REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggesstions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA, 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any oenalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
30-07-2023	Final Report	11-Sep-2017 - 10-Jul-2019

4. TITLE AND SUBTITLE
Final Report: Bridging the Hardware-Software Gap: A Proof-Carrying Approach for Computer Systems Trust Evaluation

5a. CONTRACT NUMBER
W911NF-17-1-0477

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES

University of Florida - Gainesville
219 Grinter Hall
PO Box 115500
Gainesville, FL 32611 -5500

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
71935-NC.4

12. DISTRIBUTION AVAILIBILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Yier Jin
UU	UU	UU	UU		19b. TELEPHONE NUMBER 407-823-5321

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

RPPR Final Report
as of 31-Jul-2023

Agency Code: 21XD

Proposal Number: 71935NC
Agreement Number: W911NF-17-1-0477

INVESTIGATOR(S):

Name: Yier Jin
Email: jinyier@ufl.edu
Phone Number: 4078235321
Principal: Y

Organization: **University of Florida - Gainesville**
Address: 219 Grinter Hall, Gainesville, FL 326115500
Country: USA

DUNS Number: 969663814
EIN: 596002052
Report Date: 10-Oct-2019
Date Received: 30-Jul-2023
Final Report for Period Beginning 11-Sep-2017 and Ending 10-Jul-2019
Title: Bridging the Hardware-Software Gap: A Proof-Carrying Approach for Computer Systems Trust Evaluation
Begin Performance Period: 11-Sep-2017
End Performance Period: 10-Jul-2019
Report Term: 0-Other
Submitted By: Yier Jin
Email: jinyier@ufl.edu
Phone: (407) 823-5321
Distribution Statement: 1-Approved for public release; distribution is unlimited.

STEM Degrees: 2
STEM Participants: 4

Major Goals: The globalization of the semiconductor supply chain has significantly lowered the design cost and shortened the time-to-market [TTM] of Integrated Circuits [ICs] in the electronic industry. Over the years, the semiconductor industry has been restructured and has made significant adjustments to adapt to the trend of globalization. The fabless semiconductor companies have focused on high-profit phases such as design, marketing, and sales and have outsourced chip manufacturing, wafer fabrication, assembly, and packaging to third-party companies. The growth of fabless companies has also helped in the proliferation of the intellectual property [IP] industry. The use and reuse of existing commercial IPs has enabled improvements in TTM and cost reduction. Due to globalization of the semiconductor supply chain, companies and governments have decentralized control over this industry. As a consequence, tracking the source of third-party IP cores and monitoring fabrication processes within the foundries has become increasingly difficult, creating unique security concerns for the semiconductor industry. Vulnerabilities in the pre- and post-silicon stages of an IC supply chain may cause IP piracy and allow inclusion of Trojan circuits, which can hinder the growth of the hardware industry.

In order to secure computer systems built from third-party components, security researchers both in hardware and software areas have developed countermeasures to detect malicious modifications and have proposed various solutions to validate the trustworthiness of third-party resources. In the hardware domain, hardware Trojan detection, prevention, and trust evaluation methods have been proposed at the pre- and post-silicon stages to avoid the insertion of malicious logic in ICs. In the software domain, methods have been developed for kernel integrity defense and detection of malicious kernel extensions.

While the existing methods have proved effective in securing either the software or the hardware, system-level solutions targeting the entire computer system [particularly composed of third-party software programs and hardware IPs] are lacking. The software security methods assume the trustworthiness of the underlying hardware infrastructure. Similarly, the hardware security solutions do not consider the possible threats from the firmware or OS running on top of it. As a result, these methods fail to protect those computer systems where both the hardware and the software are vulnerable to attack. The semantic gap, which characterizes the difference between the operations performed by hardware and software, has been the major obstacle for developing system level security methods across the software-hardware boundary. Due to the lack of system-level protection, malicious software may exploit hardware backdoors and cause malfunctions, or leak internal information resulting in cross-layer attacks. Cross-layer attacks can easily evade either hardware or software level detection methods and cause harm to the computer systems.

To address the security issue but to overcome the shortages of existing methods, we propose a formal trust

evaluation method for the entire computer system at the pre-deployment stage. Since the semantic gap between the hardware and the software is the main obstacle for the development of cross-boundary security protection framework, our method, instead of bridging the semantic gap, eliminates the semantic gap by converting both hardware and software into a unified framework. More specifically, both software programs and the underlying hardware infrastructure will be converted and represented by the same formal language. Therefore, the whole computer system, including hardware infrastructure described by a hardware description language [HDLs] and software program written in a programming language, will be converted from their original languages into a newly proposed formal language supported by a formal platform. Unique characteristics of the proposed unified framework include: 1] support for formal reasoning to prove formal security properties on the entire computer systems; 2] elimination of the hardware-software boundary of the computer system by converting the code for both the hardware and the software into the same formal language; and 3] design of system-level security properties instead of individual security properties for the hardware and the software.

Accomplishments: During the research period, we worked on the area of computer system security assurance by eliminating the gap between the hardware platform and the software programs running on top of the hardware platform. Specifically, we conducted research on verification process automation to help solve the scalability issue for large-scale systems. Various accomplishments have been made to achieve the final goals. Three conference papers and one journal publication are published, including one Best Paper Award in the Design, Automation and Test in Europe Conference and Exhibition [DATE], 2019. Note that DATE conference is the top conference in the area of design automation and hardware security. The details of research accomplishments are listed as follows.

1] The wide usage of hardware intellectual property cores from untrusted vendors has raised security concerns for system designers. Existing solutions for functionality testing and verification do not usually consider the presence of malicious logic in hardware. Formal methods provide powerful solutions for detecting malicious behaviors in hardware. However, they suffer from scalability issues and cannot be easily used for large-scale computing systems. To alleviate the scalability challenge, we propose a new integrated formal verification framework to evaluate the trust of system-on-chip [SoC] constructed from untrusted third-party hardware resources. This framework combines an automated model checker with an interactive theorem prover to reduce the time for proving the system-level security properties of SoCs. Another factor contributing to the scalability issue is the effort required for manual conversion of the hardware design from register transfer level [RTL] code to a domain-specific language prior to verification. Consequently, we develop an automatic code converter for translating VHSIC hardware description language [VHDL] to Formal-HDL, which is a domain specific language for representing hardware designs in the language of Coq. To demonstrate the effectiveness of our integrated verification framework and automated code conversion tool, we evaluate a vulnerable program executed on a bare metal LEON3 SPARC V8 processor and prove system security with considerable reduction in verification effort. A paper based on the work has been published in the IEEE Transactions on Very Large Scale Integration System [TVLSI], the top-tier journal in the area of hardware design and security. A copy of this paper is also attached in this report.

2] Untrusted third-party vendors and manufacturers have raised security concerns in the hardware supply chain. Among all existing solutions, formal verification methods provide powerful solutions in the detection of malicious behaviors at the pre-silicon stage. However, little work has been done towards built-in hardware runtime verification at the post-silicon stage. Towards this direction, a runtime formal verification framework is proposed to evaluate the trust of hardware during its execution. This framework combines the symbolic execution and SAT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using an SoC design with untrusted IPs. The experimentation results show that the proposed approach can provide high-level security assurance for hardware at runtime. Two papers based on the work has been accepted and presented in the Asian Hardware Oriented Security and Trust [AsianHOST 2017] which was held in October 2017 and the 2018 Government Microcircuit Applications and Critical Technology Conference [GOMACTech-18] which was held in March 2018. AsianHOST is one of the academic conferences dedicated to hardware and system security. GOAMCTech is the leading conference with audience from government and military. The copies of both papers are attached in this report.

3] We broadly define charge-domain vulnerabilities from a systematic perspective and provide abstractions for their subsets, charge-sharing and capacitive-coupling leakage paths, accordingly. Our charge-domain abstraction presents an effective metric to study and analyze such stealthy attacks. We leverage the charge-domain leakage path model and develop an information flow tracking [IFT] based detection scheme for analog/mixed-signal vulnerabilities. Compared with existing digital-only IFT methods, we design the information flow policy with the consideration of fine-grain charge-domain behaviors. An automated tool is developed to demonstrate the effectiveness of the proposed IFT on analog vulnerabilities detection. This is the first work to formally analyze the

analog vulnerabilities in digital circuits. Our paper summarizing the work receives the Best Paper Award in Design, Automation and Test in Europe Conference and Exhibition [DATE], 2019.

Training Opportunities: Supported by this project, various training and professional opportunities are provided to the graduate students who are working in this project.

One graduate student, Xiaolong Guo, got the opportunity to attend the Design Automation Conference [DAC] in San Francisco, CA in June 2018. He is one of the finalists in the 2018 DAC PhD forum. The Design Automation Conference [DAC] is recognized as the premier conference for the design and automation of electronic systems. DAC offers outstanding training, education, exhibits and superb networking opportunities for designers, researchers, tool developers and vendors.

Meanwhile, Xiaolong Guo and another PhD student, Kaveh Shamsi, attended the IEEE Symposium on Hardware-Oriented Security and Trust [HOST], which was held in DC in May 2018. HOST aims to facilitate the rapid growth of hardware-based security research and development. HOST highlights new results in the area of hardware and system security. Relevant research topics include techniques, tools, design/test methods, architectures, circuits, and applications of secure hardware.

Mr. Xiaolong Guo who was involved in the project is currently an assistant professor in the Kansas State University.

Results Dissemination: During the research period, supported by this project, we published three conference papers and one journal paper. The PI [Dr. Yier Jin] has attended and presented research outcomes in the Asian Hardware Oriented Security and Trust [AsianHOST 2018], the 2018 Government Microcircuit Applications and Critical Technology Conference [GOMACTech-18] and the 2019 Design, Automation and Test in Europe Conference and Exhibition [DATE 2019].

In the Asian Hardware Oriented Security and Trust [AsianHOST 2018], we introduced a runtime formal verification framework to evaluate the trust of hardware during its execution. This framework combines the symbolic execution and SAT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using an SoC design with untrusted IPs. The experimentation results show that the proposed approach can provide high-level security assurance for hardware at runtime. The audience of the conference is mainly from academic institutions and industrial research labs.

In the 2018 Government Microcircuit Applications and Critical Technology Conference [GOMACTech-18], a similar runtime formal verification framework was presented with the audience mostly from DoD and DoD contractors.

In the 2019 Design, Automation and Test in Europe Conference and Exhibition [DATE 2019], our work on analog security was introduced.

Honors and Awards: The PI of the project was elected as the Distinguished Lecturer by the IEEE Council on Electronic Design Automation [CEDA] in 2019. This is a great achievement for researchers in the area of hardware security and design automation.

Protocol Activity Status:

Technology Transfer: Nothing to Report

PARTICIPANTS:

Participant Type: Graduate Student (research assistant)
Participant: Xiaolong Guo
Person Months Worked: 3.00 **Funding Support:**
Project Contribution:
National Academy Member: N

Participant Type: Graduate Student (research assistant)

Participant: Raj Gautam Dutta
Person Months Worked: 3.00 **Funding Support:**
Project Contribution:
National Academy Member: N

Participant Type: PD/PI
Participant: Yier Jin
Person Months Worked: 1.00 **Funding Support:**
Project Contribution:
National Academy Member: N

International Travel:

DEU 5 days

ARTICLES:

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published
Journal: IEEE Transactions on Very Large Scale Integration (VLSI) Systems
Publication Identifier Type: DOI Publication Identifier: 10.1109/TVLSI.2017.2751615
Volume: 25 Issue: 12 First Page #: 3390
Date Submitted: 8/9/18 12:00AM Date Published: 12/1/17 5:00AM
Publication Location:
Article Title: Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification
Authors: Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, Yier Jin
Keywords: Formal Verification; Hardware Security; Hardware Trojan Detection; Model Checking; Proof-Carrying Hardware; Theorem Proving
Abstract: The wide usage of hardware intellectual property cores from untrusted vendors has raised security concerns for system designers. Existing solutions for functionality testing and verification do not usually consider the presence of malicious logic in hardware. Formal methods provide powerful solutions for detecting malicious behaviors in hardware. However, they suffer from scalability issues and cannot be easily used for large-scale computing systems. To alleviate the scalability challenge, we propose a new integrated formal verification framework to evaluate the trust of system-on-chip (SoC) constructed from untrusted third-party hardware resources. This framework combines an automated model checker with an interactive theorem prover to reduce the time for proving the system-level security properties of SoCs.
Distribution Statement: 1-Approved for public release; distribution is unlimited.
Acknowledged Federal Support: **Y**

CONFERENCE PAPERS:

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: 2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)
Date Received: 09-Aug-2018 Conference Date: 19-Oct-2017 Date Published: 19-Oct-2017
Conference Location: Beijing
Paper Title: PCH framework for IP runtime security verification
Authors: Xiaolong Guo, Raj Gautam Dutta, Jiaji He, Yier Jin
Acknowledged Federal Support: **Y**

RPPR Final Report
as of 31-Jul-2023

Publication Type: Conference Paper or Presentation **Publication Status:** 0-Other
Conference Name: Government Microcircuit Applications and Critical Technology Conference
Date Received: 09-Aug-2018 Conference Date: 12-Mar-2018 Date Published:
Conference Location: Miami, FL
Paper Title: Runtime SoC Trust Verification using Integrated Symbolic Execution and Solver
Authors: Xiaolong Guo, Jiaji He, Yier Jin
Acknowledged Federal Support: **Y**

Partners

,

I certify that the information in the report is complete and accurate:
Signature: YIER JIN
Signature Date: 7/30/23 11:42AM

Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification

Xiaolong Guo, *Student Member, IEEE*, Raj Gautam Dutta, *Student Member, IEEE*, Prabhat Mishra, *Senior Member, IEEE*, and Yier Jin, *Member, IEEE*

***Abstract*— The wide usage of hardware intellectual property cores from untrusted vendors has raised security concerns for system designers. Existing solutions for functionality testing and verification do not usually consider the presence of malicious logic in hardware. Formal methods provide powerful solutions for detecting malicious behaviors in hardware. However, they suffer from scalability issues and cannot be easily used for large-scale computing systems. To alleviate the scalability challenge, we propose a new integrated formal verification framework to evaluate the trust of system-on-chip (SoC) constructed from untrusted third-party hardware resources. This framework combines an automated model checker with an interactive theorem prover to reduce the time for proving the system-level security properties of SoCs. Another factor contributing to the scalability issue is the effort required for manual conversion of the hardware design from register transfer level (RTL) code to a domain-specific language prior to verification. Consequently, we develop an automatic code converter for translating VHSIC hardware description language (VHDL) to *Formal-HDL*, which is a domain specific language for representing hardware designs in the language of Coq. To demonstrate the effectiveness of our integrated verification framework and automated code conversion tool, we evaluate a vulnerable program executed on a bare metal LEON3 SPARC V8 processor and prove system security with considerable reduction in verification effort.**

***Index Terms*— Formal verification, hardware security, hardware trojan detection, model checking, proof-carrying hardware, theorem proving.**

I. Introduction

THE changing landscape of the semiconductor industry has increased the demand for intellectual property (IP) cores. Various factors, such as reduced time to market and lower design cost, have led to the proliferation of the IP market. Another contributor to this growth is the use of system-on-chip (SoC) platforms for mobile and Internet of Things applications. SoC is a monolithic chip containing all the essential components for mimicking the functionality of a computer. It is designed by integrating multiple IP cores from trusted and untrusted third-party vendors.

