UNIVERSITY OF CALIFORNIA SAN DIEGO

Verification and Synthesis of Information Flow Secure Hardware Designs

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

in

Computer Science (Computer Engineering)

by

Armaiti Ardeshiricham

Committee in charge:

Professor Sicun Gao, Co-Chair
Professor Ryan Kastner, Co-Chair
Professor Farinaz Koushanfar
Professor Nadia Polikarpova
Professor Deian Stefan

2020

The dissertation of Armaiti Ardeshiricham is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Co-Chair

University of California San Diego

2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Chapter 2, in part is currently being prepared for submission for publication of the material. Wei Hu, Armaiti Ardeshiricham, Ryan Kastner. The dissertation author was the co-investigator and co-author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in the Proceedings of the 20th Annual Design, Automation and Test in Europe Conference Exhibition (DATE), March 2017. Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in the Proceedings of the 36th Annual International Conference on Computer-Aided Design (ICCAD), November 2017. Armaiti Ardeshiricham, Wei Hu, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in the Proceedings of the 26th Annual Conference on Computer and Communications (CCS), November 2019. Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 6 is being prepared for the publication of the material. Armaiti Ardeshiricham, Christie Lincoln, Alvin Zhang, Lisa Luo, Amir Uqdah, Sicun Gao, Ryan Kastner. The dissertation author was the primary investigator and author of this material.

VITA

2014	B. S. in Electrical Engineering, Sharif University of Technology, Tehran, Iran.
2017	M. S. in Computer Science (Computer Engineering), University of California San Diego.
2020	Ph. D. in Computer Science (Computer Engineering), University of California San Diego.

PUBLICATIONS

Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, Ryan Kastner, "VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties", *ACM Conference on Computer and Communications Security (CCS)* , 2019.

Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, Ryan Kastner, "Property Specific Information Flow Analysis for Hardware Security Verification", *International Conference on Computer-Aided Design (ICCAD)*, 2018.

Lu Zhang, Wei Hu, Armaiti Ardeshiricham, Yu Tai, Jeremy Blackstone, Dejun Mu, Ryan Kastner, "Examining the Consequences of High-Level Synthesis Optimizations on Power Side-Channel", *Design Automation and Test in Europe (DATE)*, 2018.

Armaiti Ardeshiricham, Wei Hu, Ryan Kastner, "Clepsydra: Modeling Timing Flows in Hardware Designs", *International Conference on Computer-Aided Design (ICCAD)*, 2017.

Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, Ryan Kastner, "Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design", *Design Automation and Test in Europe (DATE)*, 2017.

Wei Hu, Lu Zhang, Armaiti Ardeshiricham, Jeremy Blackstone, Bochuan Hou, Yu Tai, Ryan Kastner, "Why You Should Care About Don't Cares: Exploiting Internal Don't Care Conditions for Hardware Trojans", *International Conference on Computer-Aided Design (ICCAD)*, 2017.

Wei Hu, Armaiti Ardeshiricham, Ryan Kastner, "Identifying and measuring security critical path for uncovering circuit vulnerabilities", *Microprocessor Test and Verification Conference (MTV)*, 2017.

Wei Hu, Andrew Becker, Armaiti Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, Ryan Kastner, "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking", *International Conference on Computer-Aided Design (ICCAD)*, 2016.

Wei Hu, Alric Althoff, Armaiti Ardeshiricham, Ryan Kastner, "Towards Property Driven Hardware Security", *Microprocessor Test and Verification Conference (MTV)*, 2016.

ABSTRACT OF THE DISSERTATION

Verification and Synthesis of Information Flow Secure Hardware Designs

by

Armaiti Ardeshiricham

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2020

Professor Sicun Gao, Co-Chair
Professor Ryan Kastner, Co-Chair

The increasing number of hardware-based security attacks along with prevalence of embedded systems in critical applications necessitate robust methods for designing digital circuits with regards to security expectations. However, state of the art secure hardware design practices are mainly based on manual code review and security audit which do not provide formal guarantees of security despite being time-consuming and cumbersome. Thus, a systematic approach for identifying and fixing security vulnerabilities in hardware designs is required.

Information flow analysis presents a coherent and systematic method for evaluation of broad types of security policies. Information flow models capture how security-critical data

propagates through a system by augmenting the design with security labels. Therefore, evaluating the security labels of the critical components could substitute manual inspection of the design for information leakage. Such analysis accelerates, automates, and formalizes assessment of information flow security properties.

This dissertation shows how various security policies such as confidentiality, integrity and timing side-channel can be formalized utilizing information flow tracking (IFT) in hardware designs. Following this formalization, this thesis presents four practical tools – RTLIFT and Clepsydra for verification of timing-sensitive information flow properties, VeriSketch for synthesis and enforcement of timing-sensitive information flow properties, and lastly an error localization framework for automated inspection of verification failures. Leveraging advances in formal methods and commercial EDA software, the presented tools augment the traditional hardware design cycle with sound and automated security analysis.

Chapter 1

Introduction

Hardware-oriented security attacks have become more prevalent and powerful as novel attack surfaces have been discovered [KGG+18, LSG+18, CCX+18, ERAG+18, KW18, MR18, KKSAG18, SP18, LYG+15]. These attacks analyze and exploit low-level hardware characteristics such as optimization paths, memory layout, timing behavior, energy usage, etc., to infer secret information about the running process. As shown extensively, many of such attacks can be lunched over the network without physical adjacency and provide efficient means for stealing secret keys, passwords, or sensitive system information.

In order to lunch a hardware-based security attack, the adversary takes advantage of subtle but measurable hardware behavior which is correlated with the data that is being processed. These exploitable hardware attributes could be due to undiscovered design flaws or intentional features which were not scrutinized with regards to security expectations. While multiple factors contribute to prevalence of such vulnerabilities; the complexity of the hardware design process plays a significant part. In a standard hardware design process, the designer is required to fully describe micro-architectural features on a cycle-by-cycle basis using register-transfer level (RTL) hardware description languages (HDLs). The verbosity and complexity of RTL HDLs opens the door for design errors and security vulnerabilities.

This problem is further aggravated by the lack of security-oriented languages and tools to specify and verify security policies. Conventional HDLs such as Verilog, VHDL or System Verilog do not directly support specification of security properties as they are traditionally built for describing design functionality. Similarly, majority of advances in EDA tools focus on analyzing functional correctness and performance requirements. Thus, the state of the art secure hardware design process requires hardware designers or verification engineers to manually inspect HDL designs for security flaws. The absence of automated hardware security verification tools requires hardware designers to be security experts in order to capture and eliminate security vulnerabilities – a requirement which is rarely met.

To enable automatic analysis of security policies, several industrial and academic tools have been developed. These tools commonly use information flow tracking (IFT) to inspect hardware designs for information leakage. IFT tools assign labels to design elements (e.g., registers, wires, memory units, etc.) to capture security-relevant information. These security labels are propagated through the design and model movement of sensitive information through the design. Various security properties such as confidentiality, integrity, isolation, and timing side-channel can be modeled by constraining the IFT labels . For instance, to analyze the integrity policy in a system, the IFT security labels capture how the data which is received from untrusted inputs influences critical design components.

IFT tools are growing in popularity for verification of digital designs as they provide a systematic and yet flexible framework for security analysis. However, multiple challenges obstruct usability and effectiveness of these tools. Many of the existing IFT tools require the designer to manually annotate the design with security labels [ZWSM15, LKO+14, LTO+11a, BM15] or work at very low level of abstraction [TWM+09, OMSK14b, TWM+09]. Relying on manual definition of security labels hinders accessibility of IFT tools as labeling increases the design time. The problem is aggravated in cases where the annotating process must be repeated according to different security policies. IFT analysis at low abstraction levels such as the gate level also

negatively affects usability of these tools since the verification time increases by moving to lower abstractions.

Furthermore, current IFT tools are merely utilized to check if a given design adheres to a set of security properties. If the verification fails, i.e., if an unexpected flow of information is discovered, the IFT tool does not facilitate localizing and fixing the error. Thus, the user needs to manually inspect and modify the design to eliminate the information leakage. Potentially, the error localization, debugging, IFT instrumentation, and verification cycle must be iterated several times. Ideally, this process should be automated to enable eliminating the information leakage with minimal effort from the user.

This dissertation introduces multiple tools and techniques to enhance hardware IFT tools and tackle some of the aforementioned shortcomings. We start by giving an overview of various security policies that can be specified using the model of information flow in Chapter 2. We introduce RTLIFT, a generic IFT tool for verifying information flow properties on RTL hardware designs in Chapter 3. Next, we extend RTLIFT information flow logic to precisely capture timing flows in RTL designs. Clepsydra, which is described in Chapter 4, uses this extended logic for verification of hardware designs against timing side-channel attacks. While RTLIFT and Clepsydra improve the security verification process, they do not provide any guidance for secure hardware design. To address this issue, we present VeriSketch in Chapter 5. VeriSketch completes a partially written hardware design with regards to information flow security and functional properties. Lastly, in Chapter 6, we propose a framework to automatically reason about verification failures and localize errors in RTL designs.

The rest of this chapter presents the outline and motivation of the remaining chapters.

1.1 Information Flow and Security Properties

Major security policies such as isolation, integrity, and confidentiality cannot be modeled by properties which describe a single trace. This is due to the fact that these policies inherently express the system behavior with respect to two traces. For instance, the confidentiality policy considers two traces which differ only in the values of the secret data. To preserve confidentiality, no difference in publicly observable behavior of the system which is executing the two traces should be noted.

Information flow models enable capturing two-trace properties through the security labels. Consequently, information flow tools are growing in popularity for reasoning about security and safety properties. Hardware IFT tools have been used at different levels of abstractions ranging from the ISA level down to the gate and the transistor level. This dissertation focuses on the RTL abstraction as it is the most prevalent way for designing digital circuits. The existing RTL IFT tools utilize different verification strategies and vary in terms of the level of automation. In this chapter we describe the basic concepts and taxonomy of these IFT tools and analyze the advantage and shortcomings of different approaches. Next, we show how information flow models can be used to specify different security policies. This is achievable by constraining the security labels of a design that is augmented with IFT logic. We focus on properties related to confidentiality, integrity, isolation, timing side-channel, and information-leaking hardware Trojans, and we provide examples of IFT security policies written for various hardware units. These properties and benchmarks will be used in the rest of this dissertation to evaluate the proposed tools and techniques.

1.2 RTLIFT: Verification of Secure Hardware Designs

Over the past decade, several tools have been developed for verification of information flow properties of RTL hardware designs [ZWSM15, LKO+14, LTO+11a, BM15]. To utilize

these tools, the user needs to annotate the design by assigning security labels to each design variable. Similar to a type-checking framework, this annotated code is then verified to ensure that the labeling system is consistent. While this semi-automated approach provide flexibility as the user can define complex and mutable security labels, it considerably increases the design time since it requires substantial annotation. Furthermore, the labeling procedure itself could be a source of error since the user should take into account all low-level details such as different execution paths, timing channels, and intermediate variables.

Ideally the IFT labeling process should be automated and require minimal inquiry from the user. GLIFT [TWM$^+$09, TWM$^+$09] alleviates this challenge by leveraging automated label inference rules and introduces an IFT technique for verification of IFT properties at the gate level. The main drawback, however, is that GLIFT requires the design to be first synthesized to a gate-level netlist. This increases the analysis time as verification is considerably slower at lower levels of abstraction. Also, the synthesis time could be large itself and should be taken into account. Moreover, once synthesized to the gate-level, high-level RTL information (e.g., conditional branches, execution paths, control flow graph, etc.) which are valuable for precise label inference are lost.

In this chapter we introduce RTLIFT [AHMK] – an information flow tracking framework for RTL hardware designs. RTLIFT requires the designer to only annotate the design's inputs with security labels. And it automatically infers the labels for the intermediate variables and outputs. This is done by defining label propagation rules (i.e., label inference rules) for all RTL constructs (e.g., Boolean operators, arithmetic operators, control constructs, etc.) as part of RTLIFt internal library. As output, RTLIFT generates an RTL design which has the same functionality as the original design and is instrumented with IFT labels. This instrumented design along with the IFT properties (i.e., constraints written over the IFT labels as described in Chapter 2) can be analyzed by off-the-shelves verification oracles to reason about information leakage. This verification can be performed by EDA simulation tools, formal methods, or hardware emulation. Once the

verification passes, the IFT labels are discarded and the original design goes through the next steps of the hardware design cycle.

As we will show in Chapter 3, RTLIFT alleviate the aforementioned issues with hardware IFT tools by providing an automated, scalable and flexible security verification framework for hardware designs written in the Verilog language. To provide more flexibility, RTLIFT introduces propagation rules with different level of precision by leveraging various RTL information.

1.3 Clepsydra: Modeling Timing Flows

Information leakage could be caused by *functional flows* or *timing flows*. Existence of *functional flow* from signal `A` to signal `B` is defined as variation in the value of signal `B` caused by changes in the value of signal `A`. *Timing flows* are a more subtle subset of information flows and refer to information leaked via the time that a signal is updated (as opposed to the value of the signal). More specifically, *timing flow* exists from signal `A` to signal `B` if the value of signal `A` affects the *time* when the value of signal `B` is ready. Timing flows are of great importance as they are the root cause of timing side-channel attacks such as Percival [Per05a] and Bernstein [Ber05] cache attacks and more recent Spectre [KGG+18] and Meltdown [LSG+18] attacks. To lunch a timing side-channel attack, the adversary monitors the execution repeatedly and measures the execution time. Then the attacker leverages statistical analysis methods to correlate the execution time with the data that is being processed. Thus, a major mitigation against timing side-channel attacks is to eliminate timing variations which are dependent on secret data.

RTLIFT [AHMK] and other existing hardware IFT tools such as SecVerilog [ZWSM15], GLIFT [TWM+09], Sapper [LKO+14], Caisson [LTO+11a], and VeriCoq [BM15] can detect information leakage caused by both functional flows and timing flows; but they fail to distinguish between the two cases. However, isolating timing flows from functional ones are critical as they are exploited in different ways. In many cases, functional flows are benign since the signal

values might be inaccessible or protected by cryptographic means. Despite that, timing flows might still be observable through repeated measurements and statistical analysis. For instance, in a crptographic core the ciphertext is protected as it is encrypted, but the time taken by the core to compute the ciphertext could contain secret information. Consequently, to show that the crptographic core is resilient against timing attacks, one needs to prove absence of timing flows in the presence of functional flows.

In Chapter 4, we show how cycle-level timing variations can be modeled in RTL hardware designs by analyzing how design registers are updated at each cycle. To do so, we formalize how timing variations are generated and propagated at RTL abstraction. We further prove that the introduced model soundly detects all timing flows and conservatively discards functional-only flows. Based on the proposed model, we introduce Clepsydra [AHK] for tracking timing flows and functional flows in separate channels. Clepsydra receives as input an RTL hardware designs written in Verilog. And it instruments the design with IFT labels and propagation logic while preserving its original functionality. Contrary to previous hardware IFT tools, Clepsydra uses disjoint sets of labels and tracking logic for timing and functional flows. As a result, one can use verification tools to prove that the instrumented hardware implementation is insusceptible to various timing side-channel attacks. We use Clepsydra to verify various hardware units such as secure cache architectures, bus arbiters, and cryptographic cores against timing side-channel attacks.

1.4 VeriSketch: Synthesis of Secure Hardware Designs

IFT tools enable evaluation of hardware designs against different security policies and extend the verification round in the hardware design flow with security analysis. To utilize these tools, the design is first instrumented with IFT labels (automatically or manually), and then it is analyzed by a verification oracle (e.g., SAT/SMT solvers or type checkers). If the verification

passes, the IFT labels are discarded and the hardware design process continues since the design is certified with respect to the security specifications. If the verification fails, however, the design should be debugged and re-evaluated manually.

Ideally, this procedure should be automated by utilizing the IFT tools in the design phase to direct the designer towards an implementation which passes the verification. One promising approach to achieve this goal is by leveraging program synthesis techniques to complete a partially-specified design according to a set of properties. Program synthesis algorithms consider a partial description of a program along with a set of properties or input/output traces and search for a complete program which satisfies both the partial design and the constraints. Program synthesis [Gul10] frameworks have been developed for automating challenging software engineering problems in different domains such as data processing, optimization, and security. However, most synthesis tools focus on functional properties. To enable synthesis of secure hardware designs, we combine program synthesis techniques and IFT tools to generate HDL designs with respect to information flow security properties.

In Chapter 5, we introduce VeriSketch [ATGK19] for synthesizing RTL hardware designs with respect to timing-sensitive information flow properties as described in Chapter 2. VeriSketch accepts partially defined Verilog designs (i.e., the sketch) along with a set of functional and IFT properties (i.e., the constraints). And it outputs a fully defined Verilog code which is proven to satisfy the constraints. To enable partially written hardware designs, we add sketch constructs to the Verilog language. The sketch constructs allows the programmer to leave the low level details of the design such as conditional branches, operation/operand selection, combinational functions, finite state machine transitions, etc., as undefined. These undefined components will be filled by the synthesis algorithm.

VeriSketch uses counterexample guided inductive synthesis (CEGIS) [SLTB$^+$06] to complete the sketch. CEGIS breaks down the synthesis problem into multiple iterations of *synthesis* and *verification*. In the verification round the design is examined against the given

properties and a counterexample is generated if the verification fails. In the synthesis round, a new designs (i.e., a complete Verilog code) is proposed by the SMT solver such that it satisfies the properties for the collected counterexamples. CEGIS iterates between synthesis and verification rounds until either the verification passes (i.e., a correct design is generated) or the synthesis fails (i.e., there is no program which satisfies all the constraints). To enable synthesis of IFT properties, VeriSketch leverages Clepsydra to instrument the design in each verification and synthesis round. In Chapter 5, we show how VeriSketch can be used to sketch and generate secure-by-construction hardware designs.

1.5 Error Localization for Hardware Designs

Formal verification tools enable analysis of a design with respect to concise specifications written as formal properties. Formal verification can simplify and accelerate debugging by eliminating the need to write simulation testbenches and looking for design corner cases; however, they provide minimal help for localizing source of errors in a given design. If the formal property fails, the verification oracle generates a counterexample which represents an input trace that falsifies the property. The counterexample trace can be used to replay the error but it does not pinpoint the source of the failure. To eliminate the error, the user needs to manually inspect the design along with the counterexample trace, localize the problem, and re-verify the design after modifying it. This process potentially might be iterated multiple times until the verification passes. And if the specification to be verified are IFT properties, the instrumentation should be re-done at each iteration as well.

Multiple techniques have been proposed to automatically localize errors and reduce the time which is spent on debugging. These techniques falls in two generic categories — statistical and formal techniques. Statistical methods require multiple simulation traces (both failing and passing traces) to correlate the failure with program snippets. Formal techniques, on the other

hand, rely on few traces and leverage formal solvers to reason about the source of the error. We focus on formal techniques since we aim to find the root cause of the failure associated with a single counterexample that illustrates the property violation.

In this chapter, we present a method for localizing single faults in RTL hardware designs. The error localization algorithm works in three steps and tries to find the minimal code snippets which if modified can eliminate the error. First, the counterexample trace is analyzed to find the input variables which are critical for reproducing the error. Next, these critical inputs are traced through the program to find the critical intermediate values. This is done by static information flow analysis. Lastly, the selected intermediate values are trimmed again to choose the ones that contribute to existence of the error. The first and last steps leverage an SMT solver to safely eliminate variables which do not affect the error. In Chapter 6 we show how the error localization method finds buggy code snippets in various HDL designs.

Chapter 2

Information Flow and Security Properties

Formal methods are commonly employed to ensure functional correctness and reveal design flaws in safety-critical and security-oriented systems. However, the major focus of formal verification techniques have been on reasoning about functional properties. And analysis of security specifications mainly relies on manual effort. More recently, several tools and techniques have been proposed to bridge this gap and enable evaluation of security properties using formal methods. Many of these tools leverage information flow models to define and verify security policies. In this chapter we show how information flow models can be used to express a wide group security policies such as integrity, confidentiality, and isolation, and we provide several examples of each of the policies written for various hardware units.

2.1 Introduction

Formal specifications concisely capture expected behavior of a design and thus can substitute exhaustive testing while providing sound guarantees for high-assurance systems. Formal verification tools are mostly utilized to ensure functional correctness using trace properties. This is due to the fact that property specification languages enable writing properties which model the behaviour of the design considering a single trace, i.e., how the design should perform given an

input trace described by the property. While trace properties are powerful to encode functional properties, modeling security policies commonly requires comparing the behavior of the design with respect to two (or more) traces. These properties which express the expected behavior of the design according to multiple traces are referred to as *hyperproperties* [CS10].

To illustrate the notion of hyperproperties, consider as an example the *integrity* policy. The integrity policy indicates that the behavior of trusted components of a design should remain intact with respect to values of untrusted data objects. This policy essentially compares the behavior of the design against multiple traces with differing values for the untrusted data. Consequently, the integrity policy cannot be represented by a single trace property. However, integrity can be represented by a hyperproperty that considers two traces which are identical except for the value of the untrusted data. To preserve integrity, the system is expected to perform identical (in terms of the trusted components) given the two traces and starting form equivalent initial states. Using formal methods, this hyperproperty can be verified for all possible values of untrusted data components to provide formal proof of integrity in all possible executions. Similarly, other security policies such as confidentiality, isolation, and timing side channels can be modeled by hyperproperties. We provide more examples of security policies which are modeled by hyperproperties in Section 2.3.

Security hyperproperties can be represented using the model of information flow. Hence, information flow tracking tools are gaining popularity for security verification. IFT tools model movement of information through a given design using labels that store security relevant information. Depending on the property to be verified, the IFT tool tracks different metadata such as whether data objects are trusted or untrusted, confidential or public, contains timing variation or not, etc. Considering the integrity policy again, the information flow model stores labels which indicate if data values are trusted or not. Using this labeling scheme, the integrity policy can be expressed by a property that considers a single trace which labels are initialized according to the trust level of the data objects. For instance, data objects which carry untrusted values are

initialized to have a *high* label while other data objects have a *low* label by default. To ensure integrity, the system is expected to maintain the *low* label for the components which are assumed to be trusted throughout execution. Maintaining a *low* label in this scenario demonstrates that untrusted values have not influenced the trusted components.

In this chapter we describe the basic notions of information flow tracking and elaborate how different security properties can be verified leveraging the model of information flow. We will use the security properties and examples presented in this chapter throughout this thesis to evaluate different IFT verification and synthesis frameworks.

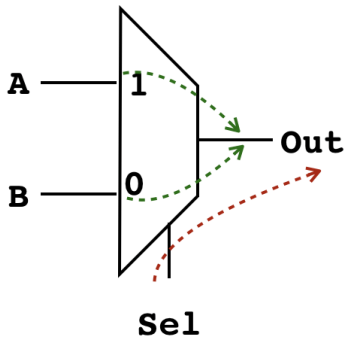
2.2 Information Flow Tracking

Information flow tracking tools model data propagation through a system to enable reasoning about security properties. To illustrate information flow in hardware modules, consider a multiplexer which determines the value of output signal `Out` by choosing from one of the two input signals `A` and `B` based on the control signal `Sel` as shown in Fig. 2.1. In this example, information flows to output `Out` from inputs `A` and `B` due to direct writes. This direct flow of information is referred to as *explicit flow* and indicates that the destination signal is directly influenced by the value of the source signal. In the same example, information also flows from the control signal `Sel` to the output `Out` since the control signal dictates which input signal should be assigned to the output `Out`. This indirect flow of information caused by conditional branches is denoted as *implicit flow*. Explicit and implicit flows in the multiplexer example are shown by green and red arrows in Fig. 2.1(b).

