

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 suggestions 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 penalty 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) 22-08-2017		2. REPORT TYPE Final Report		3. DATES COVERED (From - To) 1-Apr-2016 - 2-Aug-2017	
4. TITLE AND SUBTITLE Final Report: Bridging the Hardware-Software Gap: A Proof-Carrying Approach for Computer Systems Trust Evaluation (5.3.5)			5a. CONTRACT NUMBER W911NF-16-1-0124		
			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 Central Florida 12201 Research Parkway, Suite 501 Orlando, FL 32826 -3246			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) 66670-CS.1		
12. DISTRIBUTION AVAILABILITY 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 be construed 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 UU	b. ABSTRACT UU	c. THIS PAGE UU			Yier Jin
					19b. TELEPHONE NUMBER 407-823-5321

RPPR Final Report

as of 17-Nov-2017

Agency Code:

Proposal Number: 66670CS

Agreement Number: W911NF-16-1-0124

INVESTIGATOR(S):

Name: Yier Jin

Email: yier.jin@eecs.ucf.edu

Phone Number: 4078235321

Principal: Y

Organization: **University of Central Florida**

Address: 12201 Research Parkway, Suite 501, Orlando, FL 328263246

Country: USA

DUNS Number: 150805653

EIN: 592924021

Report Date: 02-Nov-2017

Date Received: 22-Aug-2017

Final Report for Period Beginning 01-Apr-2016 and Ending 02-Aug-2017

Title: Bridging the Hardware-Software Gap: A Proof-Carrying Approach for Computer Systems Trust Evaluation (5.3.5)

Begin Performance Period: 01-Apr-2016

End Performance Period: 02-Aug-2017

Report Term: 0-Other

Submitted By: Yier Jin

Email: yier.jin@eecs.ucf.edu

Phone: (407) 823-5321

Distribution Statement: 1-Approved for public release; distribution is unlimited.

STEM Degrees: 0

STEM Participants: 0

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.

RPPR Final Report as of 17-Nov-2017

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 conducted research on computer system security assurance by bridging the gap between the hardware platform and the software programs running on top of the hardware platform. Various accomplishments have been made to achieve the final goals. Three conference papers, one book chapter and three journal publications are published. The details of research accomplishments are listed as follows.

1) The wide usage of hardware intellectual property (IP) cores from untrusted vendors has raised security concerns in the integrated circuit (IC) industry. Existing testing methods are designed to validate the functionality of the hardware IP cores. These methods often fall short in detecting unspecified (often malicious) logic. Formal methods like Proof-Carrying Hardware (PCH), on the other hand, can help eliminate hardware Trojans and/or design backdoors by formally proving security properties on soft IP cores despite the high proof development cost. One of the causes to the high cost is the manual conversion of the hardware design from RTL code to a domain-specific language prior to verification. To mitigate this issue and to lower the overall cost of PCH framework, we propose an automatic code converter for translating VHDL to Formal-HDL, a domain specific language for representing hardware designs in Coq language. Our code converter provides support to wide variety of hardware designs. Towards the goal of speeding up the verification procedure in our PCH framework, the code converter is the important first step. The applicability of the tool is demonstrated by converting soft IP cores of AES to its Coq equivalent code. A paper based on the work has been accepted and presented in the 17th International Workshop on Microprocessor and SOC Test and Verification (MTV 2016) which was held in Austin, TX in December 2016. A copy of this paper is also attached in this report.

2) Reusable hardware Intellectual Property (IP) based System-on-Chip (SoC) design has emerged as a pervasive design practice in the industry nowadays. The possibility of hardware Trojans and/or design backdoors hiding in the IP cores has raised security concerns. As existing functional testing methods fall short in detecting unspecified (often malicious) logic, formal methods have emerged as an alternative for validation of trustworthiness of IP cores. Toward this direction, we discuss theorem proving, a category of formal methods, used in hardware trust evaluation. Specifically, proof-carrying hardware (PCH) and its applications are introduced in detail, in which we demonstrate the use of theorem proving methods for providing high-level protection of IP cores. Further, we propose an automatic tool chains to automate the whole PCH framework. A paper based on the work has been accepted and presented in the 2017 Government Microcircuit Applications and Critical Technology Conference (GOMACTech-17) which was held in Reno, NV in March 2017. A copy of this paper is also attached in this report.

3) The existing approaches, designed to ensure the trustworthiness of either the hardware IP cores or to verify software programs, rarely secure the entire computer system. The semantic gap between the hardware and the software lends to the challenge of securing computer systems. We propose a new unified framework to represent both the hardware infrastructure and the software program in the same formal language. As a result, the semantic gap between the hardware and the software is bridged, enabling the development of system-level security properties for the entire computer system. Our unified framework uses a cross-domain formal verification method to protect the entire computer system within the scope of proof-carrying hardware. The working procedure of the unified framework is demonstrated with a sample embedded system which includes an 8051 microprocessor and an RC5 encryption program. In our demonstration, we show that the embedded system is trusted if the system level security properties are provable. Supported by the unified framework, the system designers/integrators will be able to formally verify the trustworthiness of the computer system integrated with hardware and software both from untrusted third-party vendors. A paper based on the work has been published in the IEEE Transactions on Information Forensics and Security (TIFS), the top-tier journal in the area of hardware and cyber security. A copy of this paper is also attached in this report.

RPPR Final Report

as of 17-Nov-2017

4) Proof-carrying hardware intellectual property (PCHIP) is a previously proposed framework for ensuring trustworthiness of third-party hardware IP through the development of formal proofs for security properties designed to prevent introduction of malicious behavior. Based on this framework, we introduce new approaches for assuring that the secrecy of internal information in a hardware design is not compromised by design flaws or malicious hardware Trojans. Specifically, we devise two PCHIP-based information flow tracking (IFT) approaches which enhance the formal PCHIP framework with secrecy tags and/or sensitivity levels in order to provide mechanisms for proving that sensitive information does not reach undesired sites. To assist in the development of data secrecy properties, we also introduce the concept of theorem generation functions which enable generation of security theorems independent of the target circuit, thereby paving the way for proof automation. Additionally, we enhance the PCHIP framework with a hierarchy-preserving methodology and we show its utility in preventing malicious data modification which may indirectly result in sensitive information leakage, such as by modifying the secret key in a cryptographic core. This enhanced PCHIP framework also enables development of hybrid module libraries which contain hardware description language (HDL) code along with proofs of lemmas for these modules. These module libraries can then be used for hierarchically proving security properties in higher-level designs, thereby reducing the proof development burden in the general PCHIP framework. A paper based on the work has been published in the IEEE Transactions on Information Forensics and Security (TIFS), the top-tier journal in the area of hardware and cyber security. A copy of this paper is also attached in this report.

5) We introduce (1) VeriCoq-IFT, an automated PCHIP framework for information flow policies, and (2) VeriCoq-H, a hierarchy-preserving Verilog-to-Coq converter. VeriCoq-IFT aims to (i) automate the process of converting designs from HDL to the Coq formal language, (ii) generate security property theorems ensuring compliance with information flow policies, (iii) construct proofs for such theorems, and (iv) check their validity in a design, with minimal user intervention. VeriCoq-H, on the other hand, seeks to convert the entire functionality of a Verilog design to its Coq representation while preserving design hierarchy. It facilitates development of hierarchical proofs and enables construction of hybrid module libraries containing the HDL code and the corresponding reusable lemmas for each module. Applicability of our automated VeriCoq-IFT framework is demonstrated by evaluating trustworthiness of two DES encryption circuits and several genuine and Trojan-infested AES designs, along with the utility of VeriCoq-H in preventing malicious modification of sensitive data, such as the secret key of an encryption circuit. A paper based on the work has been published in the IEEE Transactions on Information Forensics and Security (TIFS), the top-tier journal in the area of hardware and cyber security. A copy of this paper is also attached in this report.

RPPR Final Report as of 17-Nov-2017

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 Oregon Programming Language Summer School (OPLSS) which is held by University of Oregon in June 2016. This summer school is to enable participants to conduct research in the programming language (PL) area, thereby contributing to improve software quality. The goal of the summer school is to provide a unique opportunity for participants to understand the current landscape in the research. This year's OPLSS presents a range of material from foundational work on semantics and type theory to advanced program verification techniques, and also introduces experience with applying the theory. For example, there are some courses introducing PL background like category theory, type theory and logical relations. Also, application of PL is considered, like automated complexity analysis and compiler verifications. Mr. Guo benefits significantly from this summer school which helps him understand the notation system. The lectures about design equivalence and the program synthesis also help Mr. Xiaolong Guo in generating formal HDL code, an important step in his research of proof-carrying hardware and trusted hardware design.