Manuscript received December 31, 2016; revised May 15, 2017 and July 25, 2017; accepted August 30, 2017. Date of publication October 2, 2017; date of current version November 22, 2017. This work was supported in part by the National Science Foundation under Grant CNS-1319105 and Grant CNS-1441667, in part by Semiconductor Research Corporation under Grant 2014-TS-2554, in part by the Army Research Office under Grant W911NF-17-1-0477, and in part by Cisco. *(Corresponding author: Yier Jin.)*

X. Guo and Y. Jin are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: guoxiaolong@ufl.edu; yier.jin@ece.ufl.edu).

R. G. Dutta is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA (e-mail: rajgautamdutta@knights.ucf.edu).

P. Mishra is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: prabhat@ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2751615

An increasing number of the third-party vendors have raised security concerns in the hardware industry. Consequently, security researchers in their respective domains have started putting in considerable effort to ensure the trustworthiness of the third-party resources. In the hardware security industry, multiple countermeasures have been developed for the verification and validation of SoCs at the pre- and postsilicon levels [1]–[11].

Among all the existing techniques, formal methods (both automated and deductive) have been most effective in detecting vulnerabilities in hardware [4]–[12]. For example, model checking is used for detecting malicious logic that corrupts data in critical registers of the third-party IP cores [11]. In a model checker, security properties, such as integrity (related to safety) and availability (related to liveliness), are represented as *traces*, and it checks all the possible traces generated by the system. If all the traces are good, then the system is said to satisfy the security properties.

However, not all security properties can be expressed as traces, such as noninterference property [13]. Moreover, model checkers run into the state space explosion problem when the system under consideration is very large. Due to these limitations of model checkers, theorem provers are being mostly used for the verification of large-scale hardware designs [5]–[7], [10].

Although these methods have proved effective in securing the hardware, system-level solutions targeting the entire SoC (particularly composed of the third-party hardware IPs) are lacking. Moreover, the existing formal verification frameworks, such as proof-carrying hardware (PCH), which rely on an interactive theorem prover for evaluating the trustworthiness of IP cores, are not scalable to SoC designs [5]–[7]. The reasons behind the scalability problem are: 1) a significant manual effort was required for converting hardware description language (HDL) code to a formal representation and 2) the lack of efficient methods for constructing machine proofs. As the size of the design increases, time required for converting HDL programs and proving security properties on the design grows exponentially. Moreover, any modification of the design required the repetition of the entire deductive process, thereby further increasing the verification time.

To solve these problems, we use an integrated automatic formal verification framework [14], where we combined a model checker with an interactive theorem prover for proving security properties on SoC. Integrating these two techniques

1063-8210 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

overcomes the state-space explosion problem in model checking approaches, and it reduces the time required for constructing machine proofs of the properties in the theorem prover. Moreover, an automatic tool proposed in [15] for the syntactic and semantic translation of register transfer level (RTL) code to a domain-specific language is used for eliminating manual effort incurred during code conversion. Compared with the manual translation in previous PCH frameworks [5], [6], [16], this tool considers all the common VHSIC HDL (VHDL) syntaxes and converts the hardware design in VHDL to *Formal-HDL*. Thus, the proposed integrated automatic framework and the automated code conversion tool can reduce effort required in formal verification of large-scale hardware designs.

The main contributions of this paper are as follows.

1) We developed a VHDL-to-Coq code converter to automate the code conversion process in the PCH framework.
2) We combined the interactive theorem prover, Coq, with the Cadence incisive formal verifier (IFV) model checker for verifying system-level security on SoC designs. The method was proposed in [14], which was the first attempt to verify security properties on large-scale hardware designs through the combination of both techniques. In this paper, we have demonstrated its applicability in an SoC design.
3) We describe the method for decomposing the hardware design and the security specification into submodules and subspecifications, respectively. These submodules and subspecifications are verified using the model checker. In the Coq platform, we combine the subspecifications to prove the security property. Following this strategy, our approach can verify large systems and, thus, help alleviate the scalability issue.

The rest of this paper is organized as follows. In Section II, we discuss previous work on malicious logic detection using formal techniques and mention the existing tools that convert hardware design written in an HDL. In Section III, we introduce the threat model and provide some relevant background on formal languages for specifying security properties, theorem provers, and model checkers. We explain our integrated framework, semantic translation of VHDL language, and elaborate on the proof construction procedure, and the automatic PCH framework in Section IV. In Section V, we provide the implementation details of the code converter and demonstrate its applicability by translating hardware designs described in VHDL to Coq language. Section VI presents the demonstrations of both our integrated verification framework, and final conclusions are drawn in Section VII.

II. Related Work

Currently, formal methods have been extensively used for the verification and validation of security properties at pre- and postsilicon stages [4]–[11]. In [4], a multistage approach was adopted for identifying suspicious signals using assertion-based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and sequential Automatic Test Pattern Generation. The PCH framework has been effective in ensuring the trustworthiness of soft IP cores [5]–[7], [9], [10].

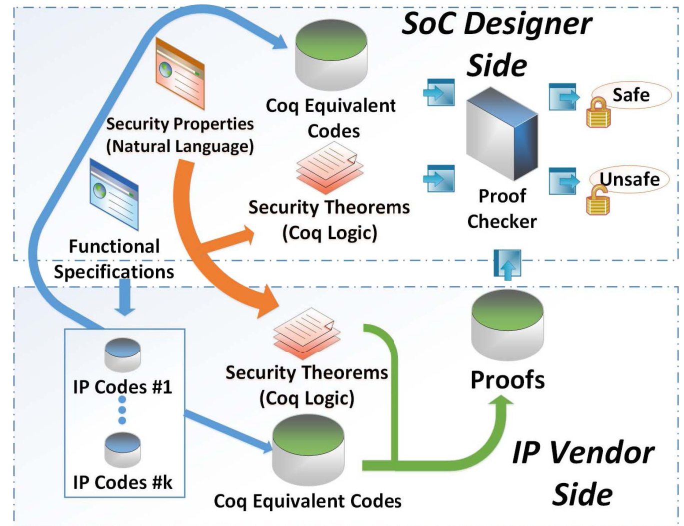

Fig. 1. Working procedure of the PCH [9].

This method was inspired from the proof-carrying code approach [17]. Drzevitzky [10] proposed the first PCH framework for dynamically reconfigurable hardware platforms. They used runtime combinational equivalence checking to verify the equivalence between the design specification and the design implementation. However, instead of using security properties, the approach verified safety policies on the design. Another PCH framework was proposed for security property verification on soft-IP cores [5]–[7], [9]. In this framework, the Coq proof assistant [18] was used to represent security properties, hardware designs, and formal proofs, as shown in Fig. 1. Coq, as a formal proof management platform, provides a formal language to write theorems, algorithms, and mathematical definitions together with an interactive proof environment. Details of the Coq can be found in Section III. However, this framework was not scalable to large SoC designs because of the extremely high conversion and verification efforts in proving security properties on large designs.

There are several methods for reducing complexity in verifying large systems in both hardware and software. In [19] and [20], algorithms were developed to improve the efficiency of proving safety through storing information from previous verification. But it was limited to Boolean systems. In [21], a highly efficient method for mining formal specifications automatically was developed, and this method was used for localizing errors in digital circuit. A scalable model checking technique was developed in [22] for verifying message passing interface systems and identifying potential deadlocks. However, all the above-mentioned methods are either limited to a specific system or not designed for security purpose. Our framework is the first attempt to apply an integrated (model checking and deductive reasoning) approach for verifying security properties on SoC designs.

In semiconductor industry, automated tools, such as equivalence checker and model checker, have been consistently used for functional verification of hardware designs [23]. Using these tools, a model represented as a transition system is verified against a set of behavioral specification stated in temporal logic. Recently, model checkers have been used for

detecting malicious signals in the third-party IP cores [4], [11]. However, these tools suffer from the state space explosion problem, and hence cannot be used for verifying large designs individually.

Some efforts have been made to combine theorem provers with model checkers for the verification of hardware systems [24], [25]. These methods try to overcome the limitations of both techniques. Some of the popular theorem provers, such as higher order logic and prototype verification system have integrated model checkers. These tools have been used for the functional verification of hardware systems. However, to the best of our knowledge, this combined technique has not been extended toward the verification of security properties on the third-party soft IP cores.

In [14], we have combined a model checker with an interactive theorem prover for verifying security properties on SoCs. Through the proposed method, we were able to significantly reduce the time required for security verifications on SoCs. However, a verification expert had to manually translate a hardware design to formal equivalent description, which required lot of effort.

A language translation tool called *VeriCoq* was developed in [26], which converted hardware designs represented in Verilog to Coq. However, *VeriCoq* required flattening the hierarchical design, which made deductive verification in Coq very challenging. Moreover, Bidmeshki and Makris [26] did not provide the details of the supported Verilog syntaxes the *VeriCoq* can support, and the demonstration in this paper was not sufficient to show the applicability of the *VeriCoq* tool to any general hardware design. Thus, we developed the VHDL-to-Coq code converter, which can convert general VHDL designs to Coq equivalent codes by supporting all common VHDL syntaxes [15]. We use this tool to improve our proposed integrated PCH framework.

III. Background

A. Attack Model and Assumptions

Malicious logic is inserted by an adversary at the design stage of the supply chain. We assume that the rogue agent at the third-party IP design house can access the HDL code and insert a hardware Trojan or backdoor to manipulate critical registers of the design. Such a Trojan can be triggered either by a counter at a predetermined time, by an input vector, or under certain physical conditions. Upon activation, it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware. In this paper, we only consider Trojans, which can be activated by a specific "digital" input vector.

We assume that the verification tools (e.g., Coq and Cadence IFV) used in our integrated framework produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design. However, the framework does not provide protection of an IP from Trojans whose behaviors are not captured by the set of security properties. Furthermore, we assume that the attacker has intricate knowledge of the hardware to identify critical registers and modify them for carrying out the attack.

B. Formal Specification Languages

Specifications are used for representing (using natural language or experimental data) the security properties of a system at a high level of abstraction. In formal specification, these properties are translated from nonmathematical description to a mathematical format using logic. This conversion helps to overcome any ambiguity in the security specifications. There are many formal specification languages, including propositional logic, temporal logic, and so on.

In the Coq proof assistant [18], behavioral specifications are written using the *Gallina* specification language. This language can also be used to represent the hardware design. In case of an automated tool, such as a model checker, specification language, such as the Property Specification Language, is used for specifying the properties or assertion of hardware designs. The directives of the PSL language, *assert*, *assume*, and *cover*, are understood by a verification tool, such as the Cadence IFV. By using the *assert* construct, a user can check at run time or at simulation time if a certain condition holds and reports a warning or an error if it does not hold. To put constrains on inputs of the design, *assume* is used and *cover* is used for specifying scenarios.

The PSL language is divided into four layers: 1) Boolean layer; 2) temporal layer; 3) verification layer; and 4) modeling layer [27]. The Boolean layer is composed of Boolean expressions that either hold or not hold over a given clock cycle. The temporal layer allows to relate the Boolean expression with time. This layer is further divided into: 1) foundation language (FL) and 2) optional branching extension (OBE). The FL is used to describe linear properties in which there is only a single successor for a current state. Therefore, FL is often used to describe traces/path. In the FL, the linear temporal logic and the sequential extended regular expression are used to represent the behavioral specifications of the system. Alternatively, OBE is based on computational tree logic and can describe multiple traces (i.e., successor states) at a time. The verification layer consists of directives, which describe how the temporal properties should be used by the verification tool. That is, the verification layer specifies the semantics for PSL directives and operators in the temporal layer. It also helps the verification tool to understand the difference between properties, which use *assert*, *assume*, and *cover* directives. The modeling layer provides a means to the model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables.

The alphabets of Boolean expressions in the Boolean layer include Boolean variables, logical connectives, relational operators, and bitwise operators. A formula (ϕ) in the Boolean layer over Boolean variable *(v)* is given as follows:

$$\phi ::= \text{true} \mid v \mid (\phi_1 \wedge \phi_2) \mid \neg\phi.$$

Here, $\wedge$ and $\neg$ are conjunction and negation operators, respectively. The rest of the Boolean connectives, $\vee$ (disjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence), can be derived from $\wedge$ and $\neg$. The PSL language also supports suffix implication operators, $\mapsto$ and $\Mapsto$, for linking two regular expressions.

C. Interactive Theorem Prover

Theorem provers are used to prove or disprove the properties of systems expressed as logical statements. Over the years, several theorem provers (both interactive and automated) have been developed for proving the properties of hardware and software systems. However, using them for the verification of large and complex systems requires excessive effort and time. Irrespective of these limitations, theorem provers have currently drawn a lot of interest in the verification of security properties on hardware. Among all the formal methods, they have emerged as the most prominent solution for verifying large designs. A recent application of an interactive theorem prover in order to ensure the trustworthiness of soft IP cores is called PCH [5], [10].

Coq is an interactive theorem prover/proof assistant, which enables the verification of software and hardware programs with respect to their specification. In Coq, programs, properties, and proofs are represented as terms in the *Gallina* specification language. By using the *Curry–Howard Isomorphism*, the interactive theorem prover formalizes both program and proofs in its dependently typed language called the *Calculus of Inductive Construction*. Correctness of the proof of the program is automatically checked using the inbuilt type checker of Coq. For expediting the process of building proofs, Coq provides a library consisting of programs called *tactics*. However, using *tactics* does not significantly reduce the time required for certifying large (consisting of hundred thousand lines of code) software and hardware programs.

