rules in a generic multi-level security system. Since we consider binary operations, these operations can be replaced by disjunction ($\vee$) and conjunction ($\wedge$).

Remark 4.8. Note that grammar of the Verilog language and similar HDLs only permits assignments to each variable in a single block as all blocks are executed in parallel. Hence, $Ctrl(v)$ can be determined by analyzing the single block in which v is used as left hand-side variable. Furthermore, continuous assignments cannot be guarded by conditional variables. Hence, IFT operators suffice to track information flow through continuous assignments.

Example 4.9. Examples of tracking explicit flows for combinational and sequential code are shown in lines 2 and 8 of Fig. 3. Explicit flows capture how information moves through logical operations and assignments from right to left. Line 9 shows an example of tracking implicit flows. Here, execution of line 7 depends on control variables `rd_req`, `stall`, and `stall_cycles`. Furthermore, value of `index` specifies which memory element is accessed. Hence, these variables implicitly affect `cache[index]` and their labels are propagated to `cache_s[index]`.

4.3 Tracking Timing Flows

VeriSketch provides the ability to track both timing flows and logical flows. This allows the designer to define properties related to timing invariance alongside those related to logical flows. Timing flows are a *subset* of logical flows [34] and can be modeled by capturing how registers can be updated at each clock cycle [3]. We describe this in more detail in the following.

Definition 4.10 ($Bal(v)$)**.** Let v be the l-value variable in the procedural assignment η. Boolean variable $Bal(v)$ declares if updates to v are balanced. An *unbalanced update* means that there exists a clock cycle where register v can either maintain its current value or get reassigned. $Bal(v)$ is statically decided by analyzing the program control flow graph.

Remark 4.11. $Bal(v)$ can be determined since Verilog grammar confines all assignments to each variable v to a single block. Hence, one can compute if v keeps its value under certain branches of that block.

Using $Bal(v)$ VeriSketch detects timing variation occurring at assignments to variable v and tracks them via v_t.

Proposition 4.12 (c.f. [3])**.** Sensitive timing variations in a sequential circuit are generated at the l-value variable v of a clocked statement if the following equation evaluates to true:

$$\neg Bal(v) \sqcap \bigsqcup \{c_{i_s} : c_i \in Ctrl(v)\} \tag{2}$$

Any register v in a given hardware design is written to at each clock edge by a set of data signals which are multiplexed using a set of control signals. The existence of a feedback loop which connects the register to itself ($\neg Bal(v)$) indicates that there are some cases when the register maintains its value. Consequently, the final value of the register may become available at different cycles resulting in a timing leak. Hence, the conjunction of unbalanced updates and control signals which carry sensitive information results in sensitive timing variation at the register. To make the analysis more precise, a new conjunction is added to check if there is any untainted ($\neg c_{i_s}$) control variable which fully controls updating the register ($Full(v, c_i)$). This enables safe downgrading of timing variations:

$$\neg Bal(v) \sqcap \bigsqcup c_{i_s} \sqcap \neg \bigsqcup (\neg c_{i_s} \sqcap Full(v, c_i)) : c_i \in Ctrl(v) \tag{3}$$

Remark 4.13. Proposition 4.12 relies on the fact that in a hardware design registers updated at clock edges and combinational logic do not introduce cycle-level timing variation. Hence, the analysis is specific to HDLs and cannot be applied to software languages.

Example 4.14. Examples of tracking timing flows in combinational and sequential blocks are shown in lines 3 and 10 − 16 of Fig. 3. Lines 12 − 16 show the logic for detecting occurrence of timing flows while lines 3 and 10 − 11 show the logic for propagating them.

Example 4.15 (Secure Cache Property Specification)**.** The root cause of timing side channel leakage is that the victim's action changes the state of the hardware in a way that affects the time it takes for the succeeding operations to complete. If the victim action depends on secret data, the subsequent timing variation reveals information about the secret data. In the cache example, the index that the victim uses to read from the cache changes the state of the cache memory by bringing in new data and evicting the adversary's data to the next level memory. If the index contains secret information (as in table-based RSA implementation), the increment in the time taken for adversary's subsequent request

discloses information about the index used by the victim process. Absence of timing information leaked from process i's access to a cache can be modeled by the following property:

```
if(pid=i) assume(index_s==High);
else assert(rd_proc_t==Low);
```

This property states that assuming that process i accesses the cache with an `index` which contains sensitive information (shown by having *High* `index_s` label), the data read afterwards by other processes should not have sensitive timing information. This is shown by having an assertion on `rd_proc_t`, which is the timing label of the data read by the processor from the cache. This property along with the instrumented cache design is given to a formal verification tool to determine if a cache implementation is vulnerable against access pattern based cache timing attacks. Writing the IFT properties is identical to formalizing the security expectations and does not require knowledge of how an attack is performed since the verification tool searches for the exact input sequence which leaks the secret data.

Notation 4.16. Throughout the examples, all input labels have a *Low* value if not specified otherwise.

Example 4.17 (PLCache Property Specification)**.** To take into account the assumption that sensitive data should be preloaded and locked in the partition locked cache before access, we rewrite the properties as follows:

```
if(pid=i&&Preloaded[addr]) assume(index_s==High);
if(pid≠i) assert(rd_proc_t==Low);
```

4.4 Enforcing Multiple Policies

In order to instrument the circuit with the appropriate IFT instrumentation, we need to know how many disjoint flow properties we will be checking. It may be the case that different security properties require unique and independent tracking logic, each with different input labels. To accommodate simultaneous analysis of these properties, VeriSketch instruments the circuit with disjoint sets of labels and tracking logic based on the number of specified flow properties.

Example 4.18. In order to specify absence of timing leakage between multiple processes sharing a cache, we need disjoint labels to track flow of information from different processes:

```
if(pid=i) assume(index_s_i==High);
else assert(rd_proc_t_i==Low);
if(pid=j) assume(index_s_j==High);
else assert(rd_proc_t_j==Low);
```

Definition 4.19 (IFT Instrumentation)**.** For any design $F(x)$, its instrumented representation, denoted by $F_{\text{IFT}}\ (x,\ x_{\text{taint}})$, has the original functionality of $F(x)$, as well as multiple lines of flow tracking logic. Here, v_{taint} defined for each variable v is a vector of tuples of labels $(v_{s_i},\ v_{t_i})$.

