TOWARDS AUTOMATED SECURITY VALIDATION FOR HARDWARE DESIGNS

Rui Zhang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2020

Approved by:

Cynthia Sturton

Leonard McMillan

Fabian Monrose

Michael Reiter

Daniel Sorin

ABSTRACT

Rui Zhang: Towards Automated Security Validation for Hardware Designs
(Under the direction of Cynthia Sturton)

Hardware provides the foundation of trust for computer systems. Defects in hardware designs routinely cause vulnerabilities that are exploitable by malicious software and compromise the security of the entire system. While mature hardware validation tools exist, they were primarily designed for checking functional correctness. How to systematically detect security-critical defects remains an open and challenging question.

In this dissertation, I develop formal methods and practical tools for automated hardware security validation. To identify and develop security-critical properties for hardware design, I developed SCIFinder, a methodology that leverages known vulnerabilities to mine and learn security invariants. I show that security vulnerabilities together with machine learning techniques can give us a set of security properties to detect both known and unknown security bugs in the OR1200 processor. I also proposed another method to develop security-critical properties by leveraging existing ones, and I built a tool, Transys, to translate security properties across similar or different versions of hardware designs. I demonstrate that translating security properties across AES hardware, RSA hardware and RISC processors is feasible and light-weight. Given the security properties, I developed Coppelia to validate the security of hardware designs. I proposed a hardware-oriented backward symbolic execution strategy to find violations and generate exploit programs. I successfully generate exploits for known security bugs on the OR1200 processor, and discovered and generated exploit programs for 4 unknown bugs across two different processors and architectures.

To my parents, and my husband.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Cynthia Sturton. Professor Sturton kindly accepted me as her student, guided and supported me throughout my PhD study. I really appreciate her extreme patience with me during my PhD. She always patiently listens to my dumb and premature ideas, and often gives constructive and insightful feedback to me. She patiently helps me practicing my presentations numerous times, and encourages me before each of my presentation.

I would like to express my gratitude to all my committee members: Professor Leonard McMillan, Professor Fabian Monrose, Professor Michael Reiter, and Professor Daniel Sorin. I am grateful for the time and wisdom they devoted, and the helpful discussions and feedbacks they provided. I am super lucky to have all of them to be my committee members. I would like to thank Professor Sorin for his supports and help, especially during my hard time.

I am thankful to my research collaborators and co-authors: Natalie Stanly, Andrew Chi, Chris Griggs, and Calvin Deutschbein, for their contributions, tremendous help and insightful comments. I would also like to thank to my colleagues and friends in all UNC security groups: Michael Brown, Alyssa Byrnes, Abhishek Singh, Qiuyu Xiao, Ziqiao Zhou, Sheng Liu, Marie Nesfield, for their friendship, inspiration and all the generous help. I am also grateful to all the people I met during these years at UNC, other institutions, and the conferences, as well as the anonymous reviewers, all of whom made my PhD life colorful.

Lastly, I would like to thank my family. I appreciate all the love, supports and encouragements from my husband, Peng. He encourages me to think bigger, and motivates me to fight harder during the dark times. Although we had a long-distance relationship, he made me feel that he was always around me. I also would like to thank my parents for their unconditional love, understanding, and supports. They are always proud of me no matter what I achieve. I cannot express how grateful I am to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABV	Assertion Based Verification
AES	Advanced Encryption Standard
AST	Abstract Syntax Tree
BFS	Breadth First Search
CNF	Conjunctive Normal Form
DAG	Directed Acyclic Graph
DFS	Depth First Search
DoS	Denial of Service
GPR	General Purpose Register
HDL	Hardware Description Language
ISA	Instruction Set Architecture
ORAM	Oblivious RAM
OVL	Open Verification Library
PC	Program Counter
PCA	Principal Component Analysis
PDG	Program Dependence Graph
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SCI	Security-Critical Invariants
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extension
SPR	Special Purpose Register
SR	Status Register
TCB	Trusted Computing Base
VM	Virtual Machine
XOM	Execute-Only Memory

CHAPTER 1
INTRODUCTION

Hardware provides the foundation of computing and represents the minimal Trusted Computing Base (TCB) for the upper-level software systems. The implication is that an attacker who controls the hardware of a machine can often gain unrestricted access to the entire machine. A well-resourced attacker with access to the hardware supply chain may incorporate a backdoor into the design. Yet an easier way exists: by leveraging existing vulnerabilities within hardware designs, the attacker can gain control of hardware using software-only exploits launched remotely from the hardware. For example, a defect in recent x86 CPUs' exception delivery logic could be exploited by a guest VM to launch a denial of service attack against its host [9]. The infamous Spectre and Meltdown attacks [67, 74] exploited the vulnerability in speculative execution to leak kernel memory. This problem is exacerbated by the growing complexities of hardware and varieties of domain-specific architectures. Indeed, security experts have warned recently that hardware is in "the crosshairs of cyberhackers" [88].

The current state of the art to find vulnerabilities in hardware designs uses hardware design verification, which combines simulation based testing and formal static analysis. The efficacy of simulation-based testing depends on the coverage of the testbenches used, and is unlikely to uncover a vulnerability that is exploitable by only a handful of possible input sequences. Formal static analysis, on the other hand, is a systematic approach to potentially combat hardware security vulnerabilities. One promising direction of formal static analysis for security validation is assertion-based verification. Each assertion is a proposition encoding a property that should always hold, and monitors the hardware signals and states named in the property. Because the expected behavior is described via a property, the assertion can potentially expose a class of seemingly different defects in the processors. The questions of what are the security-critical properties we should protect for hardware and how to efficiently develop them, however, are not clear.

Another big challenge often faced by hardware designers and security experts is: when a vulnerability is fixed or mitigated, is the problem fully solved? There are three aspects of this question: First, is the vulnerability fully or only partially fixed – can the attacker modify the original attack program or use a new

attack program to trigger the same vulnerability? Second, are similar vulnerabilities also fixed – are simple variants of the original vulnerabilities (low-hanging fruits to attackers) also fixed? Third, when we move to the next generation of the product, will the design be vulnerable to the same attacks again?

This dissertation aims to answer the above questions. I develop systematic methods and practical tools that help hardware designers efficiently detect a class of security vulnerabilities (software exploitable bugs) at design time and build more secure hardware designs. Our research efforts have explored achieving this objective from two broad lines.

The first line is identifying and automatically generating security properties for hardware designs. I developed a semi-automatic approach, SCIFinder [112], that uses known processor errata and machine learning techniques to identify security properties. The security properties are useful to protect the hardware design they are derived from, but repeating the whole process for each new design can be tedious. Thus, I built a tool, Transys [113], that automatically translates security properties from one design to similar designs to reduce the effort in developing security properties. The second line is automatically generating exploit programs. I developed an end-to-end tool, Coppelia [111], that automatically generates exploit programs. The core of Coppelia is a hardware-oriented backward symbolic execution engine. It helps designers better analyze, understand, and assess the security threat for vulnerabilities in processor designs.

1.1 Thesis Statement

In the hardware design process, security validation of open-source RISC processors designs can be automated through mining security assertions with machine learning and known security errata, translating security assertions using static analysis techniques, and generating exploit programs using backward symbolic execution.

1.2 Developing Hardware Security Properties

The current practice for hardware designers and security experts to write security-critical properties is through manually studying the processor's Instruction Set Architecture (ISA) and its documentation, such as hardware specification sheets and user manuals. This process is time consuming, error-prone, and easy to miss important security properties. To address this challenge, we introduce SCIFinder in Chapter 3.

SCIFinder semi-automatically generates Security-Critical Invariants (SCI) for processor designs. Our first insight is that hardware security essentially concerns a core set of processor functionality (e.g., exception delivery, protection rings) that the upper-level software must depend on; any bugs in this subset could result in security compromises. Thus, we can first automatically generate a large set of processor invariants that describe all aspects of software-visible processor behavior by exercising the processor with a variety of programs. Then security-critical invariants can be algorithmically identified by checking the generated invariants against known software exploitable vulnerabilities. Our second insight is that SCIs tend to have common features such as the registers and flags used in them, and machine learning techniques can be used to infer additional security SCIs based on the existing ones.

Specifically, SCIFinder starts with obtaining processor execution traces by simulating the processor's register-transfer-level (RTL) design. To ensure wide coverage of the processor states, it runs a variety of software programs. SCIFinder then derives invariants within and across these traces by using a modified version of Daikon (as the original Daikon is not specifically designed for hardware). Among the derived invariants, SCIFinder uses known processor errata to differentiate the security-critical invariants from the purely functional ones. SCIFinder relies on a human to identify which errata are security-critical. For each security-critical erratum, we craft an exploit program that triggers the vulnerability. Then SCIFinder generates execution traces from executing the exploit program on both the buggy processor and the patched processor. SCIFinder further identifies those invariants that get violated in the execution trace of buggy processor but not the patched processor's as security-critical. In the last step, SCIFinder uses machine learning techniques to infer additional security-critical invariants. SCIFinder models the probability that an invariant is non-security-critical as a function of its measured features.

SCIFinder is evaluated on the OR1200 RISC processor. It identified 19 (86.4%) of the 22 manually crafted security-critical properties from prior work and generated 3 new security properties not covered in prior work. The generated assertions were tested against 14 new vulnerabilities adapted from real-world AMD processor errata, and find the assertions stop 12 (86%) of these bug-based exploits. The following invariant shows an example of security property that SCIFinder identifies (related to privilege-escalation):

$$I \doteq risingEdge(\texttt{l.rfe}) \rightarrow \texttt{SR} = orig(\texttt{ESR0})$$

This invariant states that when returning from an exception (indicated by the `l.rfe` instruction), the status register (`SR`) should be correctly updated with the value it had before the processor entered the exception handler. `ESR0` stores that value. The $orig(\texttt{ESR0})$ denotes the value of `ESR0` before `l.rfe` is executed, while `SR` denotes the value of `SR` after the `l.rfe` instruction is executed.

The unique contribution of SCIFinder is that it demonstrates the feasibility and benefits to derive hardware security invariants from software-exploitable bugs and in turn use the derived invariants to protect the hardware against a class of vulnerabilities and zero-day attacks.

1.3 Translating Hardware Security Properties

A comprehensive set of properties describing the security requirements is useful for validating the security of hardware designs. However, developing security properties for hardware designs is challenging, and this effort needs to be repeated for each version of the design and each new design. The insight is that if we can leverage existing security properties and translate them from one design to another, we can reduce the effort in developing security properties.

To this end, I introduce Transys in Chapter 4, a tool for translating security critical properties written for one hardware design to analogous properties suitable for a second design. Transys takes a set of security properties that already developed for one hardware design, and two implementations of hardware designs as inputs. It works in three steps: first, it maps variable names by using a statistical matching method to semantically correlate variables between the two designs; second, it adjusts arithmetic expressions by using a program analysis-based re-structuring phase to make assertions functional in the new design; finally, it refines the constraints to iron out remaining issues with the constraints of the result properties. Trnasys is evaluated for translating 27 assertions written in a temporal logic and 9 properties written for use with gate-level information flow tracking, across 38 AES designs, 3 RSA designs, and 5 RISC processor designs. It successfully translates 96% of the properties (the output properties are validated by model checking). Among these, the translation of 23 (64%) of the properties achieved a semantic equivalence rate of above 60%. The average translation time per property is about 70 seconds.

Transys is the first tool to translate security properties across hardware designs. Applying Transys to hardware designs also suggests that hardware designers can keep the assertions of a single project in sync

with the design. Over the life cycle of the hardware code base, designers could use Transys to suggest updates to properties they had affected by changing the target design.

1.4 Generating Hardware Exploit Programs

Formal static analysis and simulation-based testing are powerful to help find many potential bugs in hardware designs. However, reporting a potential bug is only the first step. A reported bug, in the form of some violated assertion, could turn out to be a false alarm due to errors in the assertion itself, the formal method tools, or the simulation environment. Even if a reported violation is a true bug, it is unclear whether this bug poses security risk or not, and if so, how it may be exploited. Determining true bugs and their security implications for found violations is time-consuming, a process that involves hardware designers, security experts and formal method experts. To address the gap in the current practice, we developed Coppelia that provides an end-to-end solution that helps hardware designers systematically find potential bugs and assess the security implication of the vulnerabilities.

Our key strategy is to automatically generate exploit programs for hardware designs. In this way, hardware designers can narrow down and contextualize the true vulnerabilities in the set of reported violations. Moreover, when hardware designers fix some vulnerabilities, they can check whether exploit programs can still be generated for the updated hardware design and use this fact to validate the correctness and security implication of the vulnerability fix. Symbolic execution is a promising technique for our scenario because symbolic execution can generate a concrete input that leads to the assertion-violation path. The bug-triggering input will be essential for generating exploit programs. While symbolic execution has been extensively explored in the software world and successfully applied to relatively large software, how to perform symbolic execution on hardware designs is under-explored. Two characteristics of hardware designs require rethinking the standard symbolic execution. First, the symbolic execution of a hardware design represents an exploration of the design for a single clock cycle, but hardware executes continuously, and security vulnerabilities may only become apparent many clock cycles after the initial state. Second, security properties developed for hardware designs capture the semantics of particular signals and their connecting logic. Finding such violations is akin to finding a needle in a haystack.

Coppelia designs a novel hardware-oriented backward symbolic execution strategy to enable a targeted search through the hardware design space for rare assertion violations. To begin, Coppelia translates the

hardware design from an hardware description language implementation to C++. After translation, Coppelia generates testbenches with security-critical assertions. To build a trigger to the exploits, Coppelia leverages KLEE and implements backward symbolic execution. Starting at the point of an assert statement, Coppelia symbolically executes the design backwardly, searching for a path from an assertion-violating state back to the reset state. To handle symbolic execution across multiple clock cycles, Coppelia adopts a cycle stitching method that can generate a complete sequence of instructions that triggers a bug starting from the reset state. Coppelia also uses several optimizations to tackle the challenge of state exploration in hardware. To better contextualize and analyze the security threat, Coppelia goes beyond triggering the vulnerability. It adds a program stub to complete the exploit. These program stubs are generated according to the category of the security-critical assertion violated. Coppelia is evaluated on three CPUs of different architectures: it generates exploits for 26 out of 29 known vulnerabilities in these CPUs, all of which are successfully replayable on an FPGA board; it also finds 4 new vulnerabilities along with exploits in these CPUs.

1.5 Organization

The remaining of this thesis is organized as follows: Chapter 2 discusses the background and prior work that is related to this thesis. Chapter 3 and 4 presents our work in developing security properties for hardware designs. Chapter 3 describes the design and implementation of SCIFinder, together with an evaluation of its effectiveness in generating security properties for detecting both known and unknown security bugs on the OR1200 processor. Chapter 4 introduces Transys, presents the three phases for translating the security properties across hardware designs, and reports its uses in translating both temporal and informatino flow tracking assertions in AES, RSA, and RISC designs. Chapter 5 presents our work in using the security properties for security validation for hardware designs. Chapter 5 describes Coppelia and the details of our backward symbolic execution strategy, and shows an evaluation of using it for two different processor and architectures. Chapter 6 summarizes the contributions and concludes this dissertation.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter I place my proposed methods and tools in the wider context of hardware security defense and validation methods by describing related work and background information. I first describe several research directions in defending against different hardware vulnerabilities, including novel hardware features and mechanisms, information flow security, hardware security validation, and novel hardware description languages. Then I discuss research in extracting and mining assertions for hardware designs, as well as software designs. These research are related to our work in developing hardware security assertions. Finally, I present the background of symbolic execution, and its application in software security research and hardware validation, which are related to our work in generating exploit programs for hardware.

2.1 Approaches for Protecting Vulnerable Hardware

Major types of attacks to hardware include physical attacks, privileged software attacks, software attacks on peripherals, hardware trojans, and cache timing attacks. I describe detection methods and countermeasures to these attacks in this section.

2.1.1 Secure Processors

To defend against physical and software attacks to hardware, as well as to protect the integrity of sensitive data, different mechanisms and features have been proposed. I categorize them into academic solutions and industry solutions.

Academic Solutions

The execute-only memory (XOM) architecture [73] proposes the approach of executing sensitive code and data in isolated compartments. XOM tags each cache line with a XOM identifier, and disallows memory accesses to cache lines by compartments whose identifier mismatches the current one. AEGIS secure

processor architecture [95] protects the integrity and privacy of applications from physical attacks and software attacks. AEGIS uses physical random functions and off-chip memory protection to defend against physical attacks. It uses a security kernel to isolate compartments. Bastion [30] relies on a trusted hypervisor to provide secure compartments for running applications. The Ascend secure processor [89] leverages the Oblivious RAM (ORAM) technique which obfuscates address buses by reshuffling memory as it is accessed to defend against attacks that learns private information from DRAM memory access patterns. The Capability Hardware Enhanced RISC Instructions (CHERI) [104], based on RISC ISA, introduces a hybrid capability model to mitigate memory related vulnerabilities. CHERI includes a capability coprocessor and tagged memory to support a self-contained virtual capability system.

Industry Solutions

ORWL [5] is an open-source processor aiming to defend against physical attacks. ORWL uses the outer shell and the sensors to detect physical attempts to tamper encrypted data, and once detected, ORWL destroys all hardware encrypted data instantly. IBM SecureBlue++ [6] is an architecture that protects confidentiality and integrity of information in an application against other applications, as well as against attempts to introduce malware inside the application. An application's information is encrypted whenever it is outside the processor and other software cannot access the application's cleartext information inside the processor. ARM TrustZone [1] is a hardware architecture that creates an isolated environment to allow confidentiality and integrity of code and data. TrustZone conceptually partitions a system's resources between a secure world for the security subsystem and a normal world. The secure world resources cannot be accessed by the normal world components. TrustZone also provides hardware extensions that enables code execution from both the secure world and the normal world in a time-sliced fashion, allowing high performance security software to run alongside the normal world operating environment [1]. Intel Software Guard Extensions (SGX) [79] enables processors to execute code in an enclave that is isolated from the untrusted software, and also provides a software attestation scheme that allows a remote party to authenticate the software running inside the enclave. AMD Secure Encrypted Virtualization (SEV) [63] aims for protecting data in DRAM against physical threats as well as threats from virtual machines or hypervisors. SEV isolates a full virtual machine (VM) by tagging and encrypting all code and data to prevent data from being used by anyone other than the owner.

2.1.2 Information Flow Security

Hardware is capable of leaking information through timing, power, thermal and other covert channels. Information flow security in hardware uses dataflow tracking to track the flow of untrusted network, file and user inputs through memory. Information Flow Tracking logic can be added at the gate level [99] or register transfer level [14] of a hardware design, and can capture timing flows [13, 78] or data flows [85]. While there is a trade-off to be made between precision and performance [19, 105], these techniques can demonstrate whether sensitive inputs to a design, e.g., the key material input to a cryptographic core, is directly or indirectly visible in the output signals. As with language based verification, this approach can provide strong guarantees, but also requires modifying the original design, either by adding tracking logic or, as in the case of CPUs, redesigning from the ground up to provide provable isolation between software contexts [97, 98]. Efficiently tracking information flow in hardware has been studied [41, 101, 33, 99, 14], but this approach often requires modifying or extending the hardware architecture. Cherupalli et al. proposed a gate-level symbolic simulation tool for information flow for particular IoT applications [35].

2.1.3 Property Driven Hardware Security Validation

Assertion Based Verification for Security

Assertion based verification (ABV) is the form of testing in which assertions added to the design encode functional correctness properties, such as a request–acknowledge pattern [50]. Once assertions are added, simulation-based testing or formal static analysis may then be used to search for violations of the assertions. Both approaches have gained wide acceptance and commercial tools are available, including Cadence [2]. In simulation-based testing the aim is to achieve high code coverage using many test cases [103]; assertion violations that exist along untested paths will not be discovered. In formal static analysis the design is unrolled some number of cycles and the state space is methodically explored [25, 24, 46]. Test cases are no longer the limitation, but rather the size of the state space limits how far the design can be unrolled. Both simulation-based testing and formal static analysis are performed at design time.

Properties can be encoded as assertions and added to the design under review, at which point standard ABV techniques can be used to find property violations. Historically, functional properties were used, but recently security properties have been considered. These security properties are manually developed [21, 22, 59]. SecurityCheckers [21, 22] and SPECS [59] developed temporal logic security assertions from the

manual or specification of the RISC processors, e.g. the return from exception instruction causes the program counter (PC) to be loaded from EPCR and the status register (SR) to be loaded from ESR; unspecified custom instructions are not allowed. Information flow properties have also been manually developed [60, 62]. An example information flow property in an AES core is that the secret key should not flow to the ciphertext ready signal otherwise there would be a timing side channel. There has lately been a call for "property driven hardware security" [60, 61, 64] that advocates building security specifications into the hardware design workflow, automating the process of doing so, and developing quantifiable measures of security.

Protecting Buggy Hardware Post Deployment

Some bugs may persist to the final product and work must be done to mitigate the resulting harm. Solutions include adding redundancy to the hardware design to protect against random errors [16] and checking processor state transitions against a known set of errata signatures [83, 91, 92]. Once discovered (post-deployment), some bugs are amenable to being patched by software. These methods include micro-code patching and binary translation [57, 100, 80]. A hybrid software–hardware approach adds additional hardware to the design in the form of assertions and uses software to handle any assertion failures that occur at run-time [40, 39, 59].

2.1.4 Language Based Approaches

There is a body of work on developing new or extending current hardware description languages for secure hardware development. One language based approach uses typed hardware description languages, which can enforce security policies by construction [72, 71, 109, 110, 48]. Caisson [72] is a hardware description language (HDL) targeting statically-verifiable information-flow secure hardware designs. Caisson enforces security policies by using a lease mechanism between trusted and untrusted states during execution. Sapper [71] compiles code to synthesizable Verilog that enforces security policies by automatically deriving and inserting dynamic security checks. SecVerilog [109, 110] extends Verilog with security labels to mitigate external timing channels. SecVerilog relies on hardware designers to annotate the security lables and to distinguish between benign timing variations and and those carrying confidential information. ChiselFlow [48] is a security-typed HDL embedded in Scala. ChiselFlow reduces its trusted component by compiling to a small intermediate language that is responsible for the enforcement of security policies, and reduces programmer efforts by providing label inference for internal signals inside hardware modules. Although these

approaches can prove that a hardware design meets the security policies, they cannot verify those designs not already implemented in these languages. A second language based approach uses a formally defined language to first specify a policy and then refine the specification to a provably correct design [102, 23, 37]. Fe-Si [23] is a functional language that is a deterministic subset of Bluespec. Fe-Si embeds Coq as a meta programming tool that allows it to prove the correctness of the circuits. Kami [37], also based on Coq, verifies circuits by proving that each hardware module refines its specification modules.

2.2 Developing Security Specifications

2.2.1 Extracting assertions from hardware designs

In ABV, assertions are added to the hardware design, typically written in a hardware description language (HDL) such as Verilog or VHDL, and the design, with assertions added, is simulated with random or selected inputs. Any assertions that fire during simulation point to a bug in the design. Considering properties beyond those critical to security, there is a body of work on specification mining from hardware designs.

The IODINE [54] tool automatically extracts ABV assertions from designs. The IODINE tool looks for possible instances of known design patterns, such as one-hot encoding or mutual exclusion between signals, and creates assertions that encode the found patterns [54]. Change et al. examines frequently occurring patterns in a number signals that the user deems important based on knowledge of the domain, and then generates assertions based on the most frequent patterns [32]. Avoiding the dependence on human choice, the GoldMine system used a combination of static analysis of the hardware design to guide the data mining on the simulation data providing a more robust set of assertions [58]. While these techniques are not concerned with finding security properties, they provide lessons on how to scale assertion extraction effectively. Our work goes beyond just extraction and focuses on *finding* security critical assertions.

2.2.2 Data Mining for Security Properties of Software

Security properties in software have been found using human specified rules [96], by observing instances of deviant behavior [86, 81, 47], or by identifying instances of known bugs [107].

Tan et al. looks for patterns of security checks and the sensitive operations protected by those functions, and subsequently searches the rest of the code base for unprotected sensitive operations [96]. Moving away from human specified rules, more recent work explored extracting the latent security rules inherent in the

code by modeling the normal behavior in a more descriptive manner [86, 81]. Perkins et al. models software behavior [47] by monitoring registers and memory locations in order to create invariants that defined normal program behavior. By examining which invariants are violated during erroneous execution, they were able to define and identify deviant behavior [86]. Another technique that comes from the concept of normal and deviant behavior is modeling abnormal behavior from a previously known vulnerability or bug. Yamaguchi et al. established a pattern based vulnerability extrapolation where they define deviant behavior and examine the rest of the code base to identify similar abnormal behavior [107].

2.3 Symbolic Execution

2.3.1 Symbolic Execution Technique

Symbolic execution is a technique of analyzing a program to determine what inputs cause each part of a program to execute [65]; it is often used to check which inputs cause assertion failures. At the beginning, the inputs to a program are marked as symbolic. Then the program executes step by step, building constraints on the symbolic variables based on the program operations. When reaching the branch statements, symbolic execution forks into two paths. Each path gets assigned a copy of the program state and the path constraint at the branch statement. When paths terminate, symbolic execution uses the accumulated path constraints to determine whether assertions fires by solving the path constraints together with the assertions with a solver (e.g. stp [52], z3 [43]). Meanwhile, symbolic execution can generate concrete values of the inputs that cause errors to occur.

Directed symbolic execution [76] and execution synthesis [108] use guided symbolic execution to increase the probability of executing paths of interest. In software, backward symbolic execution has been studied to solve the goal-reachability problem [31, 76]. Otter [76] developed the call-chain-backward symbolic execution which begins at a target line and proceeds backward to the start state. Application to real-world software raises many challenges such as complicated arithmetic (such as floating point), external method calls, and data-dependent loops [45, 31].

Researchers also explored different ways to mix concrete and symbolic runs [28, 27, 29]. The symbolic engine always executes concretely on operations with concrete values only, which makes symbolic execution possible to reason over complex operations. S2E [36] presents a systematic approach to consistently cross the

symbolic and concrete boundaries. It interleaves portions of code that are concretely run with fully symbolic phases, which is done carefully to preserve the meaningfulness of the whole execution.