D. Model Checking

Model checking [28] is a method for verifying and validating models in software and hardware applications [12], [23]. In this approach, a model (Verilog/VHDL code of hardware) $\mathcal{M}$ with an initial state $\mathtt{s}_0$ is expressed as a transition system, and its behavioral specification (assertion) ϕ is represented in a temporal logic. The underlying algorithm of this technique explores the state space of the model to find whether the specification is satisfied. This can be formally stated as, $\mathcal{M},\ \mathtt{s}_0 \models \phi$. If a case exists where the model does not satisfy the specification, a counterexample in the form of a trace is produced by the model checker [29]. The application of model checking techniques, including symbolic approaches based on the reduced order binary decision diagrams (ROBDDs) and satisfiability (SAT) solving, to SoC has had limited success due to the state-space explosion problem. For example, a model with n Boolean variables can have as many as 2^n states, and a typical soft IP core with 1000 32-bit integer variables has billions of states.

Symbolic model checking (SMC) using ROBDD is one of the initial approaches used for hardware systems verification. Unlike explicit state model checking where all states of the system are represented using global state graph, the SMC represents states of the transition system using ROBDD. The ROBDD is a unique, canonical representation of a Boolean expression of the system. Subsequently, the specification to be checked is represented using a temporal logic. A model checking algorithm then checks whether the specification is true on a set of states of the system. Despite being a popular data structure for symbolic representation of states of the system, ROBDD requires finding an optimal ordering of state variables, which is an NP-hard problem. Without the proper ordering, size of the ROBDD increases significantly. Moreover, it is memory intensive for storing and manipulating BDDs of system with a large state space.

Another technique called bounded-model checking (BMC) replaces BDDs in symbolic checking with SAT solving [30]. In this approach, a propositional formula is first constructed using a model of the system, the temporal logic specification, and a bound. Then, the formula is given to an SAT solver to either obtain a satisfying assignment or to prove there is none. Although BMC outperforms BDD-based model checking in some cases, the method cannot be used to test properties (specification) when bound is large or cannot be determined.

To overcome the limitations of the model checking and the theorem proving approaches, we propose to combine these two techniques to verify security properties on SoCs designs. Specifically, we have combined an industrial model checker Cadence IFV with Coq for verifying hardware designs written in VHDL in this paper.

IV. Methodology

The existing PCH framework uses an interactive theorem prover for verifying security properties on soft IP cores, which triggers a large design overhead [5], [6], [16]. Moreover, PCH requires flattening of the hardware design before translation of the HDL code to the formal language. Design flattening increases the verification effort and adds to the risk of introducing errors during the code conversion process.

Meanwhile, model checkers, such as Cadence IFV, cannot be used for verifying systems with large state space either because of the space explosion problem. As the number of state variables $(\mathtt{n})$ in the system increases, amount of space required for representing the system and the time required for checking the system increases exponentially $[\mathtt{T(n)} = 2^{O(n)}]$.

To overcome the scalability issue and to verify an SoC, we introduce the *integrated formal verification framework* (see Fig. 2), where the security properties are checked against SoC designs. In this framework, the theorem prover is combined with a model checker for proving formal security properties (specifications). Moreover, the hierarchical structure of the SoC is leveraged to reduce the verification effort.

The entire working procedure of the proposed framework is shown in Fig. 4. In the integrated framework, the top level/module of hardware design, represented in an HDL, is first translated to the Coq equivalent codes in *Gallina*. Then, the security specification is stated as a formal theorem in Coq. In the following step, this theorem is decomposed into disjoint lemmas (see Fig. 3) based on submodules. These lemmas are then represented in the PSL specification language and are called subspecifications.[1] Subsequently, the

[1]Note that in Fig. 4, the theorem decomposition and lemmas representation are done by an Interpreter, which are out of the scope of this paper and will be discussed in our future work.

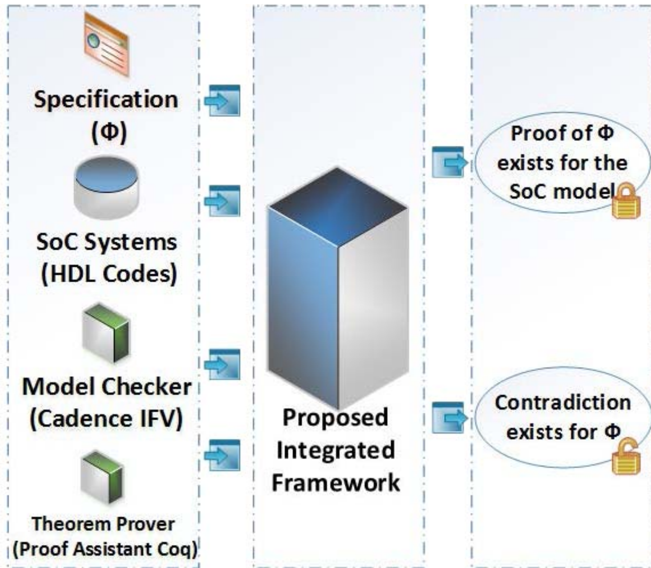

Fig. 2. Integrated formal verification framework.

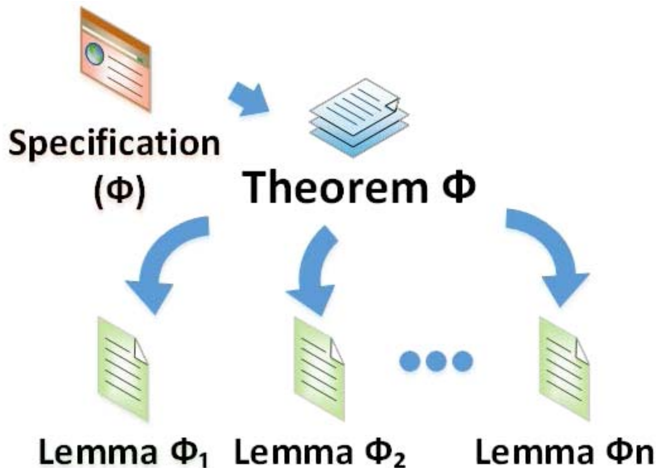

Fig. 3. Security specification (ϕ) decomposed into lemmas.

Cadence IFV verifies the submodules against the corresponding subspecifications. Submodules are functions, which have less number of state variables and are connected to primary output of the design. These functions are always from the bottom level of SoC and have no dependence relationship with each other.

The HDL code of a large design consists of many such submodules. If the submodules satisfy the subspecifications, we consider the lemmas are proved. Checking the truth value of the subspecifications with a model checker eliminates the effort required for proving the lemmas and translating the submodules to Coq. Upon proving these submodules, we then use Hoare logic to combine the proof of these lemmas to prove the security theorem of the entire system in Coq.

A. Semantic Translation

The developed semantic translation method is based on the *Formal HDL* of [16]. Using this method, the HDL code of the SoC and the informal security properties are translated to *Gallina*. During the translation process, the syntax and semantics of the HDL are represented in Coq. To preserve the hierarchical design of the SoC, we use the *module* functionality of Coq. In this paper, semantic translation is made by an automatic RTL-to-Formal code converter developed in [15]. In Section IV-B, the details of the tool will be provided.

Gallina is also used to represent the security properties (theorems) in Coq. PSL is used for representing the security lemmas written in Coq. PSL uses HDL operators, temporal operators, and regular expressions to represent the properties (assertions/specifications) of the hardware design. An industrial model checker, such as Cadence IFV, can interpret PSL properties and use them to verify the HDL code of the design.

B. Distributed Proof Construction

Proof construction procedure limits the scalability of the PCH framework to large designs [5]. Consequently, we improve scalability by combining a model checker (Cadence IFV) with a theorem prover (Coq). In Coq, the proof construction process follows Hoare-logic style reasoning, where the trustworthiness of the designs, represented in the HDL code, is determined by ensuring that the code operates within the constraints of the precondition and the postcondition. The precondition of the formal HDL code is the initial configuration of the design and the postcondition is the security theorem. The security theorem will be divided into lemmas. Then, lemmas are translated to the PSL specification language, so-called subspecifications. Similarly, the HDL code is decomposed into submodules. A model checker then determines whether the submodule satisfies the corresponding subspecification. If it is satisfied, then we can state that the lemmas are proved. Such lemmas are combined at the end to prove the system-level security theorem.

In the mentioned procedure, the decomposition method is applied to reduce the time complexity of verification significantly. As mentioned earlier, limited by the state-space explosion problem, model checker can only provide comprehensive verification to small-scale circuit. Meanwhile, in the scenario of utilizing theorem prover to large system, interactive proving leads to high manual workload. Using the proposed integrated framework, an SoC design can be decomposed into top module and submodules. Model checker is utilized to check submodules, which are all in simple and small scale, while Coq is used in verifying top module only, so that fewer interactive proofs are required. Therefore, compared with the previous PCH framework, the new approach reduces verification complexity from exponential to linear.

Application scenario of model checker, such as IFV, is checking simple and small circuit. When the system becomes larger, the time complexity of IFV will be extremely high for the reason that the proof engine of IFV will traverse all the possible states in a module. In some cases, pruning strategy will be utilized to reduce complexity. However, pruning can let Trojan bypass the checking, considering that Trojans are always hidden deeply in the circuit. On the other hand,

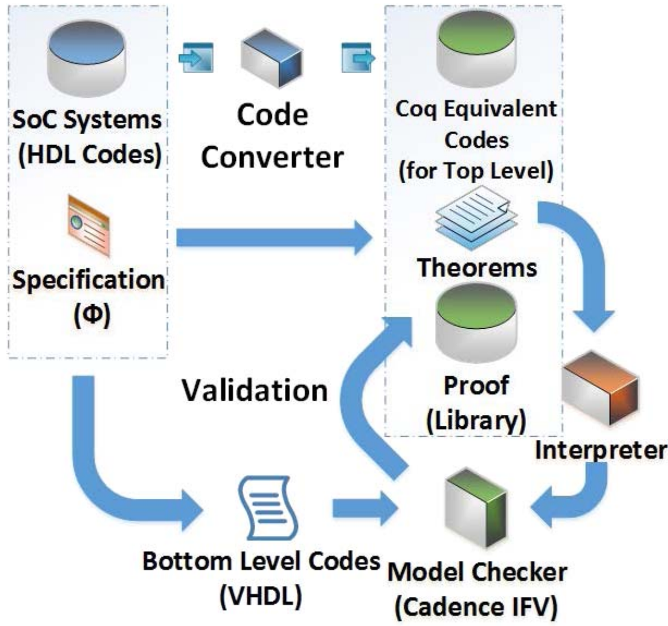

Fig. 4. Automatic PCH framework.

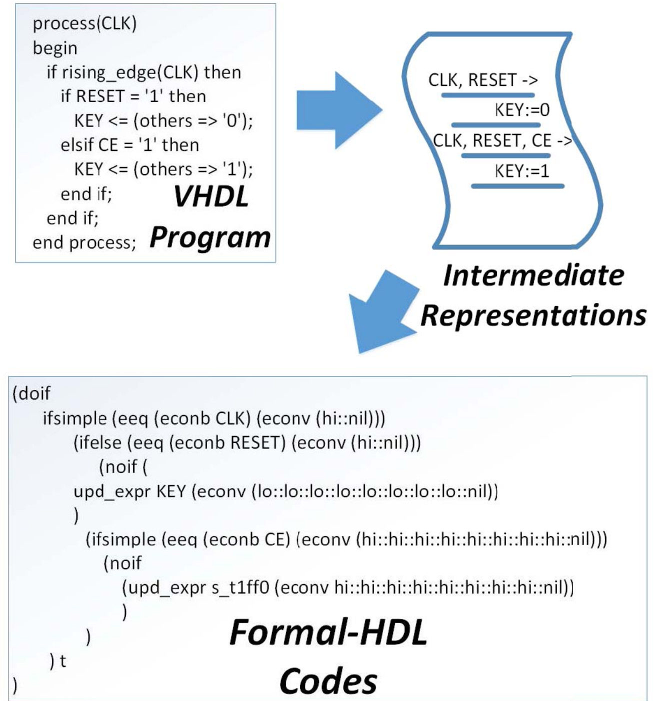

Fig. 5. Code conversion from VHDL to *Formal-HDL* through IRs.

theorem prover, such as Coq, has the capability of verifying large system instead of traversing states, but interactive proving process leads to too much manually workload. So we integrate them together, and decompose the SoC into submodules and top module. Then, IFV is utilized to check submodules, which are all in simple and small scale. Correspondingly, Coq is used in verifying top module only, and fewer interactive proofs are required.

C. Automatic PCH Framework

As mentioned in the previous sections, extending PCH method to the large-scale design, such as SoCs, was difficult due to the time required for verification. Therefore, a scalable framework for the formal verification of SoC security was required to be developed to alleviate this challenge.

Considering the PCH working procedure of Fig. 1, it is desired to maximize the steps that can be executed automatically. The software tools we develop will help facilitate these processes as shown in Fig. 4. By using Cadence IFV, parts of hardware designs can be verified automatically. On the Coq side, there is a proof checker provided by Coq platform. So the proofs can be checked in minutes or seconds. Meanwhile, in previous PCH methods, code conversion was done manually, which increases the workload and the risks of human error. Hence, an automated code converter, which would be discussed in Section V, is applied to simplify this conversion process. By automating the working procedure, scalability issue on code conversion can be solved in the enhanced PCH framework.

V. Automatic RTL-to-Formal Code Converter

As discussed in Section IV-A, semantic translation is required to convert the SoC designs to its Coq equivalent representation. As such, we have developed an automatic code converter for translating VHDL to *Formal-HDL* in [15]. By using the converter, the hardware design constructed using VHDL syntaxes is first converted to an intermediate representation (IR). Then, the IRs are translated to *Formal-HDL* as shown in Fig. 5.

In this section, the working procedure of the tool is discussed in detail. Then, the applicability of this converter is demonstrated on three hardware designs.

A. VHDL to Intermediate Representations

Our converter extends the work of [31], where a tool was developed to translate VHDL to counter automata. Their tool supported the following syntaxes of the VHDL language: entity, generic, architecture, signals, process, direct assignment, and *if-else* statement. The tool developed in this paper incorporates additional syntaxes, such as component instantiations, user-defined types, ranged types, constants, 2-D array, and case statements.

The IRs are constructed using variables V, functions T, and behavioral rules B. The expressions E can be formed by using V, arithmetical ($+, -, \times, \div, \oplus$), relational ($=, \neq, >, <, \leq, \geq$), and logical ($\neg, \bigvee, \bigwedge$). Let C be the subset of E containing all Boolean valued expressions. Then, a behavioral rule $b \in B$ can be written as

$$c \rightarrow \upsilon := e \quad (1)$$

where $c \in C$, $\upsilon \in V$, and $e \in E$. Equation (1) signifies that under a list of enabling conditions c, a variable υ is assigned to a new value, defined by an expression e. As shown in Fig. 5, most of the VHDL codes will be parsed to IRs in the form of (1).