Another graduate student, Raj Gautam Dutta, attended the Sixth Summer School on Formal Techniques (SSFT) in Menlo College. The Formal Techniques summer school is organized by Stanford Research Institute (SRI) under the initiative of Dr. Shankar Natarajan. The main objective of this summer school is to encourage research in the field of formal methods. It has been going on since 2010 at Menlo College, California. Notable scientists in the field of formal techniques from across the world are invited each year to give lecture on topics related to their research. Students from different countries attend this summer school. In this summer school, techniques based on formal logic, such as model checking, satisfiability, static analysis, and automated theorem proving were taught to the students as these methods have been finding a broad range of applications in modeling, analysis, verification, and synthesis. This year's main focus was on deductive verification techniques, which relies on automated (SMT solver) and interactive (PVS prover) theorem provers. As Mr. Dutta's research is on formal methods and particularly on verification of systems, knowledge and tools learnt in this summer school has been helping him in developing solution approaches for analyzing autonomous systems. Moreover, the summer school also helped Mr. Dutta in collaborating with researchers who work in related research areas.

Mr. Xiaolong Guo also got the opportunity to attend the IEEE Symposium on Hardware-Oriented Security and Trust (HOST) which was held in DC in May 2017. 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. Raj Gautam Dutta attended the Design Automation Conference (DAC) which was held in Austin, TX in June 2017. The Design Automation Conference (DAC) is recognized as the premier conference for design and automation of electronic systems. DAC offers outstanding training, education, exhibits and superb networking opportunities for designers, researchers, tool developers and vendors.

Results Dissemination: During the research period, supported by this project, we published three conference papers, one book chapter and three journal publications. The PI (Dr. Yier Jin) has attended and presented research outcomes in the 29th IEEE International System-on-Chip Conference (SOCC), the 17th International Workshop on Microprocessor and SOC Test and Verification (MTV 2016) and the 2017 Government Microcircuit Applications and Critical Technology Conference (GOMACTech-17).

In the 17th International Workshop on Microprocessor and SOC Test and Verification (MTV 2016), we introduced an automatic code converter for translating VHDL to Formal-HDL, a domain specific language for representing hardware designs in Coq language. Our code converter provides support to wide variety of hardware designs. Towards the goal of speeding up the verification procedure in our PCH framework, the code converter is the important first step. The applicability of the tool is demonstrated by converting soft IP cores of AES to its Coq equivalent code.

In the 2017 Government Microcircuit Applications and Critical Technology Conference (GOMACTech-17), we introduced proof-carrying hardware (PCH) and its applications in detail. We also demonstrated the use of theorem proving methods for providing high-level protection of IP cores. An automatic tool chains to automate the whole PCH framework was also presented.

RPPR Final Report
as of 17-Nov-2017

Honors and Awards: Nothing to Report

Protocol Activity Status:

Technology Transfer: Nothing to Report

PARTICIPANTS:

Participant Type: PD/PI

Participant: Yier Jin

Person Months Worked: 3.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Graduate Student (research assistant)

Participant: Xiaolong Guo

Person Months Worked: 6.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

Participant Type: Graduate Student (research assistant)

Participant: Raj Gautam Dutta

Person Months Worked: 6.00

Funding Support:

Project Contribution:

International Collaboration:

International Travel:

National Academy Member: N

Other Collaborators:

ARTICLES:

RPPR Final Report as of 17-Nov-2017

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published

Journal: IEEE Transactions on Information Forensics and Security

Publication Identifier Type: DOI

Publication Identifier: 10.1109/TIFS.2016.2621999

Volume: 12

Issue: 2

First Page #: 405

Date Submitted: 8/15/17 12:00AM

Date Published: 2/1/17 5:00AM

Publication Location:

Article Title: Eliminating the Hardware-Software Boundary: A Proof-Carrying Approach for Trust Evaluation on Computer Systems

Authors: Xiaolong Guo, Raj Gautam Dutta, Yier Jin

Keywords: Hardware trust, proof-carrying hardware

Abstract: The wide usage of hardware intellectual property (IP) cores and software programs from untrusted thirdparty vendors has raised security concerns for computer system designers. The existing approaches, designed to ensure the trustworthiness of either the hardware IP cores or to verify software programs, rarely secure the entire computer system. The semantic gap between the hardware and the software lends to the challenge of securing computer systems. In this paper, we propose a new unified framework to represent both the hardware infrastructure and the software program in the same formal language. As a result, the semantic gap between the hardware and the software is bridged, enabling the development of system-level security properties for the entire computer system. Our unified framework uses a cross-domain formal verification method to protect the entire computer system within the scope of proof-carrying hardware.

Distribution Statement: 1-Approved for public release; distribution is unlimited.

Acknowledged Federal Support: Y

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published

Journal: IEEE Transactions on Information Forensics and Security

Publication Identifier Type: DOI

Publication Identifier: 10.1109/TIFS.2017.2707323

Volume: 12

Issue: 10

First Page #: 2416

Date Submitted: 8/15/17 12:00AM

Date Published: 10/1/17 4:00AM

Publication Location:

Article Title: Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part I: Framework Fundamentals

Authors: Yier Jin, Xiaolong Guo, Raj Gautam Dutta, Mohammad-Mahdi Bidmeshki, Yiorgos Makris

Keywords: Hardware Trust, Proof-Carrying Code, Data Secrecy Protection

Abstract: Proof-carrying hardware intellectual property (PCHIP) is a previously proposed framework for ensuring trustworthiness of third-party hardware IP through the development of formal proofs for security properties designed to prevent introduction of malicious behavior. Based on this framework, we introduce new approaches for assuring that the secrecy of internal information in a hardware design is not compromised by design flaws or malicious hardware Trojans. Specifically, we devise two PCHIP-based information flow tracking (IFT) approaches which enhance the formal PCHIP framework with secrecy tags and/or sensitivity levels in order to provide mechanisms for proving that sensitive information does not reach undesired sites. To assist in the development of data secrecy properties, we also introduce the concept of theorem generation functions which enable generation of security theorems independent of the target circuit, thereby paving the way for proof automation.

Distribution Statement: 1-Approved for public release; distribution is unlimited.

Acknowledged Federal Support: Y

RPPR Final Report as of 17-Nov-2017

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published

Journal: IEEE Transactions on Information Forensics and Security

Publication Identifier Type: DOI

Publication Identifier: 10.1109/TIFS.2017.2707327

Volume: 12

Issue: 10

First Page #: 2430

Date Submitted: 8/15/17 12:00AM

Date Published: 10/1/17 4:00AM

Publication Location:

Article Title: Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation

Authors: Mohammad-Mahdi Bidmeshki, Xiaolong Guo, Raj Gautam Dutta, Yier Jin, Yiorgos Makris

Keywords: Hardware Trust, Proof-Carrying Code, Proof-Carrying Hardware IP

Abstract: Part II of this paper series focuses on automation of the extended proof-carrying hardware intellectual property (PCHIP) framework for data secrecy protection in third-party IPs, which was presented in part I. Specifically, we introduce (1) VeriCoq-IFT, an automated PCHIP framework for information flow policies, and (2) VeriCoq-H, a hierarchy-preserving Verilog-to-Coq converter. VeriCoq-IFT aims to (i) automate the process of converting designs from HDL to the Coq formal language, (ii) generate security property theorems ensuring compliance with information flow policies, (iii) construct proofs for such theorems, and (iv) check their validity in a design, with minimal user intervention. VeriCoq-H, on the other hand, seeks to convert the entire functionality of a Verilog design to its Coq representation while preserving design hierarchy.