To track flow of information, the IFT tool augments the design with labels that keep track of the *type* of the data that is being processed. Different examples of information flow tracking for the multiplexer example of Fig. 2.1 is shown in Fig. 2.2. In this example IFT labels are defined for all the signals using "`t`" variables. The IFT logic updates these labels based on how data

```
module Mux (input A, input B,
            input Sel, output Out);
always @ (*) begin
  if (sel)
    Out = A;
  else
    Out = B;
end
```

(a)



(b)

Figure 2.1: A simple example of information flow in hardware designs. (a) HDL code for a two-way multiplexer, (b) Schematic of a two-way multiplexer where explicit and implicit flows are shown by green and red arrows, respectively.

propagates through the design via different operations. These update rules are determined based on many factors such as the property to be verified, the required precision and the abstraction level. Fig. 2.2(a) shows a simple IFT instrumentation where only explicit flows are tracked, while Fig. 2.2(b) represents a labeling scheme which tracks both explicit and implicit flows. A more complex flow tracking logic takes into account the Boolean values of the signals along with the connectivity analysis and precisely capture cases where the output signal is influenced by the inputs. This is shown in Fig. 2.2(c) for the multiplexer example. The labeling system of Fig. 2.2(c) acknowledges the fact that information flows from signal `A` to output `Out` only in cases where the control signal `Sel` is set. Hence, the label propagation logic is itself guarded by a conditional branch. As it can be seen through the different possible propagation rules for the multiplexer, increasing the IFT logic precision adds to the complexity of the analysis. As we further elaborate in Chapter 3, this complexity increases the verification time.

The values that can be captured by the IFT labels are defined through a security lattice. A security lattice defines all the possible security levels and legal data movement between them as shown by several examples in Fig. 2.3. In the simplest case, a security lattice contains two levels – `low` and `high` as shown in Fig. 2.3(a). This lattice indicates that data with the `low` label

```
module Mux (input A, input B,
            input Sel, output Out);
wire A_t, B_t, Sel_t, Out_t;
always @ (*) begin
  if (sel)
    Out = A;
  else
    Out = B;
assign Out_t = A_t | B_t;
end
endmodule
```

(a)

```
module Mux (input A, input B,
            input Sel, output Out);
wire A_t, B_t, Sel_t, Out_t;
always @ (*) begin
  if (sel)
    Out = A;
  else
    Out = B;
assign Out_t = A_t | B_t | Sel_t;
end
Endmodule
```

(b)

```
module Mux (input A, input B, input Sel, output Out);
wire A_t, B_t, Sel_t, Out_t;
always @ (*) begin
  if (sel) begin
    Out = A;
    Out_t = A_t | Sel_t;
  end
  else
    Out = B;
    Out_t = B_t | Sel_t;
  end
end
endmodule
```

(c)

Figure 2.2: Examples of information flow tracking for the multiplexer shown in Fig.2.1. The highlighted parts show the code added for IFT instrumentation where IFT labels are defined for all the signals using the "t" variables. (a) A simple IFT instrumentation which tracks only explicit flows. (b) An IFT instrumentation which tracks both explicit and implicit flows. (c) A more precise IFT instrumentation which tracks both explicit and implicit flows by considering the Boolean values of the signals.

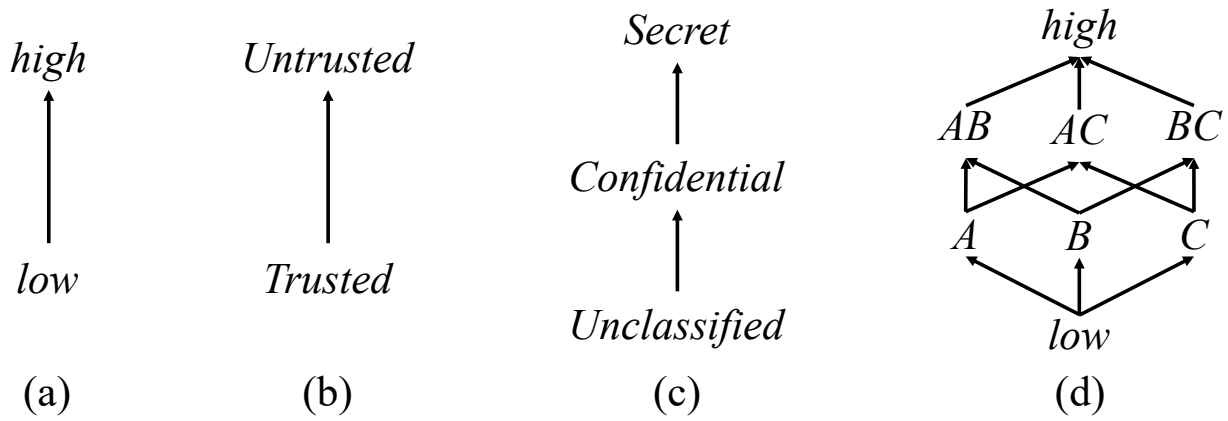


Figure 2.3: Examples of security lattices. A security lattice defines security levels and authorized relation between them. Any flow of information that is not allowed by the lattice violates security expectations. (a) A generic two-level security lattice. (b) A lattice describing the integrity policy. (c) A lattice describing the confidentiality policy. (d) A multi-layer security lattice to represent arbitrary complex policies.

is allowed to flow to and influence data with `high` label while flow of data with `high` label to data with `low` label is a security vulnerability. This simple lattice is powerful enough to model important security policies such as integrity and confidentiality. To capture the integrity policy, we simply need to mark "Trusted" values with `low` labels and "Untrusted" values with `high` labels as show in Fig. 2.3(b). Similarly, the confidentiality policy can be represented by denoting "Unclassified" values with `low` labels and "Secret" values with the `high` labels. Fig. 2.3(c) shows a slightly different security lattice with three levels to model confidentiality. If needed, lattices enable representing more complex policies as shown in Fig. 2.3(d).

2.3 Security Properties

Properties are logical formulas written over design variables using property specification languages (PSLs) and describe desired invariants in the design behavior. The model of information flow enables capturing a wider range of properties compared to what can be written and analyzed using existing PSLs and verification tools. This is feasible by writing properties over the IFT

labels that carry security metadata. These properties include a diverse set of security related behaviors such as information leakage, unauthorized access to restricted memory locations as well as undesirable interference between different execution contexts caused by design flaws, timing channels, insecure debug ports and certain types of hardware Trojans [SA14, CCF+16, HAAK16].

In the following, we show how different classes of security requirements can be modeled with IFT properties. We consider an information flow instrumentation where each design component is extended with two security labels, "s" and "t" labels for functional and timing flows, respectively. All the properties can be mapped to the two level security lattice `low` $\sqsubseteq$ `high` as shown in Fig.2.3(a). We use System Verilog Assertion (SVA) language to write these properties.

2.3.1 Confidentiality

Confidentiality properties ensure that sensitive data with `high` label never flows to an unclassified variable with `low` label. For instance, in a cryptographic core the secret key should not flow to a point that is publicly observable. Using the IFT model, this can be stated as following.

```
assume(key_s==high);
assert(pub_s==low);
```

Confidentiality properties are written to protect secret assets in a given design. This is done by marking the secret assets or inputs with a `high` security label and monitoring the label of public ports and storage units. To preserve confidentiality, all public ports and storage units should maintain a `low` label throughout the execution. Examples of confidentiality properties for various hardware designs such as crypto cores and arithmetic units are shown in Table 2.1.

2.3.2 Integrity

Integrity is the dual of confidentiality, where we mark untrusted resources with a `high` label and verify that they do not affect critical components with `low` labels. For example, in a

processor the program counter (PC) should not be overwritten by data from unprotected network. This can be modeled as following.

```
assume(ethernet_data_s==high);
assert(pc_s==low);
```

Integrity properties can be specified for any design where certain memory locations, registers or flags should be protected against unauthorized access. This is modeled by marking public access such as user or network input with a `high` label and constraining the sensitive variables to maintain a `low` security label. Table 2.1 shows integrity properties written for crypto cores, processors, and access control mechanisms.

2.3.3 Isolation

Isolation can also be enforced as an information flow security property. Isolation states that there should never be information exchange between two components with different trust levels. For example in SoC designs, trusted IP cores sitting in the secure world with `low` labels should be separated from those which are untrusted and are in the insecure domain with `high` labels. It should be noted that isolation is a two-way property as shown in the following as well as the examples of Table 2.1.

```
assume(crypto_core_s==high);
assert(lcd_ctrl_s==low);
assume(lcd_ctrl_s==high);
assert(crypto_core_s==low);
```

2.3.4 Timing Channel

Information flow models can be used to capture timing side-channels in hardware designs. These properties assess whether sensitive information can be retrieved by measuring

Table 2.1: Summary of security properties for confidentiality, integrity, and isolation.

Benchmark	Synopsis	Formal Representation
SoC Arbiter	(Confidentiality) Acknowledgement signal is not driven from sensitive requests	`assume(req_s[i]==high );` `assert(ack_s[j]==low)`
Scheduler	(Confidentiality) Grant signal is not driven from sensitive modes	`assume(mode_s[i]==high );` `assert(grant_s[j]==low)`
Crypto Core	(Confidentiality) Ready signal is not driven from secret inputs	`assume(key_s==high & plain_text_s==high);` `assert(ready_s==low)`
ALU	(Confidentiality) Ready signal is not driven from the inputs	`assume(operand1_s==high & operand2_s==high);` `assert(ready_s==low)`
Crypto Core	(Integrity) Key register is not modified by public inputs	`assume(user_inp_s==high);` `assert(key_s==low)`
Debug Unit	(Integrity) Debug flag is not modified by public inputs	`assume(user_inp_s==high);` `assert(debug_en_s==low)`
Processor	(Integrity) PC, private memory and control flow conditions are not modified by public inputs	`assume(user_ inp_s==high);` `assert(PC_s==low & private_mem_s==low & cond_s==low)`
Access Control	(Integrity) Unauthorized users cannot access protected units	`assume(user_s[i]==high);` `assert(asset_s[j] == low)`
SoC	(Isolation) Accesses to different cores on an SoC are isolated	`assume(req_s[i]==high); assert(ack_s[j]==low)` `assume(req_s[j]==high); assert(ack_s[i]==low)`
Memory	(Isolation) Memory locations are isolated	`assume(mem_s[i]==high); assert(mem_s[j]==low)` `assume(mem_s[j]==high); assert(mem_s[i]==low)`

the computation time. To precisely capture timing flows, the information flow model needs to distinguish between logical and timing flows. Take as example a floating point division unit which is expected to run in constant time independent of the value of the operands. In this case, Logical flow exists from the data inputs (i.e., the divider and dividend) to the data outputs (i.e., quotient and remainder) since the outputs are computed from the inputs. However, whether or not there is timing flow from the data inputs to the data outputs (i.e., if the arithmetic unit runs in constant time or not) depends on the implementation of the floating point unit. Using the model of information flow, existence of such timing side channel can be captured as follows.

```
assume(divider_s==high);
assume(divisor_s==high);
assert(quotient_t==low);
assert(remainder_t==low);
```

Here, we assume an information flow model where a separate channel is dedicated to track timing flows (denoted by "t" labels). To verify timing leakage, the assertions are written over these "t" labels. Table 2.2 summarizes the properties developed in [AHK, ATGK19] to analyze timing side channel in different hardware designs.

Table 2.2: Summary of properties used for detecting timing side channel

Benchmark	Synopsis	Formal Representation
Sequential Divider	Result is ready in constant time	`assume(dividend_s==high & divisor_s==high); assert(quotient_t==low & remainder_t==low)`
Sequential Multiplier	Result is ready in constant time	`assume(dividend_s==high & divisor_s==high); assert(quotient_t==low & remainder_t==low)`
Cache	Data is available in constant time	`assume(index_s[i]==high); assert(data_t[j]==low)`
SoC Arbiter	Requests are granted in constant time	`assume(req_s[i]==high); assert(ack_t[j]==low)`
Thread Scheduler	Scheduling is done in constant time	`assume(thread_active_s[i]==high ); assert(thread_grant_t[j]==low)`
AES Cipher	Cipher text is ready in constant time	`assume(key_s==high & plain_text_s==high); assert(cipher_text_t==low)`
RSA Cipher	Cipher text is ready in constant time	`assume(key_s==high & plain_text_s==high); assert(cipher_text_t==low)`

Table 2.3: Summary of properties used for detecting hardware Trojans

Benchmark	Synopsis	Formal Representation
AES-T100	Key does not flow to Antena	`assume(key_s==high); assert(Athena_s==low)`
AES-T400	Key does not flow to shift register	`assume(key_s==high); assert(TSC-SHIFTReg_s==low)`
AES-T1100	Key does not flow to capacitance	`assume(key_s==high); assert(Capacitance_s==low)`
RSA-T200	Key does not flow to count	`assume(key_s==high); assert(count_s==low)`

2.3.5 Hardware Trojan

Information flow tracking can be used to detect certain types of hardware Trojans that leak sensitive information by inserting malicious information channels in the design. For example, Trust-HUB benchmarks [STK13] include examples of Trojans added to crypto cores using a small circuitry that transfers the secret key to a public output under certain conditions. This class of hardware Trojans can be detected using a property that observes information flow from sensitive data (e.g., the secret key) to the public ports as shown in the following.

```
assume(key_s==high);
assert(pub_s==low);
```

Table 2.3 summarizes the the properties for detecting information leakage by gate level information flow tracking in the Trust-HUB [STK13] benchmarks as used by Hu et. al. [HMOK16a].

2.4 Conclusion

This chapter provides a brief overview of hardware information flow tracking tools and describes how a wide variety of security policies can be expressed leveraging the model of information flow. The properties and examples discussed in the chapter will be used in the following chapters to assess different IFT verification and synthesis frameworks.

Chapter 2, in part is currently being prepared for submission for publication of the material. Wei Hu, Armaiti Ardeshiricham, Ryan Kastner. The dissertation author was the co-investigator and co-author of this paper.

Chapter 3

Verification of Secure Hardware Designs

Information Flow Tracking (IFT) provides a formal methodology for modeling and reasoning about security properties related to integrity, confidentiality, and logical side channel. Recently, IFT has been employed for secure hardware design and verification. However, existing hardware IFT techniques either require designers to rewrite their hardware specifications in a new language or do not scale to large designs due to a low level of abstraction. In this work, we propose Register Transfer Level IFT (RTLIFT), which enables verification of security properties in an early design phase, at a higher level of abstraction, and directly on RTL code. The proposed method enables a precise understanding of all logical flows through RTL design and allows various tradeoffs in IFT precision. We show that RTLIFT achieves over 5× speedup in verification performance as compared to gate level IFT while minimizing the required effort for the designer to verify security properties on RTL designs.

3.1 Introduction

Despite decades of research on software and hardware security, today's computing technology remains vulnerable to classic exploit techniques due to the fact that many fundamental computational models and traditional design flows do not consider or prioritize security. In

the realm of hardware, security is rarely considered during design space exploration. Security vulnerabilities can originate from design flaws, which can be fully eliminated after a complete verification. Unfortunately, this is impractical due to the scale of modern chips. Furthermore, novel attack vectors like side-channel analysis undermine classic assumptions about the accessibility of internal secret information outside of a computing system. In addition, hardware designs often required incorporating third party IP cores, which may contain undocumented malicious design modifications known as backdoor.

Remedying these security vulnerabilities is best accomplished with a systemic solution. One such solution that has shown promise is information flow tracking (IFT). IFT models how labeled data moves through a system. It provides an approach for verifying that systems adhere to security policies, either by static verification during the design phase or dynamical checking at runtime. Recent work has demonstrated the effectiveness of hardware IFT in identifying and mitigating hardware security vulnerabilities, such as timing channels in cryptographic cores and caches, unintended interference between IP components of different trust, and information leakage through hardware Trojans [TWM$^+$09, TOL$^+$11, LTO$^+$11b, LKO$^+$14, BM15, ZWSM15].

Hardware IFT techniques have been deployed at different levels of abstraction. At the gate level, all logical information flows can be tracked by augmenting each logic primitive in the synthesized design netlist with additional IFT logic. While this simplifies IFT logic generation by breaking down complex language structures to lower level logic constructs, this method does not scale with design size since it relies on gate level verification. Language level methods avoid this problem by generating IFT logic at a higher level of abstraction. Previous language level IFT techniques are accomplished via designing type enforced HDL (Hardware Description Language). These techniques require that designs be rewritten or annotated in a new language in order to verify security properties, which can be a challenging task for hardware designers.

The level of abstraction can also affect the precision of the IFT logic. More specifically, the precision of IFT logic is determined by both the precision of the label propagation rules for

logic operations and the granularity of the building blocks over which IFT is deployed. Gate level IFT methods [**?**] achieve increased precision by defining precise tracking rules for a set of universal gates. On the other hand, gate level methods apply IFT at a fine granularity, and thus cannot detect higher-level dependencies (e.g., variable correlation due to reconvergent fanout) between the signals in the design.

In this work, we propose a Register Transfer Level IFT (RTLIFT) method, which allows a precise understanding of all logical information flows through RTL code. By defining precise label propagation rules for RTL expressions, we show that our method can achieve a higher level of precision as compared to gate level methods. Our IFT model is completely described with standard RTL syntax and thus eliminates the need for acquiring a new type-enforced language. Furthermore, we discuss how RTLIFT allows for separation of implicit and explicit flows, enabling various tradeoff of IFT precision, which cannot be realized using existing IFT methods. Specifically, this paper makes the following contributions:

- Proposing a method for precisely understanding all logical information flows through RTL design;
- Providing techniques that allow trading-off IFT precision and security verification performance;
- Presenting experimental results to show the improvement in IFT precision and security verification time.

The remainder of this paper is organized as follows: Section 3.2 summarizes existing IFT methods and discusses how our work differs from them. Section 3.3 deliberates the concept of precise IFT and what RTLIFT aims to improve in this work. Section 3.4 describes the implementation details. The experimental results are reported in Section 3.5. We conclude in Section 3.6.

3.2 Related Work

Software-mediated IFT, e.g., LIFT [Fen06] and RIFLE [VBC+04], use dynamic binary translation to track flows during execution. The overhead of these methods can be prohibitive, and thus motivated the use of custom architectural modifications to facilitate faster information flow tracking. Such schemes generally fall into one of two categories – integrated pervasive processor modifications (e.g., Minos [CC04] and Raksha [DKK07]) and modular core additions or coprocessor support (e.g., Flexitaint [VDSP08] and Kannan et al. [KDK09]).

Gate level information flow tracking (GLIFT) [TWM+09] performs IFT analysis directly on the "original" hardware design. It does this by creating a separate GLIFT analysis logic that is derived from the original logic, but operates independently from it. GLIFT tracks any arbitrary set of flows by labeling different hardware variables as "tainted", and tracking their effect throughout the design. The GLIFT logic is generated once, independent of the security property, and can be used to verify any IFT property. GLIFT is primarily used at design time for testing and verification but it can be used to dynamically track information flows, and thus perform runtime security checks, but the resulting tracking logic may be prohibitive for some applications [TOL+11].

Caisson and Sapper are hardware security design languages that directly generate circuits that enforce the desired IFT properties (e.g., separation and isolation). Both add a typing system to a finite state machine (FSM) language which requires that the designer assigns a security label to each register and wire. Caisson [LTO+11b] uses static types forcing it to conservatively perform replication to restrict flows of information. This can lead to significant increases in logic. Sapper [LKO+14] improves Caisson by adding a dynamic type system. This reduces the need for logic replication, but still requires that the designer learn a new language. Sapper and Caisson enforce information flow by restricting transitions between the states. Hence, the user must redesign the hardware using a new language which can be a nontrivial task.

VeriCoq-IFT [BM15] automatically converts designs from their HDL representation to the Coq formal language, eliminating the need to redesign the hardware. However, the user still needs to annotate the generated Coq code in order to analyze security properties. Furthermore, in all these three methods the flow of information is tracked conservatively since the label propagation rules are defined as updating label of the output of any operation to the highest label of its inputs. As we discuss in this work, this approach overestimates the flow of information by ignoring the functionality of the operation and the exact values of the operands.

SecVerilog [ZWSM15] extends the Verilog language with an expressive type system. SecVerilog users are required to explicitly add a security label to each variable in the code. These labels are a consequence of the security property that one wishes to verify on the design. It uses a type system to ensure that the specified information flow policy is upheld. Being a thoroughly static tool, SecVerilog uses predicate analysis in order to acquire the hardware state essential for precise flow tracking. This complicates the labeling processing, and inevitably the intricacy of precise predicate analysis leads to loss of precision compared to simulation-based and dynamic approaches. Furthermore, the designer must specify many of the flow rules when she adds labels to the variables. Ideally, this process would be automated, e.g,. as done with GLIFT, otherwise it impedes its use as a hardware security design tool. Often there are many (potentially hundreds or thousands) of IFT properties that one wishes to test or verify on a single design. This would require the user to relabel the design in order to prove each different property. For example, in a cryptographic core proving that the secret key value does not affect the timing of the output signals requires a different labeling from proving that no inputs except the key and the plain text can affect the value of the cipher text.

RTLIFT creates a new methodology that combines the benefits of these previous approaches while eliminating their drawbacks. RTLIFT works directly with existing HDL languages, and thus does not require a designer to learn a new hardware security language. Much like GLIFT, it automatically defines flow relation properties. Yet working at a higher level of abstraction

leads to many benefits including faster verification time and more flexibility in defining different types of flow relationships (e.g., implicit versus explicit). The method presented in this paper is different from available high level IFT techniques in the sense that the user does not need to redesign or annotate the code since the labels are generated automatically. Furthermore, by specifying precise tracking rules which evaluate the exact state of the hardware through simulation it achieves a higher level of precision. It improves upon GLIFT by accelerating the verification process, decreasing the false positive rate and enabling separation of implicit flow from explicit flows.

3.3 Background and Motivation

3.3.1 IFT Basics

Information flows from signal X to signal Y if and only if a change in the value of X can influence the value of Y. Information flow can model security properties related to both confidentiality and integrity:

- **Confidentiality:** Assume that X is a secret value while Y is publicly observable. In this case, an attacker can extract sensitive information by observing and analyzing the variations in signal Y. For example, Y could be the "ready" output of a cryptographic core which in a secure design should not depend on the value of the private key stored in X; otherwise there is a timing side channel which can be used to extract the secret key. In this case, we want to insure the property that X does not flow to Y.

- **Integrity:** Assume that X is an untrusted value while Y is trusted. In this scenario, we wish to insure that an attacker cannot gain unauthorized access through Y by modifying the value of X. For example, X may be an openly accessible memory location and we wish to insure that it cannot be used to influence the results of a system control register. Thus, we want to

insure again that X cannot flow to Y.

Hardware information flow tracking generally works by adding a security label to each signal, and using that to track the influence of flow (or *taint*) of a set of signals throughout the circuit. The initial taint is set based upon the desired security property, and IFT techniques are used to test or verify whether that taint can move to an unwanted part of the system as specified by the security property.

IFT can be done with various levels of precision. One approach marks the output of each operation as tainted when any of its inputs is tainted. While simple, this method is overly conservative and can inaccurately report existence of flow in certain cases (i.e., *false positives*). This inaccuracy is due to the fact that based on the functionality of the operation, a single untainted input can dominate the output, yielding an untainted output while other inputs are tainted. To avoid this imprecision, the tracking rules need to take into account both the type of the operation and the exact state of the hardware.

To clarify this idea, consider the expression `out = secret & 0x0F`, where sensitive information is stored in an 8-bit variable `secret` and we want to determine if the information from `secret` flows to the variable `out`. The most conservative and least precise approach would mark all bits of `out` as tainted since `secret` is tainted. A more precise strategy gives us slightly different answer: the secret information only flows to the four least significant bits of `out`, and the other bits should not be marked as tainted since their values are zero regardless of the value of `secret`. To achieve this level of precision, separate tracking rules for different operations shall be defined as we discuss in Section 3.4.1.Achieving this level of precision requires separate tracking rules for different operations. For example, for a 2-input AND gate the output is tainted only if both inputs are tainted, or if one input is tainted *and* the other input is high (logical `1`), which allows for the tainted input to effect the output.

3.3.2 IFT Precision-Complexity Trade-offs

Precise tracking rules impose more complexity, and hence it might be desirable to deliberately add some false positives to the IFT logic by taking a rather conservative approach. In large designs it might be beneficial to use imprecise and efficient approaches to track the flow through complex arithmetic operations, while preserving the precision for logical operations such as AND, OR, XOR, etc. As opposed to the gate level where the difference between logical and arithmetic operations is lost, considering the high level description of the design, one can define the tracking rules for various operations with different levels of precision and more flexibility.