B. Formal-HDL Representations

As shown in Fig. 1, the first step in verifying the security properties of IP cores is converting the code written in HDL into a domain specific language, so that the proof assistant can recognize and construct proofs.

The *Formal-HDL* of [16] can represent basic circuit units, combinational logic, sequential logic, and module instantiations. In [32], *Formal-HDL* is further updated to include component instantiations to preserve the design hierarchy of the SoC. In the following, we show the code conversion details of hardware designs from VHDL to Coq equivalent expressions.

1) Data Types: To represent a single regular logical value in hardware, a $value$ type is defined as an enumeration, which includes three elements—hi, lo, and x, where hi stands for high voltage or logical value 1, lo stands for low voltage or logical value 0, and x stands for all other unknown values. To define binary logical values and vectors, a *bus* type is defined as a function in *Formal-HDL*, which takes one parameter, a timing variable t, and returns a list of signal values with data type $value$. Since the *Formal-HDL* can be applied to only synchronous hardware, the variable t indicates the global clock cycle.

Listing 1 Date Types in Formal HDL Syntax

```
Inductive value := lo|hi|x.
Definition bus_value := list value.
Definition bus := nat -> bus_value.

Definition wire := bus.
Definition reg := bus.
```

2) Structural Syntax: As the most important behavior, the updates of wire and flip-flop/latch are distinguished as blocking assignment and nonblocking assignment such as in VHDL. Then, the keyword *assign* of the *Formal-HDL* is used for blocking assignment, while *update* is used for nonblocking assignment. During the blocking assignment, the bus value is updated in the current clock cycle, and in the nonblocking assignment, the bus value is updated in the next clock cycle.

Listing 2 Assignment in Formal HDL Syntax

```
Fixpoint assign (a:assignblock)(t:nat){struct a}:=
 match a with
 | expr_assign bus_one e => bus_one t = eval e t
 | assign_useless => True
 | assign_cons a1 a2 =>(assign a1 t)/\(assign a2 t)
 end.

Fixpoint update (u:updateblock)(t:nat){struct u}:=
 match u with
 | (upd_expr bus exp) => (bus (S t))=(eval exp t)
 | (updcons block1 block2) =>
                          (update block1 t)/\
                          (update block2 t)
 | upd_useless => True
 end.
```

To facilitate clock-edge specifications and synchronizations among signal assignments, processes are used in VHDL. In *Formal-HDL*, these behaviors are characterized using the following logical syntax, and constructed using propositional logic symbol $\bigwedge$.

3) Logical Syntax: To represent logical interactions between signals, arithmetic, relational, and logic operations are defined in *Formal-HDL*. For example, the logic operator *exclusive* OR is defined as a type with two input and one output

$$\text{exor} : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}. \tag{2}$$

The key word *expr* stands for expressions and is the parent type of all the logic operations.

Another commonly used form of syntax is conditional statements. According to two assignment types, conditional statements are designed as blocking if statements *adoif* and nonblocking if statements *doif*. The example of a nonblocking if statement is shown in List V-B3.

Listing 3 Non-blocking If-Then-Else Statement in Formal HDL Syntax

```
Fixpoint doif (i : ifblock)(t : nat)
 {struct i} :=
 match i with
  | (noif up) => (update up t)
  | (ifsimple exp ifb) =>
      match (eval exp t) with
       | hi::nil => (doif ifb t)
       | lo::nil => True
       | x::nil => True
       | _ => True
      end
  | (ifelse exp ifb1 ifb2) =>
      match (eval exp t) with
       | hi::nil => (doif ifb1 t)
       | lo::nil => (doif ifb2 t)
       | x::nil => (doif ifb2 t)
       | _ => True
      end
end.
```

4) Module Structure: For hardware infrastructure, the *Formal-HDL* supports hierarchical designs where basic functional blocks and low-level modules are instantiated in a high-level structure (note that processors often follow the hierarchical structure because of their high complexity). Like the $entity$ in VHDL, keywords *Module Type* are defined for circuit module definitions. And the other submodules' instantiations inside a top module are defined by using keywords *Declare Module*. Meanwhile, in each module, circuit details are described by using the keyword *Fixpoint*, which is a special syntax provided in Coq for generic primitive recursion. The input parameter of *Fixpoint* is defined as an *inductive* type, which explains how the inhabitants of the type are built by giving names to each construction rule. This specific inductive type is treated as an interface, which provides the rule of how the entities are connected. To make it clear, an example describing two submodules in a top module is shown in List V-B4.

C. Automatic Code Converter Development

In this section, we show the development of a Python-based automatic code converter, and show the results of converting

Listing 4 Circuit Module Definition in Formal HDL

```
Inductive ip_circuit :=
 | ip_submodule_1 : bus->bus->bus->...->ip_circuit
 | ip_submodule_2 : bus->bus->bus->...->ip_circuit
 | ip_topmodule : ...->ip_circuit->ip_circuit.
 ...
Module Type module_top.
 Declare Module inst_submodule_1 : ip_submodule_1.
 Declare Module inst_submodule_1 : ip_submodule_2.
 ...
 Fixpoint module_inst (m:ip_circuit) (t:nat) :=
 match m with
  | (ip_submodule_1 ...)=> inst_submodule_1.module_inst m t
  | (ip_submodule_2 ...)=> inst_submodule_2.module_inst m t
  | (ip_topmodule ...) => ... (module_inst sub1 t)
                             /\(module_inst sub2 t)
 end.
End module_top.
```

three VHDL design—Advanced Encryption Standard (AES) Encoder, Data Encryption Standard (DES), and RS232—to their Formal HDL equivalent expressions.

For parsing VHDL to IRs, we use the translator from [31]. Furthermore, we extend the tool to support more VHDL syntaxes, such as constants, component instances, choices in expressions, and user-defined types. For translating IRs to Formal HDL, mapping is built. For instance, as shown in (1), the conditions, variables, and expressions are described using if-then-else statements in Formal HDL. Fig. 5 shows the format of IRs and Formal HDL codes during the conversion of a small VHDL program.

A structural block diagram of the automated code converter is given in Fig. 6. The *VHDL2Coq* module is the access point of the program, and it reads a VHDL file and creates Coq file. The *VHDL_Process* module then generates *Formal HDL* codes based on the IRs produced by a parser. The parser is named *VHDL_Parser*, which contains *lex*, *yacc*, and *VHDL_Syn* for parsing VHDL to IR. The *parsetab* is used to accelerate the lexical analysis in parsing process. Optimization and simplification are done in *CoqFile* and *Optimize* stages. In the entire block diagram, *VHDL_Syn* is the most important part of the code converter. We develop this module to define the conversion of each element in the VHDL syntax. Later, we summarize all the syntaxes that are required to be converted. To deal with the syntax in the VHDL language, we used an object-oriented model to represent the *VHDL_Syn*. Each syntax element will be emulated as a class. And all the elements are inherited from a super class called *VHDL_Object*, which defines the common behaviors of all VHDL syntax.

To test the proposed automatic RTL-to-Formal code converter, we have applied the converter to three VHDL applications from [33]—AES core for encoding [34], Basic data encryption standard Crypto Core [35], and RS232 Communication Controller [36], and then the integer unit (IU) of LEON3. The example has been tested on a desktop with 64-bit Intel i7-3370 CPU and 16-GB RAM.

The results of the experiments are summarized in Table I. Lines of codes for each testbench, which stands for manually workload of developers, are shown in the first column.

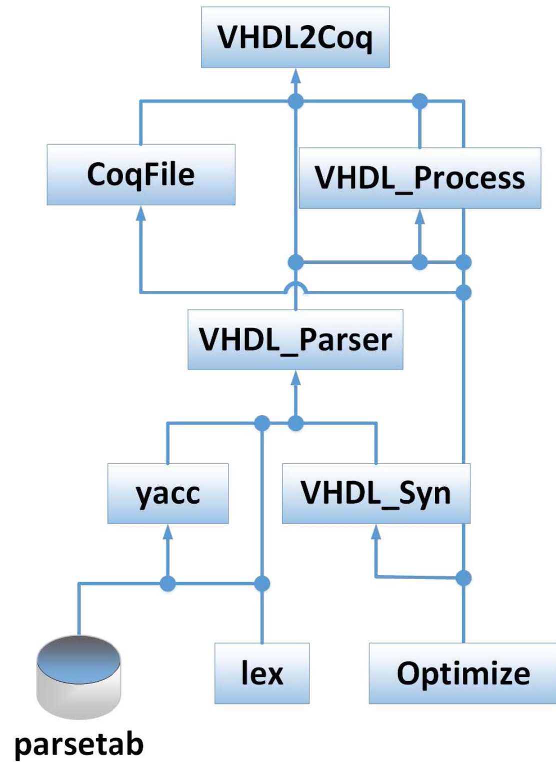

Fig. 6. Code conversion from VHDL to Formal-HDL through IRs.

The second column gives the number of registers used in the design. The next column provides the number of lookup tables applied to this circuit on a field-programmable gate array. Finally, the last column provides the time-consuming in conversion. And we can get the conclusions that: 1) time-consuming is increased with the scale of VHDL design and 2) time-consuming is acceptable for converting VHDL design to Coq equivalent expressions.[2]

VI. Case Study

To demonstrate the effectiveness of the proposed integrated verification framework supported by the automatic code converter, we consider a 32-bit LEON3 processor implementing the SPARC V8 architecture. This processor core is written in VHDL. The IU of the core, a seven-stage pipeline, is considered for verification (see Fig. 7). In order to prove the presence/absence of malicious logic that can trigger illegal data writing, we will check the signals connecting the IU to the register file.

In this experiment, we consider a hardware Trojan embedded in the processor, which may manipulate internal data. Execution of the *call* instruction is a necessary condition that triggers the Trojan. In order to understand the attack easier, we bring in the following assembly code of the subroutine, *vulnerable_function*, which is working with the assumption that the Trojan has been triggered. The payload of this Trojan can write a specific value to the register, which is used to store the return address. Then, the data overwriting can cause the redirection of the program counter.

[2] In previous PCH framework, several days are required in this conversion manually.

TABLE I
TIME CONSUMED FOR CONVERTING THE VHDL DESIGN TO FORMAL HDL EXPRESSIONS

Hardware Design	Lines of Codes	Number of Regs	Number of LUTs	Time Consuming
AES Encoder	670	326	800	1.526s
DES	1056	197	360	1.071s
RS232	291	74	97	0.144s
IU in Leon3	1707	853	1817	3.982s

```
<vulnerable_function>:
    save %sp, -200, %sp
    ...
    mov %g1, %o0
    call 0x206b4 <strcpy@plt>
    nop
    nop
    restore
    retl
```

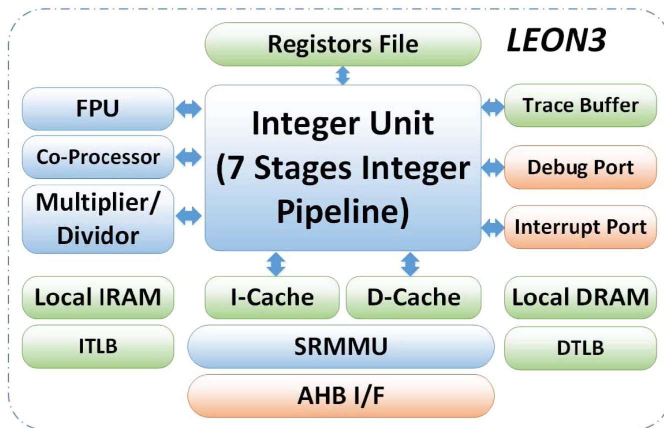

Fig. 7. Block diagram of IU of LEON3 core [37].

This code is assembled and executed on a bare metal LEON3 processor. We only consider those scenarios where the *call* instruction is followed by a corresponding *return* instruction. Due to this constraint, if a return address is overwritten, then the callee will not be able to return to the caller. When a callee is invoked using the *call* instruction by a caller in LEON3, the return address of the caller is saved in the *i7* register of callee's register window (we consider the default setting of eight register windows from *w0-w7*).

In the examined subroutine, the vulnerable instruction is *call 0 × 206b4 ⟨strcpy@plt⟩*, which corresponds to the *strcpy()* function of the C library. This function is used to copy input to a buffer. When the input is longer than the size of the stack allocated buffer, the space reserved for a register window on the stack is overwritten. Upon returning from the function, if this portion of memory is loaded into the register file, the return address is corrupted.

To detect such a vulnerability, we first measure the time required for the normal execution of the *vulnerable_function*. After executing the *retl* instruction, this subroutine returns to the *main* function of the program. During the execution of the subroutine, we continuously monitor the register where the return instruction is stored. If an attempt is made to overwrite the register, we detect it and report it.

In our experiment, we consider the return address is stored in the *i7* register of the *w7* register window. The corresponding address of the register *i7* in *w7* is *"01111111."* The write address signal, *rfi.waddr*, of the IU of the processor is used for writing the value of the return address into the *i7* register when the write enable signal, *rfi.wren*, is *"1."* Based on this, the informal security specification can be stated as *rfi.wren* and `rfi.waddr` signals should not be equal to "1" and "0111 1111," respectively, at the same time after the caller saves the return address. That is, the register *i7* containing the value of the return address of the caller should not be overwritten at any clock cycle when the write enable signal *rfi.wren* is *"1."* This specification can be also expressed as

$$\forall t\ \exists t_0,\ t_n,\ t_i \in t : (t_0 < t_i < t_n)$$
$$\wedge (rfi.wren_{t_0} \rightarrow rfi.waddr_{t_0})$$
$$\wedge \neg\ (rfi.wren_{t_i} \rightarrow rfi.waddr_{t_i})$$

where t_0 is the time when the return address is written into the *i7* register, t_n is the time when the *return* instruction is executed (used for returning to caller), and all time between t_0 and t_n is given as t_i. The specification is stated in Coq starting from $t = 1$ in the following theorem.

```
Theorem IU_Cycle_1:
forall (t : nat),
t = 1 ->
ico.data_0 t = sethi_0_g0 ->
rstn t = hi::nil ->
holdn t = lo::nil ->
irqi.run t = hi::nil->
irqi.rst t = hi::nil->
(bv_eq (rfi.wren t) (hi::nil)=lo\/
 bv_eq (rfi.waddr t) (lo::hi::hi::hi::hi::hi::hi::hi::nil)=lo).
```

The symbols *hi* and *lo* represent the high voltage and the low voltage in the circuit, respectively. The function *bv_eq* compares two binary codes and returns the result *lo* when there is a match between the codes and *hi*, otherwise. Similarly, we have written theorems for anytime between t_0 and t_n. Note that the time increases at steps corresponding to the clock cycles of the LEON3 processor.