2.3.2 Automatic Exploit Generation

Software symbolic execution has been widely explored [28, 53, 29, 27, 42, 65]. Symbolic execution is also often used in automatic exploit generation for software [26, 56, 17, 18, 29]. Typically, vulnerabilities (e.g., buffer overflows) are first found through static or dynamic analysis, and then program input satisfying identified constraints are found. We tackle similar problems but differ in that we target the hardware domain, which requires a stateful analysis across multiple clock cycles to generate a series of input for hardware, instead of a single input as in software.

The problem I tackle in this thesis is similar in nature to the problems in the software domain of automatic exploit-based generation [17] and patch-based exploit generation [26]. In those cases, as in ours, strategies are needed to focus the search toward an exploitable bug. Brumley et al. [26] focus their search by calculating the weakest precondition of a vulnerable state, and my use of backward symbolic execution is, in effect, a method for calculating a precondition, although not the weakest. However, the approach we take is geared toward handling a stateful hardware design, which may require a sequence of inputs to find a bug, rather than a single input.

2.3.3 Hardware Symbolic Simulation

Applying symbolic execution to hardware designs for verification and testing has also been studied [75, 82]. STAR [75] is a functional input vector generation tool combining symbolic and concrete simulation for RTL designs over multiple time frames. It provides high range statements and branch coverage, but is limited by the sequential depth (around 6 cycles) [75]. PATH-SYMEX is a forward symbolic execution engine that takes in ANSI-C interpretation of the RTL code [82]. Its application is limited to small RTL designs. With the purpose of reproducing bug exploits, my work focuses more on the sequential depth of the exploration with Backward Symbolic Execution Scheme and can be easily integrated to the current industrial verification flow by leveraging software verification methods.

CHAPTER 3
DEVELOPING HARDWARE SECURITY PROPERTIES

In this chapter, I present the details our semi-automatic approach, SCIFinder, for developing hardware security properties outlined in Section 1.2. I first give an overview of the proposed approach in Section 3.1. I then describe the design and implementation details in Section 3.2 and 3.3. Finally, I show our evaluation of SCIFinder on the OR1200 processor in Section 3.4.

3.1 Overview

Previously, researchers develop security assertions manually by studying the processor's instruction set architecture (ISA), identifying properties of the ISA that are critical to the security of software running on the processor, and encoding those properties as assertions [59, 21, 22]. The process requires human expertise and judgment. The process can be tedious and time consuming. Moreover, some properties that are important for security are obscure and unlikely to be identified. Furthermore, because the instruction manuals describing an ISA may be incomplete and ambiguous there are important properties which even the most thorough perusal of the ISA will be unable to uncover.

However, there is a benefit to having a human in the loop. The line between a security property and a purely functional property is blurry. Some properties seem obviously critical to security. As an example, each of the above cited works ([21, 59, 22]) includes an assertion that the supervisor signal is set only in response to a small number of well defined events. Other properties, such as the one(s) violated by Intel's infamous FDIV bug [8, 15], feel safely characterized as purely functional.

SCIFinder is a methodology and tool chain for semi-automatically generating a set of security-critical processor invariants that can be encoded as synthesizable assertions. Our approach is informed by three observations. First, detailed information about processor invariants may not exist in any specification documents; this information can only be learned by studying a running processor. Second, human expertise is still needed for, and well suited to, distinguishing security concerns from purely functional ones. And, third,

properties that are critical to security tend to have commonalities between them, for example, they concern state that is critical to security, such as the supervisor signal.

Rather than try to cull security-critical properties from the ISA, we instead generate, automatically, a large set of processor invariants that describe all aspects of processor behavior and then categorize each invariant as critical to security or not. While we wish to use human judgment to guide this process, we do not want to burden the designers with the task of combing through hundreds of thousands of invariants to perform the categorization. Therefore, we have developed two ways to algorithmically differentiate security-critical invariants from functional invariants: 1) Use published processor errata that can be shown to pose a threat to security to drive the categorization, and 2) Use statistical analysis techniques to classify the invariants. The benefit of using published errata as a starting point is the potential to create assertions with high value. These assertions will at a minimum catch actual bugs that have historically had a deleterious effect on security. The approach has the potential to be stronger than that, however. If the assertions are well crafted, they capture not just the absence of a particular bug, but the presence of a desired security property. These assertions will detect any bug that violates the protected property, even if the bug itself is entirely different from the one that first inspired the assertion. Still, the errata-based approach is limited to finding properties that have at one point been violated by a known security bug. In our second approach, we explore to conjecture that once a human has identified points along the security–functionality boundary, machine learning techniques can be used to automate the classification of additional invariants. In this way new security-critical invariants can be identified.

3.2 Design

As shown in Figure 3.1, SCIFinder has four phases. The first phase is invariant generation. We observe a processor executing a variety of programs to collect a set of likely processor invariants defined over software-visible processor states. The second phase is to classify each collected known design errata, using human expertise and judgment, as either a functional bug or a potential security vulnerability. The third phase is security-critical invariant (SCI) identification. We identify SCI as those invariants violated by the security vulnerabilities from the second phase. The fourth phase is SCI inference. We apply machine learning techniques to find additional SCI in the set of processor invariants. We next discuss phase one, three and four in detail. Details of our implementation of phase two are in Section 3.3.1.

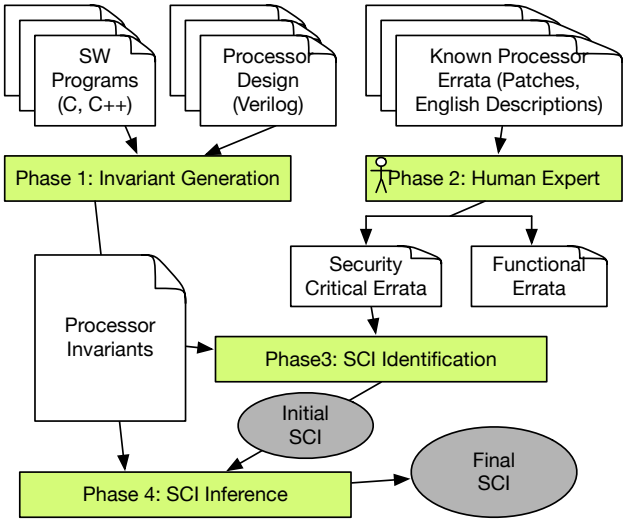

Figure 3.1: Workflow of SCIFinder.

3.2.1 Invariant Generation

In the first phase, we collect a set of likely processor invariants. We use a modified version of Daikon, a dynamic invariant generation tool, and execute the processor design in simulation with a variety of software running on it. We operate at the ISA level: we track software-visible states and consider execution of an instruction to be a single step of execution. We wish to collect meaningful processor invariants. We do this by generating a large number of processor execution traces covering as many processor states as possible, and then observing invariants within and across these traces. Some of the generated invariants are potentially security critical and will be identified as such in the following phases (see Section 3.2.3 and Section 3.2.4).

Execution Traces

We obtain the processor execution traces by simulating the processor's register-transfer-level (RTL) design. During simulation we track architectural signals and selected register values of the RTL design at each instruction boundary. To provide as much breadth as possible, we run a variety of programs including SPEC benchmarks, a Linux boot, and scientific computations (see Section 3.4.1). Our execution traces must, at a minimum, cover all the instructions in the ISA, including system calls, bit-rotation operations, word-extension operations, and interrupts and exceptions.

Daikon

From the execution traces, we use Daikon, a dynamic invariant detection tool, to gather meaningful invariants [47]. Daikon has an instrumenter and an inference engine. The instrumenter records information about variable values as a program executes, and the inference engine reads the traces produced by the instrumenter to generate invariants.

Daikon is not specifically designed for hardware, and we adapted it to suit our needs. Daikon is intended to learn software-level invariants: procedure pre- and post-conditions, class invariants, and data structure invariants. These are not directly applicable to processor execution traces; we extend Daikon to suit our hardware use case (see Section 3.2.1). Patterns often seen in hardware design, such as bit-packing several flags into a single register, are unknown to Daikon. We develop new invariant patterns that capture such non-linear relationships between variables (see Section 3.2.1). Certain processor design optimizations, such as delay slots, need to be carefully handled (see Section 3.2.1). The invariants generated by Daikon contain redundancies. Our SCI will be enforced on processors dynamically and should be concise to avoid overhead. We introduce optimizations to remove redundancy (see Section 3.2.2).

Invariant Variables

Daikon produces invariants in the form of procedure pre- and post-conditions, as well as class and object invariants. The latter two are not applicable to our hardware setting, but the first two can be adjusted to suit our needs. We are interested in ISA-level properties that hold as the processor executes; by observing processor state before and after the execution of each instruction, we can use Daikon to develop a set of pre- and post-conditions for each instruction. The pre-conditions describe properties that always hold when a particular instruction executes, and the post-conditions describe properties that always hold at the conclusion of a particular instruction, provided the pre-conditions hold. We modify Daikon's instrumenter to extract trace data from the execution logs produced by the simulation. It outputs variable values before and after each instruction is executed. The set of variables tracked should be inclusive enough for the inference engine to infer meaningful invariants including those critical to security. On the other hand, the variable set should be small enough to make invariant inferences computationally feasible.

We make the same design decision as prior work in dynamic processor verification. We include all the variables at the ISA level, that is, all registers and signals that are visible to software: all general purpose

registers (GPRs), all special purpose registers (SPRs), flags, data and address of the memory subsystem, target registers, and immediate values of the instruction. The ISA level represents a trade-off between complexity and completeness: the microarchitectural signals and registers that make up the processor implementation are abstracted away, reducing complexity. In exchange, we lose information that may be useful for constructing security properties. As an example, prior work found that an error in the processor's pipeline that modifies an instruction in flight would not be caught because the processor remains self-consistent at the ISA level. Extending our approach to capture microarchitectural information is the likely solution to this limitation. Our optimization strategies (Section 3.2.2) are a first step toward making such an extension feasible.

Invariant Patterns

Daikon invariants make comparisons between variables or between a linear combination of variables. We found this to be insufficient for capturing important properties at the hardware level. For example, a common pattern in hardware is for a 32-bit register to act as a record containing 32 (or fewer) independent bit flags. To address this, we made the Daikon instrumenter configurable. This allows users to create derived variables that can be used to define more complex invariants. For example, a derived variable that extracts bits from its parent variable can be used to generate a property indicating whether the flag that handles control flow is correctly set.

Processor Complexity

In many architectures, including the one in which we implement our tool, the processor always executes the instruction in the branch delay slot – the instruction directly after a control flow instruction (*i.e.*, branch or jump). A naive observation would infer the invariant that the next program counter (`NPC`) after a control flow instruction is equal to the current program counter plus four ($\texttt{PC} + 4$), and while true, this does not capture the important property that control might move to the target of the branching instruction after executing the instruction in the delay slot. Similarly, the naive observation would be unable to infer an invariant about the `NPC` register for any other instruction. Normally a (non-branching) instruction obeys the invariant $\texttt{NPC} = \texttt{PC} + 4$, but if the instruction ever appears in a delay slot, its `NPC` would be the address of the branch or jump target.

To allow for the generation of meaningful invariants about control flow, we treat the control-flow instruction plus the one in the delay slot as a single entity. The OpenRISC architecture, the architecture we

$$
\begin{aligned}
EXPR &\doteq EXPR_1 \mid EXPR_2 \\
EXPR_1 &\doteq OPER\ OP_1\ OPER \\
EXPR_2 &\doteq OPER\ \texttt{in}\ \{imm, imm, \ldots\} \\
OPER &\doteq VAR \mid \texttt{orig}(VAR) \mid imm \\
OP_1 &\doteq = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
VAR &\doteq \texttt{GPR} \mid \texttt{SPR} \mid flag \mid mem\_address \mid VAR \times imm \\
&\quad \mid \texttt{not}\ VAR \mid VAR\ \texttt{mod}\ imm \mid VAR\ OP_2\ VAR \\
OP_2 &\doteq \texttt{and} \mid \texttt{or} \mid + \mid -
\end{aligned}
$$

Figure 3.2: The grammar of invariant expressions. `orig()` indicates the value of a variable before the instruction executes; the default is the variable value after the instruction executes. `imm` refers to an immediate value. `in` indicates set inclusion. Boolean operators are all bitwise operators.

use in our implementation, has a single branch delay slot, so the branching instruction and the instruction in the delay slot is treated as one instruction. For those architectures with double branch delay slots (*e.g.* MIPS-X), the branching instruction and the pair of instructions following can be treated as one block.

Structure of the Invariants

From the data generated during executions we use the Daikon generator to create invariants of the format

$$I \doteq risingEdge(INSN) \rightarrow EXPR,$$

where $risingEdge(INSN)$ represents the execution of an instruction, and $EXPR$ is an expression over the tracked variables. Figure 3.2 shows the grammar for expressions in our set of invariants.

As the execution of each instruction can take several cycles, we only consider the variables as they enter and leave the instruction. We designate the value of the variables before the instruction begins with the $orig()$ prefix, and any variable without the $orig()$ prefix indicates the value after the instruction has been completed.

To give an example, we show the invariant that describes the property that privilege should correctly de-escalate:

$$I \doteq risingEdge(\texttt{l.rfe}) \rightarrow \texttt{SR} = orig(\texttt{ESR0})$$

This invariant states that when returning from an exception (indicated by the `l.rfe` instruction), the status register (`SR`) should be correctly updated with the value it had before the processor entered the exception

handler. ESR0 stores that value. The $orig(\texttt{ESR0})$ denotes the value of ESR0 before the `l.rfe` instruction is executed, while SR denotes the value of SR after the `l.rfe` instruction is executed.

We generate approximately 106,000 unique invariants which form a model describing normal processor behavior. Inherently the model we generate represents the current implementation of the processor; the correctness of our model is tied to the correctness of the implementation and design of the processor. Any errors or bugs in the specification and implementation will be reflected and remain undetected.

3.2.2 Optimization

We perform the following optimizations to rewrite the invariants in a more concise form.

Constant Propagation

Equality-to-constant invariants (*e.g.* $A = 0$) can be used to reduce the complexity of other invariants. Our constant propagation optimization is similar to the compiler optimization technique of substituting constant values at compile time [10, 11]. The propagation is performed iteratively so that any new equality-to-constant invariant can be used in subsequent substitutions.

We parse the invariants into expression trees, initialize a worklist with all the invariants, and construct a variable–value map. Then we iterate through the worklist, and for each invariant, we use the variable–value map to substitute constants for expressions where possible. For any new equality-to-constant invariant after substitution, we update the variable–value map and remove that invariant from the worklist. The process continues to iterate through the worklist until there are no new equality-to-constant invariants.

Deducible Removal

The deducible removal optimization pass removes the invariants that can be deduced from several other invariants. For example, $D < C$ is deductible from $A + B > D$ and $C > B + A$. Full deducible removal is equivalent to taking the transitive reduction of the binary relation; we remove invariants with transitive operators that can be derived from other invariants. Daikon invariants do not have complex expressions on both sides of an inequality, thus we do not perform deducible removal for cases similar to the following: $A + B > C + D$ is deducible from $A > C$ and $B > D$.

We first canonicalize invariants with transitive operators into the form of $lhs\ OP\ rhs$, where $OP \in \{>, \geq, ==\}$ ($<$ and $\leq$ will be converted accordingly), and lhs (rhs) is a sorted postfix string of the left

(right) hand side of the expression. We build a directed acyclic graph (DAG) for all generated invariants for each OP. For each invariant $I \doteq lhs\ OP\ rhs$, we add the lhs and rhs as vertices in the DAG, and an edge directed from lhs to rhs. We then compute transitive reduction of the graph to get the minimum set of invariants with the same reachability relation.

Equivalence Removal

In this optimization pass we remove redundant invariants. We cluster invariants that are logically equivalent to each other in the same class and keep only one invariant from an equivalence class. For instance, the following invariants would be grouped into two equivalence classes and only two would be retained: $(A = B), (B = A); (C + B * D > F), (F < C + D * B), (D * B + C > F)$, etc.

We determine invariant equivalence by putting every invariant into a canonical form, using the same form as used in the deducible removal pass.

3.2.3 Security-Critical Invariant Identification

The third phase relies on one of our key observations: security-critical errata are vulnerabilities precisely because they violate some underlying security property. We can use the errata to identify security-critical invariants – those that are violated when a security-critical erratum is triggered. SCI identified in this phase will protect against not just the particular vulnerability used to find it (and presumably that vulnerability has been patched in the latest version of the processor), but also against other, unknown bugs that violate the same invariant. In Section 3.4 we discuss how often this occurs within our test data.

Once we generate the set of invariants that describe normal processor behavior, our goal is to identify the subset of invariants that are crucial for security – the security critical invariants (*SCI*). One possible solution might be to use human expertise to develop a set of rules to apply. However, the rules may lack diversity: only the types of properties that a human has thought of will be represented, and prior work has shown that this approach can leave gaps in the resulting set of security properties [59]. In addition, the set of rules has to be small enough that the human can reasonably create it (*i.e.*, there cannot be an individual rule for every generated invariant), but the rules themselves cannot be too general or they risk admitting too many invariants into the set of SCI.

For these reasons, we took an empirical approach to identifying SCI in the set of generated invariants. We leverage security errata that have existed in the processor design at some point in its development lifecycle.

By definition, a program that triggers the bug must exhibit some unusual states that do not obey processor specifications. By checking which of our generated invariants are violated in the execution of a triggering program, we can approximately obtain the SCI. Because the errata are essentially programming bugs, they may occur anywhere in the design and potentially provide a more varied set of SCI than human-generated heuristics do. Because the identified SCI come directly from a security vulnerability, we know they are in fact critical to security.

To be specific, when we find a security bug from the published processor errata list or bug trackers (Section 3.3.1), we first implement the defect in an open source processor (in Verilog), creating a buggy processor. We then write a program that triggers the vulnerability, execute it on the buggy processor, and record its execution trace. Given the previously generated invariant set and the execution trace, our tool will automatically sort through the execution trace to see if *at any point* an invariant has been violated. Any violated invariants are then added to our set of candidate SCI.

Since the initial set of generated invariants may contain false positives, invariants identified as SCI in this step may not be true SCI. In order to remove these false SCI, we run the same trigger program on a correctly implemented processor (with the security defect removed) and perform the same steps of recording execution traces and checking for invariant violations. The set of violated invariants found in this phase are false positives, *i.e.* they are not true processor invariants, and can be eliminated from the final set of SCI.

One possible concern is that identified SCI are applicable only to one particular bug. In our experiments, we found that a single SCI can be identified from different bugs and it can stop multiple bugs (see Section 3.4.2). This means the SCI we extract from a particular bug are applicable to a class of bugs, a class defined by the invariant(s) violated.

3.2.4 Security-Critical Invariant Inference

In the fourth phase, we use machine learning techniques to identify additional invariants as security critical. Once we have identified a set of SCI using security-critical bugs, we apply machine learning techniques to infer which other invariants should be labeled security-critical.

The core component of the Inference step is a logistic regression model, which can be applied to classify invariants as security critical or non-security critical. We model the probability that an invariant is non-security critical as a function of its measured features. In particular, we adopt the penalized logistic regression model with elastic net penalty [114]. There are two reasons: 1) In this application the number of measured

features is larger than the number of observations (invariants). Penalized logistic regression approaches have successfully extended traditional regression models for improved accuracy in such circumstances [114]. 2) This model excels in parameter interpretability [77]. As each feature included in the model incurs a cost or penalty, it can also be used to understand which of the features are critical to security.

Here, we specify the details of the regression model. We fit the model with the elastic net penalty using the glmnet [51] package in R.

As in the typical regression framework, we let $y_i \in \{\text{security-critical}, \text{non-security-critical}\}$ be the class label for invariant i. Since y_i is binary and hence a Bernoulli random variable, we model its probability, p_i, as follows.

$$\begin{aligned} p_i &= \text{Probability}(y_i = \text{non security critical}), \\ (1 - p_i) &= \text{Probability}(y_i = \text{security critical}). \end{aligned} \tag{3.1}$$

For invariant i, we let $\mathbf{x}_i$ be its set of measured features. In our context, the features are all the ISA-level variables (Section 3.2.1) such as general purpose registers, flags, memory contents and memory addresses, and also operators such as $>$, $<$, $\neq$.

Then, we relate p_i to $\mathbf{x}_i$ as,

$$\log(\frac{p_i}{1 - p_i}) = \mathbf{x}_i^T \boldsymbol{\beta} + \beta_0. \tag{3.2}$$

Here, $\boldsymbol{\beta}$ and β_0 are the vector of regression model coefficients and the intercept term, respectively, that are fitted with glmnet. The jth entry of $\boldsymbol{\beta}$ corresponds to the jth feature and explains that feature's contribution to the odds that invariant i is not security critical. β_0 is an intercept term giving the odds of being non security critical. When fitting the model, the objective is to learn the $\boldsymbol{\beta}$ and β_0 values that best describe the observed data.

We bootstrap this model using a small set of manually labeled invariants that contain both SCI and non-SCI. The constructed model can be used not only to predict whether a given invariant is likely an SCI but also to help hardware designers and security practitioners understand which of the features are critical to security based on the learned $\boldsymbol{\beta}$. For example, in our implementation only 24 of the 158 features have non-zero coefficients in the constructed models (see Section 3.4.3).

3.2.5 False Positives

False positives can occur in the final set of SCI in two ways. The first is that our tool generates an invariant that is not truly invariant. There are two potential sources for this type: 1) the Daikon tool itself; 2) inadequate test suites for invariant generation; and 3) the unintentional use of a buggy processor during the first stage. We minimize the first and second by tuning the parameters of Daikon to be conservative in finding invariants (see Section 3.4.1) and running many programs on our processor. (Increasing test coverage reduces the number of false positives.) The third source of false positive is a limitation of our tool. We rely on human experts to manually remove this kind of false positive from the final set of SCI.

The second type of false positive occurs when our tool classifies a non-security-critical invariant as security-critical. Reducing this type of false positives requires drawing a fine line to differentiate SCIs and non-SCIs, adding more labeled data, and refining machine learning models.

3.3 Implementation

Our tool is implemented mainly in Python. The exception is the SCI inference engine which is implemented in R and Matlab. As part of our evaluation we implement assertions enforcing the SCI on the OR1200 processor. This part of the work is implemented in Verilog.

3.3.1 Security-Critical Errata

We use potential security vulnerabilities to find security-critical invariants. We first collect bugs from the popular open source processors OR1200, LEON2, LEON3, OpenSPARC-T1, and OpenMSP430. Bugs are found from the processors' bugtracker and bugzilla sites, developers' mail archives, commits to the source repositories, comments in the source code, and published lists of errata. The bugs we collect are mainly in the core of the processor; bugs in peripheral devices such as UART, Debug Unit, and Ethernet are not included.

After collecting bugs, we manually select the bugs that may be classified as security critical: for each bug in the collection, we examine the patch and description to determine whether it is vulnerable to a security attack. In doing so we follow the same guidelines used by prior efforts in manually building SCI. Namely, we look for bugs that would allow an attacker to gain privileges to read or modify processor state that would not otherwise be allowed by the ISA or that would allow the attacker to subvert core functionality of the processor such as modifying the address in a load operation. The total number of bugs we collected is 185, of

those we deem 25 as security-critical. Of those 25, we successfully reproduced and modeled 17; 8 of them were not reproducible.

Bug No.	Synopsis	Source
b1	l.sys in delay slot will run into infinite loop	OR1200, Bugzilla #33
b2	l.macrc immediately after l.mac stalls the pipeline	OR1200, Bugtracker #1930
b3	l.extw instructions behave incorrectly	OR1200, Bugzilla #88
b4	Delay Slot Exception bit is not implemented in SR	OR1200, Bugzilla #85
b5	EPCR on range exception is incorrect	OR1200, Bugzilla #90
b6	Comparison wrong for unsigned inequality with different MSB	OR1200, Bugzilla #51
b7	Incorrect unsigned integer less-than compare	OR1200, Bugzilla #76
b8	Logical error in l.rori instruction	OR1200, Bugzilla #97
b9	EPCR on illegal instruction exception is incorrect	OR1200, Mail #01767
b10	GPR0 can be assigned	OR1200, Mail #00007
b11	Incorrect instruction fetched after an LSU stall	OR1200, Bugzilla #101
b12	l.mtspr instruction to some SPRs in supervisor mode treated as l.nop	OR1200, Bugzilla #95
b13	Call return address failure with large displacement	LEON2, Amtel-errata #2
b14	Byte and half-word write to SRAM failure when executing from SDRAM	LEON2, Amtel-errata #3
b15	Wrong PC stored during FPU exception trap	LEON2, Amtel-errata #4
b16	Sign/unsign extend of data alignment in LSU	OpenSPARC T1
b17	Overwrite of ldxa-data with subsequent st-data	OpenSPARC T1

Table 3.1: Security-critical bugs implemented and used for evaluation.

Table 3.1 shows the 17 security-critical processor bugs we use. The first 12 bugs are from OR1200, 3 bugs are from LEON2, and the last 2 are from OpenSPARC T1.

Bugs b1 and b2 may allow denial-of-service (DoS) attacks. In particular, bug b1 causes the processor to run in an infinite loop and bug b2 stalls the pipeline infinitely. Although the attacks violate liveness properties, we can identify security-critical safety properties at the root of the vulnerability. For example, the SCI we identified for b1 shows that the root cause of the vulnerability is that the PC is not correctly updated.

Bug b8 can be exploited to make the processor ignore an exception that it should handle. Attackers may leverage this to bypass some security checks. For example, failing to raise a bus error exception will potentially allow users to write into protected memory area.

Bugs b6, b7, or b13 leave the processor open to insecure control flow. Attacking bug b6 or b7 will cause the processor to incorrectly set the flag that decides whether branches should be taken. As a result, the processor may execute a sequence of instructions of the attacker's choosing. Bug b13 will incorrectly set the link register, which will cause the processor to return from a function call incorrectly and thus run a sequence of unexpected instructions.