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: 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)

Date Received: 15-Aug-2017

Conference Date: 12-Dec-2016

Date Published:

Conference Location: Austin, TX, USA

Paper Title: Automatic RTL-to-Formal Code Converter for IP Security Formal Verification

Authors: Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin

Acknowledged Federal Support: Y

Publication Type: Conference Paper or Presentation

Publication Status: 1-Published

Conference Name: Government Microcircuit Applications and Critical Technology Conference

Date Received: 15-Aug-2017

Conference Date: 20-Mar-2017

Date Published:

Conference Location: Reno, NV

Paper Title: Proof-Carrying Hardware based IP Protection

Authors: Xiaolong Guo, Raj Gautam Dutta, and Yier Jin

Acknowledged Federal Support: Y

Publication Type: Conference Paper or Presentation

Publication Status: 1-Published

Conference Name: 2016 29th IEEE International System-on-Chip Conference (SOCC)

Date Received: 16-Aug-2017

Conference Date: 06-Sep-2016

Date Published:

Conference Location: Seattle, WA, USA

Paper Title: Quantifying trust in autonomous system under uncertainties

Authors: Raj Gautam Dutta, Xiaolong Guo, and Yier Jin

Acknowledged Federal Support: Y

RPPR Final Report
as of 17-Nov-2017

Proof-Carrying Hardware based IP Protection

Xiaolong Guo, Raj Gautam Dutta, and Yier Jin
Department of Electrical and Computer Engineering
University of Central Florida
yier.jin@eecs.ucf.edu

Abstract—*Reusable hardware Intellectual Property (IP) based System-on-Chip (SoC) design has emerged as a pervasive design practice in the industry nowadays. The possibility of hardware Trojans and/or design backdoors hiding in the IP cores has raised security concerns. As existing functional testing methods fall short in detecting unspecified (often malicious) logic, formal methods have emerged as an alternative for validation of trustworthiness of IP cores. Toward this direction, we discuss theorem proving, a category of formal methods, used in hardware trust evaluation. Specifically, proof-carrying hardware (PCH) and its applications are introduced in detail, in which we demonstrate the use of theorem proving methods for providing high-level protection of IP cores. Further, we propose an automatic tool chains to automate the whole PCH framework.*

I. INTRODUCTION

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 have also helped in the proliferation of the intellectual property (IP) market. The use and reuse of existing commercial IPs has enabled improvements in TTM in addition to cost reduction. However, as a result of the globalization of the semiconductor supply chain, companies and the government 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. This has created 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 the inclusion of a Trojan circuit can derail the entire hardware industry.

Many hardware Trojan detection and protection methods have been developed [1]–[5]. However, methods for detecting Trojans at the RTL level (of the pre-silicon stage) has been lacking. Furthermore, existing Trojan detection methods for RTL designs rely on a golden circuit model. Generally, such a model is not readily available for third-party soft-IP cores. As such, it further limits the applicability of existing methods for detecting Trojans implanted in the RTL design

of the hardware. The Proof-Carrying Hardware (PCH) based method could detect Trojans in third-party soft-IP cores in absence of a golden model.

Although the PCH based methods have proved effective in securing the hardware, system-level solutions targeting the entire SoC (particularly composed of third-party hardware IPs) are lacking. The reason is that existing PCH formal verification frameworks, which rely on an interactive theorem prover for evaluating trustworthiness of IP cores, are not scalable to SoC designs [6]–[8]. The reasons behind the scalability problem are 1) a significant manual effort was required for converting HDL code to a formal representation, 2) for each specific attack on each hardware application, security properties need to be designed accordingly, and 3) 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 repetition of the entire deductive process, thereby further increasing the verification time.

To solve these problems, an automatic PCH framework is proposed in this paper, which includes an automatic tool chains - auto code converter, auto properties translator, and auto proof generator. The rest of the paper is organized as follows. In Section II, we discuss previous work on hardware protection using formal techniques. In Section III, we introduce the attack model of this paper, and applicability of PCH in verifying synthesizable register-transfer level (RTL) code. Then we shown the detail of the proposed auto PCH framework in Section IV. Section V concludes the paper.

II. PROOF-CARRY HARDWARE FRAMEWORK

Proof-Carrying Hardware is an approach for ensuring trustworthiness of hardware [6], [9]–[11]. The PCH method is inspired from the proof-carrying code (PCC) [12]. Using the PCC mechanism, untrusted software developers/vendors certify their software code. During the certification process, software vendor develops *safety proof* for the safety policies provided by software customers. The vendor then provides the user with a PCC binary file, which includes the formal proof of the safety properties encoded with the executable code of the software. The customer becomes assured of the safety of the software code by quickly validating the PCC binary file in a proof checker. Efficiency of this approach in reducing validation time at the customer end led to its adoption in different applications.

Following the concept of PCC, authors in [10], [11] proposed the Proof-Carrying Hardware (PCH) framework for dynamically reconfigurable hardware platforms. In the PCH framework, authors used runtime combinational equivalence checking (CEC) for verifying equivalence between the design specification and the design implementation. A boolean satisfiability (SAT) solver was used to generate resolution proof for unsatisfiability of the combinational miter circuit, represented in a conjunctive normal form (CNF). The proof traces were combined with the bitstream into a proof-carrying bitstream by the vendor and given to the customer for validation. However, the approach did not consider exchange of a set of security properties between the customer and the vendor. Rather it considers safety policy, which included agreements on a specific bitstream format, on a CNF to represent combinational functions, and the propositional calculus for proof construction and verification.

III. PROOF-CARRYING HARDWARE BASED IP PROTECTION IN RTL CODE

A. Attack Model

In this paper, we assume that malicious logic is inserted by an adversary at the design stage of the supply chain. We also 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 through different mechanisms. Upon activation it can leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the hardware. Note that in this paper, we only consider Trojans which can be activated by a specific digital input vector.

Further, we assume that the verification tools (e.g., Coq) used in our PCH framework produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design whereas its absence indicates the presence of malicious logic. 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 to launch the attack.

B. PCH Framework Applied in RTL Code

In [6], [9], another PCH framework was proposed, which overcame the limitations of the previous framework and expanded it for verification of security properties on soft IP cores. The new PCH framework is dedicated for security properties verification on synthesizable register-transfer level (RTL) IP cores. In the framework, Hoare-logic style reasoning is used to prove the correctness of the RTL code and implementation was carried out using the Coq proof assistant [13]. As Coq supports automatic proof checking, it can help IP customers to validate proof of security properties with minimum effort. Moreover, usage of the Coq platform by

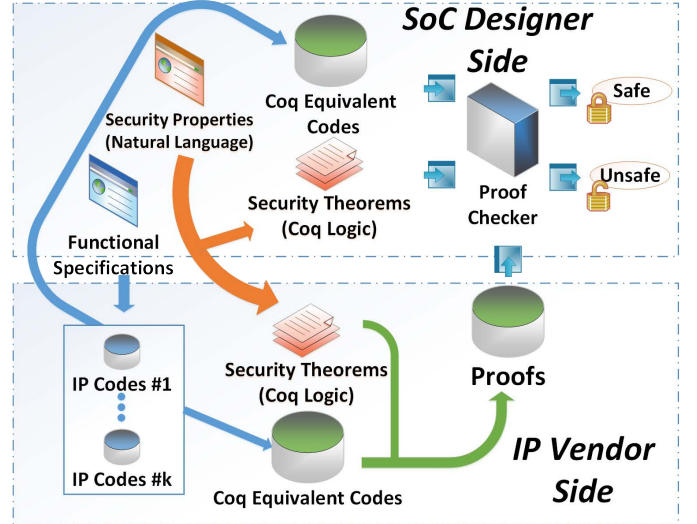


Fig. 1: Working procedure of PCH framework [14]

both IP vendors and IP consumers ensures that same deductive rules could be used for validating the proof. However, Coq does not recognize commercial hardware description languages (HDLs) and security properties expressed in a natural language. To solve this problem, semantic translation of HDLs and informal security specifications to calculus of inductive construction (CIC) was done. Based on this PCH framework, a new trusted IP acquisition and delivery protocol was proposed (See Figure 1), in which IP consumers provided both functional specifications and a set of security properties to IP vendors. IP vendors then developed the HDL code based on the functional specifications. The HDL code and security properties were then translated to CIC. Subsequently, proofs were constructed for security theorems and the transformed HDL code. The HDL code and proof for security properties were combined into a trusted bundle and delivered to the consumer. Upon receiving the trusted bundle, IP consumers first generate the formal representation of the design and security properties in CIC. The translated code, combined with formal theorems and proofs were quickly validated using the proof checker in Coq platform.

Within the PCH framework defined in [6], the most important component is the set of security properties. A complete set of properties can ensure trustworthiness of the design by detecting malicious logic if present in the IP core. As different soft IP cores often share similar security properties, an expandable centralized repository of security properties with theorem-proof pairs will be a desirable solution for reducing the verification effort and protect the design from different types of hardware Trojan attacks. Any hardware designer will be able to pick a set of security properties from the property library rather than developing properties from scratch. The selected set of properties will be able to render many modes of attack significantly difficult to implement, thereby ensuring the trustworthiness of the delivered IP cores.