In *ico.data_0 t = sethi_0_g0*, the *sethi* instruction and its operands are stored. Signals, *rstn*, *holdn*, and *irqi*, representing reset, hold, and interrupt, are not considered in our experiment.

As the VHDL code of the IU has a lot of *procedures* (shown in Fig. 8) and *functions*, we allocate their verification task to the Cadence IFV. We verify the procedure, *regaddr*, using the model checker for the corresponding informal specification—when the input signal *cwp* equals to "111," and *reg* equals to "01111," the output signal *rao* will be "01111111." An example specification (assertion) in the PSL language is shown

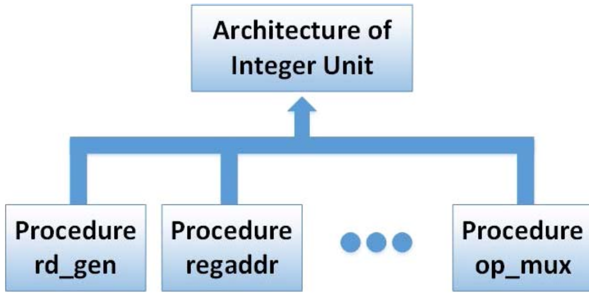

Fig. 8. Submodules in the IU of LEON3.

as follows:

$$\textbf{assert}(\{(cwp_ifv\,[1:3] = \text{“111”}) \wedge (\text{reg_ifv}\,[1:5] = \text{“01111”})\} \mapsto (\text{rao_ifv}\,[1:8] = \text{“01111111”})).$$

Here, the *cwp_ifv* register stores the value of the current window pointer *w7*, the *reg_ifv* register stores the address of the *i7* register, the *rao_ifv* register contains the address of the *i7* register of the *w7* register window, and the $\mapsto$ operator means that when the regular expression at the left-hand side holds, then the expression at the right-hand side also holds at the same clock cycle. The assertion states that when registers *cwp_ifv* and *reg_ifv* have the values of “111” and “01111,” respectively, the output register *rao_ifv* has the value “01111111.”

The above-mentioned PSL specification in the VHDL language is given as follows.

```
psl ASSERT_SubModule_regaddr:
assert
({(cwp_ifv(2 downto 0) = ``111'')  AND
  (reg_ifv(4 downto 0) = ``01111'')}|->
  (rao_ifv(7 downto 0) = ``01111111''));
```

We state the above-mentioned PSL specification as the following lemma.

```
Lemma Assert_SubModule_regaddr :
forall (t:nat),
       regaddr.cwp t = hi::hi::hi::nil ->
       regaddr.reg t = lo::hi::hi::hi::hi::nil ->
       regaddr.rao t= lo::hi::hi::hi::hi::hi::hi::hi::nil.
```

When the model checker proves that the code satisfies the specification, we can be assured that the proof of the lemma exists. By combining the proofs of all the lemmas, we were able to prove the theorem, *IU_Cycle_1*. Following this procedure, we were able to reduce the effort required for proving the security theorem in Coq. For instance, the Cadence IFV took only 0.03 s of CPU time for verifying the *Procedure regaddr* against the *ASSERT_SubModule_regaddr* specification. For the IU, there are 34 procedures and 22 functions in the source HDL codes. Hence, 56 submodules are obtained from decomposition. Considering that scale of each submodule is similar, an approximate runtime of the model checking procedure is 1.68 s. If the *Procedure regaddr* or other submodules were verified by an interactive theorem prover, such as Coq, it will take more time to complete the verification process.

Meanwhile, at the theorem prover side, we have applied the developed code conversion tool on the top module of the IU design excluding submodules (note that the equivalence checking is performed on VHDL code directly so no code conversion is required for submodules). The code conversion process was carried out on the same desktop as used in Section V. In total, the time consumed in conversion is 3.982 s.

VII. Conclusion and Discussion

In this paper, an automatic integrated formal verification framework is proposed to protect a large-scale SoC design from malicious attacks. Given that an interactive theorem prover (e.g., Coq) requires significant effort to manually verify the design and that a model checker suffers from scalablity issues, we combine these two techniques together through the decomposition of the security property as well as the design in such a way that the model checker can verify those submodules, which have much less state variables. Meanwhile, a VHDL-to-Coq code converter is developed to automate the code conversion process in the PCH framework. Two important steps are involved in construction of this tool: 1) translation of VHDL program to IR and 2) conversion of IR to *Formal-HDL*. Consequently, we automated the procedure for translating the design from HDL to *Gallina* and reduced the amount of effort required for proving the security theorem in Coq.

In the future, we plan to use our approach for detecting sophisticated hardware Trojans with the assistance of automatic tool chains. We will deliver open-source, prototype versions of the formal verification software package to the hardware security community and the formal verification community. Specifically, the generation of proofs and the library of security properties will be performed in the future. Furthermore, for decomposition, work such as detection sensitivity will be carried out.

References

[1] M. Banga and M. S. Hsiao, “Trusted RTL: Trojan detection methodology in pre-silicon designs,” in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2010, pp. 56–59.

[2] Y. Jin and Y. Makris, “Hardware Trojan detection using path delay fingerprint,” in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jun. 2008, pp. 51–57.

[3] A. Waksman, M. Suozzo, and S. Sethumadhavan, “FANCI: Identification of stealthy malicious logic using Boolean functional analysis,” in *Proc. CCS*, 2013, pp. 697–708.

[4] X. Zhang and M. Tehranipoor, “Case study: Detecting hardware trojans in third-party digital IP cores,” in *Proc. HOST*, 2011, pp. 67–70.

[5] E. Love, Y. Jin, and Y. Makris, “Proof-carrying hardware intellectual property: A pathway to trusted module acquisition,” *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 1, pp. 25–40, Feb. 2012.

[6] Y. Jin, B. Yang, and Y. Makris, “Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing,” in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2013, pp. 99–106.

[7] Y. Jin, “Design-for-security vs. design-for-testability: A case study on DFT chain in cryptographic circuits,” in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2014, pp. 19–24.

[8] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang, “Backspace: Formal analysis for post-silicon debug,” in *Proc. Int. Conf. Formal Methods Comput.-Aided Design*, 2008, p. 5.

[9] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.
[10] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *Proc. Int. Conf. Field Program. Logic Appl.*, Aug. 2010, pp. 255–258.
[11] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proc. DAC*, New York, NY, USA, 2015, pp. 112-1–112-6.
[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Proc. Model Checking Softw.*, Portland, OR, USA, May 2003, pp. 235–239.
[13] J. A. Goguen and J. Meseguer, "Unwinding and inference control," in *Proc. IEEE Symp. Secur. Privacy*, Apr./May 1984, p. 75.
[14] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *Proc. IEEE Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2016, pp. 124–129.
[15] X. Guo, R. G. Dutta, and Y. Jin, "Automatic RTL-to-formal code converter for IP security formal verification," in *Proc. 17th Int. Workshop Microprocess. SOC Test Verification (MTV)*, 2016, pp. 35–38.
[16] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2013, pp. 824–829.
[17] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles Programm. Lang. (POPL)*, 1997, pp. 106–119.
[18] INRIA. (2010). *The Coq Proof Assistant*. [Online]. Available: http://coq.inria.fr/
[19] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo, "Incremental formal verification of hardware," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design (FMCAD)*, 2011, pp. 135–143.
[20] A. R. Bradley, "Sat-based model checking without unrolling," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*, 2011, pp. 70–87.
[21] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proc. 47th Design Autom. Conf.*, 2010, pp. 755–760.
[22] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for MPI programs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2010, pp. 1–10.
[23] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technol. J.*, vol. 3, no. 1, pp. 1–14, 1999.
[24] S. Berezin, "Model checking and theorem proving: A unified framework," Ph.D. dissertation, SRI Int., Menlo Park, CA, USA, 2002.
[25] P. Dybjer, Q. Haiyan, and M. Takeyama, "Verifying haskell programs by combining testing, model checking and interactive theorem proving," *Inf. Softw. Technol.*, vol. 46, no. 15, pp. 1011–1025, 2004.
[26] M.-M. Bidmeshki and Y. Makris, "VeriCoq: A verilog-to-Coq converter for proof-carrying hardware automation," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 29–32.
[27] C. Eisner and D. Fisman, in *Proc. Model Checking Softw.*, Portland, OR, USA, 2006.
[28] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. Model Checking Softw.*, Portland, OR, USA, 2000, pp. 154–169.
[30] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, pp. 117–148, Dec. 2003.
[31] A. Smrčka and T. Vojnar, "Verifying parametrised hardware designs via counter automata," in *Proc. Haifa Verification Conf.*, 2007, pp. 51–68.
[32] X. Guo, R. G. Dutta, and Y. Jin, "Hierarchy-preserving formal verification methods for pre-silicon security assurance," in *Proc. 16th Int. Workshop Microprocess. SOC Test Verification (MTV)*, 2015, pp. 48–53.
[33] OpenCores. *Opencores Projects*. Accessed: Sep. 19, 2017. [Online]. Available: http://www.opencores.org
[34] *AES Core Modules*. Accessed: Sep. 19, 2017. [Online]. Available: http://opencores.org/project,aes_128_192_256
[35] *Basic DES Crypto Core*. Accessed: Sep. 19, 2017. [Online]. Available: http://opencores.org/ project,basicdes
[36] RS232. *RS232/UART Interface*. Accessed: Sep. 19, 2017. [Online]. Available: http://opencores.org/project,rs232_interface.
[37] Gaisler Research. *LEON3 Synthesizable Processor*. Accessed: Sep. 19, 2017. [Online]. Available: http://www.gaisler.com

Xiaolong Guo (S'14) received the double bachelor's degrees from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, and the University of London, London, U.K., in 2010, and the M.S. degree from BUPT in 2013. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of Florida, Gainesville, FL, USA.

His current research interests include the design of scalable verification methods for hardware intellectual property protection, trusted system-on-chip verification, cyber security, formal methods, program synthesis, and secure language design.

Raj Gautam Dutta (S'17) received the B.Tech. degree in electronics and communication from Visvesvaraya Technological University, Belgaum, India, in 2007, and the M.S. degree in electrical engineering from the University of Central Florida, Orlando, FL, USA, in 2011, with an emphasis on control systems, where he is currently pursuing the Ph.D. degree with the Electrical Engineering and Computer Science (EECS) Department.

His current research interests include the development of security solutions for semiconductor soft intellectual property cores by using formal verification techniques, the design of attack detection and mitigation software for autonomous systems, and the synthesis of controllers for multiagent systems.

Prabhat Mishra (S'00–M'04–SM'08) received the Ph.D. degree in computer science and engineering from the University of California at Irvine, Irvine, CA, USA.

He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA. His current research interests include the design automation of embedded systems, energy-aware computing, hardware security and trust, system validation and verification, and postsilicon debug.

Dr. Mishra has been recognized by several awards, including the NSF CAREER Award, the IBM Faculty Award, three best paper awards, and the EDAA Outstanding Dissertation Award. He is an ACM Distinguished Scientist. He serves as the Deputy Editor-in-Chief of the *IET Computers Digital Techniques*. He also serves as an Associate Editor of the *ACM Transactions on Design Automation of Electronic Systems*, the IEEE TRANSACTIONS ON VLSI SYSTEMS, and the *Journal of Electronic Testing*.

Yier Jin (M'13) received the B.S. and M.S. degrees in electrical engineering from Zhejiang University, Hangzhou, China, in 2005 and 2007, respectively, and the Ph.D. degree in electrical engineering from Yale University, New Haven, CT, USA, in 2012.

He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA. He proposed various approaches in the area of hardware security, including the hardware Trojan detection methodology relying on local side-channel information, the postdeployment hardware trust assessment framework, and the proof-carrying hardware intellectual property (IP) protection scheme. His current research interests include the areas of trusted embedded systems, trusted hardware IP cores, hardware–software coprotection on computer systems, the security analysis on Internet of Things (IoT), and wearable devices with a particular emphasis on information integrity and privacy protection in the IoT era.

Dr. Jin received the DoE Early CAREER Award in 2016 and the best paper awards from DAC'15, ASP-DAC'16, and HOST'17.

PCH Framework for IP Runtime Security Verification

Xiaolong Guo*, Raj Gautam Dutta‡, Jiaji He†, and Yier Jin*
*Department of Electrical and Computer Engineering, University of Florida
†School of Microelectronics, Tianjin University
‡Department of Electrical and Computer Engineering, University of Central Florida
guoxiaolong@ufl.edu, rajgautamdutta@knights.ucf.edu, dochejj@tju.edu.cn, yier.jin@ece.ufl.edu

***Abstract*—Untrusted third-party vendors and manufacturers have raised security concerns in hardware supply chain. Among all existing solutions, formal verification methods provide powerful solutions in detection malicious behaviors at the pre-silicon stage. However, little work have been done towards built-in hardware runtime verification at the post-silicon stage. In this paper, a runtime formal verification framework is proposed to evaluate the trust of hardware during its execution. This framework combines the symbolic execution and SAT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using an SoC design with untrusted IPs. The experimentation results show that the proposed approach can provide high-level security assurance for hardware at runtime.**

I. INTRODUCTION

The changing landscape of the semiconductor industry has increased the demand for third-party intellectual property (IP) cores. Various factors such as reduced time to market (TTM) and lower design cost have led to the proliferation of the IP market. Another contributor to this growth is the use of System-on-Chip (SoC) platforms for mobile applications. SoC is a monolithic chip containing all the essential components for mimicking the functionality of a computing system. It is designed by integrating multiple IP cores from trusted and untrusted third party vendors.

Increasing number of third-party vendors have raised security concerns in the IC industry. Due to the extremely high cost of building foundries, chip manufacturing is usually outsourced to existing foundries. Consequently, security researchers in their respective domains have started putting in considerable effort to ensure trustworthiness of third-party resources. In the hardware security industry, multiple countermeasures have been developed to protect SoC at pre- and post-silicon stages [1]–[12]. However, these methods are designed to protect the hardware only in certain scenarios [13]. A comprehensive approach is required to protect against attacks from untrusted vendors and manufacturers.