The notion of taking various levels of precision based on the functionality becomes more important when extended to the different information flow paths in the design: the data flow and the control flow. The data flow represents how the information explicitly flows, while the control flow shows all the paths that might be traversed and hence contains information regarding the implicit flow. For example, in a conditional statement the flow from the right hand side expression to the left hand side variable is explicit and the logic implementing it is within the data path. However, the value of the left hand side variable is also implicitly affected by the conditional variable which is represented in the control path. Formally, an implicit flow is the influence that a data value has on other data values by virtue of its use as a condition. Even though the conditions might not directly interact with other values, they still affect values that otherwise would have been assigned new values had the condition gone the other way. Implicit and explicit flows are not distinguishable in the gate level netlist. However, we can differentiate between them at the language level. We exploit this idea in order to adjust the complexity of the tracking logic based on the verification objective. Specifically, when searching for timing flows, which are caused by implicit flow, keeping the tracking logic associated with the data path imprecise, − hence reducing the logic complexity − and implementing the control flow's tracking logic precisely, we can realize a smaller tracking logic which does not impose additional false positives for tracking the implicit flow.

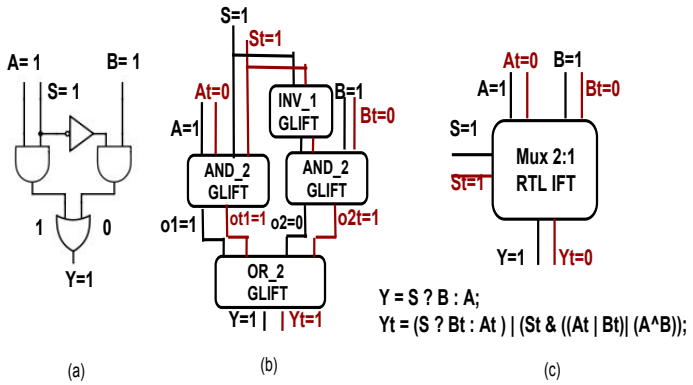


Figure 3.1: Different IFT logic for 2:1 mux. (a) Gate level structure of a mux. (b) IFT logic generated using GLIFT. (c) Precise IFT tracking logic for a 2:1 mux. RTLIFT can use either of these (or more) as an IFT library element.

3.3.3 IFT Precision

In this section we discuss how generating the IFT logic at a higher level of abstraction can improve its precision level. Gate level IFT techniques necessitate synthesizing the design to its gate level netlist before generating the IFT logic. Different logic synthesis methodologies can generate very different tracking logic via GLIFT, altering the result of the security verification [HBA+16]. Resource sharing done by the synthesis tool introduces reconvergent paths to the netlist which are not present at the language level. Reconvergent paths lead to false positives in the tracking logic since the tracking rules cannot easily take into account the exact relationship between multiple inputs of an operation. To clarify the source of such imprecision, we deliberate the gate level and high level tracking rules for a 2:1 multiplexer.

Fig. 3.1(a) shows the gate level structure of a multiplexer and Fig. 3.1(b) represents its precise gate level tracking logic. Even though the flow is precisely tracked through each single gate, when combined together the multiplexer's IFT logic contains false positives. To examine when a false positive happens, we analyze the case where both data inputs *A* and *B* are one, and their security labels *At* and *Bt* are zero, indicating being untainted. The control signal *S* and its security label *St* are both equal to one, indicating being tainted. Analyzing the gate level tracking logic, the output of both AND gates have high security labels while only one of them is on.

Consequently, both inputs to the OR gate are tainted resulting in a tainted output. Conceptually, the output of the OR gate is marked as tainted because flipping the value of its high input will change its result to zero. However, the missing part is that this flip cannot happen unaccompanied by a flip on the other input, which forces the final output to remain the same. This imprecision, in the presence of precise tracking logic for all the gates, occurs due to the reconvergence path at the input of the OR gate. The false positives provoked by the reconvergnece paths at the gate level can be avoided by generating the tracking logic at a higher level as we can see in Fig. 3.1(c).

Reconvegence paths also exist at the language level, which inevitably results in false positives. We can define precise tracking rules for gates or language constructs; however, this precision is based on the independency of the inputs. By generating the tracking logic at a higher level of abstraction and hence utilizing higher level tracking rules, we can overcome the dependency between the intermediate variables which improves the precision level. Looking back at the multiplexer example, we are able to eliminate the false positives which are caused by the dependency between the inputs of the OR gate. However, if the inputs of the multiplexer are dependent themselves, that can cause false positives which we cannot avoid. Fundamentally, precise information flow tracking is an undecidable problem as shown by Denning and Denning [DD77a]. Nonetheless, we improve IFT precision level in two ways: First, we have precise tracking rules specifically defined for each operation which take into account the exact state of hardware; second, we avoid a large class of false positives caused by the reconvergent paths in the gate level netlist by analyzing the design from a higher level of abstraction.

3.4 Implementation

In this section we elaborate details of RTLIFT implementation. RTLIFT software receives a synthesizable Verilog code along with flags specifying the precision level of the data flow IFT logic and the control flow IFT logic, and generates functionally equivalent Verilog code

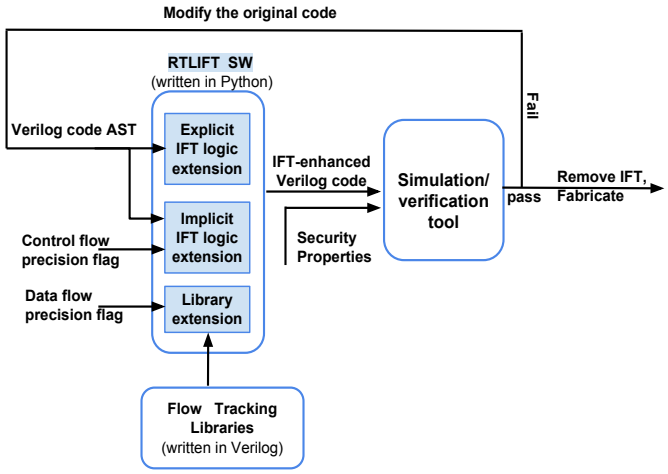


Figure 3.2: RTLIFT overview

instrumented with information flow tracking logic. Since the generated code is synthesizable, it can be analyzed by standard EDA test and verification tools allowing us to leverage decades of research on functional testing to assess security properties of hardware designs. If the IFT-enhaced design passes the security properties the original code can be used for fabrication. Otherwise, the original code should be modified and analyzed again. Fig. 3.2 gives an overview of how the tool is used. RTLIFT is realized through the following steps: Designing flow tracking libraries; enhancing the combinational circuit with tracking logic; enhancing the conditional statements with logic required for tracking the implicit flow. We describe each of these steps in the rest of this section.

3.4.1 Flow Tracking Libraries

For tracking the flow of information through an RTL code, each operation should be instrumented so it can operate both on the Boolean values and security labels of the operands. Hence, for each operation `OP` such that `Z = X OP Y` is a valid statement in Verilog, we have defined a module `OP_IFT` which receives inputs `X` and `Y` along with their security labels `X_t` and `Y_t`, and generates the output `Z` along with its security label `Z_t`. These modules are predefined and

```
module and_IFT //conservative
(Z, Z_t, X, X_t, Y, Y_t);
parameter w1,w2,w3;
output [w1-1:0] Z, Z_t;
input [w2-1] X, X_t;
input [w3-1] Y, Y_t;
assign out = X & Y;
assign out_t = X_t | Y_t;
endmodule
```

(a)

```
module and_IFT //precise
(Z, Z_t, X, X_t, Y, Y_t);
parameter w1,w2,w3;
output [w1-1:0] Z, Z_t;
input [w2-1] X, X_t;
input [w3-1] Y, Y_t;
assign out = X & Y;
assign out_t =  (X_t & Y_t) |
                (X & Y_t)|(Y & X_T);
endmodule
```

(b)

Figure 3.3: Flow tracking libraries

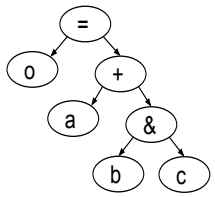
given to the RTLIFT software as an input file called "flow tracking libraries", as shown in Fig. 3.2. These libraries serve two goals: first, improving IFT precision by enabling operation-specific label propagation as opposed to the tracking rules in Caisson [LTO+11b], Sapper [LKO+14] and VeriCoq-IFT [BM15]; second, automating the computation of security labels in contrast to the approach taken in SecVerilog [ZWSM15]. We have defined two different sets of libraries, each of which calculating `Z_t` output with a different level of precision. In the *conservative* library, the label propagation rules overestimate the existence of flow by marking the output of each operation as tainted when any of its inputs are tainted, yielding a small tracking logic modeled with an OR expression. In the *precise* library, the label propagation rules are designed to minimize the number of false positives through each operation in exchange for a more complex tracking logic. The user selects which library should be used by specifying the precision flag for the data path logic. Other libraries with various precision-complexity balances can be added if required. Fig. 3.3(a) and (b) shows the IFT-enhanced modules for the AND operation available in the *conservative* and *precise* libraries respectively. In Section 3.5 we explain the design of IFT-enhanced modules for arithmetic operations in more detail.

3.4.2 Tracking Explicit Flows

Flow tracking starts by extending each bit of data, i.e., wires and registers in a given Verilog code, with a label that carries out information regarding the security properties of the data (Lines 4-6 in Algorithm 1). In this work, we consider a single bit label, where a high value

```
input [7:0] a,b,c;
output [8:0] o;
assign o = a + b&c;
```

(a)



(b)

```
input [7:0] a,b,c;
input [7:0] a_t,b_t,c_t;
output [8:0] o;
output [8:0] o_t;
wire [7:0] temp, temp_t;
and_IFT #(8,8,8) and1
        (temp, temp_t, b, b_t, c, c_t);
add_IFT #(9,8,8) add1
        (o, o_t, temp, temp_t, a, a_t);
```

(c)

Figure 3.4: Explicit flow tracking. (a) sample Verilog code. (b) Data Flow Graph of the code. (c) IFT-enhanced Verilog code.

indicates either secret or untrusted value, depending on whether we want to verify confidentiality or integrity IFT properties. To obtain a smaller IFT logic and further speed up the verification, it is possible to make a label for a multi-bit variable; this is the power of the library based approach.

Algorithm 1 IFT Logic Generation

1: Input: Verilog code AST file, IFT libraries
2: Input: $flag\_im, flag\_ex$
3: Output: IFT instrumented Verilog code
4: **for** each variable $V[n:0]$ **do**
5: define new variable $V\_t[n:0]$
6: **end for**
7: **for** each assignment Ai: $Vk := Exp(Xi, \cdots, Xj)$ **do**
8: Traverse the DFG of Ai in order
9: **for** each operation OP: $Yp = Xm\ OP\ Xn \in$ Exp **do**
10: instantiate module OP_IFT (Xm, Xm_t, Xn, Xn_t, Yp, Yp_t)
11: **if** (OP = DFG.root) **then**
12: $Vk := Yp$
13: $Vk\_t := Yp\_t \mid \mathsf{Imp\_Flow}(Ai, flag\_im)$
14: **end if**
15: **end for**
16: **end for**

After extending the variables with security labels, we replace every HDL operation with an IFT-enhanced operation as described earlier. To do so, we examine the node of each assignment statement via in-order traversal. The data flow graph is acquired using *Yosys Verilog frontend* [Wola] to transform Verilog code to its Abstract Syntax Tree (AST) representation. For

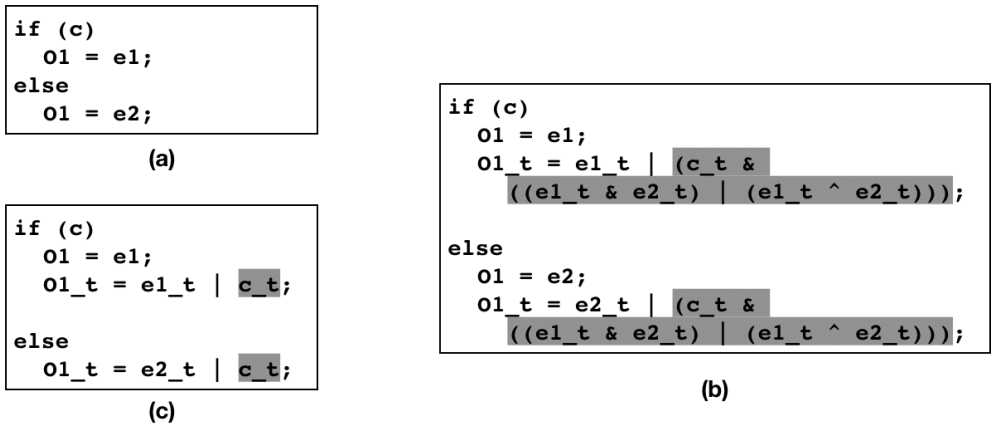


Figure 3.5: Implicit flow tracking. (a) Verilog code. (b) Imprecise IFT-extended Verilog code. (c) Precise IFT-extended Verilog code. Highlighted parts show the tracking logic for implicit flow tracking.

each operation, a module from the available libraries is instantiated (Lines 7-16 in Algorithm 1). This process is shown for a simple code in Fig. 3.4. For sequential circuits modeled as *always blocks* in Verilog language, the same approach is taken by calculating the flow of the right hand side expression outside the *always block* and updating the label of the left hand side variable using the original *always block* structure.

3.4.3 Tracking Implicit Flows

Merely tracking the explicit flow might inaccurately report the absence of flow in conditional statements by ignoring existence of implicit flows. To track these flows, for each assignment we obtain a list of variables which affect the execution of the statement. Having this list, we generate the logic required for tracking the implicit flow as shown by Algorithm 2. This logic can be generated with different levels of precision specified by "$flag\_im$". If we wish to employ a conservative IFT approach, any use of tainted condition should yield a tainted output (Lines 8-10 in Algorithm 2). While this approach captures all possible flows of information, it overestimates the actual flow. For a more precise flow tracking, we need to traverse the control flow graph in

order to figure out what other outcomes are possible for the right hand side of the assignment, assuming the conditions were flipped. Now we can model each conditional statement with a multiplexer and acquire the taint of the output using the high level flow tracking rule for multiplexer (Lines 10-13 in Algorithm 2). To better understand the idea, we analyze implicit flow tracking through a simple code shown in Fig. 3.5(a). The highlighted parts in Fig. 3.5(b) and (c) show the logic added to track the implicit flow while `e1_t` and `e2_t` represent the explicit flows from the right hand side expressions `e1` and `e2`. As it can be seen in Fig. 3.5(b), the imprecise approach marks the output of a conditional statement as tainted if the condition is tainted by employing an OR logic. Taking a precise approach, information flows from the condition to the output only if the tainted condition occurs when both inputs are tainted or they have different Boolean values.

Algorithm 2 Implicit flow tracking logic generation

```
1:  Input: assignment Ai: Vk := Exp(Xi, ··· ,Xj)
2:  Input: flag_im, Verilog code AST file
3:  Output: Verilog expression ImpFlow
4:  Extract Vk_Conds by traversing CFG of Ai
5:  if (Vk_Conds = Null) then
6:     ImpFlow := 0
7:  else
8:     if (flag_im = Imprecise) then
9:        ImpFlow := ImpFlow | ci_t; ∀ ci ∈ Vk_Conds
10:    else
11:       Extract OtherInps by traversing the CFG
12:       ImpFlow := Mux_IFT(Vk_Conds, OtherInps)
13:    end if
14: end if
15: return ImpFlow;
```

3.5 Experimental Results

We have used RTLIFT to analyze security properties on several benchmarks, and here we compare RTLIFT and GLIFT in terms of required time for verifying security policies and the

precision of the generated IFT logic.

3.5.1 Security Proofs

Cryptographic Cores: Table 3.1 shows the required time for proving IFT properties on cryptographic cores. As depicted in Fig. 3.2, we have used RTLIFT to generate the IFT logic for the design under test, which is then given to Quetsa Formal Verification tool. To inspect if information flows from input X to output Y, we need to set X's label high while all other inputs' labels are low, and observe Y's label which tells of if information can flow from X to Y or if they are isolated from each other. We have proved two properties on 32 bit and 128 bit RSA cores: 1) flow from the secret key to the cipher text and 2) flow from secret key to "ready" signal. While the former is expected as it is secured through encryption, the latter reveals an unintended flow. Since the Boolean value of the " ready" signal is not affected by the key value, the detected flow reveals a timing channel. The timeout is set for one hour for this experiments.

Furthermore, we have used the tool to check confidentiality properties on a number of trust-HUB AES benchmarks that contain hardware Trojans which leaks the secret key to an output other that the cipher text. IFT techniques can be used to detect hardware Trojans that cause unintended flows of information [HMOK16b]. Specifically, in a cryptographic core information from the secret key should only flow to the cipher text, and its flow to any other output is undesirable. Hence, we have used the tool to specify if there is a flow from the secret key to any output besides the cipher text. Our method is capable of detecting such hardware Trojans while considerably reducing the verification time compared to GLIFT (taken from reference [HMOK16b]), as reported in Table 3.1.

WISHBONE:IFT can be used to detect timing flows in SoC benchmarks [Obe14]. Here, we have used RTLIFT to inspect timing flows between cores that are connected together via the WISHBONE bus architecture. WISHBONE is a relatively simple protocol developed by the Opencores community [ope], and allows multiple devices to interact with each other by sharing a

bus. The transaction starts by a master core requesting access from a device by asserting its "cyc" signal. If the slave device is idle, access is granted to the master by setting its "ack" signal. We want to indicate if a certain master's "ack" signal is affected by the requests coming from other masters. To test this, we assume one of the master cores, $m1$, to be untrusted by setting its "cyc_t" signal high. Next, we observe "ack_t" signal from one of the trusted masters, $m2$. "ack_t" being high indicates a timing flow since we have not marked data values as tainted and $m1$ requests are affecting the time that $m2$ can start and finish its computation. This timing flow is a threat to system integrity since it can violate the real-time constraint of the master cores.

We generated both the conservative and precise IFT logic for comparison. As discussed throughout the paper, the conservative IFT overestimates the existence of information flow resulting in false positives. Our approach is to start the verification process by the conservative IFT which is smaller in terms of area. If isolation can be proved using the conservative IFT, there is no need to verify the properties on the precise version. However, if flow is detected using the conservative approach, we need to repeat the experiments using the precise IFT to avoid getting false positives.

For the original WISHBONE architecture with round robin arbiter, both conservative and precise IFT indicate existence of flow. Next, we have modified WISHBONE arbiter to enforce timing isolation. In our first model, we have implemented a TDMA arbiter. Here, the conservative IFT can prove timing isolation, eliminating the need to test the precise IFT. In our second model, we have divided the masters to two groups with time multiplexed access between the groups and round robin within each group. In this scenario, the conservative IFT reports existence of flow between the two groups, while using the precise IFT we can prove isolation. This final example shows the importance of precision of IFT logic for reducing false positives.

Table 3.1: Verification Time.

Benchmark	property	RTLIFT	GLIFT
32bit RSA	Key flows to cipher text	02:38	10:08
32bit RSA	Key flows to ready	02:14	10:17
128bit RSA	Key flows to cipher text	9:43	timeout
128bit RSA	Key flows to ready	9:36	timeout
AES_T100	Key leaks to output	01:05	06:48
AES_T1000	Key leaks to output	01:06	6:49
AES_T1100	Key leaks to output	01:07	6:46
AES_T1200	Key leaks to output	01:06	6:50

3.5.2 Precision Analysis

We have compared the precision and complexity of the IFT logic generated by RTLIFT and GLIFT for data path operations addition and multiplication, and control path logic modeled as *case statements* in Verilog language. The precision is measured by comparing the number of tainted outputs during simulation for 2^{20} random input samples. As shown by table 3.2, high level tracking rules result in less tainted flow. False positive percentage is reported as the ratio of the difference in the number of tainted flows to the total number of simulations. The complexity is reported as IFT logic area, which gives a first order estimate on testing and verification time.

We briefly explain how IFT-enhanced addition and multiplication operations are designed for the flow tracking library which is given to RTLIFT as an input file. First we have designed a full adder which receives three inputs `A`, `B` and `Cin` along with their labels `A_t`, `B_t` and `Cin_t` and generates outputs `Sum` and `Co` along with their labels `Sum_t` and `Co_t`. To find Boolean expression describing `Sum_t` and `Co_t`, we need to consider Boolean expressions of `Sum` and `Co` and find the circumstances under which the output can be flipped. Based on the Boolean equation `Sum = A` $\oplus$ `B` $\oplus$ `Cin`, the output *Sum* is tainted when any of the inputs are tainted since each input to an XOR operation can control the output. The `Co` output is high when more than two inputs are high. Hence the value of `Co` can be changed if we have control over more than one of the inputs, or we have control over only one input but the other two inputs are not equal. Next, we have employed the IF-enhanced full adder to design a ripple carry adder, and then an IFT-enhance multiplier is

Table 3.2: Precision & complexity of RTLIFT vs. GLIFT.

Operation	RTLIFT		GLIFT		
	#tainted flows	Area	#tainted flows	Area	%FP
8-bit adder	8477103	271	8535900	222	5.6%
16-bit adder	16441632	603	16524855	556	7.9 %
32-bit adder	32385907	1215	32549597	1243	15.6%
8-bit multiplier	15029971	847	15310281	1759	26.7%
16-bit multiplier	31816947	2078	32200870?	7647	36.6%
4-way case	849874	70	883810	54	3.2%
8-way case	869915	226	958070	129	8.4%
16-way case	869799	199	997874	289	12.2%

built from the adder. As it can be seen from table 3.2, generating IFT logic at a higher level of abstraction can reduce false positives rate for both data path and control path unit.

3.6 Conclusion

This paper presents RTLIFT for precisely measuring all digital flows through RTL designs in order to formally prove security properties related to integrity, confidentiality and logic side channels. RTLIFT can be directly applied on HDL codes and easily integrated into the hardware design flow through automated IFT logic augmentation. Furthermore, it enables designers to tradeoff between the complexity and precision of the IFT logic for data path elements and control path logic separately allowing for fast property-specific verification. Experimental results show that generating the IFT logic at a higher level of abstraction can increase the IFT precision and improve the performance of security verification.

Chapter 3, in full, is a reprint of the material as it appears in the Proceedings of the 20th Annual Design, Automation and Test in Europe Conference Exhibition (DATE), March 2017. Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Timing Flows in Hardware Designs

Emergence of side channel security attacks has challenged the classic assumptions regarding what data is publicly available. As demonstrated repeatedly, statistical analysis of information collected by measuring completion time of hardware designs can reveal confidential information. Even though timing-based side channel leakage can be easily exploited to breach data privacy, conventional hardware verification tools are not yet suited to assess these vulnerabilities. To acquaint the hardware design process with formal security evaluations, we introduce a model for tracking timing-based information flows through HDL codes. Based on this model, we have developed Clepsydra, a tool for automatically generating circuitry for tracking timing flows and generic logical flows within hardware designs in two distinct channels. The circuit generated by Clepsydra can be analyzed by EDA tools to detect timing leakage or formally prove constant execution time. We present proofs regarding soundness and precision of the proposed model along with results of employing Clepsydra to verify security properties on a variety of hardware units including crypto cores, bus architectures, caches and arithmetic modules.

4.1 Introduction

Variations in the time taken by a computational unit to generate results form a leakage channel that carries information regarding the data being processed. Many implementations of cryptographic algorithms are shown to have a varying runtime based on the Boolean value of secret key. Thus, an attacker familiar with the underlying algorithm can leverage statistical methods to extract the key from timing measurements [Koc, Sch00]. In security critical applications, timing based attacks have targeted various hardware units such as caches [Ber05, Per05b], shared buses [Hu92, OHI+11a] and floating point arithmetic units [AS07, AKM+15]. Being both inexpensive and pervasive, timing-based side channel attacks are attracting more attention. They can be launched at low cost since the attacker merely needs to measure the execution time of the victim process without physical access to the design. Moreover, any application encompassing data-dependent optimizations is susceptible to such attacks.

Most methods to protect against hardware based timing leakage rely on manual inspection of the HDL code, e.g., looking for sources of timing variation such as branches conditioned on secret values or data-dependent requests sent to shared resources. This can be a lengthy and cumbersome task, and it provides no formal guarantee regarding the design's security. Furthermore, such analysis only inspects the design with respect to already-known attack vectors and falls short in providing resilience against all possible timing based threats. Exhaustively testing the design to capture timing variations is also becoming impractical due to scale of modern chips. As the complexity and prevalence of hardware designs grow, so does the need for automatic and formal analysis of security properties. Over the past decade, multiple research solutions have been proposed for incorporating security analysis into traditional hardware verification tools. Many of the proposed techniques [ZWSM15, JM12, AHMK, TWM+09] enable the designers to verify security properties regarding confidentiality, integrity, and non-interference based on the notion of information flow tracking (IFT).