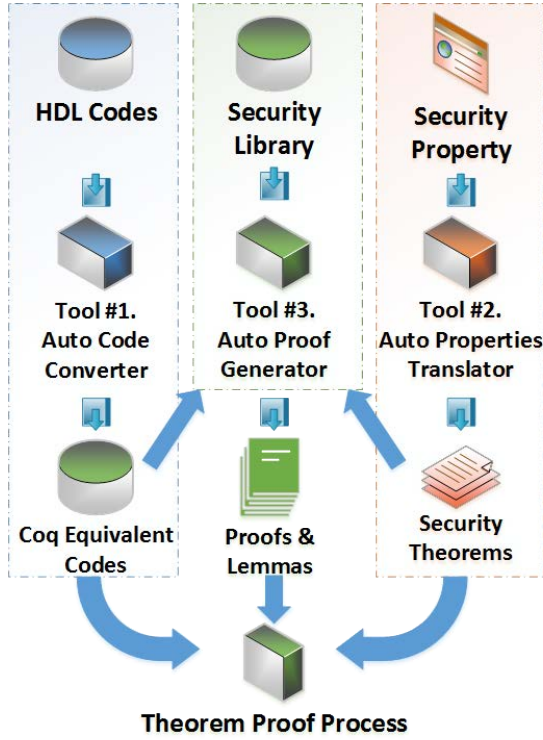


Fig. 2: Automatic PCH framework

IV. AUTOMATIC PCH FRAMEWORK

As mentioned in previous sections, extending PCH method to large-scale design such as SoCs was difficult due to the time required for verification. Therefore, development of automation tools were demanded to alleviate this challenge. The software tools we intend to develop will help facilitate the process of code conversion, security property translation, and proof generation. An overview of these tools in the PCH framework is shown in Figure 2. Code converter will convert HDL code to our *Formal-HDL*, security property translation tool will convert an informal security property given in natural language to provable security theorem represented in *Formal-HDL*, and automated proof scripts (tactics) will help in expediting proof construction in Coq.

A. Auto Code Converter

The automated code converter we discussed is considered as one of the tools applied in the automatic PCH framework as shown in Figure 2. Supported by this automatic code converter, verification experts can then ensure that the circuit on which security theorems are proved corresponds exactly to the HDL code that needs to be verified. Thus, the proposed automatic framework and the automated code conversion tool can reduce effort required in formal verification of large-scale hardware designs.

In previous PCH methods, code conversion was done manually, which increases the workload and risks of human error. In order to eliminate the burden of code conversion,

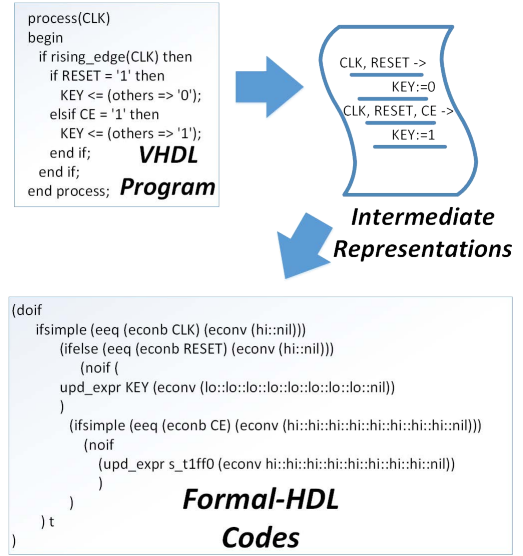


Fig. 3: Code conversion from VHDL to *Formal-HDL* through IRs

we have developed the *Formal-HDL* first in [15], a hardware description language within Coq environment. The *Formal-HDL* was further enhanced to include additional VHDL language constructs in [16]. Then in our paper [17], an automatic syntactic and semantic translation tool of RTL code to a domain-specific language is developed for eliminating manual effort incurred during code conversion. Compared to the manual translation in previous PCH frameworks [6], [7], [15], the translation tool considers all the common VHDL syntaxes and converts the hardware design in VHDL to *Formal-HDL*.

Our converter extends the work of [18], 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 developed tool here incorporates additional syntaxes such as *component instantiations*, *user defined types*, *ranged types*, *constants*, *two dimensional array*, *case statements*.

There are two important stages involved in construction of this tool. As shown in Figure 3, at the beginning, most of VHDL codes will be parsed to Intermediate Representations (IRs), which are constructed using variables, functions, and behavioural rules. Then those IRs will be converted to the formal expressions in *Formal-HDL*, which can represent basic circuit units, combinational logic, sequential logic, and module instantiations. More implementation details of the converter can be found in [17].

B. Auto Properties Translator

Meanwhile, in PCH the IP consumer (the SoC designer) provides informal (written in natural language) security properties to the IP vendor as shown in Figure 1. Upon receiving

these information, in the IP vendor side, the most challenging task in designing and verifying third-party IP cores is selecting and defining a complete set of security properties. A well-defined security property set can precisely assess the trustworthiness of a target circuit. In contrast, a poorly defined security property set may fail to detect malicious modifications or design flaws endangering the security of the circuit. Also, security properties agreed-upon by vendors and consumers for specific IP cores are less likely to be reused for different IP cores or accepted by other consumers because different users often have different security requirements.

Considering the fact that building proofs for all security properties is impossible and unnecessary, developing a standardized set of security properties and managing them in a property library would be a more attractive solution for IP vendors. A standard set of formal security properties, a set of target circuits, and the corresponding security proofs would enable a quick and effective way for consumers to decide the security property set. Specifically, choosing standard security properties rather than constructing new ones on a case-by-case basis would be preferable for both IP vendors and consumers. Therefore, as shown in Figure 2, the proposed Tool #2, auto properties translator, will reads the informal security properties, and then produce the security theorem written by *Formal-HDL* based on the pre-defined security properties.

C. Auto Proof Generator

Along with these language conventions and property translations, we plan to leverage automated proof scripts (tactics) for minimizing the time required for proof construction in Coq (see Tool #3 in Figure 2). To build such *tactics* we consider the Coq Equivalent code of the design (obtained from the VHDL code of IP module), proofs stored in the proposed security library, and the security theorems. Subsequently, such tactics will be invoked while constructing proof of new security theorems for a SoC design. This will be a critical step toward proof automation in Coq for hardware designs.

V. CONCLUSION AND FUTURE WORK

Growing reliance on hardware IPs, often gathered from untrusted third-party vendors, severely affects the security and trustworthiness of SoC computing platforms. A major concern with the hardware IPs acquired from external sources is that they may come with deliberate (often malicious) implants to incorporate undesired functionality, undocumented test/debug interface working as hidden backdoor, or other integrity issues. SoC integrators typically tend to treat these IPs as black box and rely on the IP vendors on their structural/functional integrity. As various reports suggest that such a reliance can compromise the hardware security and trust. We presented theorem proving approaches using proof-carrying hardware to enable high-level protection of IP cores.

ACKNOWLEDGMENT

This work has been partially supported by Army Research Office (ARO W911NF-16-1-0124), National Science Foundation (NSF-1319105) and Cisco.

REFERENCES

- [1] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware Trojans," in *IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 632–639.
- [2] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards Trojan-free trusted ICs: Problem analysis and detection scheme," in *IEEE Design Automation and Test in Europe*, 2008, pp. 1362–1365.
- [3] 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.
- [4] L. Lin, M. Kasper, T. Guney, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware Trojans through side-channel engineering," in *Cryptographic Hardware and Embedded Systems*, ser. LNCS. Springer-Verlag Berlin, 2009, vol. 5747, pp. 382–395.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] E. Love, Y. Jin, and Y. Makris, "Enhancing security via provably trustworthy hardware intellectual property," in *Hardware-Oriented Security and Trust (HOST)*, 2011 *IEEE International Symposium on*, 2011, pp. 12–17.
- [10] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *International Conference on Reconfigurable Computing and FPGAs*, 2009, pp. 189–194.
- [11] 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.
- [12] 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.
- [13] INRIA, "The coq proof assistant," 2010, <http://coq.inria.fr/>.
- [14] 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.
- [15] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.
- [16] X. Guo, R. G. Dutta, and Y. Jin, "Hierarchy-preserving formal verification methods for pre-silicon security assurance," in *16th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2015.
- [17] —, "Automatic rtl-to-formal code converter for ip security formal verification," in *17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2016.
- [18] A. Smrčka and T. Vojnar, "Verifying parametrised hardware designs via counter automata," in *Haifa Verification Conference*. Springer, 2007, pp. 51–68.