Bug b11 can cause the processor to execute the wrong instruction. Even though the processor would execute the instruction correctly, the instruction itself in the pipeline has been contaminated because of subtle timing constraints. This allows the attackers to change or substitute instructions according to their needs.

Attacks on bug b12 can cause `l.mtspr` (Move to Special-Purpose Register Instruction) to act as a no-op when moving the content of a general-purpose register to some special-purpose registers. This bug causes the processor state to be incorrectly updated.

Bugs b4, b5, b9, and b15 deal with the contamination of exception-related special-purpose registers. This exposes the processor to security vulnerabilities because contaminating the registers that store the pre-exception processor state can potentially lead to privilege escalation.

Bugs b3, b10, b14, b16, and b17 are related to memory access. Bugs b3 and b10 can cause an incorrect address calculation or the wrong data to be loaded or stored. Bugs b14, b16, and b17 contaminate the data transferred between memory subsystems and registers. A potential attack might be to modify secret keys by contaminating the memory address or the data itself when loading or storing the data.

We reproduced these 17 bugs in the OR1200 processor, which is a 32-bit implementation of the OpenRISC 1000 architecture with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support (MMU), and basic DSP capabilities [69]. Our processor implements the basic instruction set (*i.e.*, none of the extension modules such as floating point). It is widely used in research projects and embedded computer environments. For each bug we also developed a triggering program written in a mixture of C and assembly that attacks the buggy processor and causes the violation of some security policies during execution.

3.3.2 Assertions

We leverage the industry standard Open Verification Library (OVL) for constructing assertions. All SCI were implemented using one of four OVL assertion templates: *always*, *edge*, *next*, *delta*. *always* is used when the expression is always true; *edge* is used when the expression is true at the point when the instruction is sampled; *next* is used when the expression is true some number of clock cycles after sampling the instruction; *delta* is used when a monitored signal's updates stay within a range.

Taking the invariant we described in Section 3.2.1 as an example,

$$I \doteq risingEdge(l.rfe) \rightarrow SR == orig(ESR0),$$

the corresponding assertion for this invariant is

$$A \doteq next(INSN = l.rfe, SR = ESR0_{PREV}, 1).$$

This means expression $SR = ESR0_{PREV}$ must be true one clock cycle after instruction l.rfe is sampled. Note that we need to store the previous cycle value of $ESR0$.

3.4 Evaluation

In this evaluation we show that 1) our tool effectively generates SCI from existing security-critical bugs; 2) the generated SCI stop both the existing security-critical bugs and new bugs; 3) meaningful SCI not tied to any known security-critical bugs can be found; and 4) the automatically generated SCI represent security properties written by experts.

3.4.1 Invariant Generation

Our tool's first step is to run a variety of programs on the processor to generate candidate invariants. We collected 26GB of trace data from 17 programs; more trace data results in more accurate invariant generation. We configured Daikon with a confidence limit of 0.99, reducing the risk of generating false-positive invariants that hold by chance in our trace data set. The filters search for invariants matching our invariant grammar in Figure 3.2.

We evaluate how the number of programs affect the set of invariants generated. We use the following programs: Linux boot, SPEC benchmarks (Parser, Mesa, Ammp, Mcf, Instru, Gzip, Crafty, Bzip, Quake, Twolf, Vpr), Basicmath, Pi Calculation, Bitcount, FFT, Helloworld. The execution traces cover all 56 instructions of the OpenRISC (basic instruction set) architecture. Figure 3.3 shows the result of this evaluation. We see that running additional programs may add invariants to the result set by exercising new features of the processor. It may also eliminate some invariants from the result set that cannot be justified by the new trace.

The overall trend of Figure 3.3 indicates that as the number of programs increases, the set of unique invariants that we generate becomes stable. After adding the *twolf* benchmark, no new invariants are generated or removed. From this trend, we extrapolate that if we run enough (finite) programs on the processor, we will reach a stable set of invariants that can roughly model the behavior of a processor.

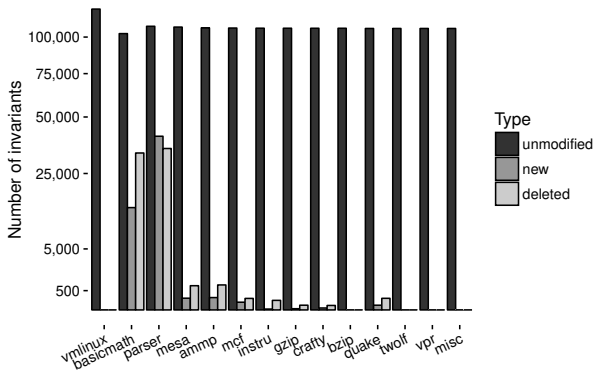

Figure 3.3: Unique invariants generated from executing programs. The X-axis is aggregative, *e.g.*, `basicmath` means invariants generated from running both `vmlinux` and `basicmath`.

	Raw	**after CP**	**after DR**	**after ER**
Invariants	106,174	106,174	90,955	88,301
Variables	210,013	171,858	170,517	167,863

Table 3.2: Effect of invariant optimizations (Section 3.2.2) in reducing the total number of invariants and variables in all invariants. CP is constant propagation; DR is deducible removal; ER is equivalence removal.

After the initial set of invariants is generated, it is optimized. Table 3.2 shows the effectiveness of different optimization passes in reducing redundant and lengthy raw invariants. The optimizations in combination achieve 17% reduction in terms of the number of invariants and 20% reduction in terms of the number of total variables in all invariants.

3.4.2 SCI Identification

The second step for our tool is SCI identification. Given a set of optimized invariants, a buggy processor and a triggering program, our tool identifies the affected SCI from the invariant set. Table 3.3 shows the number of identified SCI for each of the 17 security-critical bugs we implemented.

In total, our tool identifies SCI for 16 (94%) of the 17 bugs. Interestingly, although bug b1 and b5 are two different bugs, our tool identified the same SCI. This shows one advantage of our tool: the SCI we extract from a particular security bug are not just applicable to that bug, but rather potentially to a class of bugs. The only bug for which our tool fails to identify any SCI is bug b2. The reason is that no ISA-level invariants are violated by this bug. The bug is in the pipeline and all software-visible signals remain self-consistent. Identifying SCI for this bug would require adding microarchitectural level variables to Daikon's instrumenter and generating microarchitectural level invariants.

Bug No.	True SCI	FP	Detected
b1	2	22	✓
b2	0	N/A	×
b3	1	8	✓
b4	2	2	✓
b5	5	28	✓
b6	1	5	✓
b7	1	1	✓
b8	3	0	✓
b9	4	0	✓
b10	32	0	✓
b11	1	0	✓
b12	1	4	✓
b13	2	0	✓
b14	1	0	✓
b15	1	25	✓
b16	1	0	✓
b17	3	2	✓

Table 3.3: SCI identified from the 17 security-critical bugs we reproduced (see Table 3.1). Detected means enforcing the SCI as assertions on the processor can detect the buggy behavior dynamically.

Table 3.3 also shows more than one SCI identified per bug in some cases. This occurs for one of three reasons. The simplest is that the bug violates more than one security property. A second reason is that violating a single property may have multiple consequences. For example, in our implementation the syscall handler is always at address `0xC00`. Bug b8 violates this property and, therefore, the two invariants `l.sys` $\rightarrow$ `PC` $=$ `0xC00` and `l.sys` $\rightarrow$ `NPC` $=$ `0xC04`, where `l.sys` is the syscall instruction, `PC` is the program counter, and `NPC` is the next program counter. A third reason is that a violation may persist for multiple steps and our SCI are defined per instruction. For example, bug b10 violates the property `GPR0` $= 0$. The bug manifests in the add instruction and violates the invariant `l.add` $\rightarrow$ `GPR0` $= 0$. And, as the register is not restored to a valid state subsequent instructions violate analogous invariants, such as `l.nop` $\rightarrow$ `GPR0` $= 0$.

When more than one SCI are generated, it is attractive to think one primary property as the root cause property. However, it is possible that the identified SCI share equal weights and thus there are no immediate properties which are primary. It is also usually difficult to identify the primary property that represents the root-cause of the bug. On one hand, it is difficult to formally define what root cause is. On the other hand, identifying the primary properties is hard when the bugs are non-trivial and several properties might seem likely to be the cause of the violation. In this case, the primary properties can be identified only when the bugs are deeply understood. For example, from the description of bug b1 we might think that we need a

liveness property to detect the bug. However, when we digged into the cause of the bug, we found that the root-cause property is actually a safety property ($\texttt{l.rfe} \rightarrow \texttt{NPC} = \texttt{PC} + 4$).

The set of identified SCI may include false positives. We manually validated the identified SCI and found 7 of the bugs (43.8%) resulted in 0 false positives, while 6 of the bugs (37.5%) resulted in fewer than 10 false positives (Table 3.3). In practice, the false positives in the identified SCI can be easily spotted (e.g., an SPR must equal 0). We envision the usage scenario of our tool is that after it identifies SCI, experts would validate them before putting into a processor.

To further validate that our automatically identified SCI are useful, we enforce them as assertions in a SPECS-like system. The result shows that all the 16 security-critical bugs from which we identified SCI are detected dynamically, meaning the SCI are effective.

3.4.3 SCI Inference

In Section 3.4.2 we show that the SCI we build from the Identification step can effectively detect security-critical bugs and some identified SCI can detect multiple different bugs. In this section, we show that our tool can identify useful SCI not tied to any particular previously known bug. We use an elastic net logistic regression model to infer new SCI from existing SCI.

We start with our 88,301 invariants, each with 158 features, i.e., in our model from Section 3.2.4, $N = 88,301, P = 158$. Our model is supervised, and we leverage the results from the Identification step to provide labels to train the model. In particular, we have 54 verified SCI (*unique* SCI in Table 3.3). We label the *unique* false positives from the Identification step as non-SCI, a total of 48 invariants.

Of these 102 labeled invariants, we used 70% of the data as training data and performed the optimization of $\boldsymbol{\beta}$ and β_0 using the glmnet [51] package in R. We took $\alpha = 0.5$ and used 3-fold cross validation in the training set to choose an appropriate λ. Doing so, resulted in $\lambda = .08$. When we tested the model on the test set, we observed 90% accuracy, validating the quality of the fitted model.

In the constructed model, there were 24 non-zero coefficients from the original set of 158 features (see Table 4.6). To evaluate how these 24 features can be used to partition invariants in high-dimensional feature space, we performed principal component analysis (PCA) on the 102 labeled invariants according to this limited set of 24 selected features. Figure 3.4 shows the projection of these invariants in 2-dimensional space. As expected, using this set of features, invariants cluster adequately according to class label. This supports the model's selection of features as robust candidates for distinguishing SCI from non SCI.

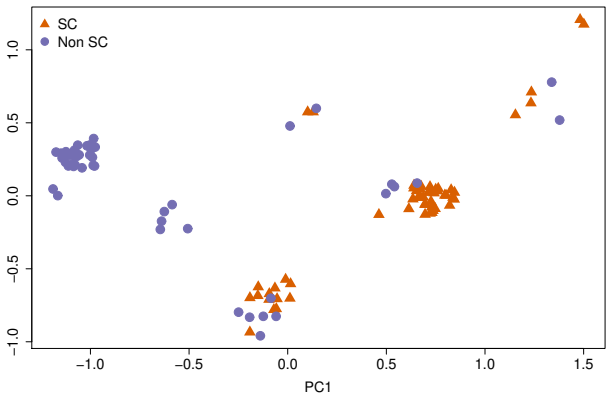

Figure 3.4: PCA using selected features. From the learned elastic net logistic regression model, 24 of the original set of 158 features had non-zero coefficients. PCA was performed using the 24 selected features on 102 SCI/non SCI. The plot shows the projection of these invariants in 2 dimensions.

Weight	**Features**			
Positive	GPR6	OP_B	ROR	DIV
	IM	MEM_{BUS}	orig(OP_A)	orig(SPR)
	orig(IM)	$<$	$\neq$	$+$
Negative	GPR0	PC	SF	WBPC
	IDPC	REG_B	orig(GPR0)	orig(NPC)
	orig(NNPC)	CONST	==	>=

Table 3.4: 24 identified features with non-zero coefficients. Features with negative weights are associated with SCI. Features with positive weights are associated with non-SCI.

We use the constructed model to further predict the entire set of 88,199 (88,301−102) *unlabeled* invariants. Table 3.5 shows the results. The model recommends 3,146 out of the 88,199 invariants as SCI. In the Identification step, we used the triggering programs to validate an identified SCI. In this Inference step, we do not have ground truth for the 88,199 invariants, but we manually examined the 3,146 recommended SCI and spotted 852 clear false positives.

These inferred SCI can be concisely described as 33 security properties that can be added, in the form of assertions, to a processor. In Section 3.4.4, we show that some of the inferred SCI represent security properties that are not covered by the SCI found in the Identification step, demonstrating the advantage of SCI inference.

Invariants	Inferred SCI	FP	Security Properties
88,199	3,146	852	33

Table 3.5: SCI inference results

No.	Security Property Description	Class	From Ident.	From Infer.
	Properties from SPECS [59]			
p1	Execution privilege matches page privilege	XR		✓
p2	SPR equals GPR in register move instructions	RU	b12	
p3	Updates to exception registers make sense	XR	b4 b9 b15	
p4	Destination matches the target	CR		✓
p5	Memory value in equals register value out	MA	b14	
p6	Register value in equals memory value out	MA	b16 b17	
p7	Memory address equals effective address	MA		✓
p8	Privilege escalates correctly	XR		✓
p9	Privilege deescalates correctly	XR		✓
p10	Jumps update the PC correctly	CF		▲
p11	Jumps update the LR correctly	CF	b13	
p12	Instruction is in a valid format	IE	b11	
p13	Continuous Control Flow	CF	b5	
p14	Exception return updates state correctly	XR	b1 b5	
p15	Reg. change implies that it is the instruction target	CR		✓
p16	SR is not written to a GPR in user mode	RU		
p17	Interrupt implies handled	XR	b8	
p18	Instr unchanged in pipeline	IE		★
	Properties from Security-Checker [22]			
p19	SPR modified only in supervisor mode	RU		✓
p20	Enter supervisor mode is on reset or exception	XR		✓
p21	Exception handling implies exception mechanism activated	XR	b8	
p22	Unspecified custom instructions are not allowed	IE		▲
p23	Exception handler accessed only during exception, in supvr mode, or on reset	XR	b8	
p24	Page fault generated if MMU detects an access control violation	MA		★
p25	UART output changes on a write command from CPU			■
p26	Only transmit cmd or initialization change Ethernet data output			■
p27	Debug Unit's value and ctrl regs only accessible from supvr mode			■

Table 3.6: Evaluation against security properties from prior work. For each property we indicate whether it was found in the identification (From Ident) or the inference (From Infer) step. The bug numbers correspond to Table 3.1. ✓means the property is found. If the property is not found it may be because it is not generated from Daikon (▲), it needs micro-architectural state (★), or it relates to HW outside the processor core (■).

No.	Security Property Description	Class	From Ident.	From Infer.
p28	Flags that influence control flow should be set correctly	CF	b6 b7	
p29	Calculation of memory address or memory data is correct	MA	b3 b10	
p30	Link address is not modified during function call execution	CF		✓

Table 3.7: New security properties generated by our tool that are not covered in prior work.

3.4.4 Representing Manually Written Security Properties

To evaluate the efficacy of our tool, we test whether it finds SCI, either from Identification or Inference, that represent the manually written security properties of the two state-of-the-art works: SPECS [59] and Security-Checker [22].

Table 3.6 shows the result. Of the 27 security critical properties from these two papers, 3 (p25, p26, p27) are security bugs outside of processor cores. These are not the target of this paper. For the remaining 24, 2 of them (p18, p24) need microarchitectural states and thus our tool cannot generate these two invariants. Thus, we mainly focus on whether our tool can identify or infer the remaining 22 security properties using the 17 security-critical bugs we reproduced.

From the Identification step, 11 (50%) of the 22 security properties are identified from 12 out of 17 bugs. There are three interesting findings. The first is that a single security property can be identified from different bugs and the identified SCI are different. For example, for bugs b4, b9, and b15, the identified SCI are different although they belong to the same security property (p3). The second is that different security properties can be identified from the same bug, *e.g.* p13 and p14 can be identified from b5. Finally, a single SCI can concisely represent multiple manually written security properties, *e.g.* p17, p21 and p23. The SCI for these properties is $risingEdge(\texttt{1.sys}) \rightarrow \texttt{PC} = \texttt{0xC00}$.

Adding the Inference step, 8 (36%) additional security properties are found. Two (p10 and p22) are not found because they do not exist in the invariant set generated with Daikon, and one (p16) is not identified as security critical although it does exists in the set of generated invariants.

Property p10 is missing because Daikon does not capture effective addresses (the immediate value shifted left two bits, sign-extended to program counter width, and then added to the address of the jump/branch instruction [69]). By adding the effective address as a derived variable to Daikon, we can generate this invariant. Property p22 is missing because it concerns custom instructions, which are part of the extended instruction set that we did not implement. (Recall, we implement the basic instruction set in our evaluation.)

Property p16 is not found by our tool, although the associated invariant does exist in our generated set of invariants. The invariant is $risingEdge(\texttt{l.add}) \rightarrow \texttt{SR} \neq \texttt{OP}_{\texttt{DEST}}$. It is neither violated by any of our implemented bugs, nor is it labeled as security critical by our logistic regression model. The latter is because in our model the $\neq$ operator is a feature with high positive weights, meaning invariants with that operator are likely to be classified as non-security-critical.

Our tool generates 3 new security properties not found by either SPECS or Security-Checker (Table 3.7). Two properties (p28, p29) are identified from bugs during the Identification phase, and one (p30) is from the Inference phase.

The property (p28) identified from bugs b6 and b7 is an example of using a derived variable, in this case one that describes the behavior of correctly setting the control flow flag. The property (p29) identified from bugs b3 and b10 is related to calculation. We note that SCIFinder is able to differentiate between calculations often used for memory addresses and others, and labels only the former as security critical. For example, the property GPR0 = 0 is often leveraged during address calculation and SCIFinder identifies multiple SCI to enforce it. Whereas invariants related to rotate calculations are not identified as security critical.

The property found during the Inference step (p30) has to do with the link address. A link address gives the location of a function call instruction and is used to calculate where program execution should return after function completion [69]. The inferred SCI states that the link address should not be modified during function execution.

3.4.5 Classification of Security Properties

The SPECS project classified security-critical processor errata into five classes (invalid register update, execute incorrect instruction, memory access, incorrect results, and exception related) [59]. Inspired by this, we classified the security properties related to the processor core into six classes: five of them are similar to the SPECS classification and we add one new class that is related to control flow. The classification results are shown in Tables 3.6 and 3.7.

CF stands for control flow related properties; **XR** stands for exception related properties; **MA** represents properties related to memory access; **IE** stands for the class of security properties that guarantee the processor will execute the correct and specified instructions; **CR** represents the class of security properties about correctly updating results.

Classifying the properties yielded two observations. The first is that SCIFinder was effective at finding properties related to exceptions (XR). Of the 27 properties identified by prior work, 9 fall into the XR category (the largest category by far – CF and MA are the next largest with 5 properties each) and SCIFinder was able to find all 9. On the other hand, SCIFinder was least effective for properties related to instruction execution (IE). Of the three identified in prior work, SCIFinder found only one. The two missed properties, p18 and p22, required microarchitectural state and analysis of custom instructions, respectively. We caution that these are observations; the total number of properties is too small to draw conclusions. However, they do suggest areas where SCIFinder may shine, as well as opportunities for future research to strengthen the SCIFinder approach.

3.4.6 Detecting Unknown Bugs

The SCI has the potential to detect new bugs that have not been seen before. We cannot measure this directly, as new bugs would only be found if we happened to run software that triggered the bug (causing the SCI assertion to fire). Instead, we took a set of bugs that we had not used in our identification or inference phases, added them to the processor, and ran software that triggers the bugs to see whether our SCI would fire. For this experiment we use the 14 AMD errata from the SPECS project. The authors reproduced the errata in the OR1200 processor and made their code public. Our tool is able to detect 12 of the 14 bugs. (By way of comparison, SPECS was also able to detect 12 bugs.) Five of these were detected by the Identified SCI, while seven were detected by the Inferred SCI. This demonstrates that our automatic SCI are not just applicable to the 17 known bugs from which they were generated, but are also useful to detect unknown bugs.

To avoid selection bias we repeat the experiment, but this time we randomly pick 14 bugs from our set of 28 (both from design documents and from AMD errata lists, excluding the 3 that use microarchitectural state), for use in the Identification and Inference steps. We use the remaining 14 bugs for testing. Of the test set, only bug b6 is not detected; the SCI for detecting b6 ($risingEdge(\texttt{l.sfleu}) \rightarrow (\texttt{OP}_\texttt{A} - \texttt{OP}_\texttt{B}) * (1 - 2 * \texttt{CF}) \geq 0$) is not found.

3.4.7 Performance

In this section, we evaluated the performance of our tool. The experiments are performed on a machine with an Intel Core i7 Processor (quad-core, 2.60GHz) and 8 GB of RAM. Table 3.8 shows the CPU time taken for each step of our tool. The whole process takes about 12 hours. The most expensive step is the

Step	Data	Size	Time hh:mm:ss
Invariant Generation	traces	26GB	11: 21 :00
Optimization	invariants	106,174	00: 00 :04
SCI Identification	invariants +bugs	88,301 +16	00: 44 :52
SCI Inference	invariants	88,301	<00: 00 :01

Table 3.8: Execution time. Except for traces, sizes are given as number of items, *e.g.*, the inference phase reads in 88,301 invariants.

	Baseline	Initial SCI	Final SCI
Logic	10073 LUTs	1.6%	4.4%
Power	3.24 W	0.13%	0.31%
Delay	19.1 ns	0%	0%

Table 3.9: Hardware overhead. The baseline is the OR1200, Xilinx xupv5-lx110t-based System-on-Chip. Initial SCI are the 14 assertions from Identification step. Final SCI are the 33 assertions from both Identification and Inference steps.

Invariant Generation for 26 GB of trace data. In practice, a full Invariant Generation step is only performed once and all subsequent generation is incremental.

Finally, table 3.9 shows the hardware overhead incurred by adding our assertions to the OR1200 design. The additional logic is less than 5% of the original design, incurs a power overhead of 0.3%, and adds no delay.

3.5 Summary

We have presented SCIFinder, a semi-automatic methodology for generating security-critical invariants (SCI). Given a list of known security-critical errata from a processor and the processor design we identify a set of SCI that can be used to dynamically verify the processor's security. Experiments show SCIFinder's practicality and effectiveness in generating meaningful SCI. It identifies effective SCI for 16 of 17 bugs from input errata plus 12 bugs from AMD errata lists. The final SCI set covers 86.4% of the manually crafted security properties from prior work and identifies 3 new properties not covered in prior work.

CHAPTER 4
TRANSLATING HARDWARE SECURITY PROPERTIES

In the previous chapter, I present a semi-automatic approach for developing hardware security properties from known vulnerabilities and machine learning technique. In this chapter, I tackle the problem of developing security properties from another perspective: one can leverage existing security properties and translate them to other hardware designs. I first present the motivation and threat model in Section 4.1. Then I give background information of hardware security properties and the problem statement in Section 4.2 and 4.3. I describes the details of our hardware security property translation tool, Transys, in Section 4.4 and 4.5. Finally, I show the evaluation results in Section 4.6.

4.1 Motivation and Threat Model

To validate the security of hardware designs, one needs a comprehensive set of properties describing the security requirements of the design. Developing such a set is challenging. The high-level goals of confidentiality and integrity of a particular security domain—and availability of a machine in general—may be well understood, but mapping these goals to the cycle-by-cycle behavior of specific registers, signals, and ports in a design is difficult, and a matter of art as much as science. In practice this effort must be repeated for each new design, even for new generations of existing designs.

I present Transys, a tool that takes in a set of security critical properties developed for one hardware design and translates those properties to a form that is appropriate for a second design. The insight that led to this work is the recent research into security specification development and security validation tools, which uses properties developed for one processor design in order to evaluate the proposed methodology on a second design [59, 60, 62]. The properties must be translated manually, and this process is mentioned only in passing, but it suggests that the properties crafted for one processor design can be made suitable for a second design.

We examine the question more closely. We investigate how the translation may be done programmatically, and we build Transys to implement our approach. We go beyond processor cores and include RSA and AES implementations in our evaluation. We examine properties from the two security verification methods in use today: assertion based verification using a restricted temporal logic, and gate level information flow tracking using `set` and `assert` tags. We find that cross-design, and in the case of a processor core, cross-architecture security specification translation is feasible and practical.

The problem statement is this: given a property written for one design, produce an equivalent property suitable for the verification of a second design.

It is not always clear what "equivalent" means. For example, prior work has demonstrated that the following policy, although relatively simple, is critical to security and holds for many pipelined RISC architectures [111]:

Policy 1. *The zeroth general purpose register (*`GPR0`*) must always contain the value 0.*

To ensure that the above policy is upheld for a particular design D, a designer might craft the following property, which if proven to hold for all possible traces of execution (along with a proof that `GPR0` is initialized to 0), will enforce the desired policy.

$$P_D \doteq \texttt{wr\_enable} \rightarrow \texttt{rf\_addr} \neq 0. \tag{4.1}$$

Property P_D states that if a write to the register file is enabled (`wr_enable`) then the register being written (`rf_addr`) is not zero—i.e., general purpose register 0 is not the target of the write.

However, the same property may not be true of a second design D', even though the design enforces the same policy. Design D' might require the following property:

$$P_{D'} \doteq \texttt{wr\_enable} \rightarrow \texttt{rf\_addr} \neq 0 \vee \texttt{rf\_data} = 0, \tag{4.2}$$

which states that writes are enabled only when `GPR0` is not the target of the write *or* when the value being written is 0. Design D' does not satisfy property P_D and an effort to verify the property will fail; however the underlying policy that we care about is upheld.