By modeling how labeled data moves through a system, IFT tools indicate if sensitive information flows to any part of the design. However, this indication is limited to a binary decision as the nature of detected flows are not specified. More specifically, IFT techniques cannot segregate functional flows from timing based ones. This could be problematic in many security applications where functional flow is expected as it is protected by encryption, while timing flows shall be eliminated. For example, in a crypto core, functional flow from the key to the ciphertext is expected while the designer needs to ensure that the time taken for the ciphertext to become available does not depend on the value of the secret key. Furthermore, it might be the case that the output is not directly accessible by untrusted parties but its timing footprint is public. For instance, completion time of a shared floating point arithmetic unit can reveal information regarding its input values, even if the output itself is hidden. Thus, to assess the security of the design with respect to side channel attacks, timing flows should be distinguishable from functional ones.

In this work, we show how timing flows can be precisely modeled, and introduce an IFT technique for capturing timing leakage of hardware designs. Our model is based on detecting and propagating sources of potential timing variations in the code by inspecting interfaces of design's registers. We introduce Clepsydra, which automatically generates the logic required for tracking timing flows and logical flows in arbitrary HDL codes. The logic generated by Clepsydra can be processed by conventional EDA tools in order to analyze timing properties of the design under test. As this logic is generated after statically analyzing all the execution paths in the design, it does not rely on activating the worst case execution path of the design in order to expose timing variation during verification.

Clepsydra is easily adoptable in the hardware design flow. It does not employ any additional HDL language features. And while it generates synthesizable logic that could be used for runtime detection, we envision its usage primarily during design time, when the Clepsydra logic is analyzed by EDA tools to verify the existence (or lack thereof) of timing-based properties.

The Clepsydra logic is only used for analysis and is discarded before manufacturing, thus it imposes no additional runtime overhead. We show how to use Clepsydra to detect timing leakage in various existing architectures, or prove that they have constant execution time. More specifically, this paper provides the following contributions:

- Modeling timing-based information flows in hardware designs;
- Developing Clepsydra for automatic generation of digital logic for testing timing behaviour of hardware designs;
- Analyzing timing-based security properties of various hardware architectures using Clepsydra.

The rest of this paper is organized as follows. Section 4.2 summarizes how IFT-based methods are employed for hardware security analysis, and how we aim to improve them. Our proposed model for tracking timing flows is introduced in Sections 4.3. In Section 4.4 we elaborate implementation of Clepsydra, and present the experimental results gathered from using Clepsydra to analyze security properties of various architectures in Section 6.6. We provide a brief summary of related work in Section 6.2, and conclude the work in Section 4.7.

4.2 Background & Motivation

In this section we elaborate how IFT-based techniques enable security analysis, and argue why current tools are inadequate for establishing formal guarantees of timing based properties. We further point out how employing a more meticulous model for segregating different forms of logical flows can resolve this issue.

4.2.1 Security Properties

Isolation of different logical components is a primary security property that hardware designers seek to provide. Two major security properties can be enforced through isolation:

- **Confidentiality**: Preventing untrusted parties from observing secret information by isolating the units which process secret data. For example, in a cryptographic hardware we want to ensure that the secret key does not leak to public outputs as a result of design flaws, hardware Trojans or side channel leakage.

- **Integrity**: Preventing unauthorized parties from modifying sensitive information. For instance, the registers storing cryptographic keys should only be accessible by trusted sources.

In order to provide sound security guarantees, information flow must be analyzed through both data channels (also known as functional channels) and timing channels. The former ensures that data does not move among isolated components, while the latter certifies that the timing footprints of the isolated entities do not form a communication channel.

4.2.2 IFT & Hardware Security Verification

IFT techniques provide a systematic approach for verifying security properties related to integrity and confidentiality. This works by assigning security labels to different signals and tracking how these labels propagate through the system. Different security properties can be tested by defining the input labels and inspecting the output labels. Precision of an IFT technique, i.e. how closely the reported flows resemble the actual flows, is directly affected by the label propagation rules.

If the label propagation rules are not comprehensive enough to capture all forms of digital flows, the design might be inaccurately marked as secure. Information can flow in both explicit

and implicit ways. In the explicit form, information flows to the output of an operation which is processing sensitive data. More subtly, data that controls conditional statements can implicitly affects the results. For instance, in the code `if(c) then x = y + z;` signal `x` is explicitly affected by signals `y` and `z`, and implicitly by signal `c`. For an IFT technique to be sound and free from false negatives, it should be capable of tracking both implicit and explicit flows.

Furthermore, label propagation rules should detect cases where flow of information is blocked. For example, if certain bits of the secret data is ANDed with zero, there will be no flow from those bits to the output of the AND gate. However, a conservative tracking rule, which assigns the highest security labels of the inputs to the output, marks all the output bits as sensitive. In contrast, a precise IFT tool, built upon stricter rules which take into account Boolean values of the operands and the operation functionality, can recognize absence of flows and avoid certain false positives [HBA+16].

4.2.3 Isolating Timing Flows

Existing IFT techniques track both *functional flows* and *timing flows* using the same set of labels and propagation rules. Thus, when a flow is detected, whether it is a functional flow or a timing flow, remains unknown. However, different applications necessitate different forms of isolation. For instance, both timing and functional isolation should be guaranteed when a cache is shared among mutually untrusting processes. But secure implementation of a cryptographic algorithm only requires elimination of timing channels as functional flows are protected by encryption. This property cannot be tested using IFT techniques which capture all forms of logical flows through a single set of labels. As the cipher is always affected by the secret key through functional flows, its security label will be raised to the security label of the key, independent of existence of timing flows. This significantly limits employment of IFT techniques for security analysis as similar scenarios happen in many applications where functional flows are inevitable but timing based flows should be eliminated.

Since conventional IFT techniques are designed for tracking all forms of logical flows, employing them to detect only timing flows results in a considerable number of false positives. As timing flows are a subset of information flows; a set of stricter propagation rules can be designed that work on a separate set of labels and track only timing flows while ignoring functional ones. In the next section we introduce a set of rules for detecting only timing flows and tracking them through the system.

4.3 Modeling Timing Flows

Timing flows exist from inputs to outputs of a circuit if the time that is taken for the outputs to become available depends on the Boolean values of the inputs. These flows can be exploited if the input signals causing them contain secret information, and the completion time of the unit can be measured by an untrusted party. For instance, consider a division unit, as shown in Fig 4.1, implemented via consecutive subtraction of the divisor from the dividend. The execution time of this algorithm depends on the input values as the number of subtractions is not fixed. This indicates that even if the Boolean value of the quotient is undisclosed, evaluating the execution time reveals information regarding the inputs. In this section we discuss how timing variations are represented in digital circuits, and develop a formal model for capturing them.

4.3.1 Characterizing Timing Flows

Completion time of a design is defined by the time when its output is updated to its final value. If no timing flow exists from input `X` to output `Y`, the time taken for `Y` to reach its final value should be constant as `X` changes. Thus, in order to detect timing flows, we need to determine whether or not the updates made to the outputs occur at constant timesteps. This can be addressed by detecting variations in the update time of all design variables, and tracking them to the final outputs. We discuss how this can be done for any arbitrary digital circuit by answering three

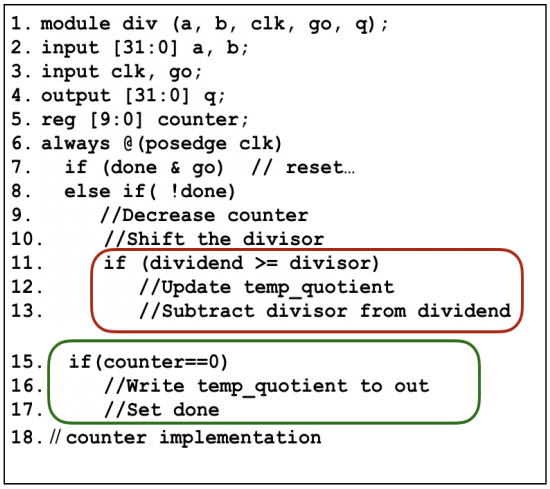


Figure 4.1: Division algorithm based consecutive subtraction. The red box shows generation of timing variations, and the green box depicts their blockage.

questions: How are timing flows generated from a set of sensitive inputs? How does the flow propagate once generated? And lastly, what are the necessary conditions for blocking the flow of timing information and enforcing constant time execution? Since we are interested in detecting timing variations in terms of clock cycles, we need to analyze the design's registers and the signals which control them.

Generation of Timing Variation: Design's registers are written to by a set of data signals which are multiplexed by controllers at each cycle. Considering a register where none of its data or control signals has timing variation but might contain sensitive data, we want to figure out the circumstances under which timing variations occur at the register's output. If the register is definitely updated at each clock cycle, there will be no timing variations. However, if occurrence of updates are tentative, i.e. there is a degree of freedom for the register to hold its current value or get a new value, timing variation could occur. If the controller signal which is deciding the occurrence of the update is sensitive, the resulting timing variation will contain sensitive information as well.

Going back to the division example in Fig 4.1, the updates made to the register `temp_quotient` are conditioned on the input. Thus, based on the Boolean values of the input signals, this register

might get its final value at different times. In Theorems 4.3.2 and 4.3.2, we show that detecting conditional updates caused by sensitive data soundly captures all timing flows while discarding functional-only ones.

Propagation and Blockage of Timing Variation: If any of the data or control signals of a register has cycle level variations, the variation can flow through the register. While simply propagating these flows soundly exposes all timing variations, it overestimates the flow when mitigation techniques are implemented to eliminate the variations. In other words, we need to be able to detect situations where timing variations are not observable at a register's output even though they are present at its input.

In division example in Fig 4.1, instead of directly writing the `temp_quotient` to the output, a wait period is taken before updating the output value. If the wait period is longer than the worst case execution time, the output gets its update at constant time steps. We will show in Theorem 4.3.2 that if there exists a non-sensitive control signal which *fully controls* the occurrence of updates to a register, it can block flow of timing variation from input to output of the register. *Fully controlling* control signal implies that the register gets a new value if and only if the controller gets a new value. Thus, the timing signature of the register output is identical to the control signal (with a single cycle delay), and is independent of its input. Implementing this policy reduces the number of false positives to some extent without imposing any false negative. In the division example, if the counter is large enough, the output value changes immediately after the `done` condition is updated, and keeps its old value while `done` does not change. Hence, all the variations to the final output are controlled by the `done` signal which is non-sensitive, indicating constant execution time with respect to inputs.

4.3.2 Theorems & Proofs

In the rest of this section we formally define IFT concepts and prove the claims we made earlier. We show that our model soundly discovers all potential timing channels by proving that

detecting tentative updates of design's registers is adequate for exposing all timing variations, and the presence of non-sensitive fully controlling signals eliminates existing timing variations. We also prove that our model ignores functional-only flows and thus is more precise for analyzing timing-based properties compared to IFT techniques which capture all logical flows. An **event** e over data set Y and time values T is shown as the tuple $e = (y,t)$ for $y \in Y$ and $t \in T$, where y and t can be retrieved by functions $val(e)$ and $time(e)$. If y is an $n-$dimensional vector, and t the number of clock ticks that has past, the inputs or outputs of a design with n ports can be represented by event e. **Trace** $A(Y,n)$ represents n events $\{e_i\}_{i=1}^{n}$ over data set Y, which are ordered by time: $time(e_i) = time(e_{i+1}) + 1$. For any trace $A(Y,n)$, its **distinct trace** d(A) is defined as the the longest sub-trace of A, where consecutive events have different values, and for any two consecutive events in A such that $val(e_i) \neq val(e_{i-1})$, e_i is in $d(A)$. For example, for trace $A = \{(10,1), (10,2), (20,3), (20,4)\}$, its distinct trace is $d(A) = \{(10,1), (20,3)\}$ since the values only change at clock cycle 1 and 3. Traces $A(X,k)$ and $A'(X,k)$ are **value preserving** with respect to set I if the only difference between their corresponding events e_i and e'_i is in the j-th element of the value vector such that $j \in I$. In IFT analysis, we are interested in the effects of a set of sensitive variables by testing the design with respect to input traces which only differ in the sensitive inputs. This idea can be modeled by using value preserving traces where I is the set of sensitive inputs.

Output of an FSM F is **completely controlled** by input J if the FSM output is updated if and only if input J is updated.

For any set of wires W, **sensitivity label set** W_s and **timing label set** W_t indicate if W carries sensitive information or timing variation, respectively.

In a sequential circuit represented by the FSM $F = (X,\ Y,\ S,\ s_0,\ \delta,\ \alpha)$, a **functional-only flow** from a set of sensitive inputs I exists if there exist two value preserving (with respect to I) input traces $A(X,k)$ and $A'(X,k)$ such that when fed to the FSM, the timesteps of the distinct traces of the outputs are equivalent, while the values of corresponding events varies. Stated

formally: if $B = \alpha(A, s_0)$ and $B' = \alpha(A', s_0)$, then:

$$\forall e_i, e_i' \in d(B), d(B')\ time(e_i) = time(e_i') \text{ and}$$

$$\exists e_j, e_j' \in d(B), \in d(B') \text{ such that } val(e_j) \neq val(e_j')$$

In a sequential circuit represented by the FSM $F = (X,\ Y,\ S,\ s_0,\ \delta,\ \alpha)$ a **timing flow** from a set of sensitive inputs I exists if there exist two value preserving (with respect to I) input traces $A(X,k)$ and $A'(X,k)$ such that when fed to the FSM, the timestep of the distinct traces of the outputs are not equivalent. Stated formally, if $B = \alpha(A, s_0)$ and $B' = \alpha(A', s_0)$, then:

$$\exists e_j, e_j' \in d(B), \in d(B') \text{ such that } time(e_j) \neq time(e_j')$$

For a combinational logic function $f : X \to Y$ its **flow tracking** function $f_s : X \times X_s \to Y_s$ determines whether or not sensitive inputs affect the outputs. If $f(x_1,...,x_n) = (y_1,...,y_m)$ then $f_s(x_1,...,x_n,x_{1_s},...,x_{n_s}) = (y_{1_s},...,y_{m_s})$, where if set of sensitive inputs $\{x_j | x_{j_s} = 1\}$ can affect value of y_i then $y_{i_s} = 1$ indicating **information flow** exists from the sensitive inputs to output y_i.

For a sequential logic function $f : X \times S \to Y$, where X, S, Y are the inputs, states, and the outputs, the **time tracking** function $f_t : X \times X_s \times X_t \times S \times S_s \times S_t \to Y_t$ determines if a set of inputs tainted with sensitive information or timing variation can affect timing variations of the output. If $f(x_1,x_2,...,x_n,s_1,s_2,...,s_l) = (y_1,y_2,...,y_m)$ then f_t $(x_1,...,\ x_n,\ x_{1_s},...,\ x_{n_s},\ x_{1_t},...,\ x_{n_t},s_1,...,\ s_l,\ s_{1_s},\ ...,\ s_{l_s},\ s_{1_t},...,\ s_{l_t}) = (y_{1_t},...,\ y_{m_t})$, where if a set of tainted inputs $\{x_j | x_{j_s} \vee x_{j_t} = 1\}$ can affect whether or not state s_i is updated then $s_{i_t} = 1$ and we say **timing flow** exists from the tainted inputs to output s_i.

The time tracking logic F_t of FSM F captures timing flows of the FSM.

Proof. To prove this theorem we show that the existence of a timing flow reduces to variations in

occurrence of updates to the output, and therefore is captured by F_t.

If a timing flow exists with respect to the set of tainted inputs I, based on Definition 4.3.2 there exist value preserving traces $A(X,k)$,$A'(X,k)$ such that :

$$\text{if } B = \alpha(A),\ d(B) = (e_1, e_2, ..., e_m)$$

$$\text{and } B' = \alpha(A'), d(B') = (e'_1, e'_2, ..., e'_m) \text{ then :}$$

$$\exists j \in [1:m] \text{ such that } time(e_j) \neq time(e'_j)$$

Consider n to be the smallest index such that $time(e_n) \neq time(e'_n)$. Without loss of generality we can assume that $time(e_n) = t_n$ and $time(e'_n) = t_n + d,\ d > 0$. Basically, we are assuming n to be the time when the new value of trace B' appears with delay d compared to trace B. We can write the elements of these two traces up to the nth element:

$$d(B) = (v_1,t_1),\ (v_2,t_2),\ ...,\ (v_{n-1},t_{n-1}),\ (v_n,t_n)$$

$$d(B') = (v'_1,t_1),\ (v'_2,t_2),\ ...,\ (v'_{n-1},t_{n-1}),\ (v'_n,t_n+d)$$

$$\forall i \in [2:n]:\ v_i \neq v_{i-1} and\ v'_i \neq v'_{i-1} \text{ Based on Definition4.3.2.}$$

The following observations can be made based on the above traces:

$$(a) : (v_n,t_n) \text{ and } (v_{n-1},t_n-1) \in B$$

$$(b) : (v'_{n-1},t_n) \text{ and } (v'_{n-1},t_n-1) \in B'$$

From (a) we can infer that value of trace B is updated at time t_n from v_{n-1} to v_n while equation (b) shows that value of trace B' is not updated at time t_n and is equal to v'_{n-1}

By Definition 4.3.2, all input events remain the same $\forall i \notin I$, meaning that the only

difference between them is the sensitive inputs. Thus, the difference in the update to the output is caused by the set of sensitive inputs and is captured by F_t based on Definition 4.3.2. □

The time tracking logic F_t of FSM F does not capture functional-only flows of the FSM.

Proof. We prove this theorem by showing that the existence of functional only flows will not impose any variations on the occurrence of updates to the output, and thus will not be captured by F_t.

If a functional-only flow exists with respect to the set of sensitive inputs I, then based on Definition 4.3.2 there exists value preserving traces $A(X,k)$,$A'(X,k)$ such that:

$$\text{if } B = \alpha(A), d(B) = (e_1, e_2, ..., e_m) \text{ and}$$

$$B' = \alpha(A'), d(B') = (e'_1, e'_2, ..., e'_m) \text{ then :}$$

$$(1)\ \forall i \in [1:m]:\ time(e_i) = time(e'_i)$$

$$(2)\ \exists j \in [1:m]:\ \text{such that } val(e_j) \neq val(e'_j)$$

We claim that there is no time t_n such that the value of one of the traces is updated while the other one is not. Without loss of generality, we show that there is no time t_n where the value of B is updated but the value of B' remains the same. We prove this via proof by contradiction.

Contradiction hypothesis: At time t_n, trace B is updated while trace B' holds its value.

Let v_n and v_{n-1} be the values of trace B at times t_n and $t_n - 1$ respectively. Based on the contradiction hypothesis, (v_n, t_n) is an event in B. Hence, $d(B)$ contains an event e_i such that $time(e_i) = t_n$. Let us assume that e_i is the ith element of $d(B)$. Similarly, assume values of trace B' in times t_n and $t_n - 1$ are v'_n and v'_{n-1}. Based on the contradiction hypothesis we know (v'_n, t_n) is not an event in $d(B)$ since B is not updated at this time. Thus, $d(B')$ does not have any event e'_j

which timestep is t_n. Hence, if we pick the ith element of $d(B')$, called e'_i, then $time(e'_i) \neq t_n$. So:

$$\exists i \; in[1:m] \text{ such that } time(e_i) \neq time(e'_i)$$

This is contradictory to the definition of functional-only flows since there could be no time t_n where the values of one of the traces is updated while the other one is not. Based on Definition 4.3.2 this is not captured by FSM_t. □

If FSM F is completely controlled by input J such that $J \notin I$, then no timing variation is observable at the output of FSM F as a result of processing traces which are value preserving with respect to set I.

Proof. We will prove this theorem via proof by contradiction.

Contradiction hypothesis: there exist value preserving (with respect to I) traces $A(X,k)$ and $A'(X,k)$ which impose timing flow at the output of FSM F which is completely controlled by input $J \notin I$.

Based on Definition 4.3.2:

$$\text{let } B = \alpha(A, s_0), \text{ and } d(B) = \{e_1, e_2, ..., e_m\}$$

$$\text{let } B' = \alpha(A', s_0), \text{ and } d(B') = \{e'_1, e'_2, ..., e'_m\}$$

$$\exists i \in [1,m] \text{ such that } time(e_i) \neq time(e'_i)$$

let n be the smallest index in the above equation such that $time(e_i) \neq time(e'_i)$. Without loss of generality we can assume $time(e_i) = t_n$ and $time(e'_i) = t_n + d$. We can write elements of $d(B)$ and $d(B')$ up to the nth element:

$$d(B) = (v_1,t_1),(v_2,t_2),...,(v_{n-1},t_{n-1}),(v_n,t_n)$$

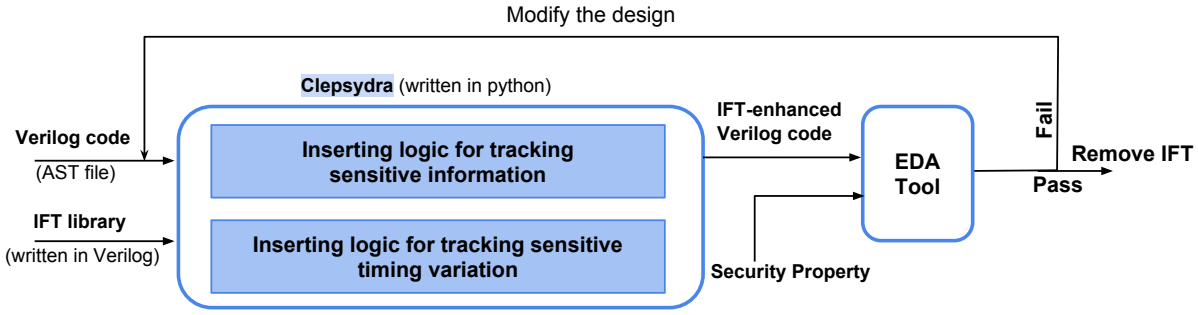


Figure 4.2: Clepsydra overview

$$d(B') = (v'_1, t_1), (v'_2, t_2), ..., (v'_{n-1}, t_{n-1}), (v'_n, t_n + d)$$

Using Definition 4.3.2, we make the following observations:

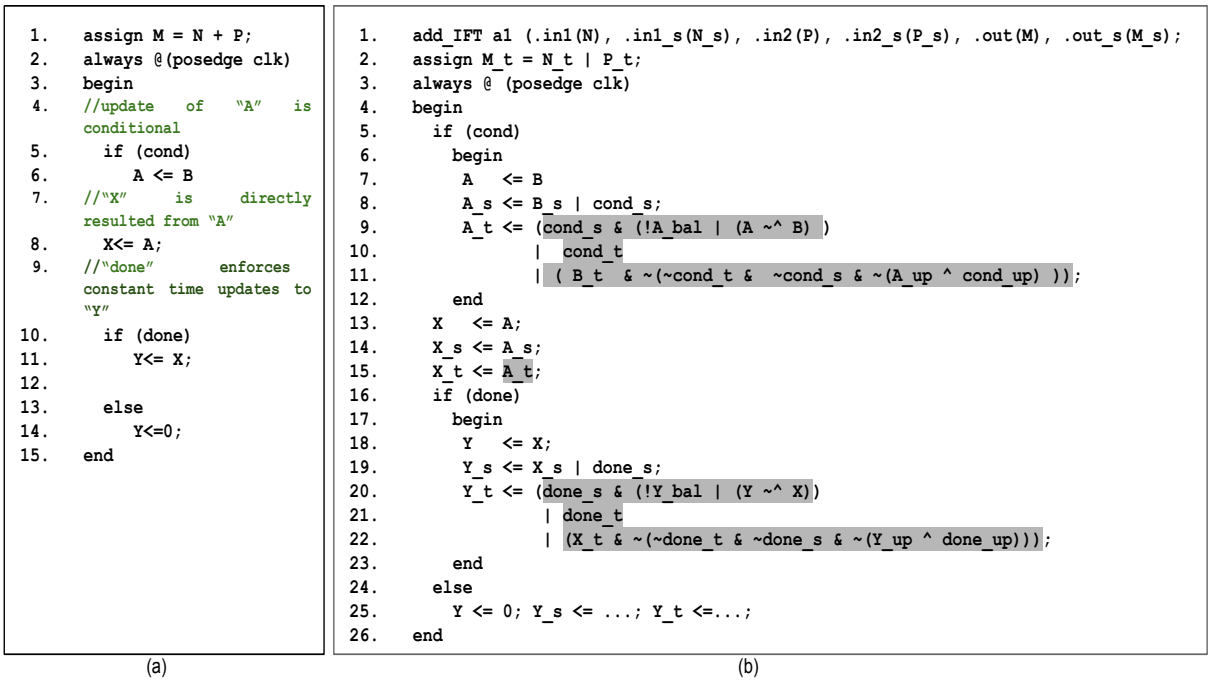
$$(a) : (v_n, t_n) \text{ and } (v_{n-1}, t_n - 1) \in B$$

$$(b) : (v'_{n-1}, t_n) \text{ and } (v'_{n-1}, t_n - 1) \in B'$$

The above equations specifies that trace B has been updated at time t_n while B' is not updated. Lets denote the sub-trace of the fully controlling input J from traces A and A' with j and j', respectively. Based on Definition 4.3.2, equation (a) indicates that input j is updated at time t_n, and from (b) we know input j' is not updated at this time. This is a contradiction as the only difference between input traces A and A' are with respect to set I, and since $J \notin I$ then j and j' should be identical. □

4.4 Clepsydra Implementation

In this section, we describe implementation details of Clepsydra. As shown in Fig 6.1, the input to Clepsydra is a hardware design described by its abstract syntax tree (AST) which is obtained by parsing its HDL representation. As output, Clepsydra generates a synthesizable



```
1.  assign M = N + P;
2.  always @(posedge clk)
3.  begin
4.  //update of "A" is
    conditional
5.    if (cond)
6.       A <= B
7.  //"X" is directly
    resulted from "A"
8.    X<= A;
9.  //"done" enforces
    constant time updates to
    "Y"
10.   if (done)
11.      Y<= X;
12.
13.   else
14.      Y<=0;
15. end
```

(a)

```
1.  add_IFT a1 (.in1(N), .in1_s(N_s), .in2(P), .in2_s(P_s), .out(M), .out_s(M_s);
2.  assign M_t = N_t | P_t;
3.  always @ (posedge clk)
4.  begin
5.    if (cond)
6.      begin
7.       A   <= B
8.       A_s <= B_s | cond_s;
9.       A_t <= (cond_s & (!A_bal | (A ~^ B) )
10.             |  cond_t
11.             | ( B_t  & ~(~cond_t &  ~cond_s & ~(A_up ^ cond_up) ));
12.     end
13.   X   <= A;
14.   X_s <= A_s;
15.   X_t <= A_t;
16.   if (done)
17.     begin
18.      Y   <= X;
19.      Y_s <= X_s | done_s;
20.      Y_t <= (done_s & (!Y_bal | (Y ~^ X))
21.              | done_t
22.              | (X_t & ~(~done_t & ~done_s & ~(Y_up ^ done_up)));
23.     end
24.   else
25.     Y <= 0; Y_s <= ...; Y_t <=...;
26. end
```

(b)

Figure 4.3: (a) Original Verilog code, (b) IFT-enhanced Verilog code generated by Clepsydra

Verilog code which has all the functionalities specified in the original design, alongside the complementary logic for propagating both timing based and generic information flows from design inputs to its outputs. The tracking logic is realized in two steps: 1) extending each variable in the design with labels *sensitivity level* and *timing level* which indicate if the variable carries sensitive information or timing variation, respectively; and 2) inserting logic for updating these labels as their corresponding variables change. The code generated by Clepsydra is then given to EDA tools for security analysis. Security properties are assessed by specifying labels of the input variables and observing the output labels after simulation, formal verification or emulation. If the output labels comply with the designers' intention, the tracking logic is discarded and the original design can be used for fabrication. In case of violating the security properties, the original design should be modified, fed to Clepsydra, and retested.