Automatic RTL-to-Formal Code Converter for IP Security Formal Verification

Xiaolong Guo*, Raj Gautam Dutta*, Prabhat Mishra[†], and Yier Jin*

*Department of Electrical and Computer Engineering, University of Central Florida

[†]Department of Computer and Information Science and Engineering, University of Florida

{guoxiaolong, rajgautamdutta}@knights.ucf.edu, prabhat@cise.ufl.edu, yier.jin@eecs.ucf.edu

Abstract—The wide usage of hardware intellectual property (IP) cores from untrusted vendors has raised security concerns in the integrated circuit (IC) industry. Existing testing methods are designed to validate the functionality of the hardware IP cores. These methods often fall short in detecting unspecified (often malicious) logic. Formal methods like Proof-Carrying Hardware (PCH), on the other hand, can help eliminate hardware Trojans and/or design backdoors by formally proving security properties on soft IP cores despite the high proof development cost. One of the causes to the high cost is the manual conversion of the hardware design from RTL code to a domain-specific language prior to verification. To mitigate this issue and to lower the overall cost of PCH framework, we propose an automatic code converter for translating VHDL to *Formal-HDL*, a domain specific language for representing hardware designs in Coq language. Our code converter provides support to wide variety of hardware designs. Towards the goal of speeding up the verification procedure in our PCH framework, the code converter is the important first step. The applicability of the tool is demonstrated by converting soft IP cores of AES to its Coq equivalent code.

I. INTRODUCTION

The improvement of manufacturing technology makes it possible to integrate billions of transistors in one chip, but it increases the difficulty of designing such large-scale circuits. To alleviate the workload of the circuit designers and to shorten the time-to-market (TTM), the hierarchical design methodology has been widely used in the IC industry with the leading example being system-on-chip (SoC) platforms. System integrators use IP cores from trusted and untrusted third-party vendors to build SoCs in a hierarchical structure.

The prevailing usage of third-party soft IP cores in SoC designs raises security concerns as current IP core verification methods focus on IP functionality rather than IP trustworthiness. Moreover, lack of regulation in the IP transaction market adds to the predicament of the SoC designers and forces them to perform verification and validation of IPs themselves. To help SoC designers in IP verification, various methods have been developed, which leverage enhanced functional testing and/or perform probability analysis of internal nodes for IP core trust evaluation and malicious logic detection [1], [2]. However, these methods were easily bypassed by sophisticated hardware Trojans [3]–[5]. Formal methods were also introduced for IP core trust evaluation [1], [6]–[10]. Among all the proposed formal methods, PCH, which originated from proof-carrying code (PCC), emerged as one of the most prevalent methods for certifying the absence of malicious logic in soft IP cores and reconfigurable logic [6]–[10]. In the PCH approach,

synthesizable register-transfer level (RTL) code of IP core and informal security properties were first represented in a domain-specific language (DSL), *Formal-HDL*, which is built on top of *Gallina*, the functional programming language of the Coq proof assistant [11]. Then, Hoare-logic style reasoning was used to prove the correctness of the RTL code in the Coq platform.

However, extending existing PCH methods to large-scale design such as SoCs was difficult due to the time required for verification [8]–[10]. This was because a significant manual effort was required for converting HDL code to a formal representation, and constructing proofs of security properties based on the design. Moreover, any modification of the design required repetition of the entire deductive process, thereby further increasing the verification time. To address these problems, an automatic tool for syntactic and semantic translation of RTL code to a domain-specific language is developed. Compared to the manual translation in previous PCH frameworks [8]–[10], this tool considers all the common VHDL syntaxes and converts the hardware design written using it to *Formal-HDL*. Apart from design translation, our tool can also convert the informal security specification to the domain specific language of Coq. Thus, this tool is an important step toward reducing effort required in deductive verification of large-scale hardware designs in tools such as the Coq platform.

The rest of the paper is organized as follows: In section II, we mention works related to ours. In section III, we introduce the threat model and describe our PCH framework. In Section IV, we provide implementation details of the code converter. Section V presents demonstrations of the proposed code converter by translating AES core in VHDL to Coq language. Final conclusions are drawn in Section VI.

II. RELATED WORK

Various methods have been proposed in the software domain to validate the trustworthiness and genuineness of software programs. These methods protect computer systems from untrusted software programs. Most of these methods place the burden on software consumers to verify the code. However, PCC switches the verification burden to software providers (software vendors/developers) [12].

A similar mechanism, called PCH, was used in the hardware domain to protect third-party soft IPs [8]–[10]. The PCH framework certifies that soft IPs are trusted if certain carefully

specified security properties hold. As shown in Figure 1, in this approach, the IP consumer provides functional specifications and security constraints to the IP vendor. Upon receiving the request, the IP vendor develops the RTL code using hardware description languages (HDLs). Before proving the trustworthiness of the RTL code with respect to the formally specified security properties in Coq, the IP vendor needs to perform semantic translation of the HDL code and informal security properties into *Formal-HDL*. After the proof has been constructed, the IP vendor provides the IP consumer with the RTL code, formalized security theorems of security properties, and proofs of security theorems. The IP consumers also translate the design and security properties to *Formal-HDL*. Then, the proof checker in Coq is used to automatically validate the proof of security theorems on the translated code. Correspondingly, the PCH and its applications were introduced in detail in [13], where the use of theorem proving methods for providing high level protection of IP cores is demonstrated.

A language translation tool called *VeriCoq* was developed in [14] which converts hardware designs represented in Verilog into Coq. However, *VeriCoq* requires a flattening the hierarchical design, which makes manual proof development very challenging. Moreover, the authors in [14] did not provide details of the supported Verilog syntaxes the *VeriCoq* can support and the demonstration in the paper is not sufficient to show applicability of the *VeriCoq* tool to any general hardware design. Thus, we develop VHDL-to-Coq code converter, which can convert general VHDL designs to Coq equivalent codes by supporting all common VHDL syntaxes.

III. BACKGROUND

A. Attack Model

In this paper, we assume that malicious logic is inserted by an adversary at the design stage of the supply chain. We also 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 through different mechanisms. 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.

Further, we assume that the verification tools (e.g., Coq) used in our PCH framework produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design whereas its absence indicates the presence of malicious logic. 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. Automatic PCH Framework

As mentioned in previous sections, extending PCH method to large-scale design such as SoCs was difficult due to the time

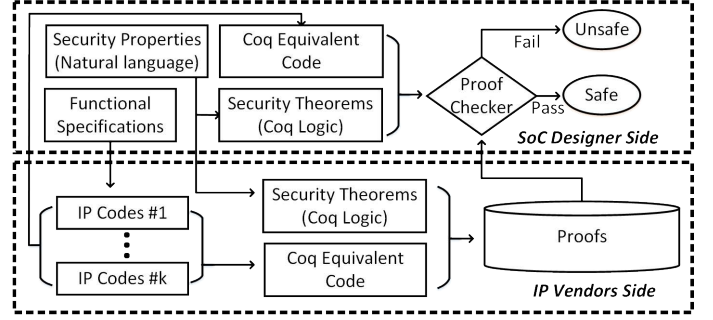


Figure 1: Working procedure of the PCH [13]

required for verification. Therefore, development of automation tools were required to alleviate this challenge. The software tools we intend to develop will help facilitate the process of code conversion, security property translation, and proof generation. An overview of these tools in the PCH framework is shown in Figure 2. Code converter will convert HDL code to our DSL, *Formal-HDL*, security property translation tool will convert an informal security property given in natural language to provable security theorem represented in *Formal-HDL*, and automated proof scripts (tactics) will help in expediting proof construction in Coq.

In previous PCH methods, code conversion was done manually, which increases the workload and risks of human error. In order to eliminate the burden of code conversion, we have firstly developed the *Formal-HDL*, a hardware description language within Coq environment. The *Formal-HDL* was proposed in [10] and was further enhanced to include additional VHDL language constructs in [15]. In this paper, we aim to design and implement the automated code converter to convert the HDL code to the *Formal-HDL* (the Tool 1 in Figure 2). Supported by this automatic code converter, verification experts can then ensure that the circuit on which security theorems are proved corresponds exactly to the HDL code that needs to be verified.

Furthermore, in PCH the IP consumer provides informal (written in natural language) security properties to the IP vendor as shown in Figure 1. Upon receiving these information, semantic translation of the informal security properties to *Formal-HDL* is carried out. This translation was also done manually in the previous PCH method. To achieve a fully automatic PCH framework, another automation tool is required to facilitate this conversion process¹.

IV. AUTOMATIC CODE CONVERTER

In the translation, the source language is VHDL and the destination language is *Formal-HDL* which is defined on top of *Gallina* of Coq. As shown in Figure 3, the hardware design constructed using VHDL syntaxes is first converted to an intermediate representation (IR). Then, the IRs are translated to *Formal-HDL*.

¹Note that the development of the property formalization tools as well as other tools for the full automatic PCH framework is out of the scope of this paper and will be discussed in our future work.