Given two properties written over the registers, signals, and ports of two different designs, it is not clear how to formally define equivalence between them. We therefore take an operational approach. We start with

observations about how properties are likely to morph from one design to another: for example, varying pipeline stages may affect in which clock cycle a signal becomes valid; flags may be laid out differently in control registers; and additional gating signals may be used in one design, but not in another. We then define a set of steps that modify property P_D in a set, limited number of ways to build a property $P_{D'}$ that is valid for design D'. We build a system that can reliably translate properties from one design to another, without requiring a formal definition of the intended high-level security policies each property is in aid of.

The gist of the approach is to do the translation in three phases: the first phase substitutes the appropriate signals, ports, and register names of the second design into the property; the second phase adjusts the arithmetic expressions and timing constraints of the newly drafted property; and the third phase refines the precondition of the new property. Transys takes as input the property to be translated and the RTL implementation of both the original design and the new design. No instrumentation or manual modeling of either design is required.

Transys does not obviate the need for human involvement in security property specification. In fact, manual review of the generated properties is a required step of the Transys workflow. Transys does, however, do much of the heavy lifting for the designer, leveraging work done by others in the community tackling the security validation of similar designs, and providing an initial set of security properties. In our evaluation, we manually analyze the new properties to decide if they are semantically analogous to the original set.

4.1.1 Threat Model

Transys is a tool to ease the development of security critical properties, and in doing so promote and encourage the security validation of hardware designs and expand the set of security critical properties validated.

The end goal is to strengthen the security of our hardware designs by eliminating bugs in the implementation or flaws in the design that are exploitable in software, post deployment, by the attacker. The attacker has knowledge of or can learn the details of the hardware design and is capable of finding and designing exploits for any bugs or flaws in the design.

Security validation is not addressing the threat of malicious trojans that get added during fabrication, nor does it prevent attacks post-deployment that involve tampering with or modifying the hardware.

Once the set of properties have been developed for a design they can be used to detect subsequent malicious modifications to the design. If the modification violates one of the security properties, the violation

$$
\begin{aligned}
LTL(\mathbf{G}, \mathbf{X}) \doteq\ & \mathbf{G}(\phi) \\
\phi \doteq\ & s \rightarrow s \\
s \doteq\ & f \mid \mathbf{X}s \\
f \doteq\ & a \mid \neg f \mid f \vee f \mid f \wedge f \mid f \rightarrow f \\
a \doteq\ & t == t \mid t \neq t \mid \mathit{true} \\
t \doteq\ & \texttt{reg} \mid N \mid \texttt{reg} + \texttt{reg} \mid \texttt{reg} - \texttt{reg} \\
& \mid \texttt{reg} << \mathrm{N} \mid \texttt{reg} >> \mathrm{N} \\
& \mid \texttt{reg}[N : N]
\end{aligned}
$$

Figure 4.1: The restricted temporal logic used by security properties expressed as assertions, where `reg` is a signal, register, or port in the design, and N is the set of natural numbers.

can be found during verification. (The method of verification matters here—model checking, execution monitors in use post-deployment, and symbolic execution can provide guarantees about coverage, whereas simulation based testing does not.) We caution, however, that Transys uses the code of the second design to build the translated property; a well crafted trojan already extant in the code can affect the final property. Manual review of the set of properties created is a required step of the Transys workflow.

4.2 Security Properties

We focus on properties developed for a hardware design at the register transfer level (RTL). Properties are written for use with a particular verification method, and each method has an associated specification language in which the properties can be expressed. We present the two main logic systems used to express hardware security properties.

4.2.1 Restricted Temporal Logic

Assertion based verification is widely used in industry for the functional validation of hardware designs. Properties expressed in a restricted temporal logic are added, in the form of assertion statements, to the RTL design and simulation-based testing or static analysis is used to find violations.

The security properties that have been developed to date make use of existing industry standard libraries for expressing assertions [55] and are written in a fragment of linear temporal logic that includes the globally ($\mathbf{G}$) and next ($\mathbf{X}$) operators with a syntactic restriction that conforms to the grammar shown in Figure 4.1. In particular, the properties are of the form $\mathbf{G}(A \rightarrow B)$, where A and B are boolean combinations of arithmetic expressions and may contain the $\mathbf{X}$ operator.

$$
\begin{aligned}
\mathit{property} &: (\mathit{set\_stmt})^* \ldots (\mathit{assert\_stmt})^* \\
&\quad |(\mathit{set\_stmt})^* \ldots (\mathit{gated\_assert\_stmt})^* \\
&\quad |(\mathit{set\_stmt})^* \ldots (\mathit{declass\_assert\_stmt})^* \\
\mathit{set\_stmt} &: \text{`set'} \ \texttt{reg} \ \text{`:='} \ \mathit{tag} \\
\mathit{assert\_stmt} &: \text{`assert'} \ \texttt{reg} \ \text{`=='} \ \mathit{tag} \\
\mathit{gated\_assert\_stmt} &: \text{`assert'} \ \texttt{reg} \ \text{`=='} \ \mathit{tag} \ \text{`when'} \ \texttt{expr} \\
\mathit{declass\_assert\_stmt} &: \text{`assert'} \ \texttt{reg} \ \text{`=='} \ \mathit{tag} \ \text{`allow'} \ \texttt{reg} \\
\mathit{tag} &: \text{`high'} \mid \text{`low'}
\end{aligned}
$$

Figure 4.2: The syntax used to track how information flows through a hardware design at the gate level. A property is a series of *set* statements over source variables and *assert* statements over sink variables. The *assert* statements may be made conditional using *when*. Declassification is done using *allow*.

4.2.2 Information Flows

The properties expressible in the temporal logic are trace properties: individual traces of execution either satisfy or violate the given property. However, properties about how information flows through the processor are not immediately expressible as trace properties, but rather require hyperproperties [38, 84]. Whereas a trace property can be defined by a set of traces—those traces that satisfy the property, a hyperproperty is defined by a set of sets of traces—those systems that satisfy the property. Properties about confidentiality, such as asserting an absence of side channels, or about integrity, such as asserting which security domains can influence the control flow of a protected domain are examples of hyperproperties.

Gate level information flow tracking requires tagging source variables with the appropriate level (e.g., "high" or "low") of information, asserting the correct level is maintained for sink variables, and deciding when to conditionally disable the assert or under what circumstances to allow declassification. The syntax of these properties are shown in Figure 4.2.

4.2.3 Hardware Security Properties

We present the security properties for three classes of designs: RISC processor cores, AES implementations, and RSA implementations. Table 4.1 shows the security properties of the OR1200 processor. These security properties are collected from the SCIFinder projects, and we renumber the properties according to their categories. Tables 4.2 and 4.3 show the security properties of the AES designs and RSA designs, respectively. These we developed manually by studying the respective specifications. Table 4.4 shows information flow properties for AES and RSA implementations. These properties are collected from work on

Type	Description
Memory Access	*P01*: Memory value in equals register value out
	P02: Register value in equals memory value out
	P03: Memory address equals effective address
	P04: Calculation of memory address or memory data is correct
Exception Related	*P05*: Execution privilege matches page privilege
	P06: Updates to exception registers make sense
	P07: Privilege escalates correctly
	P08: Privilege deescalates correctly
	P09: Exception return updates state correctly
	P10: Interrupt implies handled
	P11: Enter supervisor mode is on reset or exception
	P12: Exception handling implies exception mechanism activated
	P13: Exception handler accessed only during exception, in supvr mode, or on reset
Control Flow	*P14*: Jumps update the PC correctly
	P15: Jumps update the LR correctly
	P16: Continuous Control Flow
	P17: Flags that influence control flow should be set correctly
	P18: Link address is not modified during function call execution
Update Registers	*P19*: SPR equals GPR in register move instructions
	P20: SR is not written to a GPR in user mode
	P21: SPR modified only in supervisor mode
Correct Results	*P22*: Destination matches the target
	P23: Reg change implies that it is the instruction target
Instruction Executed	*P24*: Instruction is in a valid format
	P25: Instructions unchanged in pipeline
	P26: Unspecified custom instructions are not allowed

Table 4.1: Security properties of OR1200 processor mined from the specification.

Module	Description
Key Expansion	*P27*: The round constant for each round of the key expansion should be correct.
	P28: Round keys should be derived from the cipher key correctly.
Substitution Box	*P29*: The S-box should avoid any fixed points and any opposite fixed points.
Add Round Key	*P30*: The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.
Shift Rows	*P31*: The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset.

Table 4.2: Security critical properties of AES cryptographic hardware mined from the specification.

gate level information flow tracking [62] and were, to the best of our knowledge, developed manually. We used only a subset of the AES properties during the development of Transys. The rest of the properties we reserved for use in the evaluation.

Module	Description
RSA Top	*P32*: The output cipher should be different from the input key.

Table 4.3: Security critical properties of RSA cryptographic hardware mined from the specification.

Type	Description
Confidentiality	*P33*: The key or intermediate results should not directly flow to a point observable by an attacker.
Integrity	*P34*: The key should never be altered.
Isolation	*P35*: The intermediate encryption results are allowed to flow to output when the core is working in debug mode, but are prohibited under normal operation.
	P36: The key is safe to flow to the ciphertext while it should not flow to another location.
Timing Channel	*P37*: The secret key should not flow to the ciphertext ready signal otherwise there would be a timing side channel.

Table 4.4: Information flow security properties of cryptographic hardware.

4.3 Problem Statement

Given an RTL design D_1, a property P_{D_1} that is written in a formal logic stated over the registers, signals, and ports of design D_1, and a second design D_2, how can we produce a second property P_{D_2} that

1. is a valid property for the specification of design D_2, and

2. captures the same security policy as property P_{D_1}.

4.4 Design

Transys takes as input two hardware designs and a set of security-critical properties for the first design, and outputs a set of translated properties for the second design. For each property P of the first design, the goal is to produce a new property P' that is written over the registers, signals, and ports of the second design and that preserves the semantics of P for the second design. To achieve this goal, Transys must solve four challenges:

1. The registers, signals, and ports in the original property may not have counterparts in the second design; if they do, the counterparts will likely not have the same name.

2. The arithmetic expressions in P may not be appropriate for the second design.

No.	Original	New Format	Simplified
1	$A \to B$	$A \wedge C \to B$	$(A \wedge C) \to B$
2		$A \vee C \to B$	$(A \to B) \wedge (C \to B)$
3		$A \to B \wedge D$	$(A \to B) \wedge (A \to D)$
4		$A \to B \vee D$	$(A \wedge \neg D) \to B$

Table 4.5: Possible formats of translated assertions in the new design. The simplifications are standard propositional rewrite rules.

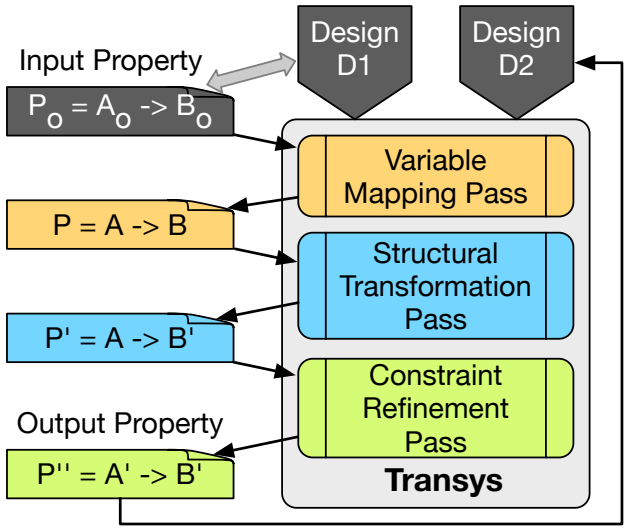

Figure 4.3: The workflow of Transys.

3. The conditions required to enforce a given policy might differ between designs. For example, in the property described in the introduction, P_D has the form $A \to B$, but $P_{D'}$ requires the form $A \to B \vee C$ to capture the same policy.

4. Policies often have to be stated across multiple clock cycles. For example, a `wr_enable` signal set in one clock cycle may be seen by the register file in the following clock cycle. Timing details depend on the specifics of an implementation and can vary across designs. The translated property will need to take that into account.

4.4.1 Overview

Transys works in three passes to address the four challenges above: variable mapping pass, structural transformation pass, and constraint refinement pass. We start with an overview of the three passes and then describe each one in detail. Figure 5.2 shows the workflow of Transys.

Type	Feature
Statistical	Variable Type (Input, Output, Wire, Reg)
	No. of Blocking Assignments
	No. of NonBlocking Assignments
	No. of Assignments
	No. of Branch Conditions
	No. of Always Block Conditions
Semantic	Variable Names
Structural	Dependence Graph Depth
	No. of Operators
	Centroid

Table 4.6: Features from AST and PDG for variable mapping.

Variable Mapping Pass. To begin, Transys maps the registers, signals, and ports named in the properties of the first design to the registers, signals, and ports (hereafter, *variables*) of the second design (Section 4.4.2). We first find the matching code windows of the two designs to narrow the scope of variables to map. We then extract statistical, semantic, and structural features of each variable, and calculate the distances between each pair of variables from the two designs. The variable pairs with shortest distance are used as mapped variables.

Structural Transformation Pass. In the next pass, Transys uses the Program Dependence Graphs (PDGs) [49] of the two designs to adjust the arithmetic expressions in the translated property. We use the PDG of the first design to learn the relationship between multiple variables in the property, and we traverse the PDG of the second design to build the arithmetic expressions of, and capture the analogous relationship between, the variables in the translated property. In practice we apply this step to only the consequent part of the property; we found the structural transformation was not needed for the antecedent. However, there is no limitation that would prevent applying this pass to the antecedent as well, should future properties require it.

Constraint Refinement Pass. In the third pass Transys refines the constraints of the property by adding terms to the boolean formula. Starting with the form $A \rightarrow B$, there are four possible modifications Transys might make. These, along with their simplified forms, are laid out in Table 4.5. The first and fourth formats represent a refinement of the original property—an added constraint under which the property holds—and Transys will produce properties that require this refinement. The second and third formats are not refinements of the original property, but rather introduce new properties of the second design. This can be seen in the "Simplified" column of Table 4.5. Transys does not produce these new properties.

4.4.2 Variable Mapping Pass

In this pass we are concerned only with mapping variables named in one design to their appropriate counterpart in the second design.

Matching Windows

Similar to feature-based image alignment approaches, we search for matching variables within a reasonable range instead of within the entire code base. Modules in the Hardware Description Language by nature are good windows for matching: it keeps the semantic meaning of some functionalities and the size of each module is often reasonable to search. As the two hardware designs for assertion translation often share the same specification, we simply match modules with their names using Equation 4.4. We thus narrow down the scope of variables to map and search the mapped variables within corresponding modules.

Extracting Features

For each variable from the two designs within the corresponding matching windows, we extract three types of features from the Abstract Syntax Tree (AST) and the Program Dependence Graph (PDG): statistical features, semantic features, and structural features (see Table 4.6).

The statistical features include: the variable type; the number of times this variable appears in the left-hand-side of blocking assignments, nonblocking assignments, and assignment statements; and the number of times it appears in the branch conditions and always block conditions. The statistical features describe local statistics of a variable within a module. These features are extracted from the AST of the design.

We observe that these statistical features can reflect how these variables are used in the code. A simple example is that the `reset` signal is often used in the branch conditions, so a signal never used in the branch conditions is unlikely to be the counterpart of the `reset` signal. Another example is that if a signal is used in the non-blocking assignments in one design, it is highly likely that we also want the counterpart signal we find to appear in the non-blocking assignments – there might be two similar signals but only one of them stores state and that is the one we are looking for; the other one might be a temporary signal in the design. Based on this observation, we include the statistical features for variable mapping.

The semantic features point to the semantic meaning of a variable. We use the variable name as a feature because it usually explains what this variable is about. For example, the variable `ex_insn` in the OR1200

processor holds the instruction in the EX pipeline stage. Different design implementations often share similar variable names for the same variable.

The structural features capture the position of a variable in a PDG. We choose three features: dependence graph depth, numbers of operators, and centroid. The dependence graph depth is the maximum length of paths of the PDG from any statement that contains the variable to the input ports of the module. The numbers of operators calculate the number of times each operation (e.g. &&, ||, $\gg$, $==$, $>$, etc.) appears in the paths from the statements to the input ports in the PDG. The centroid measures the centrality of the dependence graph [34]. We assign each operator a weight (we use the same weight for every operator) and calculate the centrality of all the paths from the variable to the input ports of the PDG. The PDGs usually capture the functionality of part of the design, and parts of the code with the same functionality usually share similar PDGs between designs. The information of the position of a variable in a PDG can help us match the variables that have similar calculation dependencies and functionalities.

Matching Variables

To match variables of two designs, we calculate distances between the features of pairs of variables, one from each design. The variable pairs with shortest distance are used for drafting the assertions.

For statistical features, we use the Euclidean distance for distance calculation:

$$d_{stat}(p, q) = \sqrt{(q_1 - p_1)^2 + \cdots + (q_n - p_n)^2} \tag{4.3}$$

For semantic features, we use the Sørensen-Dice index [44] for distance between two strings calculation:

$$d_{seman}(s_1, s_2) = 1 - \frac{2 \times |pairs(s_1) \cap pairs(s_2)|}{|pairs(s_1)| + |pairs(s_2)|} \tag{4.4}$$

where $pairs(s)$ is a set of character pairs in string s. The Sørensen-Dice index satisfies two requirements: (1) a significant substring overlap should point to a high level of similarity between strings; (2) two strings which contain the same words, but in a different order, should be recognized as being similar. The factor 2 ensures that when the two strings are exactly the same, the distance is 0.

For structural features, we use Euclidean distance (Equation 4.3). Each feature—depth, number of operators, and centroid—appears as a term in the calculation.

Design 1

```
always @(round_i)
begin
  case (round_i)
    1: rcon_o = 1;
    2: rcon_o = 2;
    3: rcon_o = 4;
    ......
end
```

Design 2

```
initial
begin
  rcon[0] = 8'h01;
  rcon[1] = 8'h02;
  rcon[2] = 8'h04;
  rcon[3] = 8'h08;
  ......
end
```

Figure 4.4: Code snippets from AES designs.

Design 1

```
assign w0 = key[127:96];
assign keyout[127:96] =
    w0^tem^rcon(rc);
```

Design 2

```
always @*
begin
  w0 = key[127:096];
  w4 = w0^subword1^{rcon1,24'b0};
  w8 = w4^subword2^{rcon2,24'b0};
  w12 = w8^subword3^{rcon3,24'b0};
  .....
end
```

Figure 4.5: Code snippets from AES designs.

We combine the three distances by assigning each of them a weight, and thus the distance between two variables is:

$$d(v_1, v_2) = \alpha d_{seman} + \beta d_{stat} + \gamma d_{struct} \tag{4.5}$$

where v_1 and v_2 are variables from the two designs respectively. When assigning values to parameters α, β, and γ, we empirically choose α to be the largest as the semantic meanings of variable names are usually similar between designs. We choose β to be the smallest as the detailed implementation are often different between designs, thus the structural information will be less similar.

4.4.3 Structural Transformation Pass

In the structural transformation pass, we amend the arithmetic expressions that make up each of the terms in the property. We start by describing the challenges we met in translating the properties after the variable mapping pass. We then discuss our observations and solutions to the challenges.

Challenges

We consider three types of structural dissimilarities between designs, which Transys must handle: mapping state to array, mapping one to many, and mapping constants.

Mapping state to array refers to the case where a variable is updated according to a state machine in one design, but in another design, the variable is an array that stores all the possible values at different states of the state machine. Figure 4.4 shows code snippets of two AES implementations of the key expansion. In Design 1, the round constant `rcon_o` changes every time the state machine changes to the next state. In Design 2, all possible values of `rcon` are stored in an array.

Mapping one to many refers to the case where a variable from one design can be mapped to several variables in another design. For example, one design might use temporary variables to store the intermediate results of long calculations or avoid large arrays, and a second design might not. Figure 4.5 shows code snippets from two AES cores. The variable `keyout` in Design 1 maps to the concatenation of variables `w0`, `w4`, `w8`, and `w12` in Design 2. Mapping many to one is the dual case and also requires structural transformation.

The last type is mapping the constant values used in one design to the analogous constant values of a second design. For example, the syscall instruction is encoded differently in OpenRISC cores versus RISC-V cores. In some cases it is possible to find a linear transformation from the constant of one design to its semantic equivalent in the second design, but in other cases, such as with the syscall encoding, it is not.

Transformation Algorithm

We observe that if in the first design, the variables in the property are related to each other, the correlation among the variables in design two are often explicitly stated in the code. Thus, we leverage the PDG to build the arithmetic expressions of, and capture the analogous relationship between, the variables in the translated property.

As shown in Algorithm 1, we first check whether in the first design, the variables in the property are in the same PDG. If not, we assume that in the second design, the variables in the translated property are also not in the same PDG. In this case, we use the translation result of the Variable Mapping Pass as the result for this pass.

Otherwise, we leverage the PDG to build the property. We take the mapped variable with the highest score (`max_var`) and check whether the other mapped variables are in the same PDG as the `max_var`. If not,

Algorithm 1: Transformation Pass

Input : The property generated from the VM Pass P
Input : A set of PDGs of the Design 1 *pdgSet1*
Input : A set of PDGs of the Design 2 *pdgSet2*
Input : A map of variable mapping scores *vScoreMap*
Output : A new property P'

```
1  newAssertSet ← ∅;
2  if in_same_pdg(P, pdgSet1) then
3  |   max_var ← max_score(P, vScoreMap);
4  |   for var in P do
5  |   |   for v in vScoreMap[var] do
6  |   |   |   if in_same_pdg(max_var, v, pdgSet1) then break ;
7  |   |   end
8  |   |   substitute(P, var, v);
9  |   end
10 |   var ← max_score(P-{max_var}, vScoreMap);
11 |   P' ← propagate(max_var, var);
12 else
13 |   P' ← P;
14 end
15 return P';
```

we move to the next variable in the vector of mapped variables and check again. We iterate until all variables in the translated property are in the same PDG as `max_var`. Then we find the variable with the second highest score (line 10).

Finally, we use a propagation algorithm in the PDG to build the new property. The propagation algorithm takes in two variables: a starting point variable, and an ending point variable (`max_var` is usually taken as the starting point). The ending point variable can be either an ancestor or a descendant of the starting point in the dependency graph. We explore both the ancestors and descendants of the starting point variable in the PDG until we hit the ending point variable. During the exploration of each node in the PDG, we replace the intermediate variables until the ending point variable is shown in the property. We stop at the ending point variable so that the property can cover the logic involving the mapped variables but does not include too long of a calculation.

There is a timing issue during the propagation. Every time we encounter a nonblocking assignment, we add a `Next` (X) to the property (or equivalently, a `prev`), indicating that there will be a delay of one clock-cycle for this assignment. Section 4.5 shows an example of how we handle the nonblocking assignment timing.

4.4.4 Constraint Refinement Pass

At this point, we have a draft property of Design 2 in the form $P' \doteq A \rightarrow B$. We first check whether P' is a valid property of Design 2. If it is, we are done. If it is not, then we continue with the constraint refinement pass. The goal of this step is to refine A to A', such that $P'' \doteq A' \rightarrow B$ is a valid property of Design 2.

We first introduce notation and define the problem; we then describe the algorithm.

Notation and Problem Statement

A hardware design unrolled for multiple clock cycles can be represented as a boolean formula ϕ in conjunctive normal form (CNF): $\phi \doteq (l_p \vee l_q) \wedge (l_r \vee l_s \vee l_t) \wedge \ldots$, which is written as a conjunction of clauses ω, where each clause is a disjunction of literals l (e.g., $\omega \doteq (l_p \vee l_q)$). A literal is either a variable x_i or its negation $\neg x_i$.

Let ϕ_{D_2} be the CNF formula representing Design 2 unrolled for some finite but unbounded number of clock cycles. P is a valid property of Design 2 if and only if the boolean formula $\phi_{D_2} \wedge \neg P$ is unsatisfiable:

$$\phi_{D_2} \models P \Leftrightarrow (\phi_{D_2} \wedge \neg P) \text{ UNSAT} \tag{4.6}$$

If $\phi_{D_2} \wedge \neg P$ is satisfiable, in other words, if P is not a valid property of Design 2, then we look for a sequence of conjuncts $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ such that the formula $F \doteq \phi_{D_2} \wedge \neg P \wedge A_1 \wedge A_2 \wedge \ldots \wedge A_n$ is unsatisfiable. Using the new conjuncts, we define P' as follows:

$$P' \doteq (A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge A) \rightarrow B \tag{4.7}$$

Then $\phi_{D_2} \wedge \neg P'$ is equivalent to F: $F \Leftrightarrow \phi_{D_2} \wedge \neg P'$, and therefore equisatisfiable with F. If we are successful in findng $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ that make F unsatisfiable, then $\phi_{D_2} \wedge \neg P'$ will also be unsatisfiable, and P' will be a valid property of the design: $\phi_{D_2} \models P'$.