Clepsydra enables analyzing timing behavior of a design with respect to any arbitrary subset of its inputs which are marked as sensitive. This facilitates modeling a variety of security

properties. For example, constant time execution can be tested by marking all the inputs as sensitive. But in many scenarios we are only interested in constant execution time with respect to certain inputs. For instance, when a cache is shared between mutually untrusting processes, timing variations caused by accesses from sensitive data is exploitable. However, variations due to cache conflicts on non-sensitive data are not valuable to the adversary. indiscriminately eliminating all timing variations results in disabling the cache as a whole. Moreover, many mitigation techniques are based on randomizing timing variations. To differentiate benign variations from sensitive ones, we should inspect the source of the variations. This is done by tracking sensitive data throughout the circuit, and extracting the sensitive timing variations from them.

Sensitive information affects computation result through both the data path and the control path, creating explicit and implicit flows. To detect explicit flows, Clepsydra replaces each data path operation with an IFT-enhanced version of it which is available as a Verilog module in a predesigned IFT library (lines 21-23 of Algorithm 3). Each IFT-enhanced operation receives the original inputs of the operation along with their sensitivity labels, and computes the outputs of the operation as well as their sensitivity labels. A simple example of replacing an add operation with an IFT-enhanced module is shown in the first line of Fig 4.3. Various complexity-precision trade-offs for the tracking logic can be explored by modifying the label propagation rules of the IFT-enhanced modules [HBA$^+$16].

To track whether or not an assignment is implicitly affected by sensitive data, we need to figure out if its execution depends on any sensitive variable. To do so, Clepsydra extracts the design's control flow graph from its AST representation, and constructs a list of control signals for each conditional assignment (lines 13-15 of Algorithm 3). Next, based on the variables in the list and their sensitivity labels the logic for tracking the implicit flow is generated and added to the explicit flow tracking logic (lines 30-32 of Algorithm 3).

Algorithm 3 Tracking logic generation

Input:Verilog code AST file
Output: IFT-enhanced Verilog code
Pre-processing:
for each register r **do**
    n := number of the paths to r
    m := number of the controllers of r
    if $n \neq 2^m$ **then**
        r_bal := 0
    else
        r_bal := 1
    end if
end for
for each conditional assignment a **do**
    traverse CFG
    a_con := list of controllers
end for
Logic insertion:
for each variable $x[n:0]$ **do**
    define $x\_s[n:0]$, $x\_t[n:0]$
end for
for each DFG operation $a = b\ op\ c$; **do**
    instantiate IFT-enhanced operation:
    $op\_IFT\ (a,\ a\_s,\ b,\ b\_s,\ c,\ c\_s)$;
    insert time tracking logic:
    $a\_t = b\_t \mid c\_t$;
end for
for each controller c **do**
    insert buffer $c\_buf <= c$
end for
for each conditional assignment $A <= B$ **do**
    insert implicit IFT logic:
    $A\_s <= B\_s \mid c\_s;\ \forall c \in A\_con.$
    $A\_t <= (B\_s\ \&\ !A\_bal)$;
    $\mid (B\_t\ \&\ !(c$ is non-sensitive and fully controlling)
end for

4.4.1 Tracking Timing Flows

Clepsydra inserts logic components at each register interface to detect if any timing variation is generated from sensitive data, and whether or not existing variations from the register input flow to the register's output (lines 33-34 of Algorithm 3). As we proved earlier, the necessary condition for formation of timing variation is existence of a register which update depends on sensitive values. To identify these cases, we need to determine if a register has the flexibility of selecting between getting a new value and holding its current value. To examine this property for each register in the design, Clepsydra statically enumerates all the paths in which the register is written to, and compares it with the total number of paths that the controllers of that register can theoretically activate. If these two numbers are unequal, a bit which indicates the updates to the register are tentative is set (lines 4-11 of Algorithm 3). Such analysis on a Verilog code is relatively easy compared to software languages since multiple writes to a register are modeled as a single multiplexer with `n` data inputs and `m` control inputs. Tentative update scenarios happen if $n \neq 2^m$ which indicates that the multiplexer has direct feedback from its output to its own input. To illustrate this idea, consider the Verilog code written in Fig 4.3(a) and the IFT-enhanced Verilog code generated by Clepsydra in Fig 4.3(b). Highlighted parts in lines 9 and 20 show the logic responsible for detecting generation of timing flows. Values of `A_bal` and `Y_bal` are statically decided by Clepsydra after analyzing the branches in the original code. An XNOR function is also added to detect cases where the register gets its value from a different variable without actually getting updated. Even though such scenarios are rare in actual designs, the logic for detecting them is added to ensure capturing cases where tentative updates are disguised by renaming the variables.

Once generated, timing variations flow directly through the subsequent registers unless special mechanism for eliminating the variations is implemented. Register `X` in Fig 4.3 directly gets its value from register `A`, thus if any timing variation is present as the output of `A`, it will unconditionally flow to `X`. As shown in the second line of Fig 4.3(b), timing variation directly

flows through combinational logic since we are interested in cycle level precision.

As we proved in the previous section if there exist any non-sensitive control signal which fully controls the updates to the register, it can block flow of timing information. To detect existence of fully controlling signals for conditional assignments to registers, Clepsydra inserts XOR gates for comparing occurrence of updates. This logic is shown in lines 11 and 22 of Fig 4.3(b). The XOR function indicates that updates of the output register and its controller are synchronous. And the inverters specify that the controller does not have any sensitive information. This logic is responsible for preventing overestimating the flow to some extent as depicted by the AND function. The logic behind `Y_up`, `A_up`, `B_up` and `done_up` are not shown in the figure for simplicity, but they are computed by XORing the current state with the next state. Since, the updates to the control signals are observable at the register output with one cycle delay, Clepsydra inserts buffers to store control values from the previous cycle in order to compute whether or not they have been updated in the previous cycle (lines 16-17 of Algorithm 3).

4.5 Experimental results

In this section we elaborate how various security properties are specified based on notion of IFT, and verified on Clepsydra logic. Table 4.1 lists the hardware designs we tested along with the assessed security properties. For each design, we briefly discuss the architectural features which create timing channels, the attack model for exploiting them, the existing mitigation techniques, and the results of our security analysis. For all of our experiments, we obtained the AST representation of plain Verilog code by parsing it using Yosis tool [Wolb], and employed Clepsydra to generate tracking logic. On the IFT-enhanced code generated by Clepsydra, effect of input `X` on timing behaviour of output `Y` can be inspected by setting the input signal `X_s` as high, and observing the value of the output `Y_t` after simulation or formal verification.

Table 4.1: summary of the designs and security properties tested using Clepsydra

Design	Security Property	Security Specification	Result
Divider	Result is ready in constant time	set dividend_s, divisor_s=H, assert(quotient_t==L)	Proved
Multiplier	Result is ready in constant time	set multiplier_s, multiplicand_s=H, assert(product_t==L)	Provable after debugging
Cache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	violated
PLcache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	Provable if sensitive data is preloaded
RPcache	Isolation bw accesses to the same line	set index_s=H, assert(data_t==L)	Provable if RNG is secure
WISHBONE, round robin	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Violated
WISHBONE, TDMA	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Proved
WISHBONE, TDMA+group	Timing isolation bw cores	set request1_s=H, assert(ack2_t==L)	Provable bw different groups
AES	Cipher is ready in constant time	set key_s, plaintext_s=H, assert(cipher_t==L)	Proved
RSA	Cipher is ready in constant time	set key_s, plaintext_s=H, assert(cipher_t==L)	Violated

4.5.1 Arithmetic Modules

For the first set of experiments, we sought proving constant time properties of arithmetic units, as variation in completion time of these units can be exploited to extract information regarding the input [AKM+15, AS07]. We tested a fixed point math library from the Opencores website [alu], which is supposed to run in constant time as claimed by its designers. In order to verify this claim, we marked data inputs of each unit as sensitive and observed the timing labels of the outputs.

The multiplication unit is based on accumulating partial products at each cycle. Thus, if the MSB bits of the multiplier are zero the result will be available faster since the partial products in the last cycles are zero. The output `ready` signal of the design is set after a counter reaches zero. After analyzing this design, we noticed that while the `ready` output is free from timing variations, the product result is not. This indicates that the result could potentially become available before the `ready` signal is raised. In order to eliminate this flow, we modified the design by adding a register which blocks the working result to write to the final output before the counter resets. After this modification, we could formally prove that the design runs in constant time using Questa Formal Verification tool. The division unit, similar to the example we had throughout

the paper, is implemented by subsequently subtracting the divisor from the dividend. Similar to the multiplication unit, a wait state is responsible for enforcing constant time updates at the final outputs. This time no timing variation was detected by our analysis as all the output variables, including the result itself, are controlled by the wait state.

This set of experiments showed that our model is capable of isolating different forms of flows, and proving absence of timing flows while functional flows exist. Furthermore, it shows that the generated tracking logic is precise enough to detect cases where timing variations are eliminated by delaying all the updates as long as the worst case scenario.

4.5.2 Cache Implementations

Cache-based side channel attacks have been repeatedly employed to break software implementations of ciphers such as RSA and AES. These attacks target implementations which use pre-computed values that are stored in the cache and accessed based on the value of the secret key. Thus, an attacker who is capable of extracting the cache access pattern of the process running the encryption can deduce information regarding the key. Percival [Per05b] has shown that an adversarial process sharing the cache with the OpenSSL implementation of the RSA cipher can retrieve cache access pattern of the victim process by inducing collisions in the cache. By remotely attacking AES implementation of the OpenSSL protocol, Bernstein [Ber05] showed that timing channels can be exploited even when the cache is not shared with an untrusted process. In his attack, Bernstein exploited the cache collisions between different requests by the victim process itself to reveal the encryption key. While these attacks vary substantially in terms of implementation, they all exploit the timing variations from the cache collisions. Several cache designs have been proposed to bar index value of sensitive accesses to affect the time that it takes for the cache to retrieve data in later cycles. We have used Clepsydra to inspect timing flows in an unsecure cache and two secure architectures, PLcache and RPcache, introduced in [WL07]. To model timing leakage via external interference, we consider two processes with isolated address

spaces sharing the same cache. Marking indexes of accesses made by one process as sensitive, we want to figure out if the data read by the other process contain timing variation. The internal interference scenario is modeled with a single process and inspecting if marking certain indexes as sensitive causes timing variation when the same lines are read with different tags.

PLcache eliminates leakage channel by letting processes to lock their data in the cache and disabling their eviction. Since sensitive data can no longer be evicted, it cannot affect the timing signature of the system. We implemented the PLcache and acquired its IFT-enhanced tracking logic from Clepsydra to test if this partitioning scheme eliminates the flow. Based on our analysis, if data with sensitive indexes is preloaded to the cache and locked, there will be no information leakage as the result of later accesses to the locked lines. However, this result is based on assuming that the preloading stage is not sensitive itself.

Next, we tested the RPcache which randomly permutes the mapping of memory to cache addresses to eliminate any exploitable relation between the collisions. When external interference between untrusting processes are detected, PRcache randomly chooses a cache line for eviction . Thus, the attacker cannot evict the victim's process sensitive information and observe whether or not that causes delay later on. In case of internal interference, collisions are handled by directly sending the data from the colliding access to the processor and randomly evicting another line. Our analysis showed that RPcache eliminates timing variations assuming that the inputs to the random number generator are not sensitive.

4.5.3 Bus Architectures

Another source of timing channel in hardware designs arises when different units are connected over a shared bus. In such scenarios, cores that are supposed to be isolated can covertly communicate by modulating the access patterns to a shared resource and affecting the time when other cores can use the same resource. Using Clepsydra, we have inspected presence of timing flows when WISHBONE interconnect architecture is used to arbiter accesses on an SoC. To

access a shared resource over WISHBONE, the master core sends a `request` signal, and waits for the arbiter to send back an `ack` signal. Timing channel between different cores can be assessed by marking the `request` signal sent by one core as sensitive and observing the timing label of the `ack` signal sent to the other core in later requests. We have tested this scenario for the original WISHBONE arbiter and two modified versions of it.

The original WISHBONE arbiter, implemented by the Opencores community [wis], is based on a round robin algorithm. Our experiments revealed existence of timing channel between the cores connected over this architecture as the grant given to one core depends on the former requests sent to the arbiter. In order to eliminate the channel, we replaced the round robin arbiter with a TDMA scheme and retested the design. In this case, no timing channel is detected as the grants are given based on a counter. In order to improve the efficiency of the arbiter, we tested a more flexible scenario where the cores are divided into two different groups. The cores within the same group are arbitrated based on a round robin algorithm, while the two groups are time multiplexed. This time our experiments showed that the two groups are isolated from each other while timing channel exists between elements of the same group.

4.5.4 Crypto Cores

Lastly, we tested timing variation in hardware implementation of RSA and AES ciphers. Since the ciphertext is computed from the key, functional flow from the key to the cipher is inevitable. Thus, existing IFT tools cannot be leveraged to inspect existence of timing channels which are not intended by the designer. Here, we have tested existence of timing channel in two ciphers from the Trusthub benchmarks [tru]. We have assessed timing flow from the secret key to the ciphertext by marking the key bits as sensitive and observing the timing label of the output. Using Questa Formal Verification tool we could prove that the AES core runs in constant time. However, for the RSA core, timing flow was detected from the secret key to the cipher as a result of insecure implementation of the modular exponentiation step. We have left comparing

timing leakage of different RSA architectures and the effectiveness of the proposed mitigation techniques as our future work.

4.6 Related Work

Over the past decade, multiple tools have been developed for tracking information flows through hardware designs. Tiwari et al [TWM+09] implemented a microprocessor which dynamically tracks information flows through the shadow logic added for each gate. Due to the huge overhead in terms of power, area and performance, gate level information flow tracking (GLIFT) has been used mostly for test and verification during design time [**?**]. Oberg et al [OMSK14b] has proved that inserting shadow logic at the gate level detects all information flows including timing and functional flows.

At a higher level of abstraction, multiple secure HDL languages have been introduced to enable designing provably secure hardware. Caisson [LTO+11a] and Sapper [LKO+14] are both FSM based languages extended with a type system. Using these languages, the designer defines a security label for each variable and restricts information flows by controlling the transactions between the states. SecVerilog [ZWSM15] adds a type system to the Verilog language for representing different security levels. SecVerilog users are required to define a type for each variable in the design based on the property they wish to enforce. SecVerilog type system statically verifies that the user defined types and the IFT property comply. While these tools facilitate secure hardware design, they require redesigning the hardware using a new language. VeriCoq [BM] automatically transfers HDL codes to Coq representation where security properties can be tested on annotated code. While all these methods employ conservative tracking rules by raising each signal's label to the highest label of its predecessor signals, RTLIFT [AHMK] allows for more flexible flow tracking rules. Based on the precision level specified by the user, RTLIFT generates the flow tracking logic for the design under test, which can be used to analyze different security

properties.

As stated earlier, all of the mentioned IFT tools track all logical information flows through a single channel and leave the nature of the detected flows as unspecified. In this work, we presented a method for tracking timing flows and generic information flows in two separate channels in order to enable analysis of a wider range of security properties.

4.7 Conclusion

In this work we presented a model for tracking timing-based information flows in hardware designs. We formally proved that our model soundly captures all timing variations and precisely separates timing flows from logical flows. We introduced an IFT tool, Clepsydra, which automatically generates tracking logic based on the proposed model. Clepsydra facilitates hardware security verification by enhancing plain HDL codes with synthesizable logic on which variety of security properties can be tested using conventional EDA tools. In our experiments, we leveraged Clepsydra to detect timing channels in different architectures, and prove absence of timing leakage when mitigation techniques such as randomization, partitioning or delaying till the worst case execution are implemented.

Chapter 4, in full, is a reprint of the material as it appears in the Proceedings of the 36th Annual International Conference on Computer-Aided Design (ICCAD), November 2017. Armaiti Ardeshiricham, Wei Hu, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Synthesis of Secure Hardware Designs

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.

5.1 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 [KGG+18, LSG+18, CCX+18, ERAG+18], 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. 6.1. VeriSketch employs Information Flow Tracking (IFT) methods to allow the definition and verification of security properties related to non-interference [TWM+09], timing invariance [ZWSM15, AHK], and confidentiality, and it extends counterexample-guided synthesis methods (CEGIS) [SL13] to hardware design.

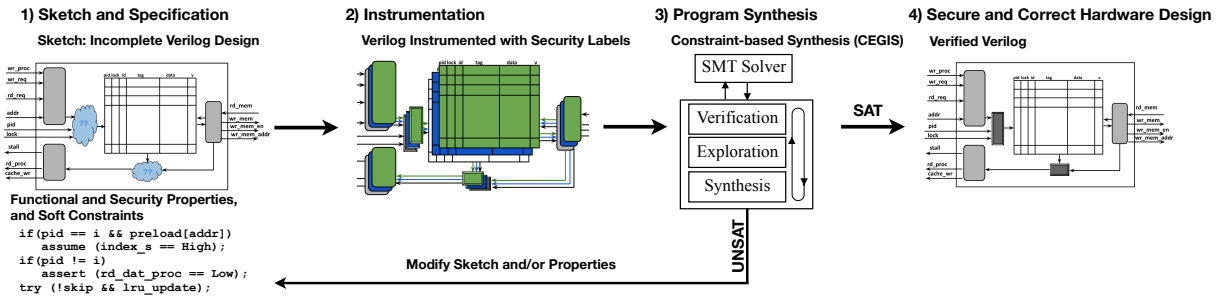


Figure 5.1: VeriSketch accepts as input an incomplete hardware design (i.e., a "sketch") and a set of functional and security properties and soft constraints. VeriSketch leverages hardware information flow tracking and program synthesis to build a Verilog design that satisfies the properties.

VeriSketch uses CEGIS to complete the sketch by breaking the synthesis problem into separate verification and synthesis sub-problems 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 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 5.3 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 5.4. Section 5.5 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 6.6.

5.2 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.

5.2.1 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 [SLTB$^+$06, ABJ$^+$13, SL13] 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.

Program synthesis techniques are commonly used to automate difficult software engineering tasks [SLTB+06, JGST10, GJTV11, PKSL16, FMBD18]. Program synthesis have been employed in different domains such as data processing [SA16, YKDC16], data completion [WDS17, FMVG+17], databases [YWDD17, ZS13], and more recently in security applications [PYIS16, HJRS17]. In the HDL domain, Sketchilog [BNI14, BNI] 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 [FJJ+12, GJJ06, HJR10, SMS13]. 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 [RWK+14, RCK+] and generate abstraction models for SoCs [SHV+17] and ISAs [HSSA16]. Similar techniques have been used at the gate level to automatically modify a netlist when errors are detected late in the design flow [CMB, WYHJ]. 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.

5.2.2 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 [LTO+11a, LKO+14, ZWSM15] or rely on automated label inference rules [TWM+09, AHMK, AHK]. 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 5.4.8, 5.4.11 and 5.4.13). VeriSketch adopts the approach from Clepsydra [AHK] 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.

5.2.3 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.

Threat Model We consider the Percival attack model [Per05b] 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 pre-computed 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 [KGG+18]. 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.

Traditional Secure Hardware Design Flow Assume that the designers decide to implement the partition locked cache (PLCache) mitigation technique [ZWSM15] 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 5.4.17), 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 5.6.3). 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.

5.3 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.

5.3.1 Main Components

Fig. 6.1 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 5.1). 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 5.4) enables reasoning about security properties alongside the functional ones. The instrumented design is given to the synthesis engine (described in Section 5.5) 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.

5.3.2 VeriSketch Language

VeriSketch extends the standard Verilog language [Dim01] with sketch constructs and security property specifications. The formal syntax is shown in Table 5.1.

Sketch Syntax Sketches are language constructs that facilitate writing partial programs [SLTB$^+$06]. VeriSketch enables users to describe a partial hardware design by combining low-level and high-level sketch constructs with the original Verilog syntax. With low-level sketches, the designer can define unknown n-bit constants ($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.

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 "$\{x_1,\ \ldots,\ x_m\} = q_j$"

Table 5.1: VeriSketch Syntax.

	$v \in Vars$	Variable
	$n \in Nums$	Constant
e	::= v — n — **uop** e — e_1 **bop** e_2 — n (??) — **step**? (v, n) — **sel** ($e_1, \ldots, e_m$) — e_1 (**bop**$_1, \ldots,$ **bop**$_m$) e_2 — (**uop**$_1, \ldots,$ **uop**$_m$) e — **comb** ($e_1, \ldots, e_m$) — **seq** ($e_1, \ldots, e_m$)	Expression
a	::= **assign** $v = e$;	Continuous Assignment
s	::=$v = e$; — $v \Leftarrow e$; — **if** (e) s_1 **else** s_2 — **begin** $s_1 \ldots s_m$ **end** — **for** ($v = n_1 : n_2$) s; v ?= e$_0$ [$e_1, \ldots, e_m$]	Procedural Assignment
γ	::= **posedge** clk — **negedge** clk — * — $\vec{v}$	Trigger
B	::= **always** @ (γ) s — a	Block
M	::= $B_1 \ldots B_m$	Module
S	::= $M_1 \ldots M_m$	Sketch
L	::= v_s — v_t	Label
p	::= v — **uop** v — v_1 **bop** v_2 — L — **uop** L — L_1 **bop** L_2 — $v \nrightarrow v$ — $v \nrightarrow_t v$	Property
C	::= **assume** (p) — **assert** (p) — **try** (p)	Spec.