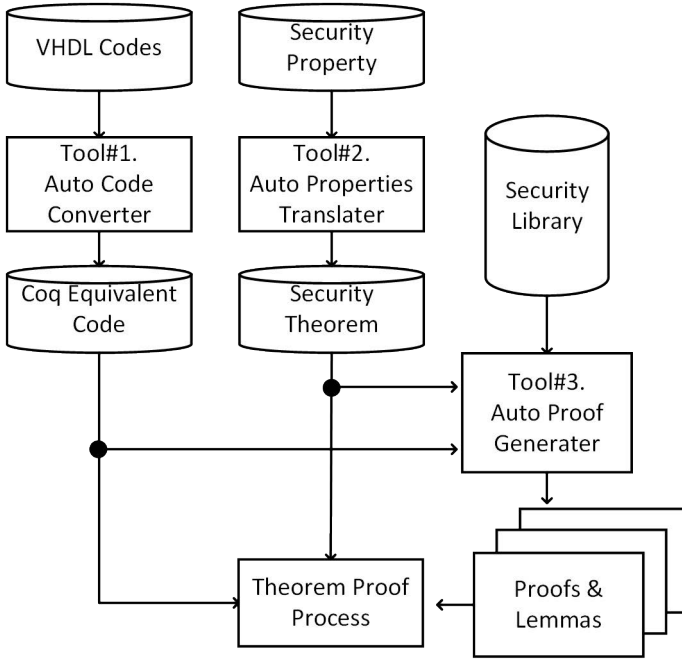


Figure 2: Automatic PCH Framework

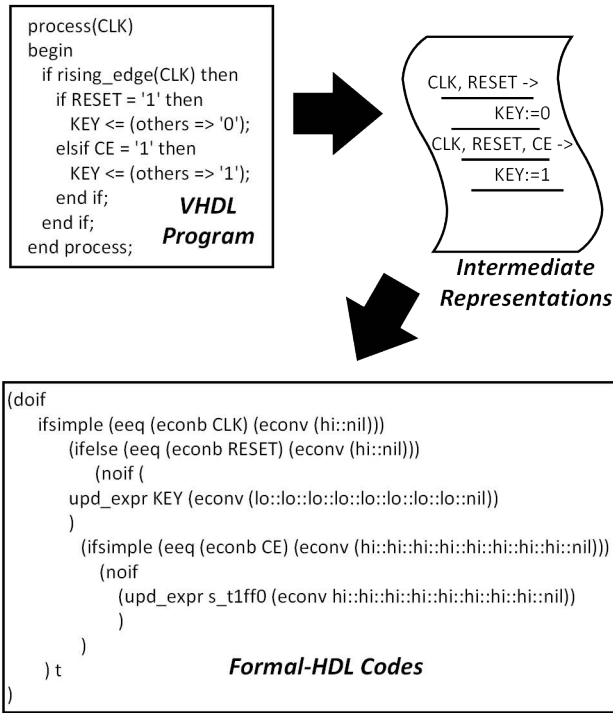


Figure 3: Code conversion from VHDL to *Formal-HDL* through IRs

A. From-VHDL to Intermediate Representations

Our paper extends the work of [16], 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 developed tool here incorporates additional syntaxes such as *component instantiations*, *user*

defined types, ranged types, constants, two dimensional array, case statements.

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

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

where, $c \in C$, $v \in V$, and $e \in E$. Equation (1) signifies that under a list of enabling conditions c , a variable v is assigned to a new value, defined by an expression e . As shown in Figure 3, most of VHDL codes will be parsed to IRs in the form of Equation (1).

B. Formal-HDL Representations

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

The *Formal-HDL* of [10], can represent basic circuit units, combinational logic, sequential logic, and module instantiations. In [15], *Formal-HDL* is further updated to include component instantiations to preserve the design hierarchy of the SoC. Below, we show 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, respectively. 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.

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

To facilitate clock-edge specifications and synchronizations among signal assignments, processes are defined in VHDL. In *Formal-HDL*, these behaviours 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:

$$exor : expr \rightarrow expr \rightarrow expr \quad (2)$$

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 non-blocking if statements *doif*.

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 sub modules' 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.

V. CASE STUDY

In this section, we show the development of a Python-based automatic code converter, and results of converting a sample VHDL design - AES encryption core, to its *Formal-HDL* equivalent expressions.

Following Section IV, for parsing VHDL to IRs, translation rules of [16] are used. Furthermore, we extend the tool to incorporate additional syntaxes of VHDL such as constants, component instances, choices in expressions, user defined types. For the translation of IRs to *Formal-HDL*, mapping will be built. For instance, as shown in equation (1), the conditions, variables, and expressions will be described by using if-then-else statements in *Formal-HDL*. Figure 3 is an example of IR and *Formal-HDL* codes during the conversion of a small VHDL program.

To test our proposed code conversion tool, we have applied it on the AES soft IP core [17], represented in VHDL. The synthesized AES design contains 326 registers and 935 logic gates. Experiments were carried out in a desktop with 64-bit Intel i7-3370 CPU and 16GB RAM. In total, the time consumed in conversion is 1.526 seconds. Thus, we can conclude that the time required for converting VHDL designs to Coq equivalent code is significantly less than the manual effort (several hours or even days) required for the same in the previous PCH framework.

VI. CONCLUSION

As an interactive theorem prover such as Coq requires lot of effort to manually verify hardware designs against security specifications, PCH framework suffers from scalability issues. In this paper, VHDL-to-Coq code converter is developed to automate the code conversion process in 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*. The results obtained while demonstrating our tool in an AES IP core shows significant reduction in the amount of effort required for translating the design from HDL to the Coq equivalent code.

ACKNOWLEDGMENT

This work has been partially supported by the National Science Foundation (NSF-1319105), the Army Research Office (ARO W911NF-16-1-0124) and Cisco.

REFERENCES

- [1] M. Banga and M. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2010, pp. 56–59.
- [2] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," ser. CCS '13, 2013, pp. 697–708.
- [3] D. Sullivan, J. Biggers, G. Zhu, S. Zhang, and Y. Jin, "FIGHT-metric: Functional identification of gate-level hardware trustworthiness," in *Design Automation Conference (DAC)*, 2014.
- [4] N. Tsoutsos, C. Konstantinou, and M. Maniatakis, "Advanced techniques for designing stealthy hardware trojans," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, 2014.
- [5] M. Rudra, N. Daniel, V. Nagoorkar, and D. Hoe, "Designing stealthy trojans with sequential logic: A stream cipher case study," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, 2014.
- [6] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *ReConFig*, 2009, pp. 189–194.
- [7] S. Drzevitzky and M. Platzner, "Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach," in *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2011, pp. 1–8.
- [8] 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 (TIFS)*, vol. 7, no. 1, pp. 25–40, 2012.
- [9] 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.
- [10] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.
- [11] INRIA, "The coq proof assistant," 2010, <http://coq.inria.fr/>.
- [12] 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.
- [13] 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.
- [14] M.-M. Bidmeshki and Y. Makris, "Vericoq: A verilog-to-coq converter for proof-carrying hardware automation," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 29–32.
- [15] X. Guo, R. G. Dutta, and Y. Jin, "Hierarchy-preserving formal verification methods for pre-silicon security assurance," in *16th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2015.
- [16] A. Smrčka and T. Vojnar, "Verifying parametrised hardware designs via counter automata," in *Haifa Verification Conference*. Springer, 2007, pp. 51–68.
- [17] AES core modules. http://opencores.org/project,aes_128_192_256.

Chapter 10

IP Trust Validation Using Proof-Carrying Hardware

Xiaolong Guo, Raj Gautam Dutta, and Yier Jin

10.1 Introduction

A rapidly growing third-party Intellectual Property (IP) market provides IP consumers with high flexibility when designing electronic systems. It also reduces the development time and expertise needed to compete in a market where profit-windows are very narrow. However, one key issue that has been neglected is the security of hardware designs built upon third-party IP cores. Historically, IP consumers have focused on IP functionality and performance than security. The negligence toward development of robust security policies is reflected in the IP design flow (see Fig. 10.1), where IP core specification usually only includes functionality and performance measurements.