Formal methods have shown their importance in exhaustive hardware security verification [3]–[6], but few of them were designed for securing post-fabrication designs. For example, in [4]–[6], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [14], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. However, model formalization and interactive proof procedures in PCH limit the scenario into static verification for design stage in the supply chain. Recently, Verifiable ASICs was proposed in [15] where an encryption protocol based approach was used against hardware Trojans in circuit manufacturing. However, their goal of ensuring correctness of computation incurred high costs in terms of complexity and overhead in hardware.

In this paper, we address the runtime hardware security verification challenge by extending our PCH framework from static to dynamic (aka runtime) with Satisfiability (SAT) solvers and symbolic executions.

SAT solvers have been used in many electronic design automation fields like logic synthesis, verification, and testing [16]–[18]. The SAT solvers are originally designed to solve the well-known Boolean Satisfiability problem, which decides whether a propositional logic formula can be satisfied given value assignments of the variables in the formula [19]. However, due to the high computational complexity, SAT solvers are not scalable to large designs.

Symbolic execution is a program analysis technique that can explore multiple paths that a program could take under different inputs [20]. Integrating these two techniques overcome the NP-Hard computation complexity issue in SAT solver and it provides a comprehensive protection by automatically checking the customized properties.

The main contributions of this paper are as follows.

- We combine the SAT solver with a static program analysis method for runtime checking of security of hardware. It is the first attempt to verify security properties on runtime hardware by combining these techniques.
- We describe the method for decomposing the hardware execution and the security specification into execution paths and sub-specifications respectively. These execution paths and sub-specifications are verified using a SAT solver.
- The work improves the study of hardware runtime verification. Our enhanced PCH framework provide comprehensive protection of hardware by complying to user specified security properties.

The rest of the paper is organized as follows: In Section II, we discuss previous work on malicious logic detection using formal techniques. In Section III, we introduce the threat model and provide some relevant background on SAT solver and static program analysis. We explain our integrated framework, formalization, and decomposition of the hardware and elaborate on the verification procedure in section IV.

978-1-5090-6008-5/17/$31.00 © 2017 IEEE

Section V presents demonstrations of our approach and final conclusions are drawn in Section VI.

II. Related Work

Formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [3]–[9], [21]. In [3], a multi-stage approach was adopted for identifying suspicious signals using assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and sequential Automatic Test Pattern Generation. The PCH framework has been effective in ensuring trustworthiness of soft IP cores [4]–[6], [8], [9]. This method was inspired from the proof-carrying code (PCC) for software assurance [22]. Drzevitzky et al. proposed the first PCH framework for dynamically reconfigurable hardware platforms [9]. They used runtime combinational equivalence checkingto verify the equivalence between the design specification and the design implementation. However, instead of using security properties, the approach verified safety policies on the design. Another PCH framework was proposed for security property verification on soft-IP cores [4]–[6], [8], [23], [24]. In this framework, the Coq proof assistant [14] was used to represent security properties, hardware designs, and formal proofs. However, this framework can only provides static verification on design stage of hardware other than the runtime of hardware.

Recently, Verifiable ASICs was proposed by Wahby et.al. [15] to verify the correctness of functionality of hardware system. In their paper, runtime (or dynamic) verification was performed by implementing an interactive encryption protocol between untrusted ICs and a second trusted ICs, where the untrusted ICs was called *Prover* and trusted ICs was called *Verifier*. It was the first attempt to compute proofs of correct execution through utilizing verifiable computation. However, for security purpose, their correctness checking method would result in high computational cost and overhead. Furthermore, their method was designed for checking specific property rather than the entire set of functional properties. In our paper, we follow the *Prover-Verifier* architecture to build the framework.

Meanwhile, many runtime hardware approaches were developed for information flow security, which could guarantee that all information flows satisfy the given security policy. For instance, GLIFT was proposed in [25] and could dynamically detect malicious logic through tracking the information flow in the hardware at runtime. Moreover, there are hardware description languages that enforce security policies by adding logic of information flow control in the hardware. Languages such as Caisson [26], Sapper [27] and SecVerilog [13] belong to this category. However, these information flow control based techniques could provide protections which only against information leakage. Our proposed runtime PCH framework can against any user specified security properties by verifying hardware designs.

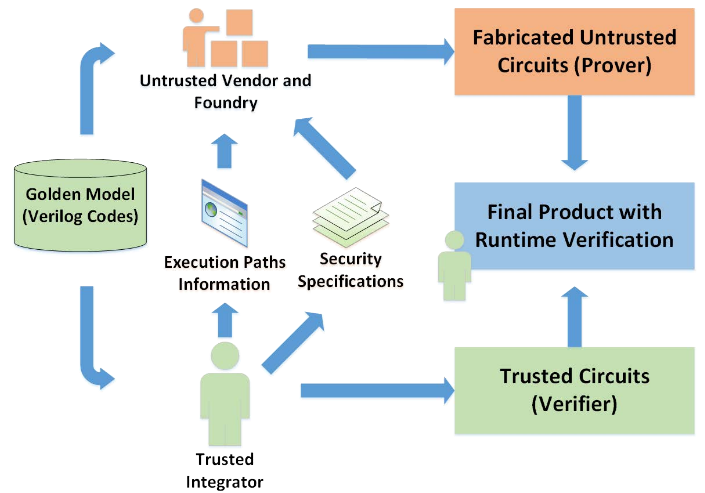

Figure 1: Working procedure of runtime PCH framework

III. Background

A. Attack Model

Hardware Trojans/Malicious logic can be inserted by adversaries at the different stages of the IC life-cycle. We assume that the rogue agents at the third-party IP design house and foundry can insert a hardware Trojan or backdoor to the fabricated circuit. Such a Trojan can be triggered either by a counter at a predetermined time, by an input vector, or under certain physical conditions. Upon activation it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware.

B. Execution Paths

Symbolic execution is a popular static program analysis technique that checks whether a software program satisfies specified properties. In this method, execution paths that the program should take are explored systematically to avoid the space explosion problem. Specifically, inputs are represented as symbols and the solvers are used to check whether there are counter examples of the property. For each path, a Boolean formula is derived to describe the conditions of the branches, while a symbolic memory is used to map variables to symbolic expressions. The Boolean formula is updated after executing the branch and the symbolic memory is updated after each assignment. In [20], a survey is provided to show many early works on symbolic execution and its applications on software testing and security.

As to the execution paths of the hardware, a path dependency graph of Verilog-HDL (Verilog) programs was proposed in [28], which could be applied for the static Verilog program analysis. In our work, after obtaining the execution paths from the golden model, the hardware designed is decomposed into segments based on these paths. Then, the foundry manufactures the ICs depending on these segments. Accordingly, the mentioned Boolean formula will be maintained for each segment and implemented in hardware in the form of the look-up table (LUT).

C. Hardware Based SAT Solver

The SAT solvers are used in a wide range of applications such as model checking of hardware and software, circuit synthesis, and testing [19]. However, due to the excessive computation time, SAT solvers are often impractical for emerging applications [29]. Early works on hardware accelerated SAT solvers was surveyed in [30]. Solvers based on FPGA [29] and GPU [31] were discussed, and all of these hardware accelerated solvers had relative limitations. As we perform decomposition of the design, the scalability of SAT solver is not an issue.

In hardware verification using SAT solvers, a circuit is first represented in conjunctive normal form (CNF). A CNF is a conjunction of many clauses and each clause is a disjunction of literals which include variables and their logical negations [32]. SAT solver is designed to figure out the satisfaction of the given CNF, which means that all clauses must get the value *True*. Furthermore, there is at least one literal that gets *True* for each clause. For most of the modern SAT solvers, Davis-Putnam-Logemann-Loveland (DPLL), proposed in [33], is applied as the kernel algorithm. In DPLL, a depth-first search (DFS) is carried out to traverse all possible variables/literals assignments as shown in Algorithm 1. In this paper, a DPLL based SAT solver is implemented in hardware in the proposed runtime PCH framework.

Algorithm 1 DPLL Algorithm

Input:
1: F ▷ A CNF formula.
Output: $Result$ ▷ A Boolean value where $True$ stands for satisfaction and $False$ stands for not-satisfaction.
2: Preprocess F;
3: **if** $F == False$ **then**
4: $\quad Result \leftarrow False$; return;
5: **end if**
6: Find the next unassigned variable, assign the value;
7: Deduce based on the assignment;
8: **if** $F == False$ **then**
9: $\quad Result \leftarrow False$; return;
10: **end if**
11: **if** The conflict happened in derivation **then**
12: Analyze the conflict
13: **if** F can be looked back upon **then**
14: look back upon
15: **else**
16: $Result \leftarrow False$; return;
17: **end if**
18: **else**
19: return to line 6.
20: **end if**

IV. Runtime Proof-Carrying Hardware

This section introduces the runtime PCH framework and provides details of the design. A *Prover* is fabricated based on the execution paths, while a DPLL SAT solver is implemented as *Verifier*. In this section, we also discuss the decomposition of security properties and distribution of proofs, which enable the SAT solver to be used for large scale application.

A. Framework Overview and Property Decomposition

A trusted circuit is designed and manufactured by a trusted foundry to verify the trustworthiness of the untrusted hardware in runtime. Similar to [15], in our proposed new PCH framework, the untrusted circuit from the third-party foundry is called *Prover* P, while the trusted circuit is called *Verifier* V as shown in Figure 1. If the verification of the security properties/theorems is successful, it indicates that the *Prover* is trustworthy. Further, *Verifier* can get all the information from *Prover*. In the case where the verification fails, the *Verifier* can disable the *Prover* at anytime.

The working procedure of our proposed framework is shown in Figure. 1, and there are mainly two entities - untrusted foundry and trusted integrator interacting in the framework. Similar to the setting in [15], the untrusted foundry gets requirements of ASICs from consumer, and then fabricates the chips as part of *Prover* depending on the functionality specifications, which is golden model in Figure 1. The other part of *Prover* is from security specifications, which will be introduced in details later in this section. Accordingly, the trusted integrator, on the side of consumer, designs an extra trusted circuit *Verifier* that can provide verification of *Prover* on runtime and then combine *Verifier* and *Prover* together to produce the runtime verification system *S*. The composition of the final system S can be presented as Equation (1).

$$S := P \wedge V \tag{1}$$

Further, the trusted integrator explores execution paths from static program analysis of the functional golden model written by hardware description language (HDL) like Verilog. In the untrusted foundry side, each execution path will be manufactured individually, and we call them individual circuit *segment*, marked as seg. So we define the functionality of circuits inside the P as F and then F is composed of many seg as shown in Equation (2), where $k \in \mathbb{Z}$ is the total number of segments.

$$F := seg_1 \wedge seg_2 \wedge \cdots \wedge seg_k \tag{2}$$

Correspondingly, security property, defined as $Prop$, would be given by the integrator and then decomposed into sub security properties, defined as $lemma$. In *Verifier* side, satisfaction of each sub property $lemma$ will be verified for the corresponding segment seg as shown in Figure 2. So the system level security property $Prop$ is constructed as Equation (3).

$$Prop := lemma_1 \wedge lemma_2 \wedge \cdots \wedge lemma_k \tag{3}$$

B. Distributed Proof-Carrying in Runtime

Along with the F, untrusted foundry requires to give proof to satisfy $lemma$ for each seg, and the proof is given in form of CNF, defined as cnf_{seg} in Equation (4) where $n \in \mathbb{Z}$ stands for index number of a list, $Tseitin$ is a transformation that converts boolean circuits to CNF [34].

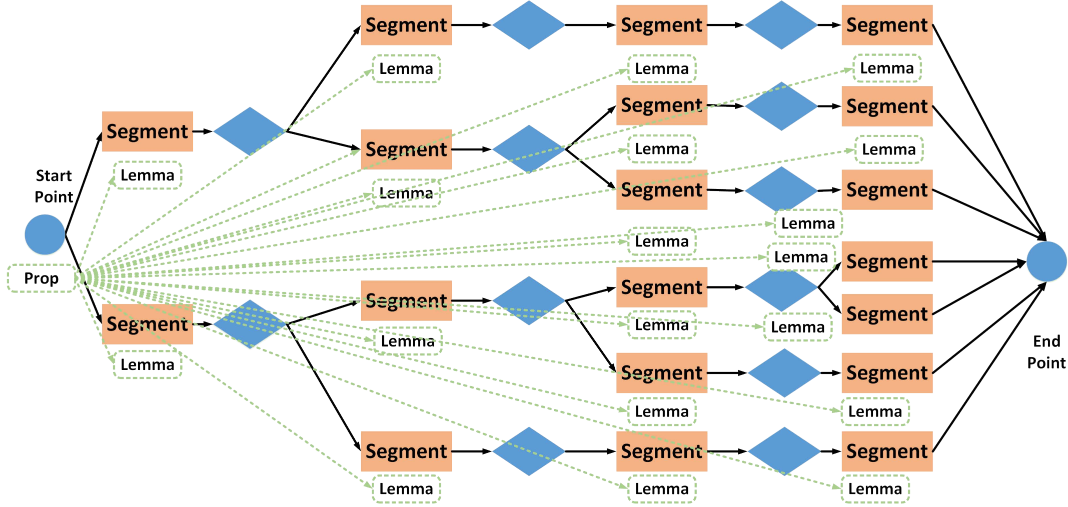

Figure 2: Circuit segments and property decomposition

$$seg_n \overset{Tseitin}{\longrightarrow} cnf_{segn} \tag{4}$$

Meanwhile, $lemma$ need to be parsed to a hardware expression $lemma_{expr}$ that can be represented by using HDL. In our proposed framework, parsing is made manually in the foundry side. After that, a $Tseitin$ transformation is utilized to convert the $lemma_{expr}$ to a CNF, noted as cnf_{la}. The procedure is presented in Equation (5).

$$lemma_n \overset{parse}{\longrightarrow} lemma_{exprn} \overset{Tseitin}{\longrightarrow} cnf_{lan} \tag{5}$$

Therefore, proof of sub property for segment is defined as a conjunction of cnf_{seg} and cnf_{lan} as shown in Equation (6). Furthermore, the entire proof in system level, noted as CNF, is composed of all the distributed cnf_n as discribed in Equation (7).

$$cnf_n := cnf_{seg} \wedge cnf_{lan} \tag{6}$$

$$CNF := cnf_1 \wedge cnf_2 \wedge \cdots \wedge cnf_k \tag{7}$$

Finally, in the following Equation (8), *Prover* is constructed from functionality part F and proof part CNF. In the runtime verification process, cnf_n would be put into the DPLL SAT sovler and verified individually. The verification details will be discussed in the following part.

$$P := F \wedge CNF \tag{8}$$

C. Design of Verifier and Runtime Verification Process

Except the segment and cnf block, the rest part of Figure 3 depicts the design of the *Verifier* which comprises a LUT and a DPLL SAT solver. The LUT in the proposed framework records information that whether the segment has been verified or not. The LUT includes two columns, where the first column contains a segment list and the second column has a binary value for each segment i.e. 1 stands for verified, 0 stands for not verified. Before the execution of a segment, the corresponding value will be checked. If the segment has been verified, then the execution continues. Otherwise, the system will be stalled and the verification of the segment is performed first.