5 SYNTHESIS

Reasoning about digital circuits can be encoded as SAT or bit-vector SMT problems, making them perfect targets for constraint-based synthesis. At a high-level, the standard synthesis problem is of the form $\exists p \forall x_v.\phi\ (p,\ x_v)$, where ϕ encodes the sketches and specifications, and the goal is to find parameters p such that the hard constraints in ϕ are satisfied for all possible inputs x_v. We now show how to extend this formulation to handle IFT instrumentation and solve for finite sequential circuits with soft constraints.

5.1 Synthesis with IFT

In order to take advantage of the IFT model within our synthesis flow, we give the parametric design $F(x_v,\ p)$ to the IFT unit. This transforms the design to $F_{\text{IFT}}\ (x_v,\ x_{v_{\text{taint}}},\ p)$ where $x_{v_{\text{taint}}}$ and F_{IFT} are the input's security labels and instrumented design (Definition 4.19). The synthesis problem over the instrumented design now includes the labels in addition to the original inputs:

$$\exists p\ \forall x.\Phi(x,\ p) \quad \text{where } \Phi(x,\ p) := Q\ (x_v,\ F_{\text{IFT}}\ (x_v,\ x_{v_{\text{taint}}},\ p)). \tag{4}$$

Here Q encodes the specifications written over the instrumented design. We use vector x to refer to the concatenation of the design inputs x_v and their taints $x_{v_{\text{taint}}}$. Note that $x_{v_{\text{taint}}}$ is constrained by the specific security properties we want to enforce.

Example 5.1. For instance, in our cache example, the cache index is initialized with a *High* label if it contains a sensitive address. And all other input labels have a *Low* label (notation 4.16). Thus, all input taints are constrained.

5.2 CEGIS for Finite Sequential Circuits

To handle sequential circuits, the CEGIS procedures need to expand over multiple cycles. To accommodate that, we extended the definition of a counterexample to capture a trace instead of a single value. Essentially, the counterexample represents a sequence of input values which take the design into an invalid state. Hence, in the synthesis stage the solver should look for a parameter such that the properties are satisfied for all the cycles triggered by the counterexample sequence. We model this by changing the original synthesis equation to:

Definition 5.2 (Synthesis Target for Sequential Circuits)**.**

$$\exists p.\ \bigwedge_{x_j \in \text{CE}} \Phi^*(x_j, p) \quad \text{where } \Phi^*(x_j,\ p)\ := \bigwedge_{k \le |x_j|} Past(\Phi(x_j, p),\ k) \tag{5}$$

Here $|x_j|$ is the length of counterexample x_j in number of cycles. $\Phi^*(x_j,\ p)$ is the conjunction of the properties over the length of each counterexample. Function $Past(v, k)$, part of System Verilog Assertion language, returns value of variable v from k previous cycles.

For the secure hardware design problems that we consider here, the bounds on the sequential depth are clear so that we can focus on tackling the synthesis aspect of the problems. With bounded depth, the verification component can be conveniently performed by standard bounded model checking (BMC). For unbounded verification, various techniques such as k-induction [15] can be used; and the framework can be naturally extended with more powerful verification methods.

5.3 CEGIS for Soft Constraints

CEGIS could potentially suggest any program which does not falsify the formal properties. Thus, the properties should effectively eliminate all undesirable programs. This makes property specification a major challenge. For example, consider the cache example where IFT properties similar to Example 4.17 are in place to eliminate side-channel leakage and synthesize the sketch from Fig. 2(a). A trivial implementation that results from this sketch and satisfies the IFT properties is a design that skips all cache accesses. While this satisfies the security properties, it is not what the designer intended to get. However, it is not clear how to formalize the property being violated in this case. The designer can potentially get around this issue by providing input/output (I/O) pairs which should be generated by the synthesized design, extending the formal properties, or shrinking the sketch such that undesirable programs are unreachable. However, all these approaches require non-trivial effort from the designer. Instead, we take an automated approach to heuristically guide the search algorithm to avoid recommending undesirable designs. We introduce *soft constraints* for specifying properties which may not hold for all cases but it is desirable if they do. Soft constraints are particularly beneficial for modeling performance attributes.

Definition 5.3 (Soft Constraint)**.** Soft constraints are logical formulas that model properties which are *preferably* true. We show soft constraints for the design being synthesized by $T(x,\ p)$.

Example 5.4. A soft constraint for synthesizing the secure cache can be defined by indicating that that having a low value for the `skip` signal and a high value for the `lru_update` signal (from Fig. 2) are desirable. While this constraint cannot be strictly enforced if one wants to eliminate timing side channel, we use it to guide CEGIS to find a design which does not skip cache writes and updates the LRU if possible. Using the *try* keyword to model the soft constraints, we rewrite the properties for synthesizing a secure cache as follows:

```
if(pid=i&&Preloaded[addr]) assume(index_s==High);
if(pid≠i) assert(rd_proc_t==Low);
try(!skip && lru_update);
```

In order to enforce soft constraints via synthesis, we extend the CEGIS algorithm to further explore the input space by searching for *positive examples*.

Definition 5.5 (Positive Example)**.** Positive example *pe* for the design synthesized with $p = p_i$ is any input trace which satisfies the specification $\Phi^*(x,\ p_i)$.

Positive examples represent cases where the design is working correctly according to the hard constraints. Positive examples are gathered after each verification round by searching the input space surrounding the newly found counterexample.

Definition 5.6 (Exploration)**.** The exploration round computes the set of positive examples *PE* by searching the design space surrounding each counterexample x. Exploration can be modeled by the following SAT problem for $x^m \in x$:

$$\exists a.\ \Phi^*(a,\ p_i) \wedge \bigwedge_{x^j \in x \wedge j \neq m} (a^j = x^j) \tag{6}$$

While the original CEGIS algorithm tries to fix the design by enforcing hard constraints on the counterexamples, we direct it to further enforce soft constraints on the collected positive examples. This is done by modifying the synthesis round to find a design such that soft constraints are held for the *maximum possible* number of collected positive examples while hard constraints are held for *all* visited counterexamples. This new synthesis problem is defined by:

Definition 5.7 (Synthesis Target for Soft Constraints T)**.**

$$\exists p.\ \bigwedge_{x_j \in CE} \Phi^*(x_j,\ p) \wedge \sum_{x_i \in PE} T^*(x_i,\ p) = n$$
$$\text{where } T^*(x_i,\ p) := \bigwedge_{k \leq |x_i|} Past(T(x_i,\ p),\ k) \tag{7}$$

The synthesis round iteratively solves Eq.7 and decreases n from $|PE|$ to zero if unsatisfiable.

Theorem 5.8. If satisfiable, CEGIS with soft constraints finds the program which enforces soft constraints on the maximum number of collected positive examples.