The prevailing usage of third-party soft IP cores in SoC designs raises security concerns as current IP core verification methods focus on IP functionality rather than IP trustworthiness. Moreover, lack of regulation in the IP transaction market adds to the predicament of the SoC designers and forces them to perform verification and validation of IPs themselves. To help SoC designers in IP verification, various methods have been developed to leverage enhanced functional testing and/or perform probability analysis of internal nodes for IP core trust evaluation and malicious logic detection [1, 2]. However, these methods were easily bypassed by sophisticated hardware Trojans [3–5]. Formal methods were also introduced for IP core trust evaluation [1, 6–10]. Among all the proposed formal methods, proof-carrying hardware (PCH), which originated from proof-carrying code (PCC), emerged as one of the most prevalent methods for certifying the absence of malicious logic in soft IP cores and reconfigurable logic [6–10]. In the PCH approach, synthesizable register-transfer level (RTL) code of IP core and informal security properties were

X. Guo • R.G. Dutta • Y. Jin (✉)

University of Central Florida, Orlando, FL 32816, USA

e-mail: guoxiaolong@knights.ucf.edu; rajgautamdutta@knights.ucf.edu; yier.jin@eecs.ucf.edu

© Springer International Publishing AG 2017

P. Mishra et al. (eds.), *Hardware IP Security and Trust*,

DOI 10.1007/978-3-319-49025-0_10

207

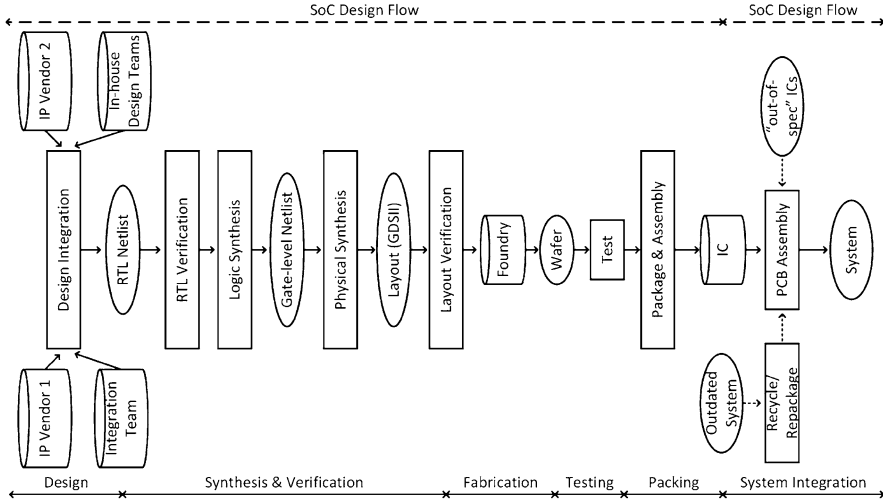


Fig. 10.1 IC design flow within the semiconductor supply chain

first represented in *Gallina*—the internal functional programming language of the Coq proof assistant [11]. Then, Hoare-logic style reasoning was used to prove the correctness of the RTL code in the Coq platform.

The rest of the chapter is organized as follows: In Sect. 10.2, we provide an overview of existing methods for IP protection, introduce the threat model, and provide some relevant background on two different formal verification approaches. In Sect. 10.3, we provide detailed explanation of the PCH method for ensuring trustworthiness of IP cores. Finally, Sect. 10.4 concludes the chapter.

10.2 Overview of Formal Verification Methods for IP Protection

To counter the threat of untrusted third-party resources, pre-silicon trust evaluation approaches have been proposed recently [1, 12, 13]. Most of these methods try to trigger malicious logic by enhancing functional testing with extra test vectors. Authors in [12] proposed a method to generate “Trojan Vectors” into the testing patterns, hoping to activate the hardware Trojans during the functional testing. In order to identify suspicious circuitry, unused circuit identification (UCI) [13] method analyzed the RTL code to find lines of code that are never used. However, these methods assume that the attacker uses rarely occurring events as Trojan triggers. Using “less-rare” events as trigger will void these approaches. This was demonstrated in [14], where hardware Trojans were designed to defeat UCI.

Admitting the limitations of enhanced functional testing methods, researchers started looking into formal solutions. Although at its early stage, formal methods have already shown their advantages over testing methods in exhaustive security verification [8, 9, 15, 16]. A multi-stage approach, which included assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and use of sequential Automatic Test Pattern Generation (ATPG), was adopted in [15] to identify suspicious signals for detecting hardware Trojans. This approach was demonstrated on an RS232 circuit and the efficiency of the approach in detecting Trojan signals ranged between 67.7 and 100 %. In [8, 9, 16], a PCH framework was used to verify security properties on soft IP cores. Supported by the Coq proof assistant [11], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. In the following section in this chapter, we will explain the PCH approach for soft IP core verification in greater details. This method uses an interactive theorem prover and model checker for verifying the design.

10.2.1 Threat Model

The IP protection methods in this chapter are based on the threat model that 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 hardware description language (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 chapter, Trojans which can be activated by a specific “digital” input vector are only considered.

Meanwhile, verification tools (e.g., Coq) used in all methods are assumed to produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design whereas its absence indicates the presence of malicious logic. However, the framework does not provide protection of an IP from Trojans whose behaviors are not captured by the set of security properties. Furthermore, there is also an assumption that the attacker has intricate knowledge of the hardware to identify critical registers and modify them in order to carry out the attack.

10.2.2 Formal Verification Methods

Formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [8, 9, 15–20]. These previous methods leverage one of the following two techniques, model checking and interactive/automated theorem proving, for design verification.

10.2.2.1 Theorem Prover

Theorem provers are used to prove or disprove properties of systems expressed as logical statements [21–28]. Over the years, several theorem provers (both interactive and automated) have been developed for proving properties of hardware and software systems. However, using them for verification on large and complex systems require excessive effort and time. Irrespective of these limitations, theorem provers have currently drawn a lot of interest in verifying security properties on hardware. Among all the formal methods, they have emerged as the most prominent solution for verifying large-scale designs.

One leading example of an interactive theorem prover is the open source tool called Coq proof assistant [11]. Coq is an interactive theorem prover/proof assistant, which enables verification of software and hardware programs with respect to their specification [25]. 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 the program and proofs in its dependently typed language called the *Calculus of Inductive Construction (CIC)*. Correctness of the proof of the program is automatically checked using the built-in type-checker of Coq. To expedite the process of building proofs, Coq provides a library consisting of programs called *tactics*. However, existing Coq *tactics* does not capture properties of hardware designs and thus does not significantly reduce the time required for certifying large-scale hardware IP cores [6–8].

10.2.2.2 Model Checker

Model checking [29] is an automated method for verifying and validating models in software and hardware applications [30–42]. In this approach, a model (Verilog/VHDL code of hardware) $\mathcal{M}$ with an initial state 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, $\mathcal{M}, s_0 \models \phi$. 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 [43, 44]. Recently, model checkers have been used for detecting malicious signals in third-party IP cores [15, 20]. The application of model checking techniques to SoCs, including symbolic approaches based on Reduced Order Binary Decision Diagrams (ROBDD) and Satisfiability (SAT) solving, has had limited success due to the state-space explosion problem [45]. For example, a model with n Boolean variables can have as many as 2^n states, a typical soft IP core with 1000 32-bit integer variables has billions of states.

Symbolic model checking using ROBDD is one of the initial approaches used for hardware systems verification [46–48]. Unlike explicit state model checking where all states of the system are explicitly enumerated, this technique model states (represented symbolically) of the transition system using ROBDD.

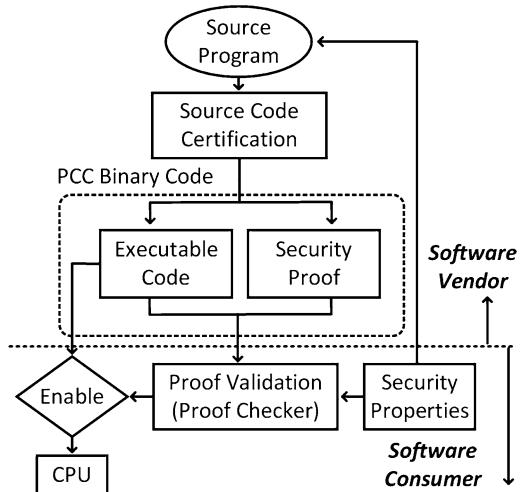
The ROBDD is a unique, canonical representation of a Boolean expression of the system. Subsequently, the specification to be checked is represented using a temporal logic. A model checking algorithm then checks whether the specification is true on a set of states of the system. Despite being a popular 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 proper ordering, the size of the ROBDD increases significantly. Moreover, it is memory intensive for storing and manipulating Binary Decision Diagrams (BDDs) of a system with a large state space.

Another technique called bounded-model checking (BMC) replaces BDDs in symbolic checking with SAT solving [49–51]. In this approach, a propositional formula is first constructed using a model of the system, the temporal logic specification, and a bound. The formula is then provided to a SAT solver to either obtain a satisfying assignment or to prove that no such assignment exists. Although BMC outperforms BDD based model checking in some cases, the method cannot be used to test properties (specification) when the bound is large or cannot be determined.