There are two possible cases when F is unsatisfiable. The first case is that the subformula $\phi_{D_2} \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is unsatisfiable. In this case, the negation of the new conjuncts $\neg(A_1 \wedge A_2 \wedge ... \wedge A_n)$ is itself a valid property of ϕ_{D_2}. We are not interested in this case as it does not relate to the original property we are translating. The second case is that $\phi_{D_2} \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is satisfiable, and $F = \phi_{D_2} \wedge \neg P \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$

Algorithm 2: Refinement Pass

Input : A CNF formula ϕ
Input : The property generated from the T Pass P'
Output : A new property with refined antecedent P''

if $\phi \wedge \neg P'$ *is UNSAT* **then** return P';
for t *in range(1,MAX_SEQ)* **do**
  $\Omega_t \leftarrow \{\omega_i | (\omega_i \text{ in } \phi) \wedge (P'_t \text{ in } \omega_i)\}$;
  for ω_i *in* Ω_t **do**
    $\Omega'_t \leftarrow \{\omega_j | (\omega_j \text{ in } \phi) \wedge (\neg l \text{ in } \omega_j) \wedge (l \text{ in } \omega_i)\}$;
    for ω_j *in* Ω'_t **do**
      $S \leftarrow \emptyset$; step $\leftarrow 0$;
      $\omega_l \leftarrow \omega_i \odot \omega_j$;
      $S \leftarrow S \cup \{l | l \text{ in } \omega_l\}$;
      while *step* < *MAX_STEP* ***or*** *False not in* ω_l ***or*** ω_l *changes* **do**
        $\omega_{ante} \leftarrow$ find_ante(ω_l, S);
        $S \leftarrow S \cup \{l | l \text{ in } \omega_{ante}\}$;
        $\omega_l \leftarrow \omega_{ante} \odot \omega_l$;
        step $\leftarrow$ step $+1$;
      end
      Ante $\leftarrow \bigwedge_{l \text{ in } \omega_l, l \neq I'_t} \lambda(l, 0)$;
      if $\phi \wedge$ *Ante is SAT* **then**
        return $P' \wedge \neg$Ante;
      else
      end
    end
  end
end
return Not Found;

is unsatisfiable. In this case, $A_1 \wedge A_2 \wedge ... \wedge A_n$ are the preconditions of the property P. This is the refinement of the constraints of the translated property.

Constraint Refinement Problem. Given ϕ_D, the CNF representation of a hardware design unrolled a finite but unbounded number of clock cycles, and a draft property P such that $\phi_D \wedge \neg P$ is satisfiable, find a sequence of n conjuncts $A_1 \wedge A_2 \wedge ... \wedge A_n$ such that:

- $\phi_D \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is satisfiable, and
- $\phi_D \wedge \neg P \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ is unsatisfiable.

Constraint Refinement Algorithm

The constraint refinement algorithm works by finding *conflict clauses* in the CNF representation of the design. For each literal l appearing in the clause ω that contains B (the consequent of the property), the

algorithm searches for a clause ω' in ϕ_D such that $\neg l$ appears in the clause. These two clauses are conflict clauses. If we force all other literals appearing in ω and ω' to evaluate to false, then ϕ_D will be unsatisfiable.

Let $\lambda(l, v)$ be a function that takes in a literal $l \in \{x, \neg x\}$ and a truth value $v \in \{\texttt{true}, \texttt{false}\}$ and returns a new literal $l' \in \{x, \neg x\}$ such that l' evaluates to `true` when l evaluates to v.

$$\lambda(l, v) = \begin{cases} x & \text{if } l = x,\ v = \texttt{true} \\ x & \text{if } l = \neg x,\ v = \texttt{false} \\ \neg x & \text{otherwise} \end{cases}$$

Given a CNF formula ϕ, if there exist conflict clauses ω_i and ω_j in ϕ, where $\omega_i = l_{i1} \vee ... \vee l_{is} \vee x_c$, and $\omega_j = l_{j1} \vee ... \vee l_{jt} \vee \neg x_c$, then $\phi \wedge \lambda(l_{i1}, 0) \wedge ... \wedge \lambda(l_{is}, 0) \wedge \lambda(l_{j1}, 0) \wedge ... \wedge \lambda(l_{jt}, 0)$ is unsatisfiable. This is because $x_c \wedge \neg x_c$ is unsatisfiable. By assigning all other literals in the two clauses ω_i and ω_j to 0, subformula $\omega_i \wedge \omega_j$ can be simplified to $x_c \wedge \neg x_c$, which is unsatisfiable. Thus, $P = \neg(\lambda(l_{i1}, 0) \wedge ... \wedge \lambda(l_{is}, 0) \wedge \lambda(l_{j1}, 0) \wedge ... \wedge \lambda(l_{jt}, 0))$ is a property of ϕ.

Algorithm 2 takes a CNF formula ϕ_D and the property to be refined P' as inputs. It first checks whether P' is a valid property of ϕ_D, if it is, the algorithm just returns P'. Otherwise, it searches for clauses that contain the property P' (line 3), and for each clause that contains P', it searches for its conflict clauses (line 5). By combining the results of these two sets of clauses, the algorithm produces the new property for ϕ_D.

Greedy Search

The results we obtained from combining ω_i and ω_j often do not include any interesting preconditions, but just a restatement of the property P'. This is because when unrolling the design together with the invariant, some clauses to connect the invariant with the design need to be added to ϕ_D. To get the preconditions, we have to search further.

We first define the resolve operator $\odot$: given two clauses ω_i and ω_j, for which there is a unique variable x such that one clause has a literal x and the other has $\neg x$, $\omega_i \odot \omega_j$ contains all the literals of ω_i and ω_j with the exception of x and $\neg x$.

Starting from the conflict clauses (line 8), we search for more clauses that can introduce potential precondition variables (line 11). ω_l keeps track of the current resolved clause. Every time we find a new conflict clause, we resolve ω_l with the new clause (line 13). The new ω_l clause can still make ϕ unsatisfiable.

We keep expanding the resolved clause, until we reach the maximum step, or `False` shows in ω_l, or ω_l does not change any more (line 10). Then we generate the antecedent from ω_l and check whether it satisfies the requirements (line 16). If yes, we output the new invariant; otherwise, we keep on searching (line 17-19).

During the search in `find_ante`, we search for clauses greedily. The goal is to keep the antecedent short to be readable and managable. Thus, every time we find a conflict clause, we only find the one that introduces one new variable to ω_l (we use a set S to keep track of the found variables).

Timing in the Assertions

A property P' is asserted at each clock cycle: $\phi \wedge \neg P' \doteq \phi \wedge \neg P'_{t=1} \wedge \neg P'_{t=2} \wedge ... \wedge \neg P'_{t=MAX\_SEQ}$. To determine the timing constraints in the assertion, the search for conflict clauses takes place only within a specific clock cycle ($\phi \wedge \neg P'_{t=t_i}$, line 2 in Algorithm 2), instead of all clock cycles together ($\phi \wedge \neg P'$). The generated property P'' from the refinement pass can contain literals in different time steps. We rank them according to the timing information, and add the delays between them.

4.4.5 Property Does not Exist

A property of one design may not be true of a second design. This can happen when the two designs implement different specifications or when one of the designs implements only part of the specification. For example, some of the AES designs we collected implemented only encryption and did not implement decryption. Thus, the properties related to decryption cannot be translated to these designs. Another example is that for RISC-V processors, there are three privilege levels, but for OpenRISC processors, there are only two privilege levels. Thus, properties related to the middle privilege level of the RISC-V processor do not have corresponding properties in the OpenRISC processors. In these cases Transys may fail to produce a translation, which is a reasonable outcome.

4.4.6 Bugs in the Code

The structural transformation and constraint refinement passes leverage the second design itself to translate the property. This raises a concern: If there is a bug in the design, it will be captured in the translated property. This is true. Transys is meant to be used as an aide to the verification team tasked with writing security critical properties of a design. Transys does the heavy lifting of producing a candidate translation,

but it does not obviate the need for human involvement in property design. A manual review of the translated properties is a required part of the workflow.

4.5 Implementation

We implement Transys based on the Yosys Open Synthesis Suite [106], a framework for Verilog synthesis. Transys is implemented in C++ with approximately 4,500 lines of code. The assertions are implemented in SystemVerilog. Each Pass is implemented as a command in Yosys: the Variable Mapping Pass and the Transformation Pass are implemented as new commands (`match_variables` and `transform`), and the Refinement Pass is implemented by modifying the `sat` command. We also implement three assisting commands for building the program dependence graphs (`build_pdg`), parsing security assertions to a standard format (`read_assertlist`) and adding assertions to the designs for refinement and validation (`append_assertlist`).

We build the PDGs on the Register Transfer Level Intermediate Language (RTLIL) representation in Yosys. Each node in the PDG is a `Cell` or a `Wire` object, which represents the netlist data; or a `Switch`, a `Case`, or a `Sync` object, which represents the decision trees and synchronization declarations; or an `assignment` block, which we build to represent the assign statements. Each edge represents either the control or data dependence. To build the PDG, we first convert the objects into nodes. An edge from node A to node B is added if the inputs to B depend on the outputs of A.

For the timing delays caused by non-blocking assignments from the Transformation Pass, we add a state machine to keep track of the signal values in different clock cycles. For example, if we have an assertion `(a == prev(b))`, the implementation of this assertion is:

```
always @(posedge clk)
begin
  prev_b <= b;
end
assert property (a == prev_b);
```

4.6 Evaluation

Our evaluation aims to answer the following questions: (1) whether Transys can successfully translate security-critical assertions from one design to another; (2) whether the translated assertions are valid and

capture the meaning of the original assertions; (3) whether Transys is practical in terms of run-time; (4) how the translation results are affected by bugs in the second design.

4.6.1 Experiment Setup and Dataset

The experiments are performed on a machine with the Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, dual-socket) and 62GB RAM. We evaluate Transys on 38 AES designs, 3 RSA designs, and 5 RISC processor designs in total.

Specifically, we collect 36 open-source AES cores from GitHub and OpenCores. Of these, 18 are implemented in Verilog and are evaluated. The remaining 20 are written in SystemVerilog, which Transys currently does not support. In addition, we collect 20 AES cores with injected trojans from TrustHub [90, 93]. We also collect 11 open-source RSA cores from GitHub, OpenCores, and TrustHub, and 3 of the them are implemented in Verilog. For CPU designs, we collect 5 open-source RISC processor, 3 of them are implementations of the OpenRISC architecture (OR1200, Espresso, Cappuccino) and 2 of them are implementations of the RISC-V architecture (OpenV, Picorv32).

To evaluate Transys on the AES and RSA designs, we draft 17 assertions for 3 designs to feed as input to Transys (see Table 4.7). We also collect 14 information-flow security assertions for AES and RSA cores from the IFT Model project [62] (see Table 4.9). These assertions are drafted for 3 AES and 3 RSA implementations, and cover properties about confidentiality, integrity, isolation and timing channels. The first 9 assertions in Table 4.9 are drafted for general AES and RSA designs, and the last 5 assertions are drafted for specific malicious designs. Thus, we use the first 9 assertions for our translation evaluation. We use the last 5 for evaluating the security impact of translated assertions (see Section 4.6.7). To evaluate Transys on the processor designs, we collect 10 security assertions for OR1200 processors from the SPECS [59], Security Checkers [21], and SCIFinder [112] projects (see Table 4.8). These assertions represent the 6 types of security properties in Table 4.1.

4.6.2 Translation Results

To evaluate whether Transys can successfully translate security-critical assertions from one design to another, we test whether it can successfully generate valid assertions for the new designs. Table 4.10 shows the main translation results. Figures 4.6, 4.7, 4.8, and 4.9 show the detailed results of the translation rate for each assertion.

A No.	Assertions
A27-01	(keysched.round_i == 1) → (keysched.rcon_o == 'h1)
A27-02	(keysched.round_i == 2) → (keysched.rcon_o == 'h2)
A27-03	(keysched.round_i == 3) → (keysched.rcon_o == 'h4)
A27-04	(keysched.round_i == 4) → (keysched.rcon_o == 'h8)
A27-05	(keysched.round_i == 5) → (keysched.rcon_o == 'h10)
A27-06	(keysched.round_i == 6) → (keysched.rcon_o == 'h20)
A27-07	(keysched.round_i == 7) → (keysched.rcon_o == 'h40)
A27-08	(keysched.round_i == 8) → (keysched.rcon_o == 'h80)
A27-09	(keysched.round_i == 9) → (keysched.rcon_o == 'h1b)
A27-10	(keysched.round_i == 10) → (keysched.rcon_o == 'h36)
A28-01	(keysched.state==4)→(keysched.next_key_reg[31:0]== keysched.next_key_reg[63:32]⊕keysched.last_key_i[31:0])
A28-02	(keysched.state==4)→(keysched.next_key_reg[63:32]== keysched.next_key_reg[95:64]⊕keysched.last_key_i[63:32])
A28-03	(keysched.state==4)→(keysched.next_key_reg[95:64]== keysched.next_key_reg[127:96]⊕keysched.last_key_i[95:64])
A28-04	(keysched.state==4)→(keysched.next_key_reg[127:96]== keysched.col_t⊕keysched.last_key_i[127:96]⊕{keysched.rcon_o, 32'h0})
A29-01	(aes_sbox.d ⊕ aes_sbox.a != 8'hff)
A29-02	(aes_sbox.d != aes_sbox.a)
A32-01	(rsa.msg_in != rsa.msg_out)

Table 4.7: Security critical assertions of cryptographic hardware. Assertion A27-01—10 and A28-01—04 are drafted for the AES09 design; Assertion A29-01—02 are for AES11; Assertion A32-01 is for RSA03. The first number in A No. refers to the property number in Table 4.2.

ANo.	Example Assertions
A01	((or1200_ctrl.ex_insn&'hFC000000)≫26=='h21)→ (or1200_rf.rf_dataw==dcpu_dat_o)
A03	((or1200_ctrl.ex_insn&'hFC000000)≫26=='h21)→ (dcpu_adr_o==operand_a+ex_simm)
A04	(or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)| (or1200_rf.rf_dataw==0)
A08	((or1200_ctrl.ex_insn&'hFC000000)≫26==9)→ (or1200_sprs.to_sr==or1200_except.esr)
A09	((or1200_ctrl.ex_insn&'hFC000000)≫26==9)→ (or1200_genpc.pc==or1200_except.epcr)
A15	((or1200_ctrl.ex_insn&'hFC000000)≫26==1)→ (or1200_rf.rf_addrw==9)
A17	((or1200_ctrl.ex_insn&'hFFE00000)≫21==1826)&(operand_a>operand_b)→ (or1200_sprs.to_sr[9]==1)
A19	((or1200_ctrl.ex_insn&'hFC000000)≫26==48)→ (or1200_sprs.spr_dat_o==operand_b)
A23	((or1200_ctrl.ex_insn&'hFC000000)≫26=='h38)→((or1200_ctrl.ex_insn&'h03e00000)≫21 ==or1200_rf.addrw)
A26	((or1200_ctrl.ex_insn&'hFC000000)≫26!='h1c)

Table 4.8: Security critical assertions of the OR1200 design. The first number in A No. refers to the property number in Table 4.1.

(1) For AES designs, the overall translation rate is 93%. The 8 failures in the Transformation Pass occur in translating A28 to the AES08 design, and A29 to the AES06 and AES12 designs. The reason that the

A No.	Assertion	Core
A36-01	set key[0] := high; assert cipher[0] == high	AES-04
A36-02	set key[1] := high; assert cipher[7:0] == high	AES-04
A36-03	set key[1] := high; assert cipher[31:0] == high	AES-04
A36-04	set key[1] := high; assert cipher[63:0] == high	AES-04
A33-06	set indata[1] := high; assert count[1] == low	RSA-03
A36-05	set inExp[1] := high; assert cipher[1] == high when ready == 1	RSA-03
A36-06	set inExp[0] := high; assert cipher[0] == low	RSA-03
A37-01	set inExp[0] := high; assert ready == low	RSA-03
A37-02	set inExp[1] := high; assert ready == low	RSA-03
A33-01	set key[0] := high; assert Antena == low	AES-T400
A33-02	set key[0] := high; assert TSC_SHIFTReg[0] == low	AES-T400
A33-03	set key[0] := high; assert Capacitance[0] == low	AES-T1100
A33-04	set key[1] := high; assert Capacitance[1] == low	AES-T1100
A33-05	set key[1]:=high; assert Capacitance[0] == high	AES-T1100

Table 4.9: Information flow assertions of cryptographic hardware. The first num in A No. refers to the property num in Table 4.4, 4.3.

Transformation Pass fails is that the highest-score variable found in the first pass is incorrect, making it impossible to find a subgraph in the PDG that includes at least two variables in the assertions.

For the AES05 design, the implementation of one module is missing in the code we collected, which caused 16 failures in the Refinement Pass. Transys can translate the assertions in the first two passes, but fails in the third pass as the code is incomplete. This shows that our first two passes do not rely on the completeness of the code base, but the third pass requires that the code should be complete. If we comment out the part of the code that instantiates the missing module in the original design, Transys can successfully translate the assertions to AES05.

(2) For AES designs with trojans, Transys successfully translates all assertions to the 20 trojan-injected AES designs. For example, as shown in Figure 4.7, Transys translates 4 AES Information Flow Tracking assertions written in the AES-04 design (a trojan-free design) to the 20 AES designs with different trojans injected. The trojans include leaking the secret key through AM radio, leakage current, spread spectrum communications, and draining the battery to cause denial-of-service [90, 93]. In this case, the translated assertions can potentially be used to detect the injected trojans.

(3) For processor designs, we translate assertions from the OR1200 to 5 processor designs in two different architectures. We found that the assertions A19 and A26 do not exist in the two RISC-V cores: A19 and A26 are about the `l.mtspr` instruction and custom instructions, which are not implemented in the two RISC-V cores.

We first evaluate the remaining 46 of the 50 total translations, and among those the translation rate is 85%. Among the 7 failed cases, 3 of them fail in the Transformation Pass and 4 of them fail in the Refinement Pass—Transys cannot find valid preconditions to make the consequent true. All the failed cases happen when we try to translate the assertions from OR1K designs to RISC-V designs: 2 of them are to the OpenV core, and 5 of them are to the Picorv32 core.

We separately evaluate the 4 translations for which the assertion does not exist in the target design. Transys successfully translates 3 of them. These 3 new assertions are valid but the policies they capture are different than the original assertions' policies. The false positive rate here is 75%.

(4) For RSA designs, we translate 1 assertion mined from the specification, and 5 Information Flow Tracking assertions. All of them are successfully translated to the new designs.

(5) We also test Transys by translating the assertions back to the original designs. Transys successfully translates all assertions back to the original designs. This implies the variable mapping pass can map the variables to themselves, and the second and third pass preserve the structure of the assertions.

Design	**AES**	**AES w/ Trojan**	**CPU**	**RSA**	**Total**
Total Translations	360	400	46	18	824
Total Succ	336	400	39	18	793
Fail in VM Pass	0	0	0	0	0
Fail in T Pass	8	0	3	0	11
Fail in R Pass	16	0	4	0	20
Total Transl. Rate	93%	100%	85%	100%	96%

Table 4.10: Main results of assertion translation for 18 AES designs, 20 AES designs with trojans, 5 processor designs, and 3 RSA designs.

4.6.3 Quality

To evaluate the quality of the translated assertions, we first check whether the translated assertions are valid for the target design using the model checking tool Cadence IFV. We then manually review the assertions alongside the design specifications to determine whether the translated assertions are semantically equivalent to the original assertions.

Validity

We check whether the translated assertions are valid by adding them to the target designs and running Cadence IFV. Figures 4.6, 4.7, 4.8, and 4.9 shows the results. For the nine Information Flow Tracking

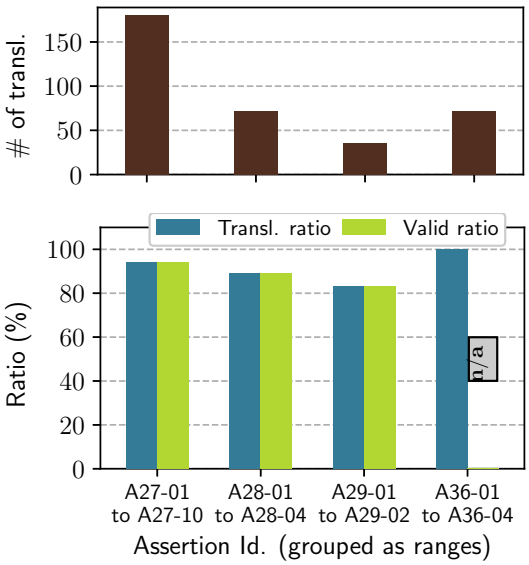

Figure 4.6: AES01—AES18 translation results: total translation number and success translation rate.

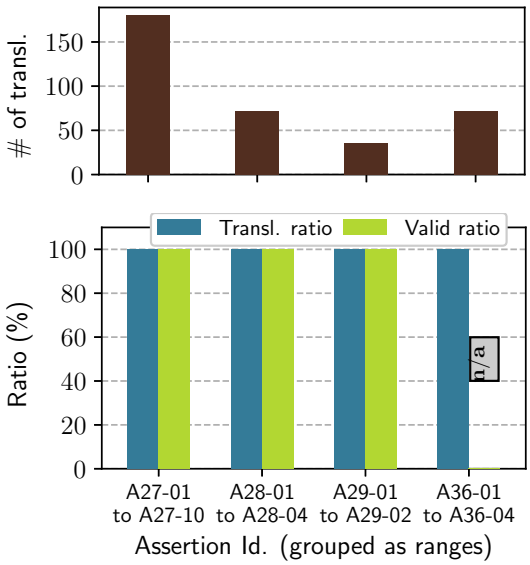

Figure 4.7: AES-T100—AES-T2100 translation results: total translation number and success translation rate.

assertions, we do not have the tool to check the validity of the translated assertions (167 in total) and thus their validity result is not available. All the other 626 translated assertions can pass verification by Cadence IFV, indicating that the assertions Transys generates are valid.

Equivalence

Figures 4.10, 4.11, 4.12, and 4.13 show the results of the equivalence checking. Type equivalence refers to the case that the translated assertion and the original assertion belong to the same type or module of security properties, as given in column 1 of Tables 4.1, 4.2, 4.3, and 4.4. Semantic equivalence refers to the case that the translated assertion and the original assertion are semantically the same.

The translation of assertions to trojan-injected AES designs achieves 100% semantic equivalence rate. For other designs, the translation of 23 (64%) assertions has type and semantic equivalence rate above 60% (between 60% and 100%). The translations of the remaining 13 (36%) assertions have type and semantic equivalence rate between 20% to 50%. The low rates mainly happen in two cases: the translation of Information Flow Tracking assertions and the translation from OpenRISC cores to RISC-V cores.

The main reason for the translated assertions to fail to capture the meaning of the original assertion is because the variable mapping pass fails to map to an accurate variable or even fails to map to the correct

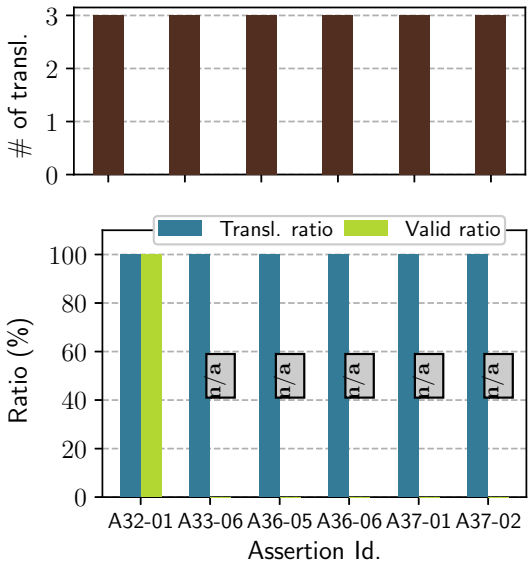

Figure 4.8: RSA01—RSA03 translation results: total translation number and success translation rate.

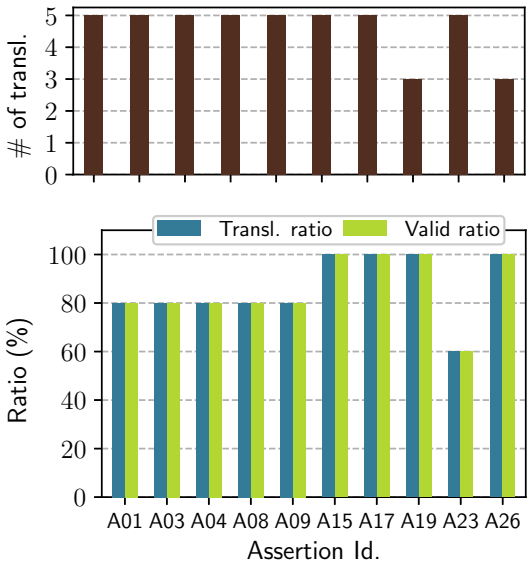

Figure 4.9: CPU translation results: total transl. number and success transl. rate.

module in the target design. In all our experiments, we choose the parameters in the Variable Mapping Phase empirically to be $\alpha : \beta : \gamma = 3 : 2 : 1$. This combination works well in most cases, but not all of them.

4.6.4 Case Studies

In this section, we show 3 examples: (1) translation from one AES design to another AES design; (2) translation from one processor design to two different processor designs from two architectures (OR1K architecture and RISC-V architecture); (3) translating an Information Flow Tracking assertion from one trojan-free AES design to a trojan-injected design.

Example 1

We show the details of translating the assertion A28-01 from AES09 to all AES designs. Table 4.11 shows the resulting assertions. For the assertions in AES02, AES03, AES12, we classify them as in the same type as the original assertion, but not as having equivalent semantics. For the assertions in AES16 and AES17, they belong to the calculation of round keys, and thus are neither type equivalent nor semantically equivalent to the original assertion.

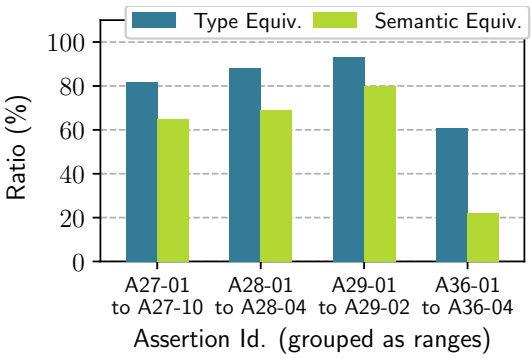

Figure 4.10: Type and semantic equivalent for AES01—AES18 designs.

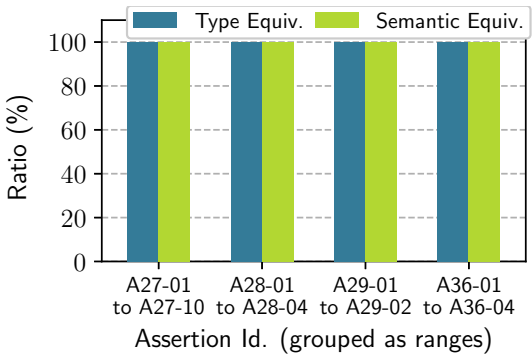

Figure 4.11: Type and semantic equivalent for AES-T100—AES-T2100 designs.