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.

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 informa-

tion 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 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 5.2: 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 5.3.1 (Sketching a Secure Cache)**.** Fig. 5.2 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. 5.2 is the amount of detail provided by the designer and conversely that which is left to be determined by VeriSketch. Fig. 5.2(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. 5.2(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.

5.4 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.

5.4.1 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 5.2. 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 5.4.1. Fig. 5.3 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.

Remark 5.4.2. 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 5.2, to ensure that variables and their

Table 5.2: VeriSketch Label Inference Rules.

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

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

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

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

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

labels are updated simultaneously.

Remark 5.4.3. 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.

5.4.2 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 5.4.4 (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.

```
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 5.3: 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.

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 [HBA$^+$16, AHMK]. 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.

Definition 5.4.5 ($Ctrl(v)$)**.** 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 5.4.6 (c.f. [OMSK14a])**.** Implicit flows via each procedural statement η with l-value

variable v can be conservatively estimated by:

$$\bigsqcup\{c_{i_s} : c_i \in Ctrl(v)\} \tag{5.1}$$

Notation 5.4.7. 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 5.4.8. 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 5.4.9. Examples of tracking explicit flows for combinational and sequential code are shown in lines 2 and 8 of Fig. 5.3. 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]`.

5.4.3 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 [OMSK14a] and can be modeled by capturing how registers can be updated at each clock cycle [AHK]. We describe this in more detail in the following.

Definition 5.4.10 ($Bal(v)$)**.** 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 5.4.11. $Bal(v)$ 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 5.4.12 (c.f. [AHK])**.** 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:

$$\neg Bal(v) \sqcap \bigsqcup \{c_{i_s} : c_i \in Ctrl(v)\} \tag{5.2}$$

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:

$$\neg Bal(v) \sqcap \bigsqcup c_{i_s} \sqcap \neg \bigsqcup (\neg c_{i_s} \sqcap Full(v, c_i)) : c_i \in Ctrl(v) \tag{5.3}$$

Remark 5.4.13. Proposition 5.4.12 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 5.4.14. Examples of tracking timing flows in combinational and sequential blocks are shown in lines 3 and $10-16$ of Fig. 5.3. 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 5.4.15 (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 5.4.16. Throughout the examples, all input labels have a *Low* value if not specified otherwise.

Example 5.4.17 (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);
```

5.4.4 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 5.4.18. 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 5.4.19 (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 $(v_{s_i},\ v_{t_i})$.

5.5 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 $\exists p \forall x_{\mathrm{v}}.\phi\ (p,\ x_{\mathrm{v}})$, 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.

5.5.1 Synthesis with IFT

In order to take advantage of the IFT model within our synthesis flow, we give the parametric design $F(x_{\mathrm{v}},\ p)$ to the IFT unit. This transforms the design to $F_{\mathrm{IFT}}\ (x_{\mathrm{v}},\ x_{\mathrm{v}_{\mathrm{taint}}},\ p)$ where $x_{\mathrm{v}_{\mathrm{taint}}}$ and F_{IFT} are the input's security labels and instrumented design (Definition 5.4.19). The synthesis problem over the instrumented design now includes the labels in addition to the original inputs:

$$\exists p\ \forall x.\Phi(x,\ p)$$

$$\text{where } \Phi(x,\ p) := Q\ (x_{\mathrm{v}},\ F_{\mathrm{IFT}}\ (x_{\mathrm{v}},\ x_{\mathrm{v}_{\mathrm{taint}}},\ p)). \quad (5.4)$$

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 $x_{\mathrm{v}_{\mathrm{taint}}}$. Note that $x_{\mathrm{v}_{\mathrm{taint}}}$ is constrained by the specific security properties we want to enforce.

Example 5.5.1. 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 5.4.16). Thus, all input taints are constrained.

5.5.2 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 5.5.2 (Synthesis Target for Sequential Circuits)**.**

$$\exists p. \bigwedge_{x_j \in \mathrm{CE}} \Phi^*(x_j, p)$$

$$\text{where } \Phi^*(x_j,\ p)\ := \bigwedge_{k \le |x_j|} \mathit{Past}(\Phi(x_j, p),\ k) \quad (5.5)$$

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 $\mathit{Past}(v,k)$, 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 [ES03] can be used; and the framework can be naturally extended with more powerful verification methods.

5.5.3 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 5.4.17 are in place to eliminate side-channel leakage and synthesize the sketch from Fig. 5.2(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 5.5.3 (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 5.5.4. 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. 5.2) 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 5.5.5 (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 5.5.6 (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 $x^m \in x$:

$$\exists a.\ \Phi^*(a,\ p_i) \wedge \bigwedge_{x^j \in x \wedge j \neq m} (a^j = x^j) \tag{5.6}$$

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 5.5.7 (Synthesis Target for Soft Constraints T)**.**

$$\exists p.\ \bigwedge_{x_j \in CE} \Phi^*(x_j,\ p) \wedge \sum_{x_i \in PE} T^*(x_i,\ p) = n$$

$$\text{where } T^*(x_i,\ p) := \bigwedge_{k \leq |x_i|} Past(T(x_i,\ p),\ k) \tag{5.7}$$

The synthesis round iteratively solves Eq.5.7 and decreases n from $|PE|$ to zero if unsatisfiable.

Theorem 5.5.8. 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.5.7 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 4 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. ∀x. C(x) :  Verilog
F (x,p) ← pre-processing (F(x))
F_IFT (x, x_taint, p) ← instrumentation (F(x,p),C(x))
P ← CEGIS (F_IFT (x, x_taint, p), C(x), C'(x))
if P ≠ unsat then
    F_syn(x) ← post-processing (F(x,p),P)
    return F_syn
else
    return unsat
end if
```

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 5.5.9. 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.5.5) 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

Algorithm 5 Counterexample guided inductive synthesis (CEGIS) for synthesizing sequential circuits with soft constraints

1: *Initial Stage:*
2: $p_i \leftarrow$ random assignment
3: $CE \leftarrow \emptyset$
4: $PE \leftarrow \emptyset$
5: **while** 1 **do**
6: *Verification Phase:*
7: $ce \leftarrow$ SAT $(\exists x.\ \neg\Phi(x, p_i))$
8: **if** ce = unsat **then**
9: return p_i
10: **else**
11: $CE \leftarrow CE \cup ce$
12: *Exploration Phase:*
13: $pe_m \leftarrow SAT\ (\exists a.\Phi^*(a,\ p_i) \wedge \bigwedge_{ce^j \in ce\ \wedge\ j \neq m} a^j = ce^j)$
14: $\Phi^*(a,\ p) := \bigwedge_{k \leq |a|} Past(\Phi(a,\ p),\ k)$
15: **if** $pe_m \neq unsat$ **then**
16: $PE \leftarrow PE \cup pe_m$
17: **end if**
18: **end if**
19: *Synthesis Phase:*
20: **for** $(l = |PE|; l \geq 0; i = i - 1)$ **do**
21: $solution \leftarrow$ SAT $\exists p. \bigwedge_{x_j \in CE} \Phi^*(x_j, p) \wedge (sum_{pe} = l)$
22: $sum_{pe} := \sum_{x_j^{'} \in PE} T^*(x_j^{'}, p)$
23: $T^*(x_j^{'}, p) := \bigwedge_{k \leq |x_j^{'}|} Past(T(x_j^{'},\ p),\ k)$
24: $\Phi^*(x_j,\ p) := \bigwedge_{k \leq |x_j|} Past(\Phi(x_j,\ p),\ k)$
25: **if** solution$\neq$unsat **then**
26: **then** $p_i \leftarrow solution$; *break* l=0 **then return** *unsat*
27: **end if**
28: **end for**
29: **end while**

design. 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 4 and 5, respectively.

5.6 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 [WL07]) 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

[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 5.3: 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

fairness between the cores.

Table 5.3 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.

5.6.1 Implementation

As shown in Fig. 6.1 and Algorithm 4 VeriSketch flow consists of an IFT engine and a program synthesis unit. The IFT tool uses the Yosys [**?**] 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 [Dut14], Boolector [BB09], Z3 [DMB08], or CVC4 [BCD$^+$11]) by using Yosys to translate Verilog to SMT-LIB2 representation.

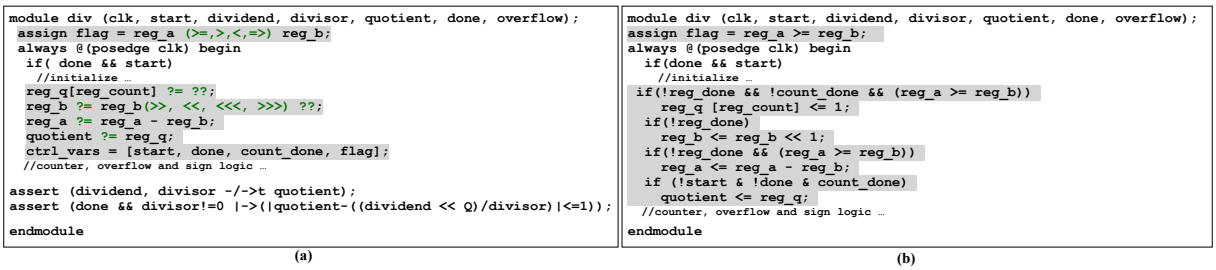


Figure 5.4: Synthesizing a constant time fixed point divider using VeriSketch. (a) Sketch of a shift-and-subtract divider where the structure of the procedural statements, operations, and constant values are left unspecified as shown by the highlighted code. Constant time and functional properties are 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.

5.6.2 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 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 [alu]. The sketch of the divider unit along with the functional and security properties are shown in Fig. 5.4(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. 5.4(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.

```
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 5.5: VeriSketch synthesizes the sketch from Fig. 5.2(a) to a fully specified Verilog design that meets the functional and security properties specified in Example 5.5.4.

Synthesized Designs. The divider unit synthesized by VeriSketch is shown in Fig. 5.4(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. 5.3) 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.

5.6.3 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. 5.2(a) and Example 5.5.4 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. 5.5 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 [Dut14] 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.

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.

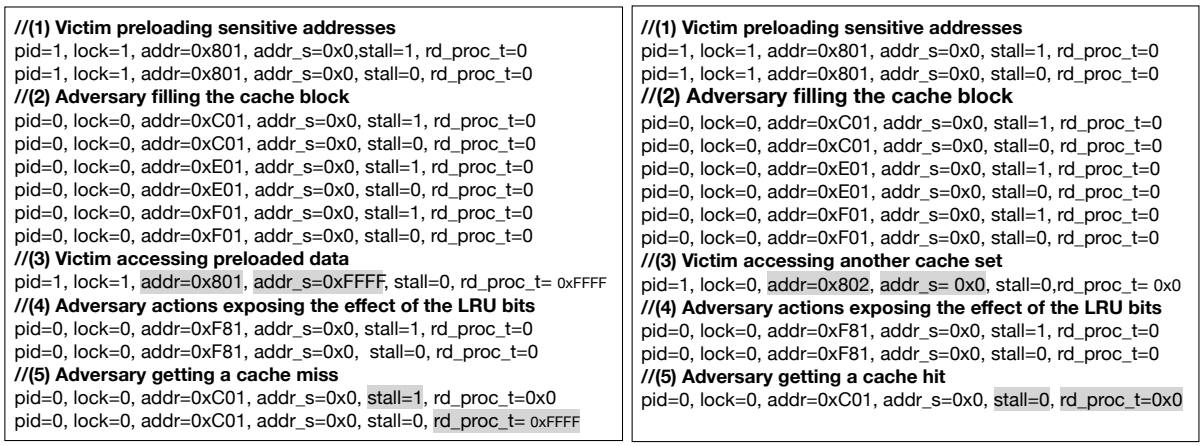


Figure 5.6: Timing leakage in PLCache. (a) Victim process (`pid=1`) accesses its locked data in stage 3. This results in a cache miss for the attacker in stage 5 (shown by `stall=1`). The verification tool captures this since `rd_proc_t` has a *High* value in stage 5. (b) Victim does not access its locked data in stage 3 and the attacker observes a cache hit in stage 5.

Fig. 5.6 shows the results of simulating the PLCache with simulation traces that resemble the extended Percival attack. In both Fig. 5.6(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. 5.6(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. 5.6(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. 5.6(a) which is a violation of the security property specified in Example 5.5.4. 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 PLCache and the sketch cache with traces that represent the extended Percival attack on LRU bits are shown in Fig. 5.7. Fig. 5.7(a) and (b) show the results of simulating PLCache and are identical to Fig. 5.6(a) and (b). Fig. 5.7(c) and (d) show the results of simulating the sketch cache. Fig. 5.7(a) and (c) show the case where the victim process (pid=1) accesses its locked data at stage 3 (case A). Fig. 5.7(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. 5.7(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

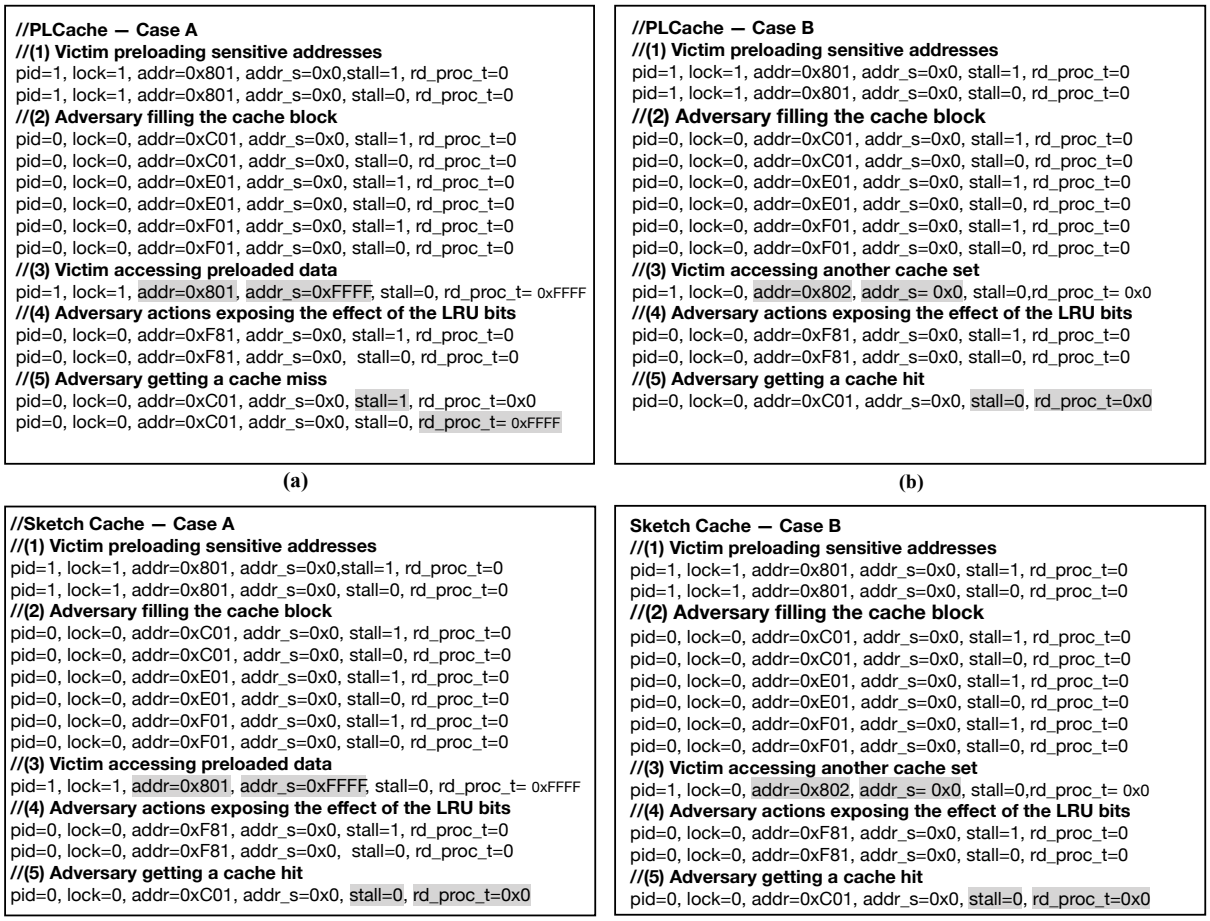


Figure 5.7: Simulating PLCache and the synthesized cache with traces representing the extended Pecival attack.

used block. In the sketch cache (Fig. 5.7(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. 5.7(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. 5.7(a) vs. Fig. 5.7(b)). In the synthesized cache, the adversary observes a cache hit in both cases (Fig. 5.7(c) vs. Fig. 5.7(d)).

Soft Constraint Analysis. As described in Section 5.5.3, 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 [FAK+12]. Fig. 5.8 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.

5.6.4 Hardware Thread Scheduler

Here we describe design of a hardware thread scheduler module for fine-grained multithreading in mixed criticality systems [BD13]. The design problem is borrowed from the

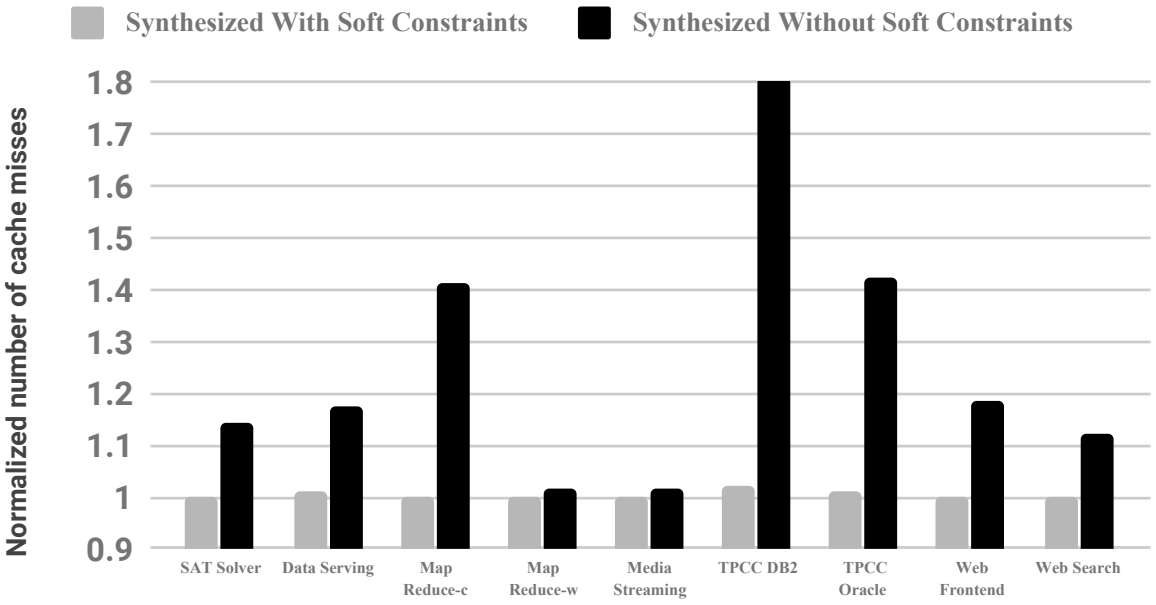


Figure 5.8: Number of cache misses for caches synthesized with and without soft constraints simulated with memory traces from CloudSuite benchmarks [FAK+12]. The numbers are normalized to the number of cache misses from a non-secure cache.

FlexPRET project [ZBSL14] 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 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 [ZBSL14], 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.

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 5.4 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

Table 5.4: 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

representation of these properties is available in Table 5.6. As shown in Table 5.4, 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 [Dut14] is used as the SMT solver for generating all the designs in these experiments.

5.6.5 SoC Arbiter

System-on-chip arbiters which mediate accesses in bus architectures have been shown to be vulnerable against timing side channel attacks [OHI+11b, OSK13]. 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.

Table 5.5: Summary of synthesized SoC Arbiters.

Design	Properties	Time (sec.)
Arbiter w/ 4 cores and 1 shared unit 338 bits	WISHBONE [MS06]	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

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 5.7.

Synthesized Designs. Table 5.5 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 5.5 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 5.5 shows the time taken for synthesis using Questa Formal Tool.

Table 5.6: 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 5.7: 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$((req_s[i] \nrightarrow_t \neg grant_s[j]) \wedge (req_s[j] \nrightarrow_t \neg grant_s[i]))$

5.7 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.

Chapter 5, in full, is a reprint of the material as it appears in the Proceedings of the 26th Annual Conference on Computer and Communications (CCS), November 2019. Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, Ryan Kastner. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Error Localization for Hardware Designs

Formal verification techniques facilitate identifying design flaws given high-level specifications. However, determining the source of the errors remains a time-consuming and manual task. To address this challenge, we propose a formal approach for error localization in Register Transfer Level (RTL) hardware designs. The presented method requires a single counterexample trace acquired from a verification tool, and it identifies a set of source code statements that includes the faulty statement. We develop a tool for capturing single errors at Verilog RTL designs and show the practicality of the proposed technique by finding errors introduced into multiple functional and control units taken from open source implementations. We further provide proof of soundness for the underlying algorithm.

6.1 Introduction

Property-based verification enables designers to model design expectations in a concise way and automatically finds counterexamples which violate the specifications (i.e., the properties). While advances in formal methods have simplified error detection, debugging using the discovered counterexamples remains a manual and tedious process taking around 60% of the time spent on verification [Fos08]. The counterexample trace generated by a formal solver (or from a failed

simulation testbench or random test) can be complex, it may span over many cycles, and thus it complicates bug analyses [CBM07]. Furthermore, the counterexample trace is an assignment over all design's primary inputs and does not indicate which parts contribute to the error.

To address these challenges, we propose a method for error localization in Register Transfer Level (RTL) hardware designs. The proposed technique considers an RTL design and its specifications along with a counterexample trace which violates the specifications. It identifies a set of suspicious code segments such that modifying them is sufficient for debugging the design. The proposed method leverages formal solvers and sound program analysis techniques to safely eliminate code segments which are irrelevant to the observed error.

The proposed error localization technique has three steps. It first reduces the counterexample trace by identifying the input variables which are irrelevant to the observed failure. While the original counterexample is an assignment to all primary inputs such that the properties are unsatisfiable, the reduced trace only contains a subset of assignments such that the properties remain unsatisfiable. We model the counterexample reduction problem as a series of calls to an SMT solvers to find input variables which values are critical for reproducing the error. In each query, a subset of inputs are set as free variables while the rest are fixed to their counterexample values, and the satisfiability of the properties is checked. If unsatisfiable, the inputs which were set as free variables are safely dropped from the counterexample. This yields a smaller and generalized counterexample by eliminating the input variables which do not contribute to the failure.

Once the relevant inputs are identified, we collect the set of program variables which are affected by those inputs using static program analysis. This is done by analyzing the data flow and control flow graph of the design and tracking how information flows from inputs to intermediate variables. The set of suspicious state variables gathered by information flow analysis could potentially be very large; however, not all elements of the set contribute to the error.

In the last step, we reduce the set of suspicious state variables by removing those that

do not contribute to the property violation. This is formulated similar to the counterexample reduction problem (from the first step) but with a minor modification. In order to look for the set of variables where changing their values could satisfy the properties we need to consider them as free variables. However, the values of intermediate variables are defined by the logic which drives them. To allow free variables, for each call to the SMT solver a temporary design is generated where state variables of interest are set as free variables by removing the logic driving them. This captures the notion that the removed logic could be potentially buggy and removing it would allow the solver to replace it with any possible value. If the solver cannot find a solution where the properties are satisfied, it indicates that the unconstrained (i.e., free) state variables do not contribute to the error. Consequently, they can be safely removed from the set of suspicious variables. By iteratively setting some state variables as free variables and removing the unsatisfiable solutions, the set of suspicious variables is gradually reduced.