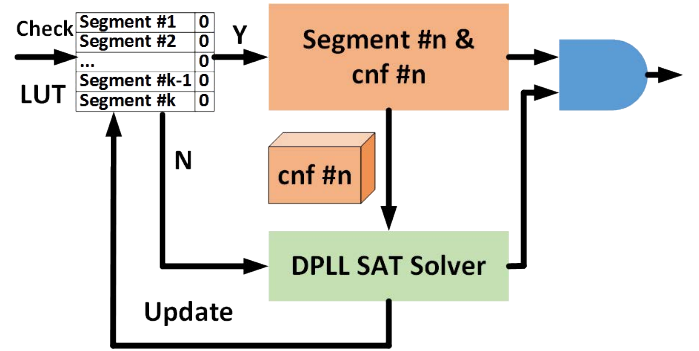

Figure 3: Structure of *Verifier*

A DPLL SAT solver is implemented based on Algorithm 1. In the verification, Proof cnf_n is delivered from *Prover* to the solver, and satisfaction of the input cnf_n will be checked. If satisfied, then the relevant value in LUT table will be updated as 1. If the given cnf_n is unsolved, then the *Verifier* will lock the segment by using an AND gate. From the system viewpoint, the above runtime verification process is represented in Algorithm 2.

V. Case Study

To demonstrate the effectiveness of the proposed runtime verification framework supported by the SAT solver, we utilize a FPGA platform implementing a RS232 program. Specifically, the RS232-T100, written in Verilog, is selected as the benchmark and obtained from [35]. The receiver side of this RS232, a micro-UART core, is considered for verification (see Figure 3). In order to prove the presence/absence of hardware Trojan, we will check the important signals like in/out interfaces.

In this experiment, we consider a hardware Trojan embedded in the benchmark RS232-T100, which manipulates output data to cause the Denial-of-Service (DoS) attack. Trigger of

Algorithm 2 Runtime Verification Process

Input:
1: P ▷ Prover
2: V ▷ Verifier
Output: $null$
3: $list_{next}, list_{cnf}$;
4: $ExePaths \leftarrow P$ ▷ Get all the execution paths
5: $SAT() \leftarrow V$ ▷ Get the SAT solver
6: $checkTable() \leftarrow V$ ▷ Get the look-up table
7: $list_{next} \leftarrow checkPath(ExePaths)$; ▷ Get the next execution paths
8: $list_{cnf} \leftarrow checkTable(list_{next})$; ▷ Whether the next execution paths are verified
9: **if** $list_{cnf}$ == null **then** ▷ All next execution paths verified
10: Go to line 7;
11: **else**
12: For each cnf in $list_{cnf}$:
13: $SAT(cnf)$;
14: **if** All $list_{cnf}$ have solutions **then**
15: Go to line 7;
16: **else**
17: Lock the circuit, return;
18: **end if**
19: **end if**

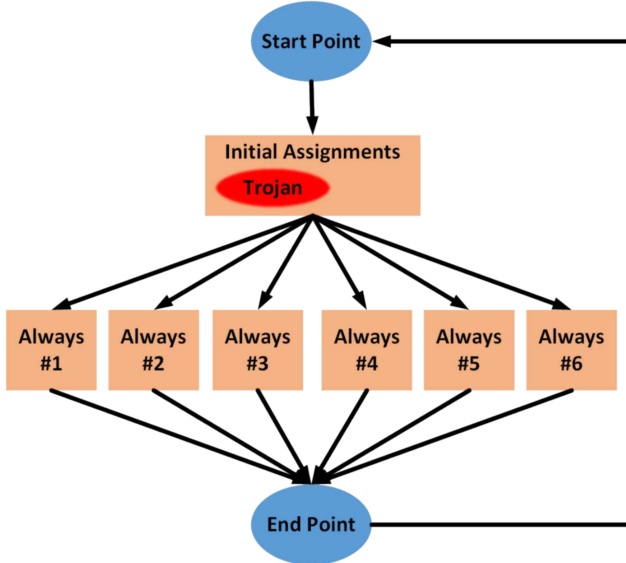

Figure 4: Decomposition of receiver part in RS232-T100

this Trojan is detecting specific values among the control signals $state$, $bitCell_cntrH$, $recd_bitCntrH$ and output signal rec_dataH in the receiver part of the micro-UART core. Once the Trojan is triggered, the payload of this Trojan can stuck the output signals rec_dataH and rec_readyH as zeros.

To detect such a DoS vulnerability, we observe the output signal in consecutive time. A heuristic property is that the output data should not always be 0 during data transmission. In our case study, we decompose and formalize the above heuristic property into $lemmas$ depending on the segments of

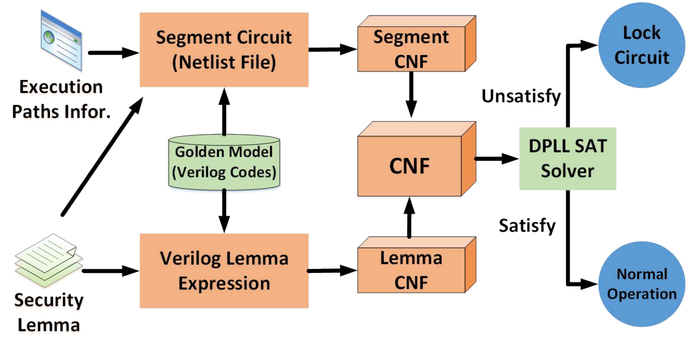

Figure 5: Working procedure of verification of a segment

the benchmark. Figure 4 indicates the execution paths of the receiver in micro-UART core and segments were set based on the paths. Similar to the Verilog program control flow graph in [28], the circuit is divided into an initial assignment part followed by parallel always parts as shown in Figure 4. Additionally, for RS232-T100 benchmark, Trojan is embedded into the initial assignment part.

Then, depending on the working procedure in Figure 5, verification is carried out on the segment which is going to be executed. In the FPGA platform, gate level netlist file is used to simulate the ASICs. The netlist file of each segment is designed and synthesized by considering both golden model and security $lemma$. Proof of the segment is in the form of CNF, and part of the CNF is from the the netlist file. Meanwhile, security $lemma$ is translated to the expressions written in Verilog. After that the expressions are converted to the rest of the CNF. Finally, a DPLL SAT solver, implemented in FPGA, is applied to check whether the solution of the input CNF exists. If the CNF is satisfied, then SAT solver return a signal to continue the execution or the system will stall.

In our case, an example security $lemma$ for the initial assignments segment in Figure 4 is formalized below:

$$\forall t \ \nexists t_0, t_n \in t : (t_0 < t_n) \ \wedge (t_n - t_0 > V_{th}) \wedge$$
$$(state_{t_0 \to t_n} = V_{wait}) \wedge (rec_dataH_{t_0 \to t_n} = 0x00)$$

Here, t is the time parameter, $state$ means the current state of the RS232 system. rec_dataH is the output port with 8 bits length of the receiver part. Also, $V_{th \in \mathbb{Z}}$ is the threshold that we set for the time interval. V_{wait} is a specific binary vector with value is $3'b011$ which implies that the system is waiting for sampling in data transmission. The $lemma$ states that if output port generates zero values in too long consecutive time during data transmission, then there is a high risk of under DOS attack.

When the SAT solver gets a solution from the given CNF, we can be assured that the proof of the $lemma$ exist. Otherwise, system will be locked for protection. In the above case, the SAT solver we developed took 4668406745 clock cycles or $9sec$ ($2ns$ per clock cycles based on our configuration) for returning an unsatisfaction conclusion for the proof/CNF of initial assignments segment, which indeed contains the Trojan. Meanwhile, the SAT solver took 7873 clock cycles or $15ms$ for returning a satisfaction conclusion for the same segment

without Trojan.

VI. Conclusion

In this paper, we give a solution for hardware runtime formal verification of security properties. The proposed runtime PCH framework integrates a static program analysis method and a SAT solver, and provides a high-level protection by verifying security properties defined by users. Furthermore, decomposition of property and distributed proofs of segments significantly reduces the computation complexity undertaken by SAT solver. The proposed method was demonstrated using FPGA and evaluated by verifying a RS232 benchmark with an embedded Trojan. Consequently, the proposed approach guarantees the security of hardware in runtime.

In future, we plan to use our approach for protecting large scale designs such as processor. Also, automated tool will be developed such as for auto generation of CNF. Furthermore, more sophisticated Trojans or attacks will be tested in different benchmarks using our runtime verification framework.

Acknowledgement

This work was partially supported by National Science Foundation (CNS-1319105), Army Research Office (W911NF-17-1-0477), Cisco and National Natural Science Foundation of China (61376032).

References

[1] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.

[2] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, 2013, pp. 697–708.

[3] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, 2011, pp. 67–70.

[4] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[5] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 99–106.

[6] Y. Jin, "Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2014.

[7] F. M. De Paula, M. Gort, A. J. Hu, S. J. Wilton, and J. Yang, "Backspace: formal analysis for post-silicon debug," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008, p. 5.

[8] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15, 2015, pp. 145:1–145:6.

[9] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 255–258.

[10] S. Wei, S. Meguerdichian, and M. Potkonjak, "Malicious circuitry detection using thermal conditioning," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1136–1145, 2011.

[11] R. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for hardware Trojan detection," in *Cryptographic Hardware and Embedded Systems*, ser. Lecture Notes in Computer Science, vol. 5747, 2009, pp. 396–410.

[12] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 49–63.

[13] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 503–516.

[14] INRIA, "The coq proof assistant," 2010, http://coq.inria.fr/.

[15] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable asics," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 759–778.

[16] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[17] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.

[18] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[19] J. Marques-Silva, "Practical applications of boolean satisfiability," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*. IEEE, 2008, pp. 74–80.

[20] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[21] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," ser. DAC '15, New York, NY, USA, 2015, pp. 112:1–112:6.

[22] G. C. Necula, "Proof-carrying code," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 106–119.

[23] X. Guo, R. G. Dutta, and Y. Jin, "Eliminating the hardware-software boundary: A proof-carrying approach for trust evaluation on computer systems," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 12, no. 2, pp. 405–417, 2017.

[24] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *IEEE Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 124–129.

[25] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 109–120.

[26] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[27] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 97–112.

[28] M. Zaki, Y. Mokhtari, and S. Tahar, "A path dependency graph for verilog program analysis," in *Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal*, 2003, pp. 109–112.

[29] J. Thong and N. Nicolici, "Sat solving using fpga-based heterogeneous computing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 232–239.

[30] A. A. Sohanghpurwala, M. W. Hassan, and P. Athanas, "Hardware accelerated sat solvers—a survey," *Journal of Parallel and Distributed Computing*, vol. 106, pp. 170–184, 2017.

[31] A. Dal Palu, A. Dovier, A. Formisano, and E. Pontelli, "Cud@ sat: Sat solving on gpus," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 27, no. 3, pp. 293–316, 2015.

[32] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[33] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[34] G. Tseitin, "On the complexity ofderivation in propositional calculus," *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[35] Trust-Hub, "Benchmarks," [Online]. https://www.trust-hub.org/benchmarks.php.

Runtime SoC Trust Verification using Integrated Symbolic Execution and Solver

Xiaolong Guo, Jiaji He, and Yier Jin
Department of Electrical and Computer Engineering, University of Florida
yier.jin@ece.ufl.edu

Abstract*—Untrusted third-party vendors and manufacturers have raised security concerns in hardware supply chain. Among all existing solutions, formal verification methods provide powerful solutions in detection malicious behaviors at the pre-silicon stage. However, little work have been done towards built-in hardware runtime verification at the post-silicon stage. In this paper, a runtime formal verification framework is proposed to evaluate the trust of hardware during its execution. This framework combines the symbolic execution and SMT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using an SoC design with untrusted IPs. The experimentation results show that the proposed approach can provide high-level security assurance for hardware at runtime.

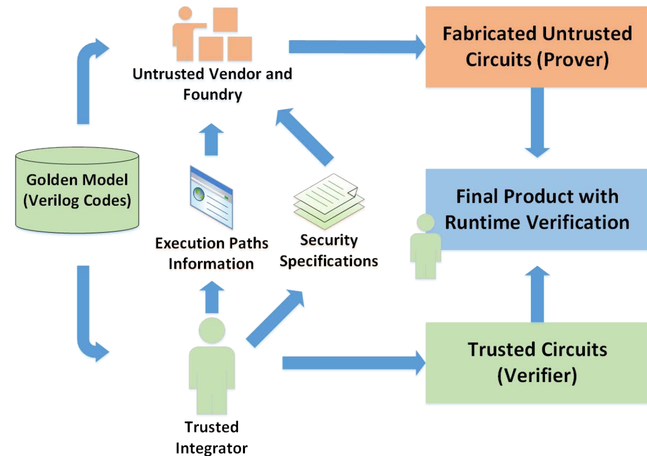

Fig. 1: Working procedure of runtime PCH framework

I. Introduction

The changing landscape of the semiconductor industry has increased the demand for third-party intellectual property (IP) cores. Various factors such as reduced time to market (TTM) and lower design cost have led to the proliferation of the IP market. Another contributor to this growth is the use of System-on-Chip (SoC) platforms for mobile applications. SoC is a monolithic chip containing all the essential components for mimicking the functionality of a computing system. It is designed by integrating multiple IP cores from trusted and untrusted third party vendors.

Increasing number of third-party vendors have raised security concerns in the IC industry. Due to the extremely high cost of building foundries, chip manufacturing is usually outsourced to existing foundries. Therefore, a comprehensive approach is required to protect against attacks from untrusted vendors and manufacturers. Formal methods have shown their importance in exhaustive hardware security verification [1]–[4], but few of them were designed for securing post-fabrication designs. For example, in [2]–[4], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [5], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. However, model formalization and interactive proof procedures in PCH limit the scenario into static verification for design stage in the supply chain.

In this paper, we address the runtime hardware security verification challenge by extending our Proof-Carrying Hardware (PCH) [3], [4], [6] framework from static to dynamic (aka runtime) with a SMT solver and symbolic executions. The working procedure of the developed runtime PCH framework is shown in Figure 1. The main contributions of this paper are as follows.