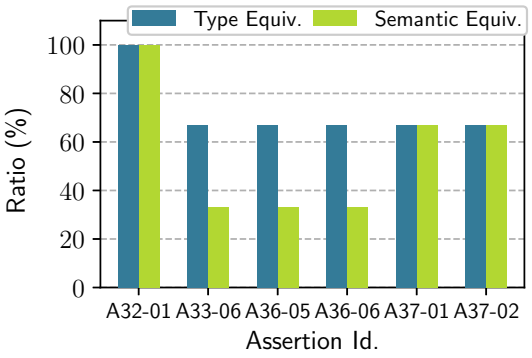

Figure 4.12: Type and semantic equivalent for RSA01—RSA03 designs.

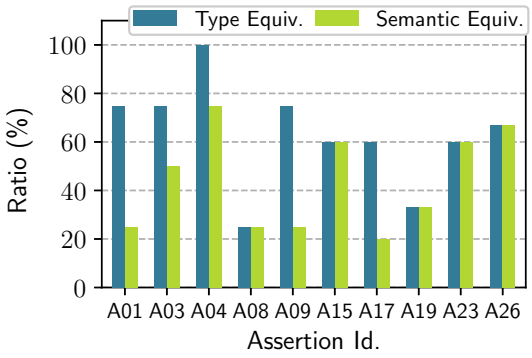

Figure 4.13: Type and semantic equivalent for CPU designs.

Table 4.12 shows the detailed results of translating assertion A28-01 from AES09 to AES03. After the Variable Mapping Pass, `keysched.next_key_reg` and `keysched.last_key_i` are both mapped to `key_exp.key_in`. The assertion generated is not valid yet. After the Transformation Pass, Transys outputs 5 assertions. These assertions are generated from the part of the PDG that contains the variable `key_exp.key_in`. Only the 5th assertion is valid. Finally, from the Refinement Pass, all the 4 assertions are refined and are valid. It is worth noting that the antecedents generated from the Refinement Pass are neither close to the part of the code of the consequent nor similar to the original code, and thus it would be difficult for a human to figure them out manually.

No.	Translation Results
Original	(keysched.state==4)→(keysched.next_key_reg[31:0]==keysched.next_key_reg[63:32] ⊕keysched.last_key_i[31:0])
AES01	(round_ctr_reg[0])&(key_mem_we)&(!round_ctr_inc)→ (key_mem_new==key[255:128])
AES02	u1.r1.t0.w0 == u1.r1.t0.key[127:96]
AES03	(key_exp.key_start==1)&(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[255:192]))| (key_exp.wr3==0)
AES04	a1.k0b == a1.k0a ⊕ a1.k4a
AES05	n.a.
AES06	(!u0.kld)→#1(u0.w[0]==prev(u0.w[0]⊕u0.subword⊕u0.rcon))
AES07	a1.k0a == prev({a1.k0[31:24] ⊕ rcon, a1.k0[23:0]})
AES08	n.a.
AES09	(keysched.state==4)→(keysched.next_key_reg[31:0]==keysched.next_key_reg[63:32] ⊕keysched.last_key_i[31:0])
AES10	AES_CORE_DATAPATH.KEY_EXPANDER.key[3]== AES_CORE_DATAPATH.KEY_EXPANDER.key_in[31:0]
AES11	(!u0.kld) → #1 (u0.w[1] == prev(u0.w[0] ⊕ u0.w[1] ⊕ u0.subword ⊕ u0.rcon))
AES12	w0_next == sbox_out ⊕ rcon ⊕ w0
AES13	w4 == key[127:96] ⊕ subword ⊕ 16777216
AES14	w4 == w0 ⊕ subword ⊕ {rcon2[31:24],24'b0}
AES15	wNext[1] == w[1] ⊕ wNext[0]
AES16	roundkey_text == mixcolumns_text ⊕ okey
AES17	roundkey_text == mixcolumns_text ⊕ okey
AES18	w7 == key[127:96] ⊕ key[95:64] ⊕ key[63:32] ⊕ key[31:0] ⊕ subword ⊕ 16777216

Table 4.11: The results of translating A28-01 to 18 AES designs.

Example 2

Table 4.13 shows the translation results for translating assertion A04 to five processor designs. The translation fails in the Refinement Pass when translating the assertion to the OpenV design. For the other designs, Transys can successfully generate valid assertions. The translated assertions for the OR1200, Espresso, and Cappuccino processors are semantically equivalent. These three designs are all implementations of the OR1K architecture and it is easier to translate assertions among them. The assertion for the Picorv32 does not capture the same semantic meaning, but it also belongs to the type of security properties that are relavent to the memory.

Pass	Translation Results
VM Pass	(key_exp.pstate==4)→(key_exp.key_in[31:0]==key_exp.key_in[63:32] ⊕key_exp.key_in[31:0])
ST Pass	(key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[255:192])
	(key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[191:128])
	(key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[127:64])
	(key_exp.pstate==4) → (key_exp.wr_data==key_exp.key_in[63:0])
	i_key == key_exp.key_in
CR Pass	(key_exp.key_start==1)&(key_exp.round[1:0]==2'b01)→#1 (key_exp.wr_data==prev(key_exp.key_in[255:192]))|(key_exp.wr3==0)
	(key_exp.key_start==0)&(key_exp.key_start_L==1) &(key_exp.round[1:0]==2'b01)→#1(key_exp.wr_data== prev(key_exp.key_in[191:128]))|(key_exp.wr3==0)
	(key_exp.key_start==0)&(key_exp.wr3==1)&(key_exp.init_wr3==1) &(key_exp.round[1:0]==2'b01)→#1(key_exp.wr_data== prev(key_exp.key_in[127:64]))|(key_exp.wr3==0)
	(key_exp.key_start==0)&(key_exp.wr3==1)&(key_exp.init_wr4==1) &(key_exp.round[1:0]==2'b01)→#1(key_exp.wr_data==prev(key_exp.key_in[63:0]))| (key_exp.wr3==0)
	i_key == key_exp.key_in

Table 4.12: Detailed results of translating A28-01 to the AES03 design. VM: Variable Mapping, ST: Structural Transformation, CR: Constraint Refinement.

No.	Translation Results
Original	(or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)| (or1200_rf.rf_dataw==0)
OR1200	(or1200_rf.rf_we==1)→(or1200_rf.rf_addrw!=0)| (or1200_rf.rf_dataw==0)
Espresso	(mor1kx_rf_espresso.rfa_o_use_last)&(mor1kx_rf_espresso.result_last[0]==0) &(mor1kx_rf_espresso.rfd_last==mor1kx_rf_espresso.rfa_r) &(mor1kx_rf_espresso.rfa_adr_i[0])&(mor1kx_rf_espresso.rfa_o[0]==0)→ (mor1kx_rf_espresso.rfa_adr_i≠0)| (mor1kx_rf_espresso.rfa_o==0)
Cappuccino	mor1kx_rf_cappuccino.rf_wradr== mor1kx_rf_cappuccino.wb_rfd_adr_i
	(mor1kx_rf_cappuccino.rf_wradr)&(mor1kx_rf_cappuccino.rf_wrdat)→ (mor1kx_rf_cappuccino.rf_wrdat==0)|(mor1kx_rf_cappuccino.rf_wraddr!=0)
OpenV	n.a.
Picorv32	picorv32.dbg_mem_rdata == picorv32.mem_rdata

Table 4.13: The results of translating A04 to 5 CPU designs.

Example 3

In this example, the Information Flow Tracking assertion A36-01 for the AES04 design is translated to the AES-T400 design. In the AES-T400 design, the injected trojan utilizes an unused pin to generate an RF signal that can be used to transmit the key bits. The leaked data can be received by an AM radio,

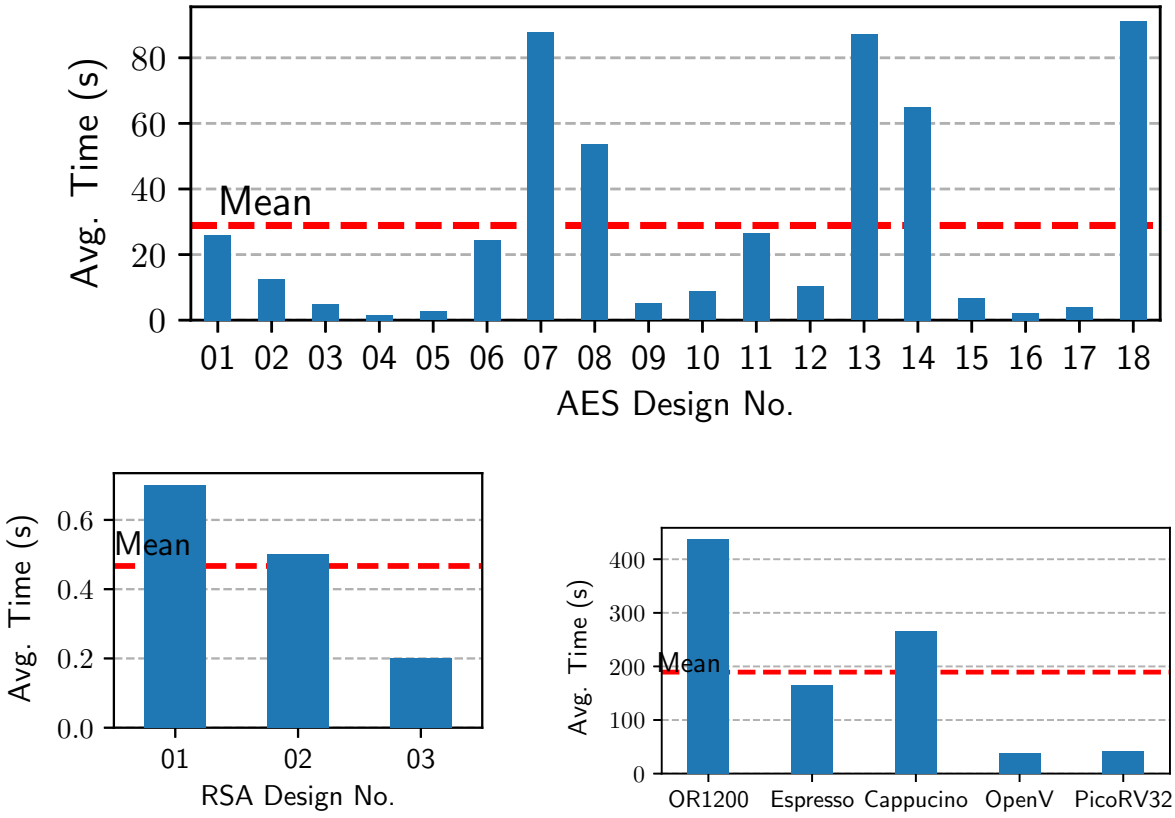

Figure 4.14: Translation time for the AES, RSA and CPU designs.

and can be interpreted with a specific beep scheme. The trojan is implemented in two additional modules: `AM_Transmission` and `Trojan_Trigger`. When a predefined plaintext is observed, the trojan will be triggered and the `AM_Transmission` module will output the key to the *Antena* signal following the beep scheme to leak data.

Ideally, the key will flow only to the output ciphertext (A36-01). The result of our translation for A36-01 to the AES-T400 design is: `set key[0] := high; assert cipher[0] == high`. This indicates that Transys can successfully translate the assertion to a new design and is not influenced by the two additional modules of the trojan.

4.6.5 Performance

We evaluate the total time it takes for Transys to translate each assertion from a source design to a target design. Figure 4.14 shows the results. The translation times for the trojan-injected AES designs are similar to the time for the trojan-free AES design, and are not shown due to space constraints.

Processor	Lines of Code
OR1200	18296
Espresso	8758
Cappucino	11902
OpenV	2211
pircov	2986

Table 4.14: Lines of code of the RISC processor designs.

We observe that the translation time varies across different designs, depending on their complexity – the translation time grows approximately linearly as the size of a design grows. Table 4.14 shows the lines of code of the RISC processor designs we used. The average times for translating one assertion for AES designs, RSA designs and CPU designs are 28.8 seconds, 0.46 seconds, and 189 seconds, respectively. For AES and RSA designs, most of the translation time is spent on the Refinement Pass. For processor designs, most of the translation time is spent on the Variable Mapping Pass. The maximum average property-translation time is 436.8 seconds for the OR1200 design. The results suggest that Transys is practical enough to be used by hardware designers on a daily basis to quickly generate security assertions through translating existing ones.

4.6.6 Effectiveness of Each Pass

We evaluate the effectiveness of each pass on translating assertions across AES and processor designs. Tables 4.15 and 4.16 show the ratio of valid results at the end of each pass. We observe that each pass increases the valid-to-invalid ratio substantially, indicating that each pass is effective.

4.6.7 Security Impact

In this section, we discuss the security impact of the translated information flow tracking assertions when there is a vulnerability in the code. Assertions A33-01—A33-05 in Table 4.9 can detect trojans in AES-T400 and AES-T1100 [62]. We translate these five assertions to the AES cores with trojans.

We do not have access to the information flow tracking tool [62] needed to add the tracking logic necessary to verify whether the translated assertions can detect trojans. Therefore, we instead compare the translated assertions with the original assertions, and compare the trojans between designs. If the assertions are logically equivalent, and the information leakage circuits are the same other than the triggering mechanism, then we infer that the translated assertions would detect the injected trojans as well.

Assertion	VM Pass	ST Pass	CR Pass
Total Transl.	360	352	336
Valid Ratio	14%	52%	93%

Table 4.15: Accumulative valid ratio of each pass for AES designs.

Assertion	VM Pass	ST Pass	CR Pass
Total Transl.	46	43	39
Valid Ratio	39%	59%	85%

Table 4.16: Accumulative valid ratio of each pass for CPU designs.

Orig Assert No.	**Trans. assert can detect trojans in**
A33-01, A33-02	AES-T1600, AES-T1700, AES-T400
A33-03, A33-04	AES-T100, AES-T1000, AES-T1100, AES-T1200
A33-05	AES-T200, AES-T700, AES-T800, AES-T900

Table 4.17: Results of security impact of translated assertions to detect trojans in AES cores.

Table 4.17 shows the results. The translated assertions of A33-01, A33-02 would detect trojans in three AES designs, and the translated assertions of A33-03—A33-05 would detect trojans in eight AES designs. For the remaining nine trojan-injected designs, we do not have assertions that can detect the trojans and therefore we cannot determine whether translated assertions would detect them.

4.6.8 Bugs in the Code

We discuss three examples to show the translation results of Transys when there is a bug in the design. For different types of bugs, the translation results of Transys can be: failing to translate, outputting trivially true assertions, or propagating the bug to the resulting assertions.

Translation Failed

The first example shows the case of translation failure. In the AES05 design we mentioned in Section 4.6.2, part of the code base is missing. When we use Transys to translate the assertions to the AES05 design, we get the error message in the Refinement Pass showing that some modules or cells are not part of the design. Thus, one possible reason for translation failure is missing parts of the code. This corresponds to the case of no refinement output at all.

Trivial Assertions

The second example shows the case that a certain constraint should be explicitly stated in the design, but it is not. We show the `GPR0` bug in the OpenRISC cores. In the OR1K specification, the general purpose register `R0` should always be set to zero [69]. A violation of this property can lead to malicious modification of the memory data or memory address in calculation. This bug exists in both the Espresso and the Cappuccino designs [111].

We translate the assertion that enforces `R0` to always be 0 (A04 in Table 4.8) from the OR1200 to both the Espresso and the Cappuccino designs. The results are shown in Table 4.13.

The result assertion for the Espresso design can be simplified to:

`(mor1kx_rf_espresso.rfa_adr_i≠0)→ (mor1kx_rf_espresso.rfa_adr_i≠0).`

The result assertion for the Cappuccino design can be simplified to:

`(mor1kx_rf_cappuccino.rf_wraddr≠0)→ (mor1kx_rf_cappuccino.rf_wraddr≠0).`

In both cases, the assertions are trivially true ($A \rightarrow A$) and there are no other valid and meaningful assertions. Thus, a bug in the design due to missing constraints is reflected in translation results that only have trivially true assertions.

Overly Restrictive Assertions

The third example shows the case that some malicious or buggy code are explicitly added in the design. For the AES assertion A29-02 from the AES11 design, Transys successfully translate it to the AES18 design: `aes_sbox.a != aes_sbox.d`. This assertion states the security property that the S-box should avoid any fixed points. We then maliciously modify the S-box design in AES18 such that when the input to the S-box is `8'hff`, it should output `8'h16` but instead outputs `8'hff`. We then run Transys to translate this assertion again and we get the new assertion: `(aes_sbox.a[7] ≠ aes_sbox.d[7]) → (aes_sbox.a ≠ aes_sbox.d)`. This new assertion is valid for the buggy design. With the additional antecedent, hardware experts can easily identify the bug and the condition to trigger it.Thus, a malicious bug in the design can manifest itself in the translated assertions (typically as additional antecedents).

4.7 Summary

In this work, we advocate building security properties for new designs by leveraging existing properties. We present Transys, an automated tool that translates given security assertions from one hardware design to another in three passes—variable mapping pass, structural transformation pass and constraint refinement pass. Transys is able to translate 27 temporal logic assertions and 11 information flow tracking assertions across 38 AES designs, 3 RSA designs, and 5 RISC processor designs. The overall translation success rate is 96%. Among them, the translations of 23 (64%) assertions achieve semantic equivalence rates of above 60%. The average translation time per assertion is about 70 seconds.

CHAPTER 5
GENERATING HARDWARE EXPLOIT PROGRAMS

In this chapter, I present the details of Coppelia, the exploit generation tool outlined in Section 1.4. I present the overview and challenges in Section 5.1. I then describe the details and implementation of the three components of Coppelia in Section 5.2 and 5.3. I show the evaluation in Section 5.4.

5.1 Overview and Challenges

The current state of the art for finding errors in processor designs is to use formal static analysis or simulation-based testing. However, neither method is complete. We develop here a third option: software-style symbolic execution for hardware designs. It systematically explores paths in hardware designs to uncover errors.

Uncovering a potential bug is only the first step during a security validation process. Hardware designers must then assess the severity and security implication of each found bug. Our work takes an end-to-end approach by automatically generating software exploits to expose potential vulnerabilities. In particular, for each found bug, the tool generates a sequence of instructions that will trigger the bug plus a program stub that carries an exploit payload. The payload stub is generated based on the violated security properties. Together, the trigger and the payload stub form a complete exploit program to demonstrate a possible, concrete attack.

Generating the exploits not only allows hardware designers to uncover and reproduce vulnerabilities with concrete test cases, but also helps them contextualize, analyze and assess the security implications of a potential vulnerability. Furthermore, by using whether an exploit can be generated as a criterion, hardware designers can validate patches and refine assertions.

5.1.1 Challenges

Two characteristics of hardware designs require rethinking the standard symbolic execution typically used in the software domain. The symbolic execution of a hardware design represents an exploration of the

design for a single clock cycle, but hardware executes continuously, and security vulnerabilities may only become apparent many clock cycles after the initial state. Symbolic execution can never provide exhaustive coverage for systems with infinite execution trees [66]. As a further complication, the large state space of a modern processor design precludes joining redundant states during exploration, as is sometimes done for software designs [42].

Second, security properties developed for hardware designs capture the semantics of particular signals and their connecting logic. By contrast, security properties developed for software are applicable throughout a code base. For example, invalid- or missing-bounds checks occur throughout a software code base, and a symbolic execution engine that looks for such violations is likely to find more examples just by exploring more broadly and deeply. Compare this to, say, the security-critical property of some RISC architectures that the general purpose register `R0` should always be set to zero. A violation of this property will occur only in that part of the design that touches the `R0` register. Finding such violations is akin to finding a needle in a haystack; if an exhaustive search is not possible, a strategy is needed to focus the search toward the target. We show how we tackle these challenges in the following sections.

5.2 Design

We first provide an overview of the three phases of Coppelia: preprocessing, building a trigger, and adding the payload. We then describe each phase in detail in the following sections. Figure 5.2 shows the workflow of Coppelia.

We are targeting vulnerabilities in a processor design that are exploitable, post-deployment, by software. We assume the attacker does not modify the processor design, but is capable of finding vulnerabilities that exist within the design. Post deployment, we assume the attacker is able to send network packets, execute a particular sequence of instructions, or both on the target machine.

5.2.1 Overview of Coppelia

Coppelia takes as input an HDL implementation of a hardware design and a set of security assertions.

Preprocessing. To begin, Coppelia translates the RTL hardware design from an HDL implementation to C++. We use the Verilator tool [7] for this step and can translate designs written in Verilog or SystemVerilog, although the basic approach would apply to other HDLs as well. Translating the RTL design to C++ allows

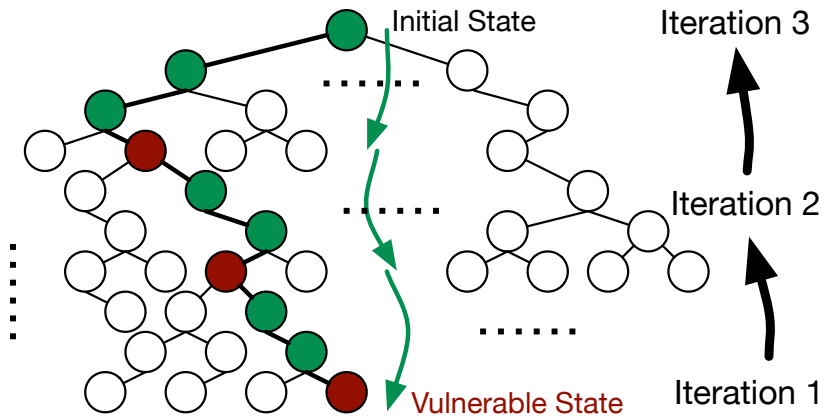

Figure 5.1: Backward symbolic execution strategy: We search for a path from the last cycle to the first cycle (black arrows). Within each cycle, we symbolically execute the hardware design forwardly (green arrows).

us to take advantage of KLEE [27], a mature symbolic execution engine, and use it as the foundation of Coppelia. We discuss this step further in Section 5.2.2. After translation, Coppelia adds the security-critical assertions to the generated testbench and compiles the newly translated design to LLVM bytecode using the Clang compiler [3].

Building a trigger. A vulnerability is defined as a processor state s_n in which a security-critical assertion is violated. Assertions are boolean-valued functions written over (a subset of) the state-holding elements of the processor. They encode desired security properties, and can express safety properties, but not liveness or hyperproperties [38]. The goal is to find a sequence of inputs $i_0, i_1, \ldots, i_k$ that take the system from the initial state s_0 to the violating state s_e. Coppelia builds the sequence backwards, first finding input i_k then i_{k-1} and so on. Each input is found by symbolically exploring the processor (Figure 5.1).

Adding the payload. To better contextualize and analyze the security threat, Coppelia goes beyond triggering the vulnerability. It adds a program stub to complete the exploit. These program stubs are generated according to the category of the security-critical assertion violated. We describe this step in detail in section 5.2.6.

5.2.2 Preprocessing: Transcompiling RTL to C++

In the first phase, we use Verilator [7] to translate the RTL Verilog to logically equivalent C++ code. Verilator is an open source Verilog simulator. It compiles the synthesizable subset of Verilog into cycle-accurate C++ or SystemC code.

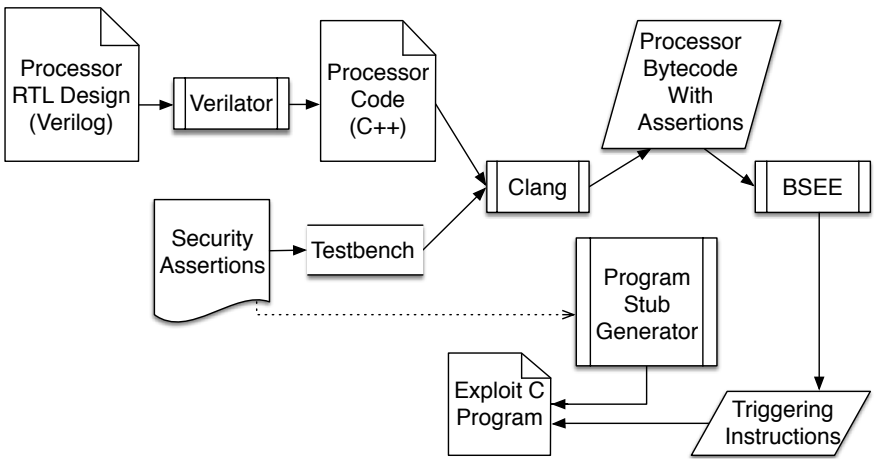

Figure 5.2: Workflow of Coppelia. The process labeled BSEE is the backward symbolic execution engine.

Verilator starts with a preprocessing step in which it propagates parameters, determines expression widths, eliminates dead code, unrolls loops, and inlines modules and tasks. It also eliminates any possible 3-state by replacing don't-care values (X) with random values. Next, Verilator does the translation. The translation of Verilog blocking statements is straightforward as these statements are semantically similar to that of straight-line C++ code. On the other hand, the translation of Verilog non-blocking statements requires additional analysis as there is no semantic equivalent in C++ to the simultaneous execution of multiple statements. Verilator imposes an order on non-blocking assignments and introduces temporary C++ variables so that the resulting straight-line C++ code produces a faithful simulation of the Verilog's behavior at each clock cycle boundary. Finally, Verilator cleans up the code, corrects expression widths, and outputs the result in C++ [94].

In the resulting C++ code, each class corresponds to a module in the Verilog code. The hierarchy of the C++ classes matches the hierarchy of Verilog modules. The interface to the top C++ class is an `eval()` function that calls the functions inside each class necessary to simulate the processor design for a single clock transition. There are two major loops inside the `eval()` function: the initialize loop and the main change loop. The initialize loop executes the initialization statements and propagates the initial values through the design. The main change loop executes circuit logic and propagates value changes to each module. Two calls to the eval() function represent a single clock cycle.