The proposed framework employs open source verification tools (Yices [Dut14] and Yosys [Wolb]) as shown in Fig. 6.1. While several techniques have been introduced for error localization in gate-level netlists, our framework assists debugging at a high-level of abstraction where it is easier and faster for the designer to fix the bugs. Using formal solvers, a single counterexample trace suffices for the presented technique to succeed, contrary to statistical methods which require multiple failing and passing traces to identify faulty code segments [ANCRL07, PV16, PKL$^+$09, LNZ$^+$05, WPF$^+$10, WWQZ08]. We test our framework by localizing errors added to open source functional and control units implemented in behavioral Verilog RTL.

The rest of this paper is organized as follows. Section 6.2 summarizes the previous work. We formalize the problem definition and error model in Section 6.3. Sections 6.4 and 6.5 provide details of the proposed method and proof of correctness of the algorithm, respectively. We describe the implementation details and experimental results in Section 6.6 and conclude in Section 6.7.

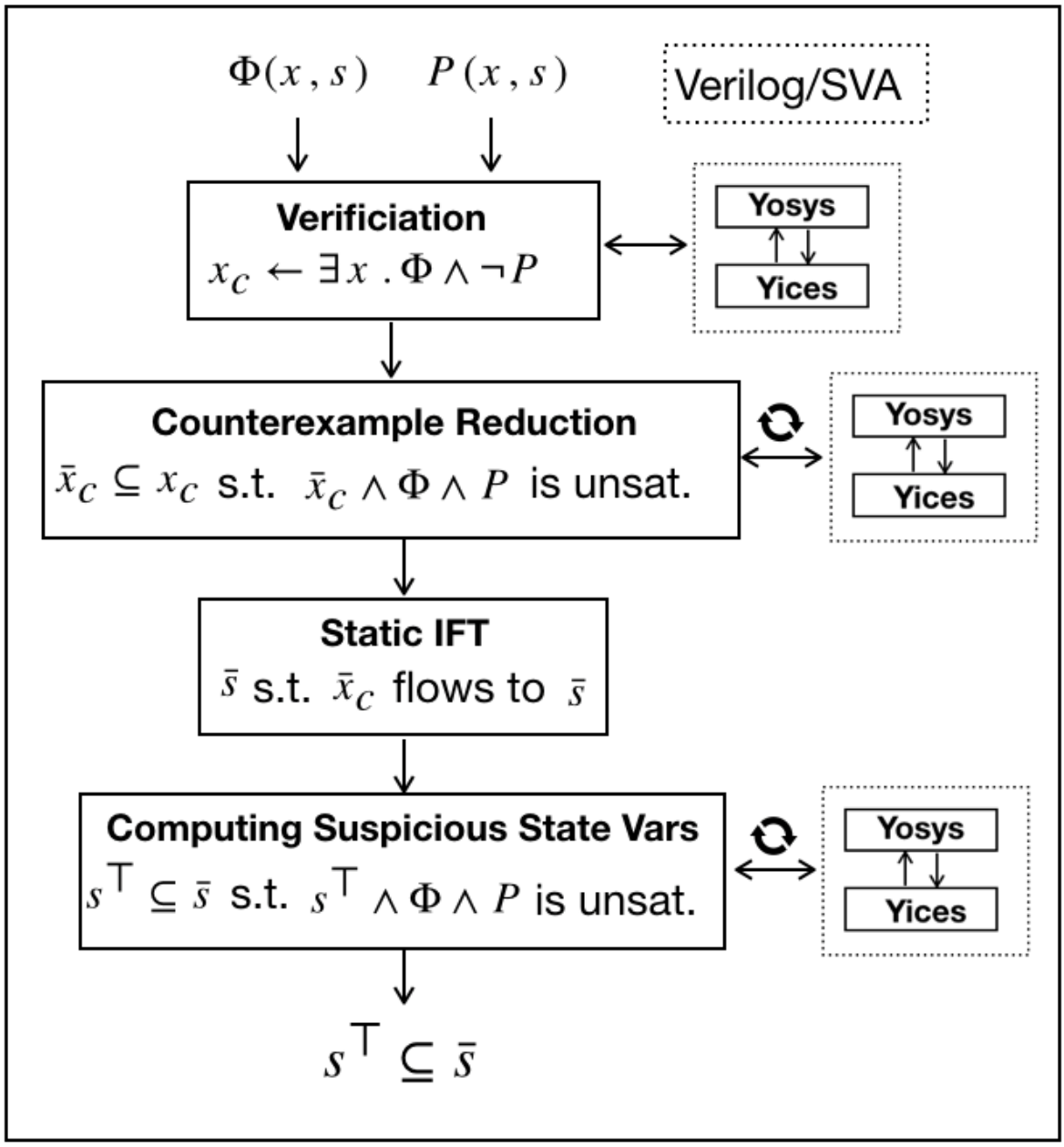


Figure 6.1: Overview of the presented error localization methodology. The proposed framework receives RTL design Φ and properties P such that verification fails. It identifies a set of suspicious source code expressions $s^\top$ from Φ that contains the bug.

6.2 Related Work

Error localization and automated debugging techniques based on statistical or formal approaches have been explored both in the software and hardware domains.

Statistical approaches rely on collecting information about program execution across multiple simulation traces to find correlation between program entities and the observed errors. Such information can be collected by instrumenting the design with Boolean predicates [LAZJ03, LNZ+05, ANCRL07, WPF+10], information flow tracking code [MFF16], invariant analysis [PKL+09], and program Spectra [AZVG07, MFF16]. Once the execution information is available, different statistical metrics rank the program statements specifying how probable they are to contribute to the error [WWQZ08, PV16, CKF+02, JHS02]. Program analysis techniques such as program slicing [Wei81, ZGG07] can further increase the accuracy of statistical methods by finding the parts of the program which are located on the failure path [MLD+14, CFR+99]. These methods have also been combined with program synthesis techniques to automatically fix the errors once they are localized [NQRC13, LCL+17].

Formal approaches leverage formal solvers to locate errors. Several solutions have been proposed for localizing errors in gate-level netlists by modeling it as a Boolean satisfiability problem. In debugging with constraint satisfaction [SVAV05, SFBD08, SV07], each gate in the netlist is instrumented with extra logic which models existence of a fault. The added logic is a multiplexer which chooses the original gate or a free variable to indicate that the gate is faulty. The instrumented circuit is replicated for all the failing test vectors and the SAT solver is queried to find a satisfying solution for the added multiplexers. The injected logic in constraint satisfaction is defined for primitive gate types and cannot be directly applied at RTL. Model-based diagnosis [Rei87, FSW99] computes conflict sets to search for fault candidates that share logic with these sets. In the hardware domain, model-based diagnosis have been used for error localization in structural RTL [PW06]. Error localization can also been modeled as a MaxSAT

problem [CSMSV10, JM11, MYR15] by finding the maximum satisfiable part of the program, i.e., the non-erroneous part. Using interpolating theorem provers, computing error invariants [ESW12] have shown promises in localizing bugs in sequential code.

Our proposed solution differs from previous work in error localization in hardware designs as it can localize errors in arbitrary RTL designs and is not restricted to specific representations such as techniques developed for gate-level netlists [SVAV05, SFBD08, SV07, CSMSV10] or structural RTL [PW06]. This facilities debugging earlier in the hardware design flow and at an abstraction level which is easier for the designer to repair the code. Furthermore, using formal techniques, the proposed method only requires a single failing trace.

6.3 Preliminaries

Here we formalize the error localization problem for a given hardware design and a set of properties. While we consider formal properties to describe the expected behaviour of the design, one can replace the properties by a list of input/output pairs without affecting the proposed technique.

6.3.1 Problem Definition

We represent the sequential[1] hardware design that we are interested in debugging by the transition function $\mathsf{tran}(x_i, s_i, s_{i+1})$ where x_i and s_i are the input and state variables at step i. The following formula encodes the design with initial state s_0 unrolled up to depth k:

$$\Phi(x,s) := \mathsf{init}(s_0) \wedge \bigwedge_{i=0}^{k-1} \Big(\mathsf{input}(x_i) \wedge \mathsf{tran}(x_i, s_i, s_{i+1})\Big)$$

Notation 6.3.1. Both x and s are sets of variables with different bitwidths. We use the notation x_i

[1] We formulate the problem for a generic sequential circuit, but the same reasoning can be applied to combinatorial circuits as well.

to refer to the elements at cycle i and use x^n to denote the n-th element which is a single variable.

Let $P(x,s)$ encode the desired properties of the circuit. Then a counterexample is a satisfying assignment to $\Phi(x,s) \wedge \neg P(x,s)$. Any counterexample x_c is a cube (conjunctive clause) over the input variables x and can be written as:

$$x_c := \bigwedge_{x^n \in x} x^n = x_c^n$$

The above formula indicates assigning value x_c^n to the n-th input variable x^n. For simplicity, we write $C(x,x_c)$ to denote the assignment of x_c values to x variables. Note that the counterexample assigns a value to each input variable at each execution cycle.

We know that $\Phi(x,s) \wedge P(x,s) \wedge C(x,x_c)$ is an unsatisfiable formula, because x_c is chosen to falsify the property P. We can now analyze the values in x_c to find the core reason for property violation and reduce the counterexample x_c.

Definition 6.3.2 (Reduced Counterexample)**.** Given counterexample x_c such that $\Phi(x,s) \wedge P(x,s) \wedge C(x,x_c)$ is an unsatisfiable formula; reduced counterexample $\bar{x}_c \subseteq x_c$ is a minimal sub-cube of x_c such that $\Phi(x,s) \wedge P(x,s) \wedge C(x,\bar{x}_c)$ is unsatisfiable.

Reduced counterexample $\bar{x}_c$ only assigns values to a subset of inputs and leaves the rest as free variables. The reduced counterexample identifies a set of input variables which contribute to the property violation. We describe an algorithm for efficiently computing $\bar{x}_c$ in Section 6.4.2.

For error localization, however, we are interested in finding the set of state variables from s which cause the error. To do so, we need to find the unsatisfiable core on the state variables.

Definition 6.3.3 (Minimal Sub-Cube $s^\top$)**.** Given reduced counterexample $\bar{x}_c$, $s^\top \subseteq s$ is the minimal subset over state variables such that $\tilde{\Phi}(x+s-s^\top, s^\top) \wedge P(x,s) \wedge C(x,\bar{x}_c)$ is unsatisfiable.

Here, $\tilde{\Phi}(x+s-s^\top, s^\top)$ represents the same design as $\Phi(x,s)$ except that variables which are not included in $s^\top$ are set as free variables by removing the logic driving them and making

them inputs. We will describe how $\tilde{\Phi}$ is built in Section 6.4.3. $s^{\top}$ indicates a set of state variables which values are critical for violation of the property. This set contains the suspicious state variables which are sufficient for debugging the design. s

Definition 6.3.4 (Debugging)**.** Given design $\Phi(x,s)$, specification $P(x,s)$, and counterexample x_c, the set of state variables $s^{\top} \subseteq s$ is sufficient for debugging Φ according to P and x_c if there exists design Φ' such that $\Phi' \wedge x_c \wedge P$ is true and Φ' is equivalent to Φ for all variables in $s - s^{\top}$. This can be written as:

$$\exists \Phi'.\Big(\bigwedge_{\forall s^i \notin s^{\top}} (\Phi'.s^i = \Phi.s^i) \wedge \Phi' \wedge x_c \wedge P\Big)$$

Notation 6.3.5. We use $\Phi.s^i$ to refer to the logic which drives state variable s^i in design Φ. This logic could be a combinatorial or sequential expression, a constant, a port assignment in module instantiation, etc..

In Section 6.5 we show that $s^{\top}$ is sufficient for debugging the design since it is proven to include the faulty state variable which should be modified.

6.3.2 Error Model

We consider debugging designs where a single bug is added to the RTL implementation such that it modifies the logical expression assigned to a single state variable. A state variable here refers to RTL variables which could have multiple bitwidths. *Hence, a single fault at RTL could map to multiple gate-level faults. Furthermore, a single fault could trigger multiple counterexamples or failing test vectors*. Note that if the buggy logical expression drives multiple other state variables, it is still included in our error model and can be soundly detected. We formalize the error model used in this work as follows.

Definition 6.3.6 (Error Model)**.** Design $\Phi'(x,s)$ has a single fault compared to the design $\Phi(x,s)$ if and only if the two designs only differ in the logical expression assigned to a single state variable s^i.

6.4 Error Localization

Algorithm 6 Given erroneous design $\Phi(x,s)$ and properties $P(x,s)$, error_loc(Φ,P) returns a set of state variables $s^{\top}$ which includes the faulty state.

```
1: Input: Φ(x,s), P(x,s): Verilog, System Verilog Assertion
2: Output: s^⊤ ⊆ s s.t. faulty state s^i ∈ s^⊤.
3: x_c ← SAT (∃x.Φ(x,s) ∧ ¬P(x,s))
4: if x_c = unsat then
5:     then return ∅
6: else
7:     x̄_c ← sub_cube (Φ,P,x,x_c)
8:     s̄ ← static_ift (Φ,x̄_c)
9:     s^⊤ ← sub_cube (Φ,P,s̄,x̄_c)
10:    return s^⊤
11: end if
```

6.4.1 Overview

Algorithm 6 describes the overview of the presented error localization technique. Given design $\Phi(x,s)$ and properties $P(x,s)$, the design is first verified against the specifications to find counterexample x_c that violates the properties. The counterexample is an assignment to all design inputs for k cycles such that if replayed the properties are false at cycle k. This indicates that formula $x_c \wedge \Phi \wedge P$ is unsatisfiable.

In the first step toward error localization we reduce the counterexample by eliminating the maximal number of assignments such that the reduced counterexample still falsifies the properties. This can be stated as finding minimal $\bar{x}_c \subseteq x_c$ such that $\bar{x}_c \wedge \Phi \wedge P$ is unsatisfiable. We find $\bar{x}_c$ by relaxing concrete assignments to the symbolic variables and checking the satisfiability. We use the notation $x_c[n \mapsto ?]$ to denote the assignment on the input variables according to the counterexample x_c but leaving the n-th input variable free. We can use the following formula for analysis:

$$\psi(x^n) := \Phi(x,s) \wedge P(x,s) \wedge C(x, x_c[n \mapsto ?]) \tag{6.1}$$

Formula $\psi(x^n)$ is used to find conditions on x^n such that the properties are true. If satisfiable, then we know that there exists an assignment on x^n such that the properties hold. That means x^n is a key variable whose value is crucial for determining whether the properties are true or false. On the other hand, if $\psi(x^n)$ is unsatisfiable, we know that the value of x^n does not affect the value of the properties, and thus it can be eliminated from the counterexample.

We can find $\bar{x}_c$ by checking the satisfiability of $\psi(x^n)$ formulas for all n. Taking this approach, the required number of SAT queries is linear with respect to the number of input arguments ($|x|$). In Section 6.4.2 we describe a binary search algorithm for computing $\bar{x}_c$ such that the number of SAT queries is linear with respect to $|\bar{x}_c|$. We refer to this algorithm as sub_cube.

The reduced counterexample identifies the input variables which values are important for the failure. Once these inputs are identified, we leverage a static information flow tracking engine to find all the intermediate variables which are affected by the critical inputs. The flow tracking algorithm described in Section 6.4.4 finds the set $\bar{s} \subseteq s$ which values are driven by some input x^n in $\bar{x}_c$. We refer to this algorithm as static_ift.

The last step of the procedure is to further minimize $\bar{s}$ by eliminating the variables which values do not affect the error. This is a very similar problem to that of counterexample reduction. Note that values of all intermediate variables can be computed by replaying the counterexample. We use $C(x,s,\bar{x}_c,s_c)$ to denote assignment $\bar{x}_c$ and s_c values to x and s variables. Similarly $s_c[n \mapsto ?]$ denotes that the n-th state variable is a free variable. However, in Φ values of state variables are determined by the logic which drives them (i.e., they are not free variables.). Hence, to actually have s^n as a free variable we define $\tilde{\Phi}(x+s^n, s-s^n)$ by removing the logic which drives s^n in Φ and turning s^n to a primary input. Now we us the following formula to compute $s^\top$ (Defn. 6.3.3):

$$\tilde{\psi}(s^n) := \tilde{\Phi}(x+s^n, s-s^n) \wedge P(x,s) \wedge C(x,s,\bar{x}_c, s_c[n \mapsto ?]) \tag{6.2}$$

We use formula $\tilde{\psi}(s^n)$ to decide if state variable s^n is critical for debugging Φ by finding

conditions on s^n that make the properties true. If $\tilde{\psi}(s^n)$ is satisfiable, it indicates that there exists an assignment on s^n which eliminates the error. Thus by altering the logic that drives s^n in the original design Φ one can potentially debug the design. In this scenario, s^n will be included in $s^\top$. If unsatisfiable, we can conclude that s^n is irrelevant to the error and should not be included in $s^\top$. Note that we only need to check satisfiability of $\tilde{\psi}$ for the intermediate variables which are marked as tainted by static_ift. Having temporary $\tilde{\Phi}$ designs, we use the sub_cube algorithm to compute $s^\top$.

6.4.2 Computing Unsatisfiable Sub-Cube

Here we describe the sub_cube algorithm for computing minimal sub-cube of an unsatisfiable formula. Given a cube c and formula ρ such that formula $c \wedge \rho$ is unsatisfiable, finding the minimal unsatisfiable sub-cube of c is defined as computing $\bar{c} \subseteq c$ such that $\bar{c} \wedge \rho$ is still unsatisfiable. This directly corresponds to the problem we are interested in solving by replacing c and ρ with the counterexample x_c and formula $\Phi \wedge P$, respectively.

A simple linear method for computing the minimal sub-cube is described in Algorithm 7. A literal is dropped from the cube at each iteration of the algorithm; and the satisfiability of the equation is checked (lines $4-6$). If satisfiable (i.e., if a positive example *pe* is found) we know that the literal which was dropped is critical for the formula to remain unsatisfiable and thus is added to the sub-cube (lines $7-8$). Dropping a literal from the counterexample cube is equivalent to setting it as a free variable. As stated earlier, if we want to have the intermediate variables as free variables we need to modify the design Φ. This is shown in line 5 of the algorithm and is discussed in more detail in Section 6.4.3.

To reduce the number of SAT queries for computing the minimal sub-cube, we describe a binary search algorithm based on Bradley et al. algorithm for computing prime implicates [BM07] for inductive strengthening of properties [Bra11]. Given a disjunctive clause c and formula ρ such that $\rho \rightarrow c$, the prime implicate problem [BM07] is stated as finding the minimal sub-clause $\bar{c}$

Algorithm 7 Given erroneous design $\Phi(x,s)$, properties $P(x,s)$, a set of variables $v \subseteq x \cup s$, and counterexample v_c, sub_cube_linear returns $\bar{v} \subseteq v$ such that $\Phi \wedge P \wedge C(v, \bar{v}_c)$ is unsatisfiable.

Input: $\Phi(x,s)$, $P(x,s)$, $v \subseteq x \cup s$, $v_c = \bigwedge_{v^n \in v} (v^n = v_c^n)$
Output:$\bar{v} \subseteq v$ *s.t.* $\Phi \wedge P \wedge \bigwedge_{v^n \in \bar{v}} (v^n = v_c^n)$ is unsat.
$\bar{v} \leftarrow \emptyset$
while $v_c \neq Null$ **do**
  $\Phi' \leftarrow$ extend_free_vars$(\Phi, head(v_c))$
  $pe \leftarrow SAT\ (\exists x.\Phi' \wedge P \wedge \bigwedge_{v^n \in \bar{v}} (v^n = v_c^n) \wedge tail\ (v_c))$
  if $pe = sat$ **then**
    $\bar{v} \leftarrow \bar{v} \cup head\ (v_c)$
  end if
  $v_c \leftarrow tail\ (v_c)$
end while
return $\bar{v}$

Algorithm 8 Given erroneous design $\Phi(x,s)$, properties $P(x,s)$, a set of variables $v \subseteq x \cup s$, and counterexample v_c, sub_cube returns $\bar{v} \subseteq v$ such that $\Phi \wedge P \wedge C(v, \bar{v}_c)$ is unsatisfiable.

Input: $\Phi(x,s)$, $P(x,s)$, $v \subseteq x \cup s$, $sup = \emptyset$,
$v_c = \bigwedge_{v^n \in v} (v^n = v_c^n)$
Output: $\bar{v} \subseteq v$ *s.t.* $\Phi \wedge P \wedge \bigwedge_{v^n \in \bar{v}} (v^n = v_c^n)$ is unsat.
$\bar{v} \leftarrow v$
if $\text{len}(\bar{v}) = 1$ **then**
  return $\bar{v}$
end if
$r,\ l \leftarrow$ split $(\bar{v})$
$\Phi' \leftarrow$ extend_free_vars(Φ, l)
$pe \leftarrow SAT\ (\exists x.\Phi' \wedge P \wedge \bigwedge_{v^n \in\ sup\ \cup\ r} (v^n = x_c^n))$
if $pe = unsat$ **then**
  return sub_cube(Φ, P, r, sup, v_c)
end if
$\Phi' \leftarrow$ extend_free_vars(Φ, r)
$pe \leftarrow SAT\ (\exists x.\Phi' \wedge P \wedge \bigwedge_{v^n \in\ sup\ \cup\ l} (v^n = v_c^n))$
if $pe = unsat$ **then**
  return sub_cube(Φ, P, r, sup, v_c)
end if
$\hat{r} :=$ sub_cube$(\Phi, P, r, sup \cup l, v_c)$
$\hat{l} :=$ sub_cube$(\Phi, P, l, sup \cup \hat{r}, v_c)$
return $\hat{l} \cup \hat{r}$

such that $\rho \rightarrow \bar{c}$. Our sub-cube minimization problem can be formulated as the dual of computing prime implicates.

To reduce the number of SAT queries in Algorithm 7, in each iteration of Algorithm 8 half of the literals are constrained by their counterexample value while the rest are free. The split function divides the variables which have not been analyzed yet to two separate groups r and l of roughly equivalent sizes. First, r variables are constrained to their counterexample values and l variables are free. If unsatisfiable, all l variables can be safely removed from the cube. And the algorithm iterates over r (lines $8-12$, ignoring *sup* for now.). If satisfiable, we cannot make any decision about l variables. Next, l variables are constrained while r variables are free. Similarly, if unsatisfiable the algorithm iterates over l (lines $13-17$).

If both queries are satisfiable, no variable can be immediately eliminated from the cube. In this case, the algorithm first assumes that all l variables are included in the sub-cube and iterates over r. This is modeled by adding l to the auxiliary set *sup* which variables are always constrained by the counterexample (lines 9 and 14) and is initially empty. This result in $\hat{r} \subseteq r$, which is the sub-cube of r computed with respect to l. Next, $\hat{r}$ is added to the auxiliary set and $\hat{l}$ is computed by iterating over l. The final sub-cube contains both $\hat{r}$ and $\hat{l}$ (lines $18-20$).

6.4.3 Computing Sub-Cube on State Variables

The algorithms described in Section 6.4.2 can be used to compute the minimal unsatisfiable sub-cube on both the input variables and state variables. The only difference between the two cases is that for finding the sub-cube over the state variables we need to be able to set them as free variables for analysis. For instance, to check if state variable s^n should be included in the unsatisfiable sub-cube $s^\top$, we need to check satisfiability of $\tilde{\Psi}(s^n)$ which requires having s^n as a free variable. To do so we generate temporary design $\tilde{\Phi}(x+s^n, s-s^n)$ which is equivalent to the original design $\Phi(x,s)$ except that s^n is turned into a primary input by removing the logic which defines it.

Algorithm 9 describes the extend_free_var(Φ, v) procedure which generates design $\tilde{\Phi}$ from Φ by turning a set of state variables v into primary inputs. For each variable $v^n \in v$, the algorithm first finds the hierarchical path p and module g in which v^n is defined. Next, it generates new module g_p from g by adding v^n_{free} as a new input which drives v_n (lines 4-7). In order to have v^n_{free} as a primary input of $\tilde{\Phi}$, extended module g_p replaces the original g in the hierarchical path p. Lastly, any module on path p which instantiates g is extended with input v^n_{free} (lines $8-11$).