Proof Outline. Each synthesis round solves Eq.7 by setting $n := |PE|$ initially and decrease n if unsatisfiable. Hence, if satisfiable, parameter p represents the design where soft constraints are held for maximum $n \leq |PE|$. □

ALGORITHM 1: Given sketch $F(x)$, hard constraints $C(x)$, and soft constraints $C'(x)$, VeriSketch generates $F_{\text{syn}}(x)$.

Input : $F(x), C\ (x),\ C'\ (x)$: VeriSketch
Output: $F_{syn}(x)$ s.t. $\forall x.\ C(x)$: Verilog

$F\ (x,p) \leftarrow$ pre-processing $(F(x))$
$F_{\text{IFT}}\ (x,\ x_{\text{taint}},\ p) \leftarrow$ instrumentation $(F(x,p), C(x))$
$P \leftarrow$ CEGIS $(F_{\text{IFT}}\ (x,\ x_{\text{taint}},\ p),\ C(x),\ C'(x))$
if $P \neq$ *unsat* **then**
    $\text{F}_{\text{syn}}(x) \leftarrow$ post-processing $(F(x,p), P)$
    return F_{syn}
else return unsat

Soft constraints are ignored in the *verification* round since they do not necessarily hold for all input traces. This means that the equisatisfiability of the synthesis problem does not change as *soft constraints* are added. Hence, one can add soft constraints without worrying about making the problem unsatisfiable.

Theorem 5.9. Soft constraints do not impact satisfiability of the synthesis problem.

Proof Outline. The synthesis parameter, the verification equation, and hence the domain of valid programs remain the same by adding soft constraints. Furthermore, the synthesis equation in each round reduces to the original synthesis equation (i.e., Eq.5) in the worst case. Thus, the satisfiability does not change. □

Synthesis by soft constraints combines techniques from property-based and example-based synthesis by automatically searching for examples which should be generated by the synthesized design.

ALGORITHM 2: Counterexample guided inductive synthesis (CEGIS) for synthesizing sequential circuits with soft constraints

Input : $\Phi\ (x,\ p),\ T\ (x,\ p)$
Output: p_i s.t. $\forall x.\ \Phi(x, p_i)$

Initial Stage:
$p_i \leftarrow$ random assignment
$CE \leftarrow \emptyset$
$PE \leftarrow \emptyset$
while *1* **do**
  Verification Phase:
  $ce \leftarrow$ SAT $(\exists x.\ \neg\Phi(x, p_i))$
  if ce = unsat **then** return p_i
  else
    $CE \leftarrow CE \cup ce$
    Exploration Phase:
    $pe_m \leftarrow SAT\ (\exists a.\Phi^*(a,\ p_i) \wedge \bigwedge_{ce^j \in ce\ \wedge\ j \neq m} a^j = ce^j)$
    $\Phi^*(a,\ p) := \bigwedge_{k \leq |a|} Past(\Phi(a,\ p),\ k)$
    if $pe_m \neq unsat$ **then** $PE \leftarrow PE \cup pe_m$;
  Synthesis Phase:
  for $(l = |PE|; l \geq 0; i = i - 1)$ **do**
    $solution \leftarrow$ SAT $\exists p.\ \bigwedge_{x_j \in CE} \Phi^*(x_j, p) \wedge (sum_{pe} = l)$
    $sum_{pe} := \sum_{x_j^{'} \in PE} T^*(x_j^{'}, p)$
    $T^*(x_j^{'}, p) := \bigwedge_{k \leq |x_j^{'}|} Past(T(x_j^{'},\ p),\ k)$
    $\Phi^*(x_j,\ p) := \bigwedge_{k \leq |x_j|} Past(\Phi(x_j,\ p),\ k)$
    if solution≠unsat **then** $p_i \leftarrow solution$; *break*
    else if l=0 **then return** *unsat*
  end
end

Alternatively, one can manually specify the positive examples; however, defining traces of examples for sequential circuits may be challenging itself. The overall VeriSketch flow and CEGIS algorithm for synthesizing sequential circuits with soft constraints are shown in Algorithm 1 and 2, respectively.

6 EXPERIMENTS

We now demonstrate four examples of security-critical hardware designs that are successfully synthesized by VeriSketch.

- **Constant Time Arithmetic Units** We implement fixed point arithmetic units which run in constant time. We use IFT specification to model constant time behaviour and non-synthesizable[1] portion of the Verilog language to model functional properties.
- **Leakage-free caches:** We add sketch constructs (following the partition lock methodology [48]) to traditional cache architectures and synthesize two caches (direct mapped and 4-way set associative) which are resilient against timing side channel attacks. We model resilience against timing side channel attacks as IFT properties and add soft constraints to model performance traits.
- **Hardware thread schedulers:** We synthesize schedulers for fine-grained multithreading in mixed criticality systems by defining properties regarding confidentiality between threads, guaranteed scheduling frequency, and timing predictability. We define three sketches of different size and synthesize each to satisfy different combinations of the properties along with soft constraints modeling efficiency and fairness.
- **System-on-chip (SoC) arbiters:** We synthesize arbiters to mediate access in bus architectures by enforcing (one or multiple of) non-interference, access control, priority, and fairness between the cores.

[1]Synthesizable in this case refers to the portion of the language that can be mapped to a gate-level netlist. Complex Verilog operators can only be used in simulation.

Table 3: Summary of synthesized designs in terms of lines code for the sketch, synthesized code and specifications.

Design	Sketch LoC VeriSketch	Spec. LoC VeriSketch	Syn. LoC Verilog/AST
Fixed Point Arithmetic	59	33	107/961
Direct Mapped Cache	243	73	379/3809
4-way Set Associative Cache	303	73	512/6098
Hardware Thread Scheduler	73	92	365/2308
SoC Arbiter	57	80	487/4262

Table 3 shows the code size for the biggest synthesized design in each experiment set. These numbers are reported in terms of lines of code written in VeriSketch language for the sketch and specification (i.e., the formal testbench) and in Verilog and AST for the synthesized code. We will first explain the implementation details of the framework and then discuss the synthesized designs.

6.1 Implementation

As shown in Fig. 1 and Algorithm 1 VeriSketch flow consists of an IFT engine and a program synthesis unit. The IFT tool uses the Yosys [49] front-end parser to get the AST representation of the Verilog design. It then analyzes the design's data and control graph along with the security properties to generate the corresponding information flow tracking logic. It writes back the instrumented design in Verilog. The instrumented Verilog design is then given to the synthesis unit to search for the ideal parameter. The program synthesis unit makes calls to a SAT/SMT solver for *verification*, *exploration*, and *synthesis*. This unit can either use a commercial EDA tool (Questa Formal Tool from Mentor Graphics) or open source solvers (Any of Yices2 [14], Boolector [8], Z3 [12], or CVC4 [5]) by using Yosys to translate Verilog to SMT-LIB2 representation.