The input signals remain stable during a single execution of the `eval()` function, meaning inputs will only change at clock tick boundaries. This assumption ensures the circuit model converges and improves the efficiency for the code analysis.

5.2.3 Background, Notation, and Definitions

Before describing how we build the trigger, we review symbolic execution, introduce notation, and define the problem.

In standard forward symbolic execution input values are replaced with symbols that represent the set of possible values in the domain of the function. The symbolic exploration of a program can be represented by a tree $\mathcal{E}$. Each path through the tree represents a path of execution taken during the symbolic exploration. Each node represents a line of code in the program; the root node represents the entry point and the leaves represent an exit point. Associated with each node is the current program state – the valuation of variables – and a path condition. The path condition (π) for node n defines constraints over the program's input domain such that if the program is run with input values satisfying the constraints, execution would be driven down the path from root to n.

The symbolic exploration of a processor – achieved by symbolically exploring two consecutive calls to the `eval()` function – corresponds to one clock cycle of the design. The root node of the resulting tree represents the state of the processor at a clock-cycle boundary and each leaf of the tree represents a possible next-state of the processor. When referring to a processor state we are referring to a root or leaf node in the symbolic execution tree, not an internal node; the root and leaf nodes represent the processor at cycle boundaries.

We will refer to a tuple (n, i, π) associated with a symbolic exploration tree $\mathcal{E}$. The tuple defines a particular leaf node of interest n, the inputs i that would guide execution from the root node down the path to leaf node n, and the path constraints π associated with leaf node n. We also define a *test case* as a satisfying solution to a path constraint. A test case is one set of concrete input values that will drive the processor down the path associated with the path constraint.

The execution of multiple clock cycles in the processor is represented by multiple symbolic explorations of the design (see Figure 5.4). Each leaf node of a tree $\mathcal{E}_j$ becomes a root node for a tree $\mathcal{E}_{j+1}$ representing the next exploration of the design, i.e., the next clock cycle of the processor.

We aim to find a sequence of inputs that will take the processor from the reset state to an error state. We define the problem in terms of symbolic exploration trees.

Problem Statement. Given s_e, an error state of the processor in which a security-critical assertion is violated, find a sequence of symbolic exploration trees $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_k$, and for each tree a particular leaf node $n_0, n_1, \ldots, n_k$ such that

- The root node of the first tree $\mathcal{E}_0$ is the reset state of the processor,
- The leaf node n_k associated with tree $\mathcal{E}_k$ represents the error state of the processor, and
- The leaf node n_j associated with tree $\mathcal{E}_j$ can be *matched* to the root node of tree $\mathcal{E}_{j+1}$.

We say the leaf node of one tree can be *matched* to the root node of a second tree if and only if the nodes are compatible: concrete values are equal and constraints over symbolic values given in one node are mutually satisfiable with constraints over symbolic values given in the second node.

If the above requirements are satisfied then the sequence of path constraints $\pi_0, \pi_1, \ldots, \pi_k$ provided by the sequence of leaf nodes $n_0, n_1, \ldots, n_k$ define the sequence of inputs to the processor that will take the processor from an initial state to the error state.

5.2.4 Building the trigger: Backward Symbolic Execution

An error state that is M clock cycles away from the initial state will only be found after $2M$ iterations of the eval() loops. The search space for forward symbolic execution is exponential in the number of loop iterations and becomes untenable for even small values of M. (See Section 5.2.4 for a discussion of the search complexity.)

The key insight of our work is that hardware is well suited to a backward search strategy for symbolic execution. The specificity of security assertions in hardware designs make them amenable to such a targeted search strategy, and the lack of dynamically linked libraries, pointers, and complex computation makes the backward strategy possible.

Rather than start at the processor's initial state and search forward, Coppelia uses backward symbolic execution to start at an error state and search backward. In the first iteration, the backward symbolic execution engine looks for a procesor state s that can reach an assertion failure in one clock cycle, given the right set of inputs. If such an s is found, the problem is ideally reduced: from finding a path of length M from the initial state to the error state to finding a path of length $M - 1$ from the initial state to state s. The backward symbolic execution engine continues in this way, stepping back from the error state toward the initial state, one clock cycle at a time.

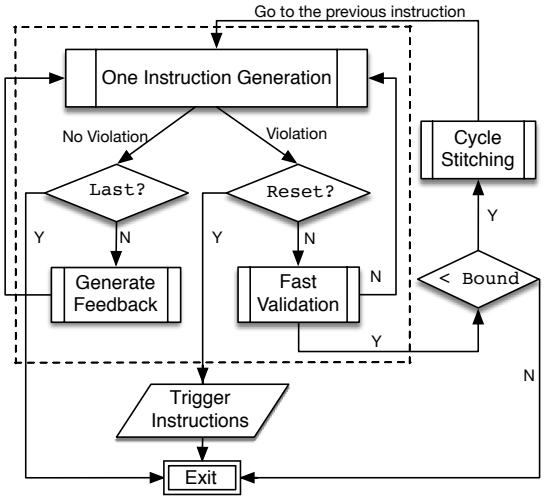

Figure 5.3: Workflow of Backward Symbolic Execution

We cannot, in the general case, be sure that each iteration actually reduces the problem. An intermediate state s may not be reachable from the initial state, or we may find ourselves stitching together a path that has a loop and never converges toward the initial state. We introduce heuristics to help the backward symbolic execution engine identify unreachable states, loops, or paths that are not tending toward the initial state.

Backward Symbolic Execution Engine

We describe the workflow of our hardware-oriented backward symbolic execution engine (see Figure 5.3). In the following sections, we describe each step in detail.

1. **One Instruction Generation:** In the first iteration, the engine initializes input and internal signals to be symbolic values and explores the processor design for one complete clock cycle. In pipelined RISC processors, one clock cycle represents the completion of one instruction. In subsequent iterations, input signals are made symbolic, but internal signals may be partially constrained or concrete. (Sections 5.2.4 and 5.2.4.)

2. **Assertion Violation:** When the engine encounters an assertion violation, it produces a path constraint describing the precondition necessary to reach that error state. If the processor's reset state can satisfy the constraint, the backward symbolic execution engine is done. It outputs the trigger instruction(s) and Coppelia moves to the next phase: adding the payload.

3. **Fast Validation:** If the processor's reset state does not satisfy the current path constraint, the engine does a fast validation of the current intermediate state. This step uses heuristics to eliminate intermediate states that are less likely to bring the search closer to the reset state.

4. **Bound Checking:** If the current state passes the fast validation, the engine then checks whether the sequence of instructions generated so far exceeds a bound. The bound is a tunable parameter to the engine.

5. **Stitching Cycles:** If the length of the sequence is within the bound, the engine stitches the current state to the previously found state and continues on to the next iteration of the One Instruction Generation step.

6. **Feedback Generation:** When any of the preceding steps fail, the engine goes back to the prior iteration and, using feedback generated during prior runs, continues exploration in a new direction.

One Instruction Generation

In the first iteration, the backward symbolic execution engine starts the search for a security property violation from an unconstrained processor state. It sets both the input and the internal signals to symbolic values, and then explores the processor design until it reaches a state that violates the security property. If exploration completes and no assertion violation is found, Coppelia returns with a result of `no violation found`. Otherwise, the resulting exploration tree, $\mathcal{E}_k$, has a leaf node n_k that represents the error state (s_e) of the processor. Associated with that leaf node is the path condition π_k that describes the sufficient constraints on processor state and input signals such that the processor will move from the constrained state (s_{e-1}) to the error state in a single clock cycle. In addition to the constraints, the engine returns a satisfying solution to the constraints over input signals. These concrete input values will form the last instruction in the trigger sequence.

In the next iteration, the engine again starts the search from an unconstrained processor state. This time the engine is looking for s_{e-1}, a state that satisfies the constraints returned in the prior iteration, but not s_e. If such a state is found, the engine returns a path condition π_{k-1} and a satisfying solution to the constraints over the input signals. These concrete input values will form the penultimate instruction.

Iterations continue in this way, searching backward through trees $\mathcal{E}_k, \mathcal{E}_{k-1}, \ldots, \mathcal{E}_0$ until we reach the initial processor state. In the following sections we discuss the heuristics and optimizations we introduce to help the search converge toward an initial state.

Stateful Signals

A naive implementation of hardware oriented symbolic execution might make all variables of type `reg` symbolic because these internal signals can store state. However, the resulting exploration tree is too large. Using this set-up, we ran Coppelia for one clock cycle. After 24 hours it had generated over 1 million test cases – each is a different leaf node in the tree – but had not triggered any assertions.

We identify those signals that can be safely left concrete without affecting completeness of the search. First, `reg` signals are used in one of two ways in a hardware design: as part of sequential logic in which case they store state from a previous clock cycle, or as part of combinational logic in which case their value depends only on input signals in the current clock cycle. Using static analysis, we identify those signals which depend entirely (albeit, possibly indirectly) on input signals and do not make those symbolic in each iteration of exploration. Second, not all `reg` signals are relevant for a particular security property. Only those signals in the property's cone of influence are made symbolic. Section 5.2.5 describes the dependency analysis that Coppelia performs to identify which signals to make symbolic.

Fast Validation

At the end of each successful iteration j, the backward symbolic execution engine checks the following: are the constraints given in path condition π_j satisfied by the initial state? If so, Coppelia has found a successful trigger and moves on to the next phase, appending the payload.

If not, in order to steer the search toward the initial state, we introduce two rules to eliminate those intermediate states that are less likely to quickly lead back to the initial state. These rules form the fast validation step.

Empirically, we found that if the number of variables whose values are different from the initial state is small, we are more likely to be able to back track to an initial state. We set the number of differing variables to be:

$$\mathit{diff}((n_j, i_j, \pi_j), (n_0, i_0, \pi_0)) \leq \lfloor |s|/4 \rfloor + 1 \tag{5.1}$$

where (n_0, i_0, π_0) is the tuple associated with the initial tree $\mathcal{E}_0$, (n_j, i_j, π_j) is the tuple associated with an intermediate tree $\mathcal{E}_j$, *diff* calculates the number of different values between two tuples, and $|s|$ represents the number of internal symbolic variables. With this rule, at most a quarter of internal state variables may differ from their reset state.

The second rule targets loops that are preventing backward progress toward the initial state. We enforce that each new iteration should produce a tuple (n_j, i_j, π_j) that is not the same as any previously generated tuples:

$$(n_j, i_j, \pi_j) \notin \{(n_l, i_l, \pi_l) \mid j < l \leq k\} \tag{5.2}$$

If the values of internal signals are the same as ones already generated, we are not making any progress in this run and risk entering an infinite loop. Thus, if the generated (s_j, i_j, π_j)-tuple is a repeat, Coppelia will keep running until a different tuple is found.

Bound Checking

As a final heuristic, Coppelia uses bounded checking to counter the fact that the sequence of trees may never converge toward the initial state. We set a bound for the exploit length. If the trace of inputs generated so far exceeds the bound, Coppelia will exit with a message that it did not find an exploit within the bound.

Stitching Cycles

If the length of the sequence is within the bound, we stitch the current clock cycle to the previous clock cycle and continue with the next iteration. The sequence of trees must be stitched together appropriately, making sure a leaf node of one tree correctly aligns with the root node of a tree previously generated.

Ideally, in order for the results of cycle $\mathcal{E}_j$ and cycle $\mathcal{E}_{j-1}$ to align, we need to replace the values of internal signals in node n_{j-1} with the path constraint π_j obtained in node n_j. This ensures completeness – we will not miss a possible test case. However, the complexity of this method is similar to forward symbolic execution (see Section 5.2.4). The more cycles we symbolically execute, the longer the path constraints will be and the more complicated the queries will be to the SMT solver. In Coppelia, we adopt a light-weight approach. The insight is that while each clock cycle is explored symbolically, the individual cycles can be stitched together using only concrete values. This sacrifices completeness for speed: after each iteration, we find satisfying solutions to a subset of the internal signals and use these conrete values to partially define the state to search for in the next iteration. This will no doubt lead us to miss some possible violating paths. In practice, we can iterate, incrementally replacing concrete values with constrained symbols if no assertion violations are found.

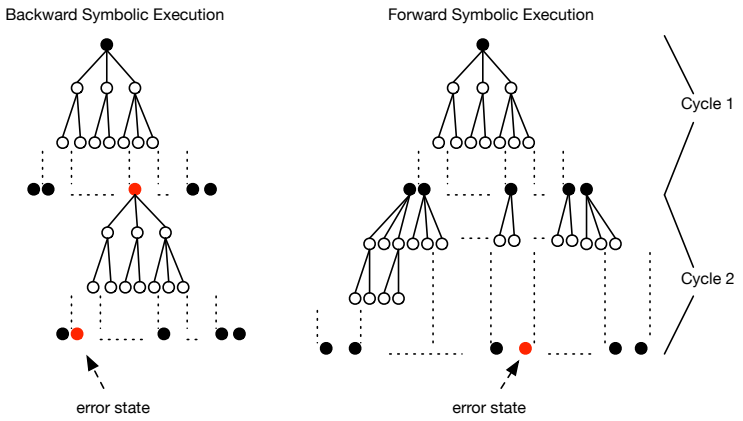

Figure 5.4: Comparison of backward and forward symbolic execution for 2 clock cycles.

Feedback Generation

If the engine finishes exploring all paths and no violations are found and this is not the first iteration (Figure 5.3), it means a violation was found in previous runs but the engine chose a wrong path, either because of the fast validation, the light-weight stitching, or because it stopped exploring after finding one violation. In this case Coppelia will go back to the previous runs and continue the exploration. Coppelia generates a feedback to the engine including which instruction causes the violation and what test cases have been explored. When rerunning that instruction generation, Coppelia only explores the specific instruction and skips the test cases already explored.

Forward and Backward Symbolic Execution

In forward symbolic execution, in the first clock cycle, a tree with N_f leaves will be explored (the N_f black dots in the first layer of the symbolic execution tree on the right in Figure 5.4). In the second clock cycle, the tree must be explored again, once for each of the N_f leaves. Exploring forward M clock cycles has complexity $O(N_f^M)$.

The complexity for backward symbolic execution is $O(N_b \cdot M)$, where M is the number of cycles to execute. Note that the N_b here is larger than the N_f in forward symbolic execution because the internal signals are set to be symbolic values which increases the paths to explore. On the other hand, in general only $j \leq N_b$ paths are explored because exploration stops once the error state is found. This is illustrated on the left side of Figure 5.4.

5.2.5 Building the Trigger: Optimizations

Each iteration of the symbolic exploration of the processor is expensive. We introduce the following optimizations tailored for hardware designs to improve the speed.

Preconditioned Symbolic Execution

As an optimization, Coppelia constrains the opcodes to only instructions in the architecture to force the SMT solver to return legal instructions.

We also add constraints to support bit level representations. In KLEE, the minimum width supported is a byte. However, hardware signal widths are not necessarily byte multiples. Thus, we add constraints to inform the symbolic execution engine of the value range of such signals. For example, for a signal of width n, we constrain the value of the signal to be less than or equal to $(2^n - 1)$.

Path Selection Heuristic

We observe that if Coppelia is exploring the right processor instruction, it will find the vulnerability in a short time. However, it often takes a long time before Coppelia starts exploring the right instruction. (In our experience, the engine spends more than three hours to analyze paths under 13 instructions.) To find vulnerabilities more efficiently, Coppelia uses a hybrid search heuristic. Coppelia selects the next symbolic execution state to run by interleaving together breadth-first search and depth-first search to both explore as many processor instructions in as short a time as possible and explore each instruction in as much depth as possible. Each of them are run in a fixed number of times (chosen heuristically). We run depth-first more than breadth-first to allow enough time for the engine to explore the paths for each instruction.

Cone of Influence Analysis

In Coppelia, we apply a cone of influence (CoI) analysis to reduce the search space. The analysis is performed at the LLVM level during the static analysis phase, and removes signals from the design whose values do not affect, directly or indirectly, the value of signals in the security-critical assertions.

In developing the CoI analysis we found that performing the analysis at the function level was too conservative and led to little or no pruning. Almost every function was found to affect the function containing the assertion, but not all those functions affected the assertion itself. Therefore, we perform the dependency

Algorithm 3: Cone of Influence Analysis

Input : A list of vars in the assertions varsInAssert
Output : A list of nodes in the graph nodeSet

```
1  trackedInstrs ← ∅;
2  nodeSet ← ∅;
3  dg ← BuildDependencyGraph();
4  for v ∈ varsInAssert do
5      vLocSet ← GetVarLocation(V);
6      for loc ∈ vLocSet do
7          nodes, instrs ← DependenceAnalysis(dg, loc, trackedInstrs);
8          nodeSet ← nodeSet ∪ nodes;
9          trackedInstrs ← trackedInstrs ∪ instrs;
10     end
11 end
```

analysis at the instruction level. On the other hand, pruning at the instruction level was too costly. Program completeness could not be guaranteed and the symbolic execution engine had to check at each instruction whether to execute it or not. Therefore, we perform the pruning at the function level. Any function containing at least one instruction affecting a signal in the assertion is kept; all other functions are pruned. This hybrid approach allows us to prune aggressively while maintaining program completeness and keeping the run-time overhead low.

The first step of our CoI analysis (Algorithm 3) is developing an interprocedural dependency graph. Each function forms a node and an edge from node a to node b is added if the inputs to b depend on the outputs of a. The second step is performing dependency analysis for the signals in the security-critical assertions. We extract the target signals in the assertions and get the location of these signals. Starting from these locations, we search backward through functions to track the LLVM instructions these signals depend on.

Compiler Optimizations

Verilator provides different levels of compiler optimizations for improving simulation performance [20]. We initially disabled optimizations and used $-O0$ flag because higher optimization levels adversely affect code readability and complicates the application of security-critical assertions because many of the signals and variables asserted over can be optimized out. Although using the $-O0$ flag confers significant readability benefits and eases assertion application, it slows the symbolic execution (Section 5.4.4). In Coppelia, we use the compiler optimizations to improve performance (Section 5.4.4) and modify the assertions for the optimized code.

Cat.	Description	Bug No.	No. of Stubs	Avg. LoC
CF	Control flow related	b20, b21, b27	2	15
XR	Exception related	b02, b03, b07, b08, b09, b10, b11, b14 , b15, b18, b19, b23, b29	3	29
MA	Memory access related	b17, b22, b24, b28, b30, b31	2	16
IE	Correct instructions	b06, b12	2	12
CR	Correctly updating results	b01, b04, b05, b13	2	13

Table 5.1: Program stub categories for each bug and implementation details.

5.2.6 Adding the Payload: Program Stubs

The sequence of instructions generated by the symbolic execution engine only triggers the bug. To better understand the security implications, we generate and append a payload to complete the exploit. This is based on our observation that although the triggers may differ, the same payload is often used across multiple exploits. Thus, we can use similar stubs for similar exploit situations.

Coppelia generates these program stubs according to the category of the security-critical properties being violated. We classified the security-critical properties into five classes as in the SCIFinder project: `CF`: control flow related properties, `XR`: exception related properties, `MA`: memory access related properties, `IE`: properties to ensure execution of the correct and specified instructions, and `CR`: properties about correctly updating results.

5.3 Implementation

Coppelia is primarily implemented in C++ and Python. We build the state exploration part on top of KLEE, and the CoI analysis is written as LLVM passes. When we implement security assertions on the OR1200 processor in Cadence IFV as part of the evaluation, we use SystemVerilog.

5.3.1 Testbench Generation

Coppelia provides an automatic process to generate a testbench environment within which to verify the processor design. This environment provides stimulus to input ports, simulates the design, and checks for violations of security assertions.

We first make all inputs symbolic and then assign these symbolic values to input ports. The symbolic values are constrained by preconditions in order to generate legal instructions (Section 5.2.5). The whole processor design is simulated twice for each clock cycle (Section 5.2.2). The simulation runs for as many

clock cycles as there are pipeline stages to allow signals' values to be propagated through the entire pipeline. At the end of each clock cycle, we check whether security-critical assertions are violated.

5.3.2 Translating Security Assertions

The initial security assertions that we collected (Section 5.4.1) are developed specifically for the OR1200 processor, which is a 32-bit implementation of the OR1k architecture with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support, and basic DSP capabilities. As part of our effort to find new bugs in different platforms and architectures, we also manually translate these assertions to the Mor1kx-Espresso processor (OR1k architecture) and the PULPino-RI5CY processor (RISC-V architecture).

The Mor1kx assertions correspond directly to OR1200 assertions because of their shared architecture so we need only adapt the assertions to different variables and pipeline stages. Assertions for the PULPino processor differ at a deeper level. We need to first verify that the examined security properties are still applicable to the new architecture. To do so we check both the RISC-V specification and the PULPino processor specification. We then adapt the assertions to appropriate variables and pipeline stages.

5.3.3 Program Stubs

For each category of the security-critical properties, we implement a few program stubs to complete the exploits. For some bugs, the instruction traces generated by symbolic execution cannot be directly connected to the program stubs. We manually implement the connecting code. Table 5.1 shows the number of stubs for each category and the average lines of code. As an example, the `R0` bug belongs to the memory access related category. The symbolic execution engine generates an instruction sequence that stores a non-zero value to `R0`. We then generate a program stub (in C) that exploits the bug by triggering a memory access instruction that expects `R0` to be zero. This demonstrates that an attacker using this bug can exploit it to write data to a memory locations as specified by the attacker.

5.4 Evaluation

We evaluate Coppelia across multiple CPU designs to study its efficacy and its practicality. Our evaluation aims to answer the following research questions: 1) Can Coppelia effectively generate high-quality exploits for known CPU security bugs? 2) How does Coppelia perform compared to hardware model checking

No.	Synopsis	Instructions Generated			Replayable		
		Coppelia	Cadence	EBMC	Coppelia	Cadence	EBMC
b01	Privilege escalation by direct access	2	1	1	✓	×	×
b02	Privilege escalation by exception	2	×	×	✓	-	-
b03	Privilege anti-de-escalation	1	1	1	✓	✓	✓
b04	Register target redirection	3	1	1	✓	×	×
b05	Register source redirection	1	1	1	✓	✓	✓
b06	ROP by early kernel exit	50	1	3	✓	×	×
b07	Disable interrupts by SR contamination	1	1	1	✓	✓	✓
b08	EEAR contamination	1	×	×	✓	-	-
b09	EPCR contamination on exception entry	2	×	×	✓	-	-
b10	EPCR contamination on exception exit	2	1	8	✓	✓	✓
b11	Code injection into kernel	2	1	1	✓	✓	✓
b12	Selective function skip	1	1	1	✓	×	×
b13	Register source redirection	1	1	1	✓	✓	✓
b14	Disable interrupts via micro arch	2	1	1	✓	✓	✓
b15	l.sys in delay slot will enter infinite loop	2	×	×	✓	-	-
b16	l.macrc immediately after l.mac stalls the pipeline	-	-	-	-	-	-
b17	l.extw instructions behave incorrectly	4	1	7	✓	×	×
b18	Delay Slot Exception bit is not implemented in SR	1	×	×	✓	-	-
b19	EPCR on range exception is incorrect	1	×	×	✓	-	-
b20	Comparison wrong for unsigned inequality with different MSB	3	1	1	✓	×	×
b21	Incorrect unsigned integer less-than compare	5	×	×	✓	-	-
b22	Logical error in l.rori instruction	5	×	×	✓	-	-
b23	EPCR on illegal instruction exception is incorrect	2	×	×	✓	-	-
b24	GPR0 can be assigned	2	1	6	✓	×	×
b25	Incorrect instruction fetched after an LSU stall	-	-	-	-	-	-
b26	l.mtspr instruction to some SPRs in supervisor mode treated as l.nop	3	×	×	✓	-	-
b27	Call return address failure with large displacement	2	1	1	✓	×	×
b28	Byte and half-word write to SRAM failure when executing from SDRAM	1	1	1	✓	✓	✓
b29	Wrong PC stored during FPU exception trap	2	×	×	✓	-	-
b30	Sign/unsign extend of data alignment in LSU	1	1	-	✓	✓	-
b31	Overwrite of ldxa-data with subsequent st-data	1	1	-	✓	✓	-

Table 5.2: Generating exploits of collected bugs. The first 14 bugs are from SPECS [59] and the last 17 bugs are from SCIFinder. The Instructions Generated column shows the number of instructions generated; the Replayable column shows whether the generated exploits can be replayable on an FPGA board. × means either the triggering information cannot be generated or the generated exploit is not replayable.

tools? 3) Is Coppelia practical for use on full-scale CPU designs, and what effect do our optimizations have on performance? 4) Can Coppelia be used to expose, and generate complete exploits for, new CPU security-critical bugs?

```
assign a_lt_b = comp_op[3] ? ((a[width-1] & !b[width-1]) |
  (!a[width-1] & !b[width-1] & result_sum[width-1]) |
  (a[width-1] & b[width-1] & result_sum[width-1])) :
  (a < b);              // Bug Free Version
  result_sum[width-1];  // Buggy Version
```

Listing 1: A security bug from OR1200 processor Bugzilla.

5.4.1 Dataset and Experiment Setup

For our evaluation, we collected 31 security-critical bugs (Table 5.2) of the OR1200 processor from SPECS [59] and SCIFinder. We collected 35 security-critical assertions from SPECS [59], Security Checkers [21], and SCIFinder. We translated 30 assertions for the Mor1kx processor, and 26 assertions for the PULPino processor. The experiments are performed on a machine with Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, a dual-socket server) and 62G of available RAM.

5.4.2 Generating Exploits for Known Bugs

To evaluate the efficacy of our tool against a ground truth, we test whether it can find and generate exploits for the known bugs we collected. These security-critical bugs are implemented in the OR1200 processor and we test Coppelia on the core of the processor. We run Coppelia by making both input signals and internal signals symbolic and executing backward toward the reset state.

Table 5.2 summarizes the results. For bug b16 we did not have an assertion. Bug b25 is a bug outside of the OR1200 core. Thus, we are not able to generate exploits for these two cases. In the remaining 29 cases, Coppelia is able to automatically generate exploits to expose the known bug for all of them. Overall, the generated exploits are concise, frequently only one or two instructions (excluding the size of the stubs). We can also see that for bugs that involve multiple cycles, Coppelia can indeed generate a series of instructions to exercise these deep error states.