- We combine a SMT solver with a static program analysis method for runtime checking of security of hardware.
- The work improves the study of hardware runtime verification. Our enhanced PCH framework provides comprehensive protection of hardware by complying to user specified security properties.

II. Background and Related Work

A. Attack Model

Hardware Trojans/Malicious logic can be inserted by adversaries at the different stages of the IC life-cycle. We assume that the rogue agents at the third-party IP design house and foundry can insert a hardware Trojan or backdoor to the fabricated circuit. Such a Trojan can be triggered either by a counter at a predetermined time, by an input vector, or under certain physical conditions. Upon activation it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware.

B. Related Work

In [3], [4], [6], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [5], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. However, this framework can only provides static verification on design stage of hardware other than the runtime of hardware. In [7], a SAT solver is utilized

to enhance the PCH to be applicable in runtime scenario. Still, expressiveness of the SAT solver is not powerful enough so that security properties are difficult to be formalized in such framework.

Verifiable ASICs was proposed by Wahby et.al. [8] to verify the correctness of functionality of hardware system. In their paper, runtime (or dynamic) verification was performed by implementing an interactive encryption protocol between untrusted ICs and a second trusted ICs, where the untrusted ICs was called *Prover* and trusted ICs was called *Verifier*. It was the first attempt to compute proofs of correct execution through utilizing verifiable computation. However, for security purpose, their correctness checking method would result in high computational cost and overhead. Furthermore, their method was designed for checking specific property rather than the entire set of functional properties.

C. Background

Satisfiability (SAT) solvers have been used in many electronic design automation fields like logic synthesis, verification, and testing. The SAT solvers are originally designed to solve the well-known Boolean Satisfiability problem, which decides whether a propositional logic formula can be satisfied given value assignments of the variables in the formula. Based on SAT solver, satisfiability modulo theories (SMT) solver is derived by including serval first-order theories, such as arithmetic, bit-vectors, quantifiers, etc [9]. However, due to the high computational complexity, there is no hardware implementation for SMT solvers, and the software based SMT solver are not scalable to large designs.

Symbolic execution is a program analysis technique that can explore multiple paths that a program could take under different inputs [10]. In this method, execution paths that the program should take are explored systematically to avoid the space explosion problem. Specifically, inputs are represented as symbols and the solvers are used to check whether there are counter examples of the property. For each path, a Boolean formula is derived to describe the conditions of the branches, while a symbolic memory is used to map variables to symbolic expressions. The Boolean formula is updated after executing the branch and the symbolic memory is updated after each assignment. Integrating these two techniques overcome the NP-Hard computation complexity issue in SAT solver and it provides a comprehensive protection by automatically checking the customized properties.

III. Runtime Proof-Carrying Hardware

In this paper, we give a solution for hardware runtime formal verification of security properties. The proposed runtime PCH framework integrates a static program analysis method and a hardware based SMT solver, and provides a high-level protection by verifying security properties defined by users.

In detail, a trusted circuit is designed and manufactured by a trusted foundry to verify the trustworthiness of the untrusted hardware in runtime. Similar to [8], in our proposed new PCH framework, the untrusted circuit from the third-party foundry is called *Prover*, while the trusted circuit is called *Verifier* as shown in Figure 1. If the verification of the security properties/theorems is successful, it indicates that the *Prover* is trustworthy. Further, *Verifier* can get all the information from *Prover*. In the case where the verification fails, the *Verifier* can disable the *Prover* at anytime.

There are mainly two entities - untrusted foundry and trusted integrator interacting in the developed framework (see Figure 1). At first, the untrusted foundry gets requirements of ASICs from consumer, and then fabricates the chips as part of *Prover* depending on the functionality specifications, which is golden model in Figure 1. The other part of *Prover* produces a conjunctive normal form (CNF), which is a combination of proof and secure specifications. The CNF will be delivered from *Prover* to the solver, and satisfaction of the CNF will be checked. If satisfied, then the execution of circuit will be continue. If the given CNF is unsolved, then the *Verifier* will lock the circuit. Accordingly, the trusted integrator, on the side of consumer, designs an extra trusted circuit *Verifier* that can provide verification of *Prover* on runtime and then combine *Verifier* and *Prover* together to produce the runtime verification system *S*. The composition of the final system S can be presented as Equation (1).

$$S := P \wedge V \tag{1}$$

Further, the trusted integrator explores execution paths from static program analysis of the functional golden model written by hardware description language (HDL) like Verilog. In the untrusted foundry side, each execution path will be manufactured individually, and we call them individual circuit *segment*, marked as seg. So we define the functionality of circuits inside the P as F and then F is composed of many seg as shown in Equation (2), where $k \in \mathbb{Z}$ is the total number of segments.

$$F := seg_1 \wedge seg_2 \wedge \cdots \wedge seg_k \tag{2}$$

Correspondingly, security property, defined as $Prop$, would be given by the integrator and then decomposed into sub security properties, defined as $lemma$. In *Verifier* side, satisfaction of each sub property $lemma$ will be verified for the corresponding segment seg as shown in Figure 2. So the system level security property $Prop$ is constructed as Equation (3).

$$Prop := lemma_1 \wedge lemma_2 \wedge \cdots \wedge lemma_k \tag{3}$$

Along with the F, untrusted foundry requires to give proof to satisfy $lemma$ for each seg, and the proof is given in form of CNF, defined as cnf_{seg} in Equation (4) where $n \in \mathbb{Z}$ stands for index number of a list, $Tseitin$ is a transformation that converts boolean circuits to CNF [11].

$$seg_n \xrightarrow{Tseitin} cnf_{segn} \tag{4}$$

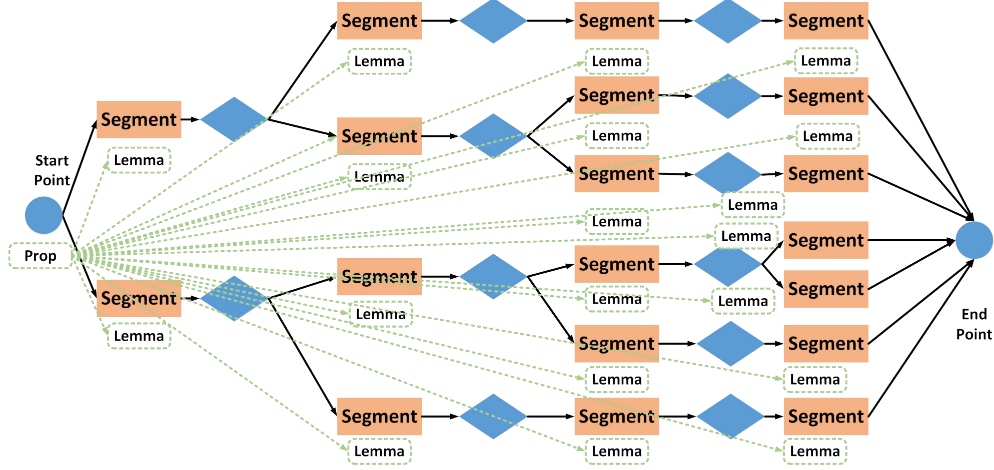

Fig. 2: Circuit segments and property decomposition

Meanwhile, $lemma$ need to be parsed to a hardware expression $lemma_{expr}$ that can be represented by using HDL. In our proposed framework, parsing is made manually in the foundry side. After that, a $Tseitin$ transformation is utilized to convert the $lemma_{expr}$ to a CNF, noted as cnf_{la}. The procedure is presented in Equation (5).

$$lemma_n \xrightarrow{parse} lemma_{exprn} \xrightarrow{Tseitin} cnf_{lan} \tag{5}$$

Therefore, proof of sub property for segment is defined as a conjunction of cnf_{seg} and cnf_{lan} as shown in Equation (6). Furthermore, the entire proof in system level, noted as CNF, is composed of all the distributed cnf_n as discribed in Equation (7).

$$cnf_n := cnf_{seg} \wedge cnf_{lan} \tag{6}$$

$$CNF := cnf_1 \wedge cnf_2 \wedge \cdots \wedge cnf_k \tag{7}$$

Finally, in the following Equation (8), *Prover* is constructed from functionality part F and proof part CNF. In the runtime verification process, cnf_n would be put into the DPLL SAT sovler and verified individually. The verification details will be discussed in the following part.

$$P := F \wedge CNF \tag{8}$$

Except the segment and cnf block, the rest part of Figure 3 depicts the design of the *Verifier* which comprises a LUT and a DPLL SAT solver. The LUT in the proposed framework records information that whether the segment has been verified or not. The LUT includes two columns, where the first column contains a segment list and the second column has a binary value for each segment i.e. 1 stands for verified, 0 stands for not verified. Before the execution of a segment, the corresponding value will be checked. If the segment has been verified, then the execution continues. Otherwise, the system will be stalled and the verification of the segment is performed first.

A DPLL SAT solver is implemented based on Algorithm 1. A typical existing SMT solver is constructed based on the SAT solver, which is shown in the Figure 4. Specifically, in the proposed framework, the SMT solver is developed to get the extra constrains from a high level, while the CNF is input to the SAT solver directly. In the verification, Proof cnf_n is delivered from *Prover* to the solver, and satisfaction of the input cnf_n will be checked. If satisfied, then the relevant value in LUT table will be updated as 1. If the given cnf_n is unsolved, then the *Verifier* will lock the segment by using an AND gate.

Algorithm 1 DPLL Algorithm

Input:
1: F ▷ A CNF formula.
Output: $Result$ ▷ A Boolean value where $True$ stands for satisfaction and $False$ stands for not-satisfaction.
2: Preprocess F;
3: **if** F == $False$ **then**
4: $Result \leftarrow False$; return;
5: **end if**
6: Find the next unassigned variable, assign the value;
7: Deduce based on the assignment;
8: **if** F == $False$ **then**
9: $Result \leftarrow False$; return;
10: **end if**
11: **if** The conflict happened in derivation **then**
12: Analyze the conflict
13: **if** F can be looked back upon **then**
14: look back upon
15: **else**
16: $Result \leftarrow False$; return;
17: **end if**
18: **else**
19: return to line 6.
20: **end if**

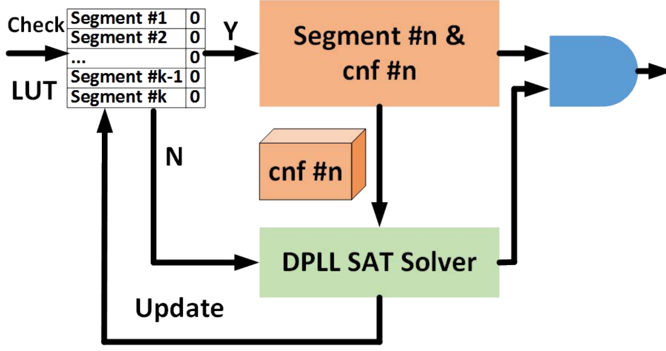

Fig. 3: Structure of *Verifier*

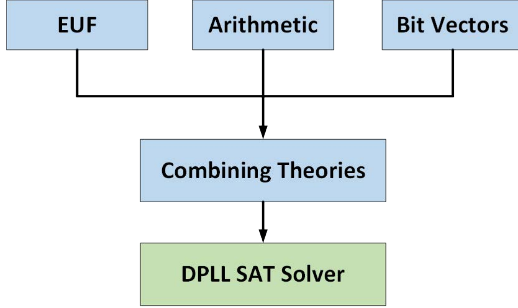

Fig. 4: Hardware based SMT solver structure

IV. Case Study

To demonstrate the effectiveness of the proposed runtime verification framework supported by the SMT solver, we utilize a FPGA platform implementing a RS232 program. Specifically, the RS232-T100, written in Verilog, is selected as the benchmark and obtained from [12]. The receiver side of this RS232, a micro-UART core, is considered for verification. In order to prove the presence/absence of hardware Trojan, we will check the important signals like in/out interfaces.

In this experiment, we consider a hardware Trojan embedded in the benchmark RS232-T100, which manipulates output data to cause the Denial-of-Service (DoS) attack. Trigger of this Trojan is detecting specific values among the control signals and output signal in the receiver part of the micro-UART core. Once the Trojan is triggered, the payload of this Trojan can stuck the output signals and as zeros.

In the above case, an example security property is formalized below:

$$\forall t \ \ \nexists t_0, t_n \in t : (t_0 < t_n) \ \wedge (t_n - t_0 > V_{th}) \wedge$$
$$(state_{t_0 \to t_n} = V_{wait}) \wedge (rec_dataH_{t_0 \to t_n} = 0x00)$$

Here, t is the time parameter, $state$ means the current state of the RS232 system. rec_dataH is the output port with 8 bits length of the receiver part. Also, $V_{th \in \mathbb{Z}}$ is the threshold that we set for the time interval. V_{wait} is a specific binary vector with value is $3'b011$ which implies that the system is waiting for sampling in data transmission. The $lemma$ states that if output port generates zero values in too long consecutive time during data transmission, then there is a high risk of under DOS attack.

As a result, the SAT solver (kernel of the proposed SMT solver) took 4668406745 clock cycles or $9sec$ ($2ns$ per clock cycles based on our configuration) for returning an unsatisfaction conclusion for the proof/CNF of initial assignments segment, which indeed contains the Trojan. Meanwhile, the SAT solver took 7873 clock cycles or $15ms$ for returning a satisfaction conclusion for the same segment without Trojan.

V. Conclusion

In this paper, we give a solution for hardware runtime formal verification of security properties. The proposed runtime PCH framework integrates a static program analysis method and a SMT solver, and provides a high-level protection by verifying security properties defined by users. The proposed method was demonstrated using FPGA and evaluated by verifying a RS232 benchmark with an embedded Trojan. Consequently, the proposed approach guarantees the security of hardware in runtime.

Acknowledgement

This work was partially supported by National Science Foundation (CNS-1812071), Army Research Office (W911NF-17-1-0477) and Cisco.

References

[1] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, 2011, pp. 67–70.

[2] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[3] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 99–106.

[4] Y. Jin, "Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2014, pp. 19–24.

[5] INRIA, "The coq proof assistant," 2010, http://coq.inria.fr/.

[6] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15, 2015, pp. 145:1–145:6.

[7] X. Guo, R. G. Dutta, J. He, and Y. Jin, "PCH framework for ip runtime security verification," in *Asian Hardware Oriented Security and Trust (AsianHOST)*, 2017, (to appear).

[8] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable asics," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 759–778.

[9] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[10] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[11] G. Tseitin, "On the complexity ofderivation in propositional calculus," *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[12] *https://www.trust-hub.org/*.