6.2 Constant Time Arithmetic Units

The Verilog language supports multiplication and division operators; however, these operators cannot be directly mapped to hardware by EDA tools due to their complexity. For instance the statement "`assign c=a\b;`" requires the EDA tool to build a divider

```
module div (clk, start, dividend, divisor, quotient, done, overflow);
 assign flag = reg_a (>=,>,<,=>) reg_b;
 always @(posedge clk) begin
  if( done && start)
    //initialize …
  reg_q[reg_count] ?= ??;
  reg_b ?= reg_b(>>, <<, <<<, >>>) ??;
  reg_a ?= reg_a - reg_b;
  quotient ?= reg_q;
  ctrl_vars = [start, done, count_done, flag];
  //counter, overflow and sign logic …

assert (dividend, divisor -/->t quotient);
assert (done && divisor!=0 |->(|quotient-((dividend << Q)/divisor)|<=1));

endmodule
```

(a)

```
module div (clk, start, dividend, divisor, quotient, done, overflow);
assign flag = reg_a >= reg_b;
always @(posedge clk) begin
  if(done && start)
    //initialize …
 if(!reg_done && !count_done && (reg_a >= reg_b))
    reg_q [reg_count] <= 1;
  if(!reg_done)
    reg_b <= reg_b << 1;
  if(!reg_done && (reg_a >= reg_b))
    reg_a <= reg_a - reg_b;
  if (!start & !done & count_done)
    quotient <= reg_q;
  //counter, overflow and sign logic …
endmodule
```

(b)

Figure 4: Synthesizing a constant time fixed point divider using VeriSketch. (a) Sketch of a shift-and-subtract divider where the structure of the procedural statements, operations, and constant values are left unspecified as shown by the highlighted code. Constant time and functional properties are modeled by IFT and built-in Verilog operators, respectively. (b) The divider unit generated by VeriSketch. The highlighted parts show the code that is generated automatically.

which runs in a single cycle. As this is not feasible in most cases, the complex operations can only be used in simulation and the hardware designers need to implement arithmetic units using low level operators. These arithmetic units run in multiple cycles and could have early termination based on the operands' values which leaks information about the values. We use VeriSketch to design fixed point multiplier and divider units which run in constant time independent of their operands' values.

Sketches and Properties. We sketch a shift-and-add multiplication unit and a shift-and-subtract division unit for fixed point computation as described in [1]. The sketch of the divider unit along with the functional and security properties are shown in Fig. 4(a). We leave the structure of the procedural statements undefined using "?=" construct and ask the synthesizer to find the correct control logic and cycle-level register updates using the list of the control variables in the design (`start`, `done`, `count_done` and `flag`). Here, `start` and `done` indicate the beginning and end of the computation while `count_done` shows that the counter has reached its maximum value. Variable `flag` is defined in the sketch. For simplicity, control signals `ctrl_vars` are globally defined for all assignments. We also use low-level sketch constructs to leave operations and constant values undefined. The first assertion in Fig. 4(a) describes constant time requirements using IFT operators. The second property states that the `quotient` computed by the sequential circuit should differ from the value computed by the built-in operations by at most one bit. This error value is equivalent to 2^{-Q} where Q is the number of bits used to represent the fractional segment. The dividend is shifted by `Q` bits to follow the fixed point representation.

Synthesized Designs. The divider unit synthesized by VeriSketch is shown in Fig. 4(b). VeriSketch finds the appropriate control signal to guard execution of the procedural statements as shown by the `if` statements. The last statement ensures that the final output `quotient` is updated at a constant time, even though the intermediate variable `reg_q` may contain the final result sooner. This example shows how the IFT unit safely downgrades timing variations (Eq. 3) from `reg_q` to `quotient` since `count_done` fully controls timing of the updates to the `quotient`. We skip reporting the details of the synthesized multiplier as it is similar to the divider.

```
module Sketch_Cache(…);
 assign skip =
  (!rd & wr & !hit & lru_block[m])| (rd & !wr & !hit & lru_block[m])
  |(!rd & !wr & hit & lru_block[m])|(!rd & !wr & !hit & lru_block[m])|
  (!rd & !wr & hit & !lru_block[m]);
 assign lru_update =
  (rd & !wr & waiting & stall & !lock)|(rd & !wr & waiting & !stall & !lock)|
  (rd & !wr & !waiting & stall  & !lock)|(rd & !wr & !waiting & !stall & !lock)|
  (!rd & wr & waiting  & !stall & !lock)|(!rd & wr & !waiting & stall  & !lock)|
  (rd & wr  & waiting  & !stall & lock) |(rd & wr  & waiting  & !stall & !lock)|
  (rd & wr  & !waiting & !stall & lock );
always @ (posedge clk)
  if(!skip)
   //cache rd/wr
   if(lru_update)
    //update LRU
  else
   //direct memory access
endmodule
```

Figure 5: VeriSketch synthesizes the sketch from Fig. 2(a) to a fully specified Verilog design that meets the functional and security properties specified in Example 5.4.

6.3 Leakage-Free Cache

We use VeriSketch to modify an existing (non-secure) cache implementation such that it defends against timing attacks. We define sketch and properties for this set of experiments as shown in Fig. 2(a) and Example 5.4 for both a direct mapped and a 4-way set associative cache (with the difference that the direct mapped cache does not require LRU logic). Fig. 5 shows the output of VeriSketch Synthesizing a fully specified and functional Verilog design. We only show the parts of the code that is automatically generated. The synthesized skip logic indicates that when a read or write request result in a cache miss, it should skip the cache and go through direct memory access if the block to be evicted is locked. The cache design created by VeriSketch does not update the LRU state when a locked cache block is accessed, and hence eliminates the timing leakage in the original PLCache. Note that as the *comb* syntax is mapped to a BDD, it generates logic for certain input combinations that do not occur in execution (e.g., having both a read and write request). Using Yices2 [14] as the SMT solver, the synthesis process takes around six and eight hours for the direct mapped and set associative caches, respectively. The synthesis time in this set of experiments are considerably longer compared to the ones reported in the rest of the examples and are dominated by the time taken to perform bounded model checking in the verification rounds. This

is due to the fact that formally verifying and reasoning about memory elements take large amount of time. This can be alleviated by abstracting the unrelated data path or giving hints to the solver on what the relevant variables are. We leave this problem for future work.