Algorithm 9 Given design $\Phi(x,s)$ and a set of state variables v, extend_free_vars returns new design $\tilde{\Phi}$ where v variables are turned into primary inputs.

```
1:  Input: Φ(x,s), v ⊆ x ∪ s
2:  Output: Φ̃(x+v, s−v)
3:  Φ̃ ← Φ
4:  for v^n ∈ v − x do
5:      Find module g and hierarchical path p where v^n is defined
6:      In Φ̃ define module g_p from g by adding new input v^n_free
7:      In g_p replace right hand-side of v^n with v^n_free
8:      for module h on path p instantiating g do
9:          In Φ̃ Define module h_p by adding new input v^n_free
10:         Replace each instantiation of g with g_p and h with h_p
11:     end for
12: end for
13: return Φ̃(x+v, s−v)
```

6.4.4 Static Information Flow Analysis

The model of information flow [Den76, DD77b] enables analyzing how information moves in a given design. Information flows from input variable x^n to state variable s^m if some change in x^n value could alter s^m value while all other input variables remain the same. Information can flow through both the data path and the control path called *explicit* and *implicit* flows, respectively. Explicit flows refer to cases where the source variable (x^n here) directly affects the output of an operation which controls the destination variable (s^m here). More subtly and via implicit flow, variable x^n can affect value of s^m if it is used in a conditional statement which

controls s^m.

Here, we use information flow analysis to find the set of state variables which are affected by the critical inputs identified by the reduced counterexample $\bar{x}_c$. More specifically, we collect the set of state variables $\bar{s} \subseteq s$ such that $\bar{x}_c$ flows to $\bar{s}$. Algorithm 10 describes the method used for finding $\bar{s}$ which we call the set of tainted state variables. This set initially includes the critical inputs from the reduced counterexample $\bar{x}_c$ and is iteratively extended with tainted state variables. To do so, for each element $s^n \in \bar{s}$ we find all the state variables which computations depend on s^n. This is done by finding all assignments[2] where s^n is used as a right hand-side or conditional variable, tracking both explicit and implicit flows. This is done by traversing the data flow and control flow graphs of the design extracted from its abstract syntax tree (AST) representation. The set of tainted variables $\bar{s}$ is then extended by the left hand-side variables of these assignments. The procedure terminates when all elements of $\bar{s}$ have been analyzed. Note that the information flow analysis does not collect the constants defined in the design since they are not driven by inputs. However, we add all the constants to the set of suspicious variables as they may potentially include the bug.

Algorithm 10 Given design $\Phi(x,s)$ and a set of design inputs $\bar{x}$, static_ift returns a set of state variables $\bar{s}$ which values are influenced by $\bar{x}$.

1: Input: $\Phi(x,s), \bar{x}_c \subseteq x$
2: Output: $\bar{s} \subseteq s$ *s.t.* $\bar{x}_c$ flows to $\bar{s}$.
3: $\bar{s} \leftarrow \bar{x}_c$
4: **for** $s^n \in \bar{s}$ **do**
5: $\quad s^n_{\mathsf{exp\_flow}} :=$ l-values of assignments where s^n is an r-value
6: $\quad s^n_{\mathsf{imp\_flow}} :=$ l-values of assignments where s^n is a condition
7: $\quad \bar{s} \leftarrow \bar{s} \cup s^n_{\mathsf{exp\_flow}} \cup s^n_{\mathsf{imp\_flow}}$
8: **end for**
9: **return** $\bar{s} - \bar{x}_c + \Phi.\mathsf{Consts}$

The described method collects the tainted state variables while preserving their locality. Hence, variables which are relevant (e.g., if they belong to the same instantiation of a module)

[2] Assignment here refers to all continuous, procedural (blocking and non-blocking) and module port assignments in Verilog.

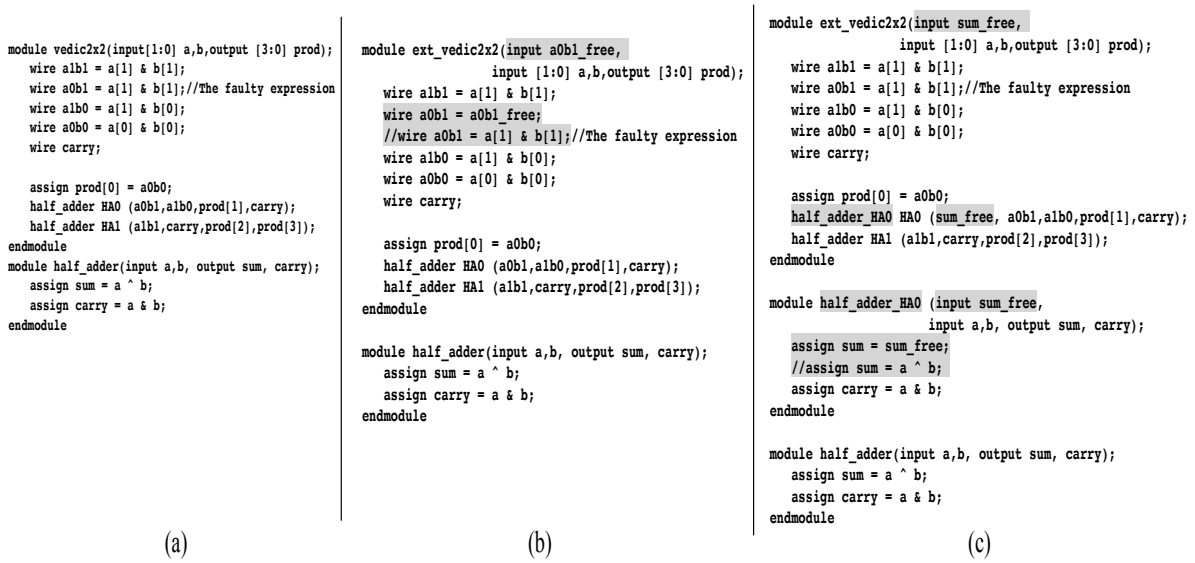


Figure 6.2: A buggy implementation of a 2bit Vedic multiplier and samples of temporary modules generated during the error localization process. Highlighted parts show how the temporary designs are different from the original one. (a) Buggy implementation where fault is added to the logic driving variable a0b1. (b) Temporary design generated to compute satisfiability of formula $\tilde{\psi}(\mathsf{a0b1})$ by turning a0b1 to a free variable. (c) Temporary design generated to analyze effect of variable HA0.sum on the failure. This design is used to check satisfiability of formula $\tilde{\psi}(\mathsf{HA0.sum})$ from equation 6.2.

will be next to each other. This helps the binary search algorithm to eliminate groups of variables as related variables will be roughly grouped together.

Recently, several information flow tracking tools have been introduced to enable security verification of hardware designs [TWM+09, JM12, ZWSM15, AHMK, AHK, JDSZ18]. These tools provide a precise technique for capturing flow of information by dynamically analyzing the Boolean values of design's variables (e.g., the branch directions are precisely computed) [HBA+16]. Here we rely on a light-weight static information flow tool which only analyzes connectivity of different variables. This lack of precision is by intention as we later need to analyze the design by setting some variables as free and considering all possible values.

Example 6.4.1. Take as an example a 2bit Vedic multiplier where a bug is manually added to its RTL implementation as shown in Fig. 6.2 (a). The output of the error localization tool analyzing the buggy design is shown in Fig. 6.3. In this simple example both inputs are critical for the

```
Critical inputs: ['a = 2'b10', 'b = 2'b10']
Tainted state variables:
   ['a1b1', 'a0b1', 'a1b0', 'a0b0', 'HA0.sum'
    'HA0.carry', 'HA1.sum', 'HA1.carry', 'carry']
Suspicious state variables: ['a0b1']
```

Figure 6.3: Output of the error localization tool analyzing the buggy 2bit Vedic multiplier from Fig. 6.2 (a). "Critical inputs" are the input variables which appear in the reduced counterexample. "Tainted state variables" are variables which are driven by the critical inputs and are collected by the information flow tracking unit. "Suspicious state variables" are the variables which the tool has marked as potentially buggy and include the faulty state. The variables are reported by their hierarchical names.

property violation. "Tainted state variables" (i.e., $\bar{s}$) are wires and registers which are driven by these inputs. In this small example all state variables are marked as tainted. "Suspicious state variables" (i.e., $s^{\top}$) are the state variables which should be inspected for debugging the design. In this example the tool can precisely locate the bug. However, in most cases several benign variables are also included in $s^{\top}$ as the tool conservatively eliminates irrelevant variables.

Two samples of the temporary files generated while computing the unsatisfiable sub-cube on the state variables are shown in Fig. 6.2(b) and (c). The difference between the original modules and the temporary versions are highlighted. In Fig. 6.2(b), new input a0b1_free is added which drives a0b1. Thus, a0b1 can be seen as a free variable which could take any value as is not constrained by the inputs a and b. This modified design corresponds to $\tilde{\Phi}$ from equation 6.2 and is used to compute $\tilde{\psi}$(a0b1). In this example $\tilde{\psi}$(a0b1) is satisfiable since modifying the value of a0b1 can eliminate the bug. Thus, a0b1 is included in the set of suspicious variables $s^{\top}$. Fig. 6.2(c) shows the temporary design generated to analyze the effect of HA0.sum on the observed error. In this case an extended version of the half adder module is generated by adding new input sum_free which drives its sum output. In the top level module the half adder which drives HA0.sum (i.e., instantiation HA0) is replaced by the new extended half adder. By adding sum_free as input to the top level module, the output sum of the half adder HA0 can be seen as free variable. This new design is used to check satisfiability $\tilde{\psi}$(HA0.sum). In this case $\tilde{\psi}$ is unsatisfiable and hence HA0.sum is not included in the set of suspicious variables $s^{\top}$.

Table 6.1: Summary of error localization experiments on benchmarks from OpenCores [ope]. "#of Suspicious Variables" shows the number of RTL expressions which are marked as potentially buggy by the tool.

Design	SVA Property	#of Design Variables	#of Suspicious Variables	Time sec.
Arbiter – TDMA 4 cores, 2 shared units	Grant given to at most n cores assert ($ Countones (grant) $\leq n$)	26	3	3.4
Arbiter – TDMA 4 cores, 2 shared units	Equal share between cores n and m accessing unit i over K time steps assert $\left(\Sigma_{k=0}^{K-1}$ $Past(grant$_i$[n],k) $=\Sigma_{k=0}^{K-1}$ $Past(grant$_i$[m],k)$\right)$	26	1	2.1
Arbiter – Round Robin 4 cores, 2 shared units	Grant given to at most n cores assert ($ Countones (grant) $\leq n$)	64	3	34.3
Arbiter – Round Robin 4 cores, 2 shared units	Grant only given to a core which requested assert ($Past(req$_i$,n) $\rightarrow$ grant$_i$ & $Past($req$_i$,n))	64	1	27.7
12bit Ripple Adder	assert($a + b + cin == sum + cout * 2^{12}$)	44	1	1.3
12bit Ripple Adder	assert($a + b + cin == sum + cout * 2^{12}$)	44	4	4.5
4bit Vedic Multiplier	assert($a * b == prod$)	72	9	23.5
4bit Vedic Multiplier	assert($a * b == prod$)	72	10	25.8
8bit Vedic Multiplier	assert($a * b == prod$)	88	2	29.6
8bit Vedic Multiplier	assert($a * b == prod$)	88	6	31.5

6.5 Soundness Analysis

Theorem 6.5.1 (Soundness)**.** Given design Φ with single fault model, specification P, and counterexample x_c such that x_c falsifies P; state variable s^i which should be modified in order to eliminate the error represented by x_c is included in $s^\top$.

Proof. We show this via proof by contradiction. As contradiction hypothesis assume that $s^i \notin s^\top$. Consider Φ' representing a design which differs from Φ only in a single state. Using the theorem obligation and the single fault error model (Def. 6.3.6) we can write:

$$\exists \Phi'.\Phi' \wedge P \wedge x_c \wedge \Phi.s^i \neq \Phi'.s^i \wedge \bigwedge_{s^j \neq s^i} \Phi.s^j = \Phi'.s^j$$

Hence, the following equation which checks the satisfiability of $\tilde{\Phi}$ with respect to P and x_c when s^i is set as free variable is satisfiable:

$$\tilde{\Phi}(x, s^i, s - s^i) \wedge P \wedge x_c$$

From this we know that $\tilde{\psi}(s^i)$ as defined by equation 6.2 is satisfiable. Hence, s^i is included in $s^\top$ by definition of the sub_cube algorithm. This falsifies the contradiction hypothesis.

□

Proposition 6.5.2 (Upper Bound)**.** To compute $s^{\top} \subseteq \bar{s}$, sub_cube algorithm described in Algorithm 8 makes at most $O\big((|s^{\top}|-1)+|s^{\top}|lg\frac{|\bar{s}|}{|s^{\top}|}\big)$ SAT queries.

The proof of the above proposition along with the lower bound on the number of queries required to compute minimal subset $s^{\top}$ can be found in [BM07]. Note that $s^{\top}$ can be directly acquired from a SAT solver in form of the *unsatisfiable core*. However, such set is not necessary minimal [BM07].

6.6 Evaluation

We have implemented the described methodology in Python and using open source verification tools as shown in Fig. 6.1. The developed framework analyzes hardware designs written in Verilog RTL along with a set of System Verilog Assertion properties. Before each SAT query, Yosys [Wolb] is called to translate the Verilog design and properties to SMT-LIB2 [smt] format which can be analyzed by any SMT solver. We have used Yices2 [Dut14] as the SMT solver in our experiments due to its good support for theory of bit-vectors. The information flow analysis unit also uses Yosys front-end parser to translate the Verilog design to its abstract syntax tree (AST) representation. The final output of the tool, as shown by the simple example in Fig. 6.2 is a set of suspicious state variables where the logic assigned to them includes the bug.

We have tested our error localization framework on multiple designs acquired from OpenCores [ope]. We have written a set of SVA formal properties for each design and verified them by bounded model checking (BMC). Next, we manually add a bug to the RTL implementation and locate the bug using our framework. These bugs include errors in the logical expressions, wrong module port assignments, and incorrect state encoding and state transition in finite state machines (FSM). Note that each RTL bug could potentially result in several faults in the gate-level netlist. We use SVA properties to state high level design expectations and to automatically search for a

counterexample trace which triggers the added bug. However, the SVA properties can be easily replaced by a set of test vectors which model the expected input/output relation.

Table 6.1 shows the results of our experiments in terms of the number of state variables reported as suspicious and the required time. In all cases, the actual bug is contained in the set of suspicious state variables. In most cases the set of suspicious variables includes some correct variables as well. This is expected as some variables besides the faulty one could potentially affect the property, and the tool is conservative in eliminating them. In our experiments, the tool marks around 6% of the state variables as suspicious on average. This could considerably reduce the time that the designer needs to spend on debugging as she only needs to focus on a small part of the design. In Table 6.1, "# of Design Variables" refers to the total number of RTL wires and registers (with various bitwidths) and are counted from the AST representation. The reported time includes the time spent on all steps of Algorithm 6 including the time required for finding the counterexample in the verification round. All experiments are run using a desktop computer and 2.3 GHz Intel Core i5 CPU.

6.7 Conclusion

This work presents a formal approach for localizing single faults in Verilog RTL designs. The proposed error localization framework receives a buggy hardware design and a set of formal properties which are violated. And it identifies a set of source code expressions which includes the error. The proposed technique simplifies debugging at a high-level of abstraction by allowing the designer to focus on a substantially smaller portion of the design (around 6% in our experiments) which is marked as suspicious. The proposed solution relies on formal solvers to soundly eliminate the code segments which do not contain the error.

Chapter 6 is being prepared for the publication of the material. Armaiti Ardeshiricham, Christie Lincoln, Alvin Zhang, Lisa Luo, Amir Uqdah, Sicun Gao, Ryan Kastner. The dissertation

author was the primary investigator and author of this material.

Bibliography

[ABJ+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, 2013.

[AHK] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. Clepsydra: Modeling timing flows in hardware designs. In *International Conference on Computer-Aided Design (ICCAD), 2017*.

[AHMK] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Proceedings of the 2017 Conference on Design, Automation & Test in Europe*.

[AKM+15] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

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

[ANCRL07] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 5–15. ACM, 2007.

[AS07] Onur Aciicmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography. Workshop on*. IEEE, 2007.

[ATGK19] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1623–1638, 2019.

[AZVG07] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BD13] Alan Burns and Robert Davis. Mixed criticality systems-a review. pages 1–69, 2013.

[Ber05] Daniel J Bernstein. Cache-timing attacks on aes, 2005.

[BM] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. Toward automatic proof generation for information flow policies in third-party hardware ip. In *Hardware Oriented Security and Trust (HOST), 2015*.

[BM07] Aaron R Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 173–180. IEEE, 2007.

[BM15] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 29–32. IEEE, 2015.

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

[BNI14] Andrew Becker, David Novo, and Paolo Ienne. Sketchilog: Sketching combinational circuits. In *Proceedings of the conference on Design, Automation & Test in Europe*, 2014.

[Bra11] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[CBM07] Kai-hui Chang, Valeria Bertacco, and Igor L Markov. Simulation-based bug trace minimization with bmc-based refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(1):152–165, 2007.

[CC04] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[CCF+16] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto. Secure embedded architectures: Taint properties verification. In *2016 International Conference on Development and Application Systems (DAS)*, pages 150–157, May 2016.

[CCX+18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[CFR+99] Edmund M Clarke, Masahiro Fujita, Sreeranga P Rajan, T Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 298–313. Springer, 1999.

[CKF+02] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

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

[CS10] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CSMSV10] Yibin Chen, Sean Safarpour, Joao Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(11):1804–1817, 2010.

[DD77a] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[DD77b] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[Dim01] Jordan Dimitrov. Operational semantics for verilog. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 161–168. IEEE, 2001.

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Dut14] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[ERAG+18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[ESW12] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *International Symposium on Formal Methods*, pages 187–201. Springer, 2012.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 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.

[Fen06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[FJJ+12] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 548–563. Springer, 2012.

[FMBD18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 420–435. ACM, 2018.

[FMVG+17] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices*, 52(6):422–436, 2017.

[Fos08] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In *International Conference on Computer Aided Verification*, pages 5–10. Springer, 2008.

[FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.

[GJJ06] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. *Retrofitting legacy code for authorization policy enforcement*. IEEE, 2006.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

[Gul10] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM, 2010.

[HAAK16] W. Hu, A. Althoff, A. Ardeshiricham, and R. Kastner. Towards property driven hardware security. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 51–56, Dec 2016.

[HBA+16] Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. Imprecise security: quality and complexity tradeoffs for hardware information flow tracking. In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pages 1–8. IEEE, 2016.

[HJR10] William R Harris, Somesh Jha, and Thomas Reps. Difc programs by automatic instrumentation. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 284–296. ACM, 2010.

[HJRS17] William R Harris, Somesh Jha, Thomas W Reps, and Sanjit A Seshia. Program synthesis for interactive-security systems. *Formal Methods in System Design*, 51(2):362–394, 2017.

[HMOK16a] W. Hu, B. Mao, J. Oberg, and R. Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):44–52, Aug 2016.

[HMOK16b] W. Hu, B. Mao, J. Oberg, and R. Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):44–52, 2016.

[HSSA16] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *ACM SIGPLAN Notices*, volume 51, pages 237–250. ACM, 2016.

[Hu92] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[JDSZ18] Zhenghong Jiang, Steve Dai, G Edward Suh, and Zhiru Zhang. High-level synthesis with timing-sensitive information flow enforcement. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.

[JHS02] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[JM11] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[JM12] Y. Jin and Y. Makris. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *2012 IEEE 30th VLSI Test Symposium (VTS)*, pages 252–257, 2012.

[KDK09] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 105–114. IEEE, 2009.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[Koc] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Advances in Cryptology - CRYPTO'96*.

[KW18] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *ACM Sigplan Notices*, volume 38, pages 141–154. ACM, 2003.

[LCL+17] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[LKO+14] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. Sapper: A language for hardware-level security policy enforcement. *ACM SIGARCH Computer Architecture News*, 42(1):97–112, 2014.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *Acm Sigplan Notices*, 40(6):15–26, 2005.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[LTO+11a] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Notices*, volume 46, pages 109–120. ACM, 2011.

[LTO+11b] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[MFF16] Jan Malburg, Alexander Finder, and Görschwin Fey. Debugging hardware designs using dynamic dependency graphs. *Microprocessors and Microsystems*, 47:347–359, 2016.

[MLD+14] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[MS06] Milica Mitić and Mile Stojčev. A survey of three system-on-chip buses: Amba, coreconnect and wishbone. In *Proc. 41st Int. Conf. Inform. Commun. Energy Syst. Technol.(ICEST)*, pages 282–285. Citeseer, 2006.

[MYR15] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 448–458. IEEE Press, 2015.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[Obe14] Jason Kaipo Oberg. *Testing Hardware Security Properties and Identifying Timing Channels*. PhD thesis, UC San Diego, 2014.

[OHI+11a] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Design Automation Conference (DAC)*, pages 254 –259, June 2011.

[OHI+11b] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in i2c and usb. In *Proceedings of the 48th Design Automation Conference*, pages 254–259. ACM, 2011.

[OMSK14a] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner. Leveraging gate-level properties to identify hardware timing channels. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(9):1288–1301, 2014.

[OMSK14b] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. Leveraging gate-level properties to identify hardware timing channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(9):1288–1301, 2014.

[ope] *The reference community for Free and Open Source gateware IP cores*. `https://opencores.org/`.

[OSK13] Jason Oberg, Timothy Sherwood, and Ryan Kastner. Eliminating timing information flows in a mix-trusted system-on-chip. *IEEE Design & Test*, 30(2):55–62, 2013.

[Per05a] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan 2005*, Ottowa, Canada, May 2005.

[Per05b] Colin Percival. Cache missing for fun and profit, 2005.

[PKL+09] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[PV16] Debjit Pal and Shobha Vasudevan. Symptomatic bug localization for functional debug of hardware designs. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 517–522. IEEE, 2016.

[PW06] Bernhard Peischl and Franz Wotawa. Automated source-level error localization in hardware designs. *IEEE design & test of computers*, 23(1):8–19, 2006.

[PYIS16] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. Type-driven repair for information flow security. *CoRR*, abs/1607.03445, 2016.

[RCK+] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009*.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[RWK+14] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. User-guided device driver synthesis. In *OSDI*, pages 661–676, 2014.

[SA14] Pramod Subramanyan and Divya Arora. Formal verification of taint-propagation security properties in a commercial soc design. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, Leuven, BEL, 2014. European Design and Automation Association.

[SA16] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. *ACM SIGPLAN Notices*, 51(6):326–340, 2016.

[Sch00] Werner Schindler. A timing attack against rsa with the chinese remainder theorem. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 109–124. Springer, 2000.

[SFBD08] Andre Suelflow, Goerschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 77–82. ACM, 2008.

[SHV+17] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. Template-based parameterized synthesis of uniform instruction-level abstractions for soc verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[SL13] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

[SMS13] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[smt] SMT-LIB: The Satisfiability Modulo Theory Library. `http://smtlib.cs.uiowa.edu`.

[SP18] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[STK13] H. Salmani, M. Tehranipoor, and R. Karri. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 471–474, Oct 2013.

[SV07] Sean Safarpour and Andreas Veneris. Abstraction and refinement techniques in automated design debugging. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007.

[SVAV05] Alexander Smith, Andreas Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1606–1621, 2005.

[TOL+11] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 189–200. ACM, 2011.

[tru] `https://www.trust-hub.org/`.

[TWM+09] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[VBC+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254. IEEE Computer Society, 2004.

[VDSP08] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Fourteenth International Symposium on High Performance Computer Architecture (HPCA)*, pages 196–206, New York, NY, USA, 2008. ACM.

[WDS17] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):62, 2017.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[wis] *The reference community for Free and Open Source gateware IP cores*. `https://opencores.org/opencores,wishbone`.

[WL07] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, 2007.

[Wola] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[Wolb] Clifford Wolf. Yosys Open SYnthesis Suite. `http://github.com/cliffordwolf/yosys`.

[WPF+10] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[WWQZ08] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 42–51. IEEE, 2008.

[WYHJ] Bo-Han Wu, Chun-Ju Yang, Chung-Yang Ric Huang, and Jie-Hong Roland Jiang. A robust functional eco engine by sat proof minimization and interpolation techniques. In *Proceedings of the International Conference on Computer-Aided Design, 2010*.

[YKDC16] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *ACM SIGPLAN Notices*, volume 51, pages 508–521. ACM, 2016.

[YWDD17] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.

[ZBSL14] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. Flexpret: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 101–110. IEEE, 2014.

[ZGG07] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.

[ZS13] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 224–234. IEEE Press, 2013.

[ZWSM15] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 503–516. ACM, 2015.