For each generated exploit, we verify its ability to expose a vulnerability by running it on an FPGA board (DE0Nano). Each exploit contains a generated stub according to the type of the security assertion triggered by the bug (see Table 5.1). As shown in Table 5.2, all the exploits are successfully replayed on the FPGA board.

As an example, Listing 1 shows a security-critical bug (b20) from the OR1200 processor Bugzilla database (Bugzilla #51 [4]). The code snippet is from the ALU module in the OR1200 processor. It shows

```
void foo() {
  printf("Attack success!\n"); // Payload
}
int main() {
  gotoUserMode();              // Payload
  asm volatile (               // Trigger
    l.movhi r16 0x8000;
    l.nop;
    l.sfgtu r16 r0;);
  jumpToFoo();                 // Payload
}
```

Listing 2: The exploit program generated by Coppelia.

the logic to determine whether operand `a` is less than operand `b`. The buggy implementation works fine in most cases, but it fails for the `l.sfgtu` (set flag greater than equal) instruction. According to the OpenRISC specification [70], the instruction `l.sfgtu rA, rB` compares the contents of general-purpose registers `rA` and `rB` as unsigned integers. If the value of the first register is greater than the value of the second register, the compare flag is set; otherwise the compare flag is cleared. However, with this bug, if the highest-order bit in register `rA` is 1 the compare flag will not be set, even if `rA` is greater than `rB`. An attacker can exploit this bug to control which branch to execute. The security bug violates the security-critical assertion: the comparison flag should be set correctly. Listing 2 shows the generated exploit. (The full payload is abbreviated for space reasons.) The total CPU time required for generating this exploit is 9m40s.

5.4.3 Comparison with Model Checking

A current standard for hardware verification is model checking. In this section, we compare Coppelia against the commercial hardware model checking tool, Cadence's Incisive Formal Verifier (IFV), and against a research tool, EBMC [68]. We use each tool to look for the known bugs from Section 5.4.2 and compare the results with Coppelia. We add the same constraints (Section 5.2.5) in both Cadence IFV and EBMC. The results are shown in Table 5.2.

We make several observations:

(1) Cadence successfully finds and generates triggers for 18 bugs and EBMC for 16 bugs.

(2) Cadence fails to find or generate triggers for 11 bugs and EBMC fails for 13 bugs. All of them are found by Coppelia.

Among these bugs, 8 of them (b02, b08, b09, b15, b18, b19, b23, b29) are related to exception handling for managing privilege levels in the processor. Although we could not determine the exact reason why Cadence and EBMC fail to find these bugs, we note that the relevant properties for these bugs all include the condition `(wb_insn == syscall)`. However, both Cadence and EBMC can find bug b14, which also relies on that same condition.

Bugs b21, b22, b26 are related to accessing register files. The OR1200 processor uses two dual-port RAMs for implementing register files. These two RAMs are written and read at the same time so that the processor can read two registers within a single clock cycle. However, we find that `(operand_b == 0)` is always true when running both model checking tools. This means data reading from `ram_b` is always 0, which is incorrect. We suspect that Cadence and EBMC build an incorrect model for the two RAMs.

EBMC fails to find and generate triggers for bugs b30 and b31 because it fails to parse assertions with deep hierarchies.

(3) As a tool designed for assertion verification rather than exploit generation, Cadence IFV only generates intermediate results when a property is invalidated. By contrast, the complete trigger is generated in Coppelia. For example, for bug b24 (the `R0` bug described in the introduction) Cadence generates the single instruction `l.addi r0, r1, 0`. This instruction will only trigger the bug if `r1` already holds a non-zero value, which is not the case for the reset state (`r1` is set to 0 at reset). In the traces Cadence generates, a number of signals are not in the reset state. It is nontrivial for designers to set the processor to a particular state in order to trigger the assertion. Table 5.2 shows that 12 exploits are not directly replayable from the reset state. For EBMC, we have similar results. Although EBMC returns multiple instructions, they are not always directly replayable from the reset state.

(4) We currently remove the memory from the processor and only run these tools on the processor core. When adding the memory back, it took Cadence several hours to build the model. It is necessary to rerun formal builds every time the verilog is changed so this would be a significant impediment to rapid development of bug fixes. Coppelia does not require long model building time but it fails to handle the memory because the queries to the solver are too long. We have not done optimizations for memory models but research on optimizing symbolic execution for arrays is ongoing [87] and could be incorporated into future versions of Coppelia.

5.4.4 Effects of Optimizations

No.	Original	Hybrid Search		Compiler Optimizations		CoI Analysis		Overall Speedup
	Time	Time	Speedup	Time	Speedup	Time	Speedup	
b05	3h50m5s	3m41s	62.47x	0m14s	15.54x	2m11s	0.11x	104.58x
b09	>24h	0m3s	>28800x	15m59s	0.004x	4m37s	3.46x	>311.91x
b10	19h30m49s	35m55s	32.60x	15m54s	1.16x	2m11s	7.32x	536.25x
b13	>24h	0m3s	>28800x	0m15s	0.22x	2m12s	0.11x	>654.55x
b24	19h31m33s	35m40s	32.85x	16m20s	2.18x	2m33s	6.42x	406.27x
b27	>24h	>6h	-	17m38s	>27.22x	11m29s	1.54x	>125.40
Avg.	>19h	>1.2h	>11545x	11m3s	>7.72x	4m12s	3.16x	>356.49x

Table 5.3: Effects of optimizations. This table is aggregative, e.g. Compiler Optimizations means that Coppelia is running with both Hybrid Search and Compiler Optimizations on. Time columns show the CPU time. Speedup columns show the relative improvements in CPU time compared to previous columns.

To evaluate the effectiveness of our optimizations (Section 5.2.5), we first randomly select six bugs which require only one instruction to trigger (examining longer bugs without optimizations took on the order of several days). For each bug, we make input signals symbolic and run Coppelia for one clock cycle, starting from the reset state. We show how each optimization influences the performance of symbolic execution. In the Original KLEE setup, we use KLEE's default settings, i.e., random search heuristic, 2000M maximum memory consumption, and counter example cache enabled. In the Hybrid Search setup, we enable the hybrid search heuristic. Specifically, we start with BFS and alternate the BFS and DFS. The BFS is set to run 10,000 times and the DFS is set to run 500,000 times. In the Compiler Optimizations configuration, we enable Verilator's optimizations when generating C++ code while keeping KLEE's settings the same as the previous column. In the CoI Analysis setup, we enable the CoI analysis in addition to all the settings in the previous column.

Table 5.3 summarizes the results, from which we make the following observations: (1) Adding all the optimizations yields an average overall speedup of two-to-three orders of magnitude compared to the original KLEE. On average, each optimization can enhance the performance by about one order of magnitude. (2) On average, the Hybrid Search heuristic improves the performance the most. (3) Applying all optimizations does not necessarily yield the best performance. For example, for bug b09 and b13, applying only the hybrid search heuristic can reduce the searching time to only 3 seconds, but adding other optimizations increases the search time.

Table 5.4 shows the result of the Cone of Influence Analysis. Running the CoI Analysis can prune out a number of functions for symbolic execution. The effects of the CoI Analysis mainly depend on the

No.	Func	Func Left	LLVM Instr	Instr Left
b05	47	34 (72.3%)	12501	11505 (92.0%)
b09	47	33 (70.2%)	12458	11427 (91.7%)
b10	47	33 (70.2%)	12475	11444 (91.7%)
b13	47	34 (72.3%)	12504	11508 (92.0%)
b24	47	34 (72.3%)	12474	11478 (92.0%)
b27	47	34 (72.3%)	12485	11489 (92.0%)

Table 5.4: Details of the Cone of Influence Pruning.

Optimization Level	Total LoC in C++
O0	14118
O3	8587 (61%)

Table 5.5: Details of the Compiler Optimizations.

security-critical assertions added. For the six bugs we picked, the first five have three variables in the assertions and the last one has four variables in the assertion. On average, the CoI Analysis prunes out 8% of the LLVM instructions and 30% of the functions. Table 5.5 shows that using `O3` level in the Compiler Optimizations can reduce 39% of the C++ code generated. Figure 5.5 compares the performance among different search heuristics. The upper figure shows the number of instructions covered in the generated test cases as time changes. The BFS covers the most instructions in a given amount of time. The lower figure shows the number of test cases per instruction generated as time changes. The DFS generates the most test cases per instruction in the given amount of time. Our hybrid search heuristic combines the advantages of both the BFS and the DFS heuristics.

5.4.5 Performance

For the 29 bugs Coppelia successfully generates exploits, 18 (62%) out of 29 of the exploits are generated within 15 minutes, demonstrating that Coppelia can be a practical quality control tool for hardware vendors. However, 2 (7%) out of 29 took a longer time (over 2 hours) to generate even for bugs involving only a single instruction. We find two reasons for the longer time: 1) Coppelia takes longer to reach the target instruction either because making internal signals symbolic increases the symbolic execution states to explore or because the instruction is near the end of the queue of all instructions to explore. 2) The bug is deep in the pipeline (in the 4th or 5th stage) and increasing the pipeline stages can dramatically increase the number of symbolic execution states. If we only run Coppelia for the target instruction (instead of all the instructions in the ISA), the time for generating the exploits can be reduced to only a few minutes.

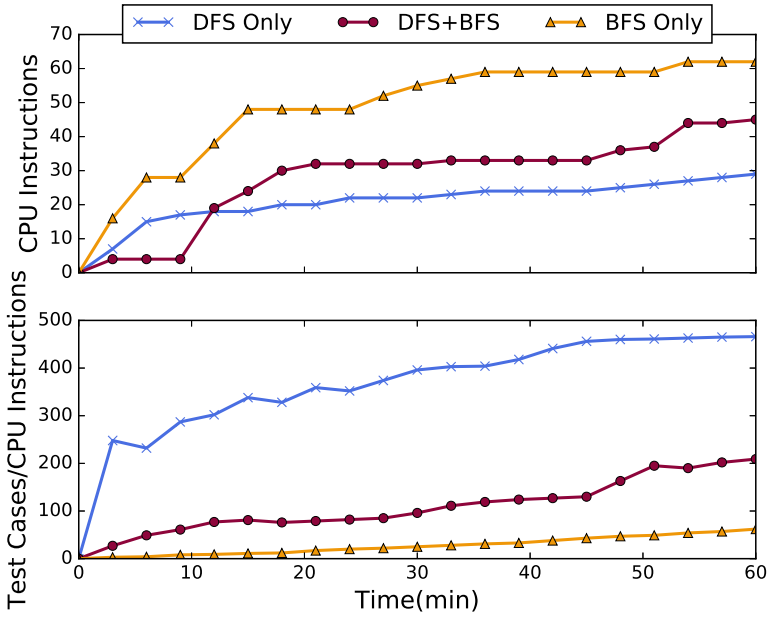

Figure 5.5: Comparison of different search heuristics.

No.	Processor	Security Property	Instructions	Replayable(ZedBoard)
b32	Mor1kx-Espresso	Calculation of memory address / data is correct	2	✓
b33	PULPino-RI5CY	Privilege escalates correctly	1	✓
b34	PULPino-RI5CY	Privilege deescalates correctly	1	✓
b35	PULPino-RI5CY	Jumps update the target address correctly	1	✓

Table 5.6: New security-critical bugs and exploits found in Mor1kx-Espresso and PULPino-RI5CY Processor.

5.4.6 Finding New Bugs

In this section, we examine Coppelia's efficacy in finding unknown bugs on new platforms and architectures. We run Coppelia on two new processors: Mor1kx-Espresso and PULPino-RI5CY. The Mor1kx is the most recent implementation of the OR1k architecture. We evaluate our tool on the Espresso core which is a 32-bit implementation with 2-stage integer pipeline and delay slot. The PULPino is an open-source single-core 32-bit low-power processor based on the RISC-V architecture. We evaluate our tool on the RI5CY core, which is an in-order, RV32-ICM implementation with 4-stage integer pipeline and DSP extensions. Table 5.6 shows the new security bugs and their exploits we found in Mor1kx-Espresso processor and PULPino-RI5CY processor.

Bug b32 is the same as the motivating example `R0` bug. This bug was not fixed in the OR1200 processor and we still found it in the new generation of OpenRISC processor. This shows that security-critical bugs can persist to the next generation of processor designs.

Items	No. of Assertions
Total Assertions	35
Pass Check	29
Fail Check (Bugs not fixed)	2
Fail Check (Wrong assertions)	4

Table 5.7: Security Patch Verification.

Bug b33 allows incorrect escalation of privilege. According to the RISC-V specification, when the `EBREAK` instruction is executed, the privilege mode's `epc` register should be set to the address of the `EBREAK` instruction itself [12]. However, in RI5CY processor, we found that when the `EBREAK` instruction is executed, the `epc` register is not correctly updated. This is security critical because when the processor returns to user mode, it will jump to an incorrect address.

Bug b34 allows incorrect de-escalation of privilege. In the RISC-V specification, to return after handling a trap, the `SRET` instruction sets the `pc` to the value stored in the `epc` register [12]. However, `pc` is not set correctly when `SRET` is executed in the RI5CY implementation. This can be exploited by redirecting the program counter to an address of the attacker's choosing.

Bug b35 incorrectly updates the target `pc` of the jump instruction. The RISC-V specification states that the target address of the indirect jump instruction is calculated by adding the 12-bit signed I-immediate to the register `rs1`, then setting the least-significant bit of the result to zero [12]. However, in the processor, the LSB is never set to 0; the implementation does not meet the specification. This may be leveraged by the attacker to silently redirect the `pc`.

5.4.7 Verify Patches and Refine Assertions

While running Coppelia on known bugs (Section 5.4.2), we also check the assertions by running Coppelia both on the buggy processor and on the patched processor expecting an exploit and no exploit respectively. While this is true for most cases, we sometimes observe that even after a bug is removed, Coppelia can still generate an exploit. This happens because either the processor is still buggy or because the assertions that we use based on prior work are not true assertions (the assertion does not consider some uncommon situation introduced by a correct patch because the assertions are collected from a dynamic simulation). As shown in Table 5.7, for the 35 assertions we collected, 29 of them pass this check. For the 6 assertions that fail, 2 fail because the processor is still buggy (these 2 assertions pass the check after the bugs are fixed) and 4 are not true assertions.

This phenomenon implies that in addition to using Coppelia to generate exploits, we can also use Coppelia to verify whether a security patch indeed fixed a vulnerability, and to iteratively refine an initial set of assertions.

5.5 Summary

We have presented Coppelia, an end-to-end tool for analyzing and contextualizing the security threat of hardware vulnerabilities. Given a processor design and a set of security properties, Coppelia generates C programs with inline assembly that exploit bugs within the design. Coppelia is able to generate exploits for 29 known bugs on the OR1200 processor, and discovered and generated exploit programs for 4 unknown bugs across two different processors and architectures.

CHAPTER 6
CONCLUSION

6.1 Summary

With many attacks on hardware appearing, guaranteeing the security of hardware designs is more and more important nowadays. However, the hardware security validation tools are still not ideal yet, and require lots of manual efforts. This dissertation explores how to develop security-critical assertions and how to build security validation tools for better assisting hardware designers in building secure hardware. This dissertation concludes that security validation of open-source RISC processor designs can be automated through mining security assertions with machine learning and known security errata, translating security assertions using static analysis techniques, and generating exploit programs using backward symbolic execution. This is shown by: (1) SCIFinder, a semi-automatic approach that leverages both known vulnerabilities and machine learning technique to identify security properties, can generate security properties to detect unknown security vulnerabilities in a RISC processor; (2) Transys, a tool that translates both temporal and information flow tracking security properties to new hardware designs, can efficiently build security properties for the new design; (3) Coppelia, a tool with the hardware-orientied backward symbolic execution strategy, can generate exploit programs and help to enforce the security properties for the hardware designs. Although assertion mining and symbolic execution techniques have been explored in the software domain, we still need to make adaptations to hardware specific features when applying them to the hardware domain. Currently, all these work are for RISC processors and simple crypto cores. The security properties are in only restricted temporal logic. To handle more complicated hardware designs, such as modern processors, we need to make our tools more scalable. To detect more sophisticated hardware vulnerabilities, such as the spectre and meltdown attacks, we need to generate security-critical hyperproperties and build validation tools for these properties. I leave these two aspects to be addressed in future research.

6.2 Future Directions

This thesis explores how to automate the process of detecting hardware vulnerabilities. However, this automation is just a first step towards providing strong security guarantees to hardware designs. An next step would be to address the question: what security guarantees do these verification methods provide – what can be detected and what cannot? Defining the scope of the security problems these methods can solve will provide a better understanding of both the power and limitations of these tools; and can provide insights into what problems remain to be addressed. Patching the vulnerabilities is also an important aspect of defending the hardware against various kinds of attacks. Automatically fixing the security vulnerabilities in hardware designs through program synthesis techniques is a potentially promising direction.

Considering the future of the hardware security field, there are two big questions that I consider paramount for researchers and hardware designers to consider. First, just like with software security, hardware attackers and defenders are in a game in which clever attacks and defenses are carefully designed to beat each other but no one ever wins. Thus, we have the question: to what point do we think our hardware designs are secure? Is there a line here? Second, hardware is often the trusted computing base for the entire system, providing ever new and more complex security primitives to software. At the same time its complexity only continues to grow. Do we really need the hardware to be so large and complicated? How can we redesign the hardware to balance the performance with security? It is time to rethink how we should build hardware in the future.

BIBLIOGRAPHY

[1] ARM TrustZone. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[2] Cadence Verification Suite. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification.html.

[3] Clang: a C language family frontend for LLVM.

[4] Comparison wrong for unsigned inequality with different MSB.

[5] ORWL Wiki. https://wiki.orwl.org/.

[6] SecureBlue++: CPU Support for Secure Execution. https://wiki.orwl.org/.

[7] Verilator. https://www.veripool.org/wiki/verilator.

[8] Intel Pentium Processor Statistical Analysis of Floating Point Flaw. *Intel White Paper*, July 2004.

[9] Xen Security Advisory CVE-2015-5307,CVE-2015-8104 / XSA-156. http://xenbits.xen.org/xsa/advisory-156.html, Nov 2015.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[11] F. E. Allen. Program optimization. In *Annual Review in Automatic Programming, vol. 5*, 1969.

[12] Krste Asanovic Andrew Waterman. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10. https://riscv.org/specifications/privileged-isa/, 2017.

[13] A. Ardeshiricham, W. Hu, and R. Kastner. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design*, 2017.

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

[15] Desire Athow. Pentium FDIV: The Processor Bug That Shook the World. *techradar.pro*, October 2014.

[16] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1999.

[17] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Network and Distributed System Security Symposium*, 2011.

[18] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Commun. ACM*, 2014.

[19] A. Becker, W. Hu, Y. Tai, P. Brisk, R. Kastner, and P. Ienne. Arbitrary precision and complexity tradeoffs for gate-level information flow tracking. In *2017 54th ACM/EDAC/IEEE Design Automation Conference*, 2017.

[20] Jeremy Bennett. High Performance SoC Modeling with Verilator. http://www.embecosm.com/appnotes/ean6/embecosm-or1k-verilator-tutorial-ean6-issue-1.html, 2009.

[21] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security Checkers: Detecting processor malicious inclusions at runtime. In *Proceedings of 2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011.

[22] Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *Proceedings of 2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2012.

[23] Thomas Braibant and Adam Chlipala. Formal Verification of Hardware Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 2013.

[24] Daniel Brand. Verification of Large Synthesized Designs. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, 1993.

[25] R. Brayton and A. Mishchenko. ABC: An Academic Industrial-Strength Verfication Tool. In *Comuter Aided Verification*. Lecture Notes in Computer Science, 2010.

[26] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[28] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[29] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[30] D. Champagne and R. B. Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.

[31] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[32] Po-Hsien Chang and Li C Wang. Automatic Assertion Extraction via Sequential Data Mining of Simulation Traces. In *Design Automation Conference, 2010 15th Asia and South Pacific*, 2010.

[33] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, and F. T. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *2008 International Symposium on Computer Architecture*, 2008.

[34] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[35] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori. Software-based Gate-level Information Flow Security for IoT Systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[36] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[37] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.*, 2017.

[38] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 2010.

[39] K. Constantinides and T. Austin. Using Introspective Software-Based Testing for Post-Silicon Debug and Repair. In *Design Automation Conference, 2010 47th ACM/IEEE*, 2010.

[40] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-Based Online Detection of Hardware Defects Mechanisms, Architectural Support, and Evaluation. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[41] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[42] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[44] Lee R. Dice. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 1945.

[45] Peter Dinges and Gul Agha. Targeted Test Input Generation Using Symbolic-Concrete Backward Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[46] D.Lin, E.Singh, C.Barrett, and S.Mitra. A structured approach to post-silicon validation and debug using symbolic dquick error detection. In *Proceedings of the IEEE International Test Conference*, 2015.

[47] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 2007.

[48] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[49] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[50] Harry Foster. Applied Assertion-Based Verification: An Industry Perspective. *Found. Trends Electron. Des. Autom.*, 2009.

[51] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models. *R package version*, 2009.

[52] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.

[53] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 2012.

[54] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. In *Proceedings of 42nd Design Automation Conference*, 2005.

[55] Mike Turpin Harry Foster, Kenneth Larsen. Introduction to the new accellera open verification library. 2006.

[56] Sean Heelan. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. 2009.

[57] L. C. Heller and M. S. Farrell. Millicode in an IBM zSeries Processor. *IBM Journal of Research and Development*, 2004.

[58] Stav Hertz, David Sheridan, and Shobha Vasudevan. Mining Hardware Assertions with Guidance from Static Analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2013.

[59] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[60] 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*, 2016.

[61] W. Hu, A. Ardeshiricham, and R. Kastner. Identifying and Measuring Security Critical Path for Uncovering Circuit Vulnerabilities. In *2017 18th International Workshop on Microprocessor and SOC Test and Verification*, 2017.

[62] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. Property Specific Information Flow Analysis for Hardware Security Verification. In *Proceedings of the International Conference on Computer-Aided Design*, 2018.

[63] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`.

[64] R. Kastner, W. Hu, and A. Althoff. Quantifying hardware security using joint information flow analysis. In *2016 Design, Automation Test in Europe Conference Exhibition*, 2016.

[65] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[66] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[67] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy*, 2019.

[68] Daniel Kroening and Mitra Purandare. EBMC: The enhanced bounded model checker.

[69] Damjan Lampret. OpenRISC 1200 IP Core Specification, 2001.

[70] Damjan Lampret. OpenRISC 1000 Architecture Manual. `https://github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf?raw=true`, 2014.

[71] 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. In *Proc. ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[72] 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*, 2011.

[73] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[74] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium*, 2018.

[75] Lingyi Liu and Shabha Vasudevan. STAR: Generating input vectors for design validation by static analysis of RTL. In *IEEE International Workshop on High Level Design Validation and Test Workshop*, 2009.

[76] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis*, 2011.

[77] Shuangge Ma and Jian Huang. Penalized Feature Selection and Classification in Bioinformatics. *Briefings in bioinformatics*, 2008.

[78] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner. Quantifying timing-based information flow in cryptographic hardware. In *2015 IEEE/ACM International Conference on Computer-Aided Design*, 2015.

[79] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[80] A. Meixner and D. J. Sorin. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, 2008.

[81] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[82] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. Hardware Verification using Software Analyzers. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2015.

[83] S. Narayanasamy, B. Carneal, and B. Calder. Patching Processor Design Errors. In *2006 International Conference on Computer Design*, 2006.

[84] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V Deshmukh, and Taylor T Johnson. Hyperproperties of Real-Valued Signals. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2017.

[85] 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*, 2011.

[86] 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*, 2009.

[87] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating Array Constraints in Symbolic Execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2010.

[88] Tekla S. Perry. Why Hardware Engineers Have to Think Like Cybercriminals, and Why Engineers Are Easy to Fool. *IEEE Spectrum*, 2017.

[89] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas. Design and Implementation of the Ascend Secure Processor. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[90] H. Salmani, M. Tehranipoor, and R. Karri. On Design Vulnerability Analysis and Trust Benchmarks Development. In *2013 IEEE 31st International Conference on Computer Design*, 2013.

[91] Smruti Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder, and Josep Torrellas. Patching Processor Design Errors with Programmable Hardware. *IEEE MICRO*, 2007.

[92] Smruti R. Sarangi, Abhishek Tiwari, and Josep Torrellas. Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware. In *IEEE MICRO*, 2006.

[93] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. Benchmarking of Hardware Trojans and Maliciously Affected Circuits. *Journal of Hardware and Systems Security*, 2017.

[94] Wilson Snyder. Verilator. `https://www.veripool.org/papers/verilator_philips_internals.pdf`, 2005.

[95] G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A Single-Chip Secure Processor. *IEEE Design Test of Computers*, 2007.

[96] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the 17th Conference on USENIX Security Symposium*, 2008.

[97] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[98] 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[99] 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 *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[100] S. G. Tucker. Microprogram Control for SYSTEM/360. *IBM Systems Journal*, 1967.

[101] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008.

[102] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification*, 2015.

[103] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[104] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[105] Wei Hu, A. Becker, A. Ardeshiricham, Yu Tai, P. Ienne, D. Mu, and R. Kastner. Imprecise security: Quality and complexity tradeoffs for hardware information flow tracking. In *2016 IEEE/ACM International Conference on Computer-Aided Design*, 2016.

[106] Clifford Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`.

[107] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, 2011.

[108] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[109] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-Based Control and Mitigation of Timing Channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[110] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[111] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. In *Proceedings of the International Symposium on Microarchitecture*. IEEE/ACM, 2018.

[112] Rui Zhang, Natalie Stanley, Chris Griggs, Andrew Chi, and Cynthia Sturton. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[113] Rui Zhang and Cynthia Sturton. Transys: Leveraging Common Security Properties Across Hardware Designs. In *Proceedings of the Symposium on Security and Privacy*. IEEE, 2020.

[114] Hui Zou and Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2005.