6.3.1 Security Analysis of Sketch Cache vs. PLCache. The PLCache is resilient against the original Percival attack as the victim's access to its preloaded data results in a cache hit and does not evict the attacker's data. However, accessing preloaded data changes the LRU bits of that cache set. More specifically, accessing the preloaded data marks the locked block as the most recently used block in the set; and it prioritizes other blocks in the set for eviction. Consequently, *even though accessing locked data does not evict the attacker's data directly, it prioritizes eviction of the attacker's data.* In order to exploit this subtle change in the state of the cache, we extend the Percival attack such that the adversary can observe the effect of the change in the LRU bits. This is done by adding an extra stage to the attack where the attacker tries to evict its own data. If the attacker is able to evict its data (i.e., the attacker observes an increased access time in the next access), it indicates that the attacker's data has been prioritized for eviction as a result of the victim's action. The Percival attack is extended as suggested by the counterexample trace collected while verifying the PLCache.

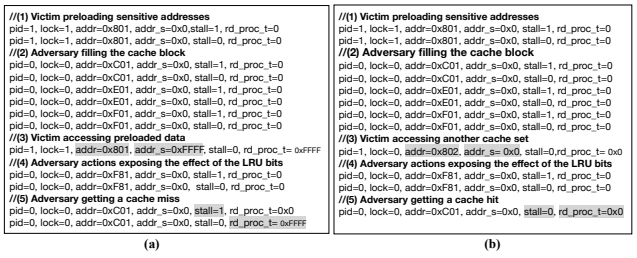


Figure 6: Timing leakage in PLCache. (a) Victim process (`pid=1`) accesses its locked data in stage 3. This results in a cache miss for the attacker in stage 5 (shown by `stall=1`). The verification tool captures this since `rd_proc_t` has a *High* value in stage 5. (b) Victim does not access its locked data in stage 3 and the attacker observes a cache hit in stage 5.

Fig. 6 shows the results of simulating the PLCache with simulation traces that resemble the extended Percival attack. In both Fig. 6(a) and (b) the victim process first preloads and locks its data (stage 1). Next, the adversary fills the cache set, but fails to evict the locked block (stage 2). Fig. 6(a) represents the case where the victim accesses its locked data at stage 3 making the locked data the most recently used block and the attacker's data the least recently used block. Fig. 6(b) represent the case where the victim accesses some other cache set and leaves the LRU bits unmodified (i.e., the locked block remains the least recently used block). In stage 4, the adversary aims to observe the change in the LRU bits by trying to evict the its own data that was used to fill cache in stage 2. In case (a), adversary's access to the cache set evicts its own data since victim's action from stage 3 has prioritized eviction of the attacker's data. In case (b), the attacker's access to the cache is skipped because the least recently used block is locked and cannot be evicted. The adversary is able to observe the difference in victim's action from stage 3 at stage 5 through timing variation. In case (a) the adversary experiences a cache miss (i.e., increased cache access time) while in case (b) adversary's access results in a cache hit. This difference is shown in the value of the `stall` signal in simulation. The IFT instrumentation shows a high value for `rd_proc_t` at stage 5 of Fig. 6(a) which is a violation of the security property specified in Example 5.4. VeriSketch synthesizer mitigates this vulnerability by generating the `lru_update` logic such that accessing locked blocks does not change the LRU bits (and any other hardware state). Results of simulating the synthesized cache by the same traces are available in Appendix B.

6.3.2 Soft Constraint Analysis. As described in Section 5.3, performance related soft constraints are essential for synthesizing a practical cache. In order to analyze the effect, we simulate the caches which are synthesized with and without soft constraints using memory traces from the CloudSuite benchmarks [19]. Fig. 7 shows cache misses for simulating 4-way set associative caches of size 32KB with one million memory traces for each application. All numbers are normalized to the number of misses for a non-secure cache of the same size. As shown by the graph, the cache which is synthesized with soft constraints has a considerable lower miss rate.

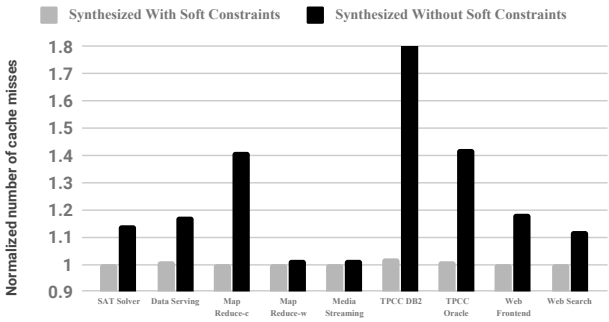


Figure 7: Number of cache misses for caches synthesized with and without soft constraints simulated with memory traces from CloudSuite benchmarks [19]. The numbers are normalized to the number of cache misses from a non-secure cache.

6.4 Hardware Thread Scheduler

Here we describe design of a hardware thread scheduler module for fine-grained multithreading in mixed criticality systems [9]. The design problem is borrowed from the FlexPRET project [55] which implements a processor dedicated to real time needs. We have expanded the scheduler design by introducing confidentiality requirements and automatically generating different modifications of it. The scheduler decides which hardware thread should execute at each clock cycle based on inputs from the operating system. These inputs consists of two vectors `freq` and `mode`. `freq` specifies the expected execution frequency for the threads, and `mode` describes

Table 4: Summary of synthesized thread schedulers.

Sketch Size	Prop.	Time(sec.)[Syn., Ver., Exp.]		
		-	E	F
72bits	V G C	9, 3, -	11, 5, 2	14, 4, 3
	V G P	7, 4, -	13, 4, 3	10, 3, 2
	V G C P	9, 3, -	12, 4, 2	15, 5, 3
192bits	V G C	50, 12, -	175, 17, 14	187, 19, 14
	V G P	114, 18, -	140, 19, 11	221, 22, 15
	V G C P	105, 20, -	205, 24, 15	209, 21, 15
232bits	V G C	185, 17, -	338, 27, 23	877, 29, 31
	V G P	357, 20, -	831, 32, 28	1592, 38, 38
	V G C P	412, 23, -	781, 32, 27	2900, 71, 44

different traits of each thread. These traits state if the thread has *hard real-time* or *soft real-time* requirements, whether or not it carries sensitive information, and if it is active or asleep at the given cycle.

Sketches and Properties. The scheduler sketch consists of two FSMs and one combinational function written with *seq* and *comb* syntax, respectively. The first FSM outputs a `thread_id` based on the given frequencies `freq`. The second FSM generates a new `thread_id` according to the result of the first FSM and the `mode` signal. The combinational function selects between the outputs of these two FSMs. This implements two interleaving schedulers where details of the scheduling schemes are unspecified. We have modeled different properties regarding real-time requirements, fairness, confidentiality, and efficiency as hard and soft constraints. The real-time properties, borrowed from [55], include timing predictability for *hard real-time* threads, and guaranteed expected frequency for *soft real-time* threads. Timing predictability requires the scheduler to give the *hard real-time* the exact frequency that they asked for. Guaranteed frequency on the other hand, requires the scheduler to give the *soft real-time* threads at least what they asked for. This enables the scheduler to assign *soft real-time* threads to any empty slots (for instance caused by others being asleep). Hence, the *soft real-time* threads can have expected frequency of zero and still get to execute. Both of these properties are modeled as hard constraints. We also model fairness for the extra quota given to *soft real-time* threads as soft constraints. The confidentiality requirement states that activity status of sensitive threads should not be revealed. We model this as an IFT property by assigning *High* labels to active/asleep bit of sensitive threads, and asserting that the scheduler output should maintain a *Low* label. Enforcing this property changes how the scheduler assigns empty slots to available *soft real-time* threads. Lastly, an efficiency property – modeled as a soft constraint – synthesizes a scheduler which selects active threads for execution. If written as a hard constraint, the problem becomes unsatisfiable due to cases where no active thread is available for scheduling. This experiment illustrates how soft and hard constraints are used in property-based program synthesis frameworks. While security and safety requirements are modeled as hard constraints since they should be held unconditionally, soft constraints are helpful for modeling properties regarding system performance.

Table 5: Summary of synthesized SoC Arbiters.

Design	Properties	Time (sec.)
Arbiter w/ 4 cores and 1 shared unit 338 bits	WISHBONE [32]	248
	WISHBONE w/ priority for core 1	162
	Priority-based access	616
	WISHBONE w/ no access for core 1	171
	TDMA	128
	Non-interference b/w all cores	157
	Non-interference b/w cores 1&2	113
Arbiter w/ 4 cores and 3 shared unit 1014 bits	U_1: Non-interference U_2: Non-interference bw/ cores 1&2 U_3: WISHBONE	312
	U_1: Non-interference U_2: Non-interference bw/ cores 1&3 U_3: WISHBONE w/o access for cores 2&3	278
	U_1: WISHBONE w/o access for core 3 U_2: Priority-based access U_3: WISHBONE	719

Synthesized Designs. In order to show how the sketch size affects synthesis time, we generate the circuitry from three different templates. We gradually add the sketch constructs and decrease the manually specified details to observe the effect. Synthesis results are shown in Table 4 where the property abbreviations are as follows. V: Valid thread id, C: Confidentiality for sensitive threads, P: Predictability for hard real-time threads, G: Guaranteed frequency, E: Only Scheduling available threads, F: Fairness between soft real time threads. The formal representation of these properties is available in Table 6 in Appendix B. As shown in Table 4, the synthesis time increases proportionally to the sketch size mostly due to the increase in the time spent on synthesis. In the first set of experiments we only leave the combinational select logic unspecified, and implement everything else manually. For the other two rounds, we replace the FSMs with sketches as well. For each set, we synthesize the sketch using various combinations of the discussed properties. The synthesis time increases as soft constraints are added. This increase is mainly caused by multiple *synthesis* stages which fail and are replayed by relaxing the problem. Collecting positive examples does not contribute much to the overall time. Yices2 [14] is used as the SMT solver for generating all the designs in these experiments.

6.5 SoC Arbiter

System-on-chip arbiters which mediate accesses in bus architectures have been shown to be vulnerable against timing side channel attacks [33, 35]. The vulnerability arises as different cores which are requesting access to a shared unit can infer about each others access pattern based on the time they are granted access themselves. We model timing side channel elimination as IFT properties to enforce non-interference between mutually untrusted cores. We further specify various functional properties and synthesize multiple SoC arbiters from generic FSM sketches.

Sketches and Properties. To synthesize the arbiter module, we have sketched three FSMs where state transitions are left unspecified. The one-hot encoded `req` and `grant` signals indicate the incoming requests and the given grant at each clock cycle. The first

two FSMs are defined using *seq* syntax with different sets of inputs. The first one takes `req` and `grant` as inputs, and the second one models a smaller FSM where state transitions are independent of the incoming requests. While the second FSM models designs that can be generated by the first one, it can more quickly synthesize arbiters where the scheduling is independent of the input (e.g., TDMA policy). The third sketch models an FSM which groups different cores in disjoint sets. Finally, we sketch a combinational logic which selects one of the FSMs. We define two sets of sketches modeling an arbiter module which mediates between four cores sending requests to one and three shared units. We define properties regarding access control, non-interference, and priority-based scheduling to synthesize different arbiters. The formal representation of these properties is available in Table 7 in Appendix B.

Synthesized Designs. Table 5 shows the result of synthesizing different arbiters by combining different sets of properties. Note that while the sketch includes multiple FSMs, only one of them is chosen and synthesized by CEGIS. Using this strategy, the sketch can be automatically selected from a pool of available sketches eliminating the need to explicitly determine a single template for synthesis. The first four designs from Table 5 are synthesized by the first most generic template. The next two designs are generated from our second template. Lastly, adding non-interference properties between two cores results in using the third template where different cores are appropriately placed in separate groups. As we can see from the results, adding IFT properties speeds up the synthesis procedure because these properties constrain the high-level structure of the design. In the next round of experiments, we replicated the templates to synthesize an arbiter which mediates accesses to three shared units with distinct policies. U_i in the table refers to shared uint number i. The last column of Table 5 shows the time taken for synthesis using Questa Formal Tool.

7 CONCLUSION

This work presents a semi-automated and security-oriented methodology for designing hardware with formal proof of security. The proposed design framework consists of language support for sketching digital circuitry, and a set of techniques for translating partially written HDL codes into complete designs that provably comply with the designers' functional and security specifications. The proposed flow speeds up and simplifies the lengthy process of hardware design and verification, and acquaints the traditional design flow with automated enforcement of security properties. We have shown how combining program synthesis techniques with the model of information flow enables generating hardware units which are correct and secure by construction.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant no. CNS-1527631 and CNS-1563767.

REFERENCES

[1] [n. d.]. *The reference community for Free and Open Source gateware IP cores.* https://opencores.org/project,verilog_fixed_point_math_library.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013.* IEEE, 1–8.

[3] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. [n. d.]. Clepsydra: Modeling Timing Flows in Hardware Designs. In *International Conference on Computer-Aided Design (ICCAD), 2017.*

[4] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. [n. d.]. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. In *Proceedings of the 2017 Conference on Design, Automation & Test in Europe.*

[5] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification.* Springer, 171–177.

[6] Andrew Becker, David Novo, and Paolo Ienne. [n. d.]. Automated circuit elaboration from incomplete architectural descriptions. In *Signals, Systems and Computers, 2013 Asilomar Conference on.*

[7] Andrew Becker, David Novo, and Paolo Ienne. 2014. SKETCHILOG: Sketching combinational circuits. In *Proceedings of the conference on Design, Automation & Test in Europe.*

[8] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 174–177.

[9] Alan Burns and Robert Davis. 2013. Mixed criticality systems-a review. (2013), 1–69.

[10] Kai-Hui Chang, Igor L Markov, and Valeria Bertacco. [n. d.]. Fixing design errors with counterexamples and resynthesis. In *Design Automation Conference, 2007. Asia and South Pacific.*

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[13] Jordan Dimitrov. 2001. Operational semantics for Verilog. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific.* IEEE, 161–168.

[14] Bruno Dutertre. 2014. Yices 2.2. In *International Conference on Computer Aided Verification.* Springer, 737–744.

[15] Niklas Eén and Niklas Sörensson. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science* 89, 4 (2003), 543–560.

[16] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 693–707.

[17] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 420–435.

[18] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[19] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012). http://infoscience.epfl.ch/record/173764

[20] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. 2012. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification.* Springer, 548–563.

[21] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. 2006. *Retrofitting legacy code for authorization policy enforcement.* IEEE.

[22] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

[23] William R Harris, Somesh Jha, and Thomas Reps. 2010. DIFC programs by automatic instrumentation. In *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, 284–296.

[24] William R Harris, Somesh Jha, Thomas W Reps, and Sanjit A Seshia. 2017. Program synthesis for interactive-security systems. *Formal Methods in System Design* 51, 2 (2017), 362–394.

[25] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified synthesis: automatically learning the x86-64 instruction set. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 237–250.

[26] Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. 2016. Imprecise security: quality and complexity tradeoffs for hardware information flow tracking. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on.* IEEE, 1–8.

[27] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, 215–224.

[28] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).
[29] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 97–112.
[30] 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 *ACM SIGPLAN Notices*, Vol. 46. ACM, 109–120.
[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).
[32] Milica Mitić and Mile Stojčev. 2006. A survey of three system-on-chip buses: AMBA, coreconnect and wishbone. In *Proc. 41st Int. Conf. Inform. Commun. Energy Syst. Technol.(ICEST)*. Citeseer, 282–285.
[33] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2011. Information flow isolation in I2C and USB. In *Proceedings of the 48th Design Automation Conference*. ACM, 254–259.
[34] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner. 2014. Leveraging gate-level properties to identify hardware timing channels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 33, 9 (2014), 1288–1301.
[35] Jason Oberg, Timothy Sherwood, and Ryan Kastner. 2013. Eliminating timing information flows in a mix-trusted system-on-chip. *IEEE Design & Test* 30, 2 (2013), 55–62.
[36] Colin Percival. 2005. Cache missing for fun and profit. (2005).
[37] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.
[38] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Type-Driven Repair for Information Flow Security. *CoRR* abs/1607.03445 (2016). arXiv:1607.03445 http://arxiv.org/abs/1607.03445
[39] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. [n. d.]. Automatic device driver synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009.*
[40] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis.. In *OSDI*. 661–676.
[41] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *ACM SIGPLAN Notices* 51, 6 (2016), 326–340.
[42] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495.
[43] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 404–415.
[44] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2013. Fix Me Up: Repairing Access-Control Bugs in Web Applications.. In *NDSS*.
[45] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2017. Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).
[46] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 109–120. https://doi.org/10.1145/1508244.1508258
[47] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 62.
[48] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*.
[49] Clifford Wolf. [n. d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/. ([n. d.]).
[50] Bo-Han Wu, Chun-Ju Yang, Chung-Yang Ric Huang, and Jie-Hong Roland Jiang. [n. d.]. A robust functional ECO engine by SAT proof minimization and interpolation techniques. In *Proceedings of the International Conference on Computer-Aided Design, 2010.*
[51] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 508–521.
[52] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 63.
[53] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.
[54] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing sql queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 224–234.
[55] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 101–110.

APPENDIX

A SECURITY ANALYSIS OF SKETCH CACHE VS. PLCACHE

Results of simulating PLCache and the sketch cache with traces that represent the extended Percival attack on LRU bits are shown in Fig. 8. Fig. 8(a) and (b) show the results of simulating PLCache and are identical to Fig. 6(a) and (b). Fig. 8(c) and (d) show the results of simulating the sketch cache. Fig. 8(a) and (c) show the case where the victim process (pid=1) accesses its locked data at stage 3 (case A). Fig. 8(b) and (d) represent the case where the victim does not access its locked data and accesses a different cache set (case B). In this simple example we use the 4 lowest bits as index; hence, all addresses except for `0x802` map to the same cache set.

The extended Percival attack on the LRU bits of a partition locked cache comprises five stages. In the first stage the victim process preloads and locks its sensitive data. Next, the attacker fills the cache set but is not able to evict the cache block which contains the locked data. At this point, the cache set includes the victim's locked data as well as the attacker's data. Furthermore, the attacker knows that its data is the most recently used data as it was just accessed. Now, consider case A where the victim process accesses its locked data at stage 3. This access results in a cache hit. In PLCache (Fig. 8(a)) this access changes the LRU bits by making the locked data the most recently used block and consequently making the attacker's data the least recently used block. In the sketch cache (Fig. 8(c)) this access does not modify the LRU bits and the locked block remains the least recently used. In case B, the LRU bits remain unmodified in both PLCache and the sketch cache (Fig. 8(b) and (d)). In stage 4, the attacker aims to observe the potential changes in the LRU bits. This is done by the attacker trying to bring new data to the cache in order to force an eviction in the set. In PLCache, the attacker is able to evict its own data in case A since it had become the least recently used. However, it cannot evict its data in case B because the locked data is the least recently used block which cannot be evicted. In the synthesized cache, the attacker is not able to force an eviction in any of the cases. In stage 5, the attacker accesses the data which was brought to the cache in stage 2. In PLCache, this results in a cache miss in case A and a cache hit in case B. Hence, the attacker can observe the difference between the two cases through the timing variation. In our simulations, this timing variation manifests itself through the value of the `stall` signal and the timing label of the data which is read by the processor `rd_data_proc_t` (Fig. 8(a) vs. Fig. 8(b)). In the synthesized cache, the adversary observes a cache hit in both cases (Fig. 8(c) vs. Fig. 8(d)).

B PROPERTIES FOR SYNTHESIZING THREAD SCHEDULERS AND SOC ARBITERS

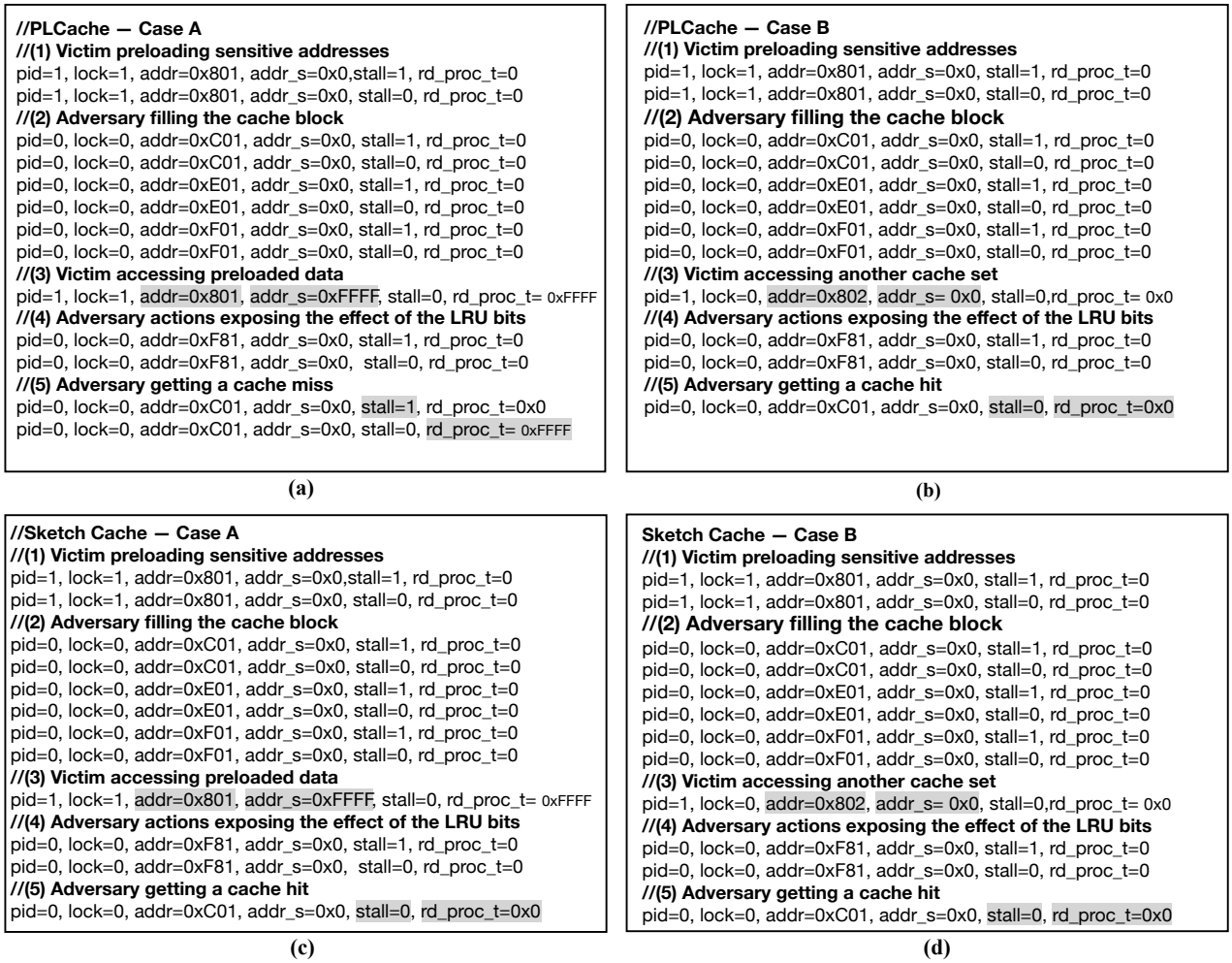


Figure 8: Simulating PLCache and the synthesized cache with traces representing the extended Pecival attack.

Table 6: Summary of properties used for synthesizing thread schedulers.

Synopsis	Formal Representation
(V) Valid thread id	$valid \mapsto assert(thread\_id < n)$
(C) Confidentiality of sensitive threads	$\forall i.(i.sensitive) \mapsto assume\ (i.active_s = High) assert\ (thread\_id_t = Low)$
(P) Predictability of hard real-time threads	$\forall i.\ (i.hardRT) \mapsto assert\ (i.freq + i.sleep = i.count)$
(G) Guaranteed frequencies	$\forall i.\ assert\ (i.freq + i.sleep \leq i.count)$
(E) Scheduling available threads	$\forall i.\ \neg i.active \mapsto try\ (thread\_id \neq i)$
(F) Fairness for soft real-time threads	$\forall i, j.(\neg i.hardRT\ \wedge\ \neg j.hardRT)) \mapsto try\ ((i.count - i.freq) = (j.count - j.freq))$

Table 7: Summary of properties used for synthesizing SoC arbiters.

Synopsis	Formal Representation
Grant given to at most n cores	$assert(Countones(grant) \leq n)$
Grant given to a core which requested	$assert(Past(req) \mapsto (\forall i.\ grant[i] \mapsto Past(req[i])))$
Stabilizing the grant while a core is using	$assert((grant[i] \wedge\ Past(req[i])) \mapsto Stable(grant))$
Equal share	$assert(\bigwedge_{i=0}^{n-1}(\bigwedge_{j=i+1}^{n-2}(Past(grant,\ period * i) = Past(grant,\ period * j))))$
Denying access to core #i	$assert(\neg grant[i])$
Prioritizing core #i	$assert(Past(req[i]) \mapsto grant[i])$
Priority-based access	$assert(\bigwedge_{j=0}^{i-1} \neg Past(req[j]) \wedge (Past(req[i]) \mapsto grant[i]))$
Non-interference between all cores	$assert(req_s \nrightarrow_t \neg grant_s)$
Non-interference between cores #i, j	assert$((req_s[i] \nrightarrow_t \neg grant_s[j]) \wedge (req_s[j] \nrightarrow_t \neg grant_s[i]))$