10.3 Proof-Carrying Hardware Framework for IP Protection

Various methods have been proposed in the software domain to validate the trustworthiness and genuineness of software programs. These methods protect computer systems from untrusted software programs. Most of these methods lay burden on software consumers to verify the code. However, *proof-carrying code* (PCC) switches the verification burden to software providers (software vendors/developers). Figure 10.2 outlines the basic working process of the PCC framework.

Fig. 10.2 Working procedure of the PCC framework [52]



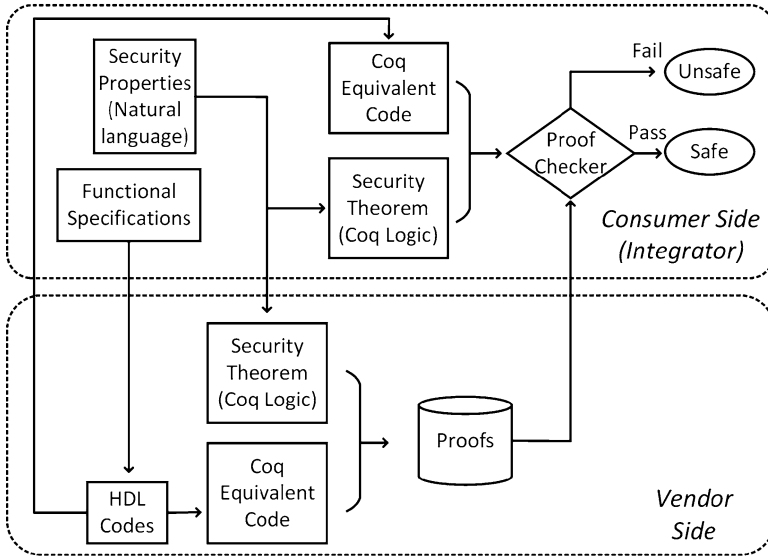


Fig. 10.3 Working process of the PCH framework [17]

During the *source code certification* stage of the PCC process, the software provider verifies the code with respect to the *security property* designed by the software consumer and encodes the formal proof of the security property with the executable code in a *PCC binary file*. In the *proof validation* stage, the software consumer determines whether the code from the potentially untrusted software provider is safe for execution by validating the PCC binary file using a proof checker [52].

A similar mechanism, referred to as Proof-Carrying Hardware (PCH), was used in the hardware domain to protect third-party soft IP cores [8–10]. The PCH framework ensures trust-worthiness of soft IP cores by verifying a set of carefully specified security properties. The working procedure of the PCH framework is shown in Fig. 10.3. In this approach, the IP consumer provides design specifications and informal (written in natural language) security properties to the IP vendor. Upon receiving the request, the IP vendor develops the RTL design using a hardware description language (HDL). Then, semantic translation of the HDL code and informal security properties to Gallina is carried out. Subsequently, Hoare-logic style reasoning is used for proving the correctness of the RTL code with respect to formally specified security properties in Coq. As Coq supports automatic proof checking, it can help IP customers validate proof of security properties with minimum efforts. Moreover, usage of the Coq platform by both IP vendors and IP consumers ensures that the same deductive rules will be used for validating the proof. After verification, the IP vendor provides the IP consumer with the HDL code (both original and translated versions), formalized security theorems of security properties, and proofs of these security theorems. Then, the proof checker in Coq is used by the IP consumer to quickly validate the proof of security theorems on the

translated code. The proof checking process is fast, automated, and does not require extensive computational resources.

10.3.1 Semantic Translation

The PCH based IP protection method requires semantic translation of circuit design in HDL to Coq's specification language, *Gallina*. Consequently, a *formal-HDL* is developed in [10], which includes a set of rules to enable this translation. These rules can help represent basic circuit units, combinational logic, sequential logic, and module instantiations. The *formal-HDL* is further extended in [53] to capture hierarchical design methodology, which is used for representing large circuits such as SoC. A brief description of the *formal-HDL* is given below

- *Basic Circuit Units:*

In the *formal-HDL*, basic circuit units are the most important components and they include signals and buses. During the translation, three digital values are used for signals: high, low, and unknown. To represent sequential logic, a *bus* type is defined as a function, which takes timing variable t and returns a list of signal values as shown in *Listing 10.1*. All circuit signals are of *bus* type and their values can be modified either by a blocking assignment or a nonblocking assignment (shown in *Listing 10.1*). Moreover, inputs and outputs are also defined as *bus* type.

Listing 10.1 Basic Circuit Units in Semantic Model

```

Inductive value := lo|hi|x.
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition input := bus.
Definition output := bus.
Definition wire := bus.
Definition reg := bus.

```

- *Signal Operations:* Logic operations such as *and*, *or*, *not*, and *xor*, as well as bus comparison operations such as checking for bus equality: *bus_eq* and less-than: *bus_lt* are designed to handle *bus* in *Gallina*. The conditional statement of RTL code such as *if...else...* checks whether signals are on or off. To incorporate this functionality in Coq, a special function, *bus_eq_0*, which compares the bus value to *hi* or *lo* is added.

Listing 10.2 Signal Operations in Semantic Model

```

Fixpoint bv_bit_and (a b : bus_value) {struct a} : bus_value :=
  match a with
  | nil => nil
  | la :: a' =>
    match b with
    | nil => nil
    | lb :: b' => (v_and la lb) :: (bv_bit_and a' b')
    end
  end.

Definition bus_bit_and (a b : bus) : bus :=
  fun t:nat => bv_bit_and (a t) (b t).

Fixpoint bv_eq_0 (a : bus_value) {struct a} : value :=
  match a with
  | hi :: lt => lo
  | lo :: lt => bv_eq_0 lt
  | nil => hi
  end.

Definition bus_eq_0 (a : bus) (t : nat) : value := bv_eq_0 (a t).

```

- *Combinational and Sequential Logic:* The definition of signals, expressions, and their semantics paves the way for converting RTL circuits into Coq representatives. Combinational and sequential logic are higher level logic descriptions constructed on top of buses. The keyword *assign* of the *formal-HDL* is used for blocking assignment, while *update* is mainly used for nonblocking assignment. During the blocking assignment the bus value will be updated in the current clock cycle and in the nonblocking assignment the bus value will be updated in the next clock cycle.

Listing 10.3 Signal Operations in Semantic Model

```

Fixpoint assign (a:assignblock) (t:nat) {struct a} :=
  (* Blocking assignment *)
  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} :=
  (* Nonblocking assignment *)
  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.

```

- *Module Definitions:* Module definition/instantiation is critical when dealing with hierarchical circuit structures, but it is never a problem for Verilog (and VHDL), as long as interfacing signals and their timing are correctly defined. Concerning the task of security property verification, however, treating a sub-module as a functional unit by ignoring its internal structure may cause problems. Security properties that are proven for the top level module and all its sub-modules do not

guarantee that the same properties will hold for the whole hierarchical design, where attackers can easily insert hardware Trojans to maliciously modify the interface without violating security properties proven for all modules separately. As a result, the operation of module definition/instantiation should be defined in a way that the details of sub-modules are accessible from the top level module so that any security properties, if proven, remain valid for the whole design. Thus, in PCH we flatten the hierarchical design such that the sub-modules and their interfaces are transparent to the top module. *module* and *module-inst* are key words for module definitions and instantiations. In [53], a new syntax for representing modules is introduced in Coq, which preserves the hierarchical structure and does not require design flattening.

The underlying formal language of the Coq proof assistant, *Gallina*, is based on dependently typed lambda calculus and it defines both types and terms in the same syntactical structure. During the translation process, syntax and semantics of the HDL are translated to *Gallina* using the *formal-HDL*.

10.3.2 Data Protection Through Information Flow Tracking

Among all potential RT-level malicious modifications, sensitive information protection has been a research topic within the cybersecurity domain for decades. Various approaches have been developed, relying on safe languages and software level dynamic checks, to detect buffer overflow attacks and format string vulnerabilities. These methods suffer from the limitation that they either have high false-alarm rates or would cause significant performance overhead. Taking these limitations into consideration, researchers invented new information protection schemes based on hardware–software co-design, where the hardware infrastructure is actively involved in dynamic information flow tracking. This new trend has proven successful in improving detection accuracy and lowering performance overhead, at the cost of hardware level modifications. For example, authors in [54] proposed a dynamic information flow tracking framework with all internal storage elements equipped with a security tag.

Authors in [55] focused on pointer tainting to prevent both control data and non-control data attacks. Besides information flow tracking, the hardware is also enhanced to help prohibit information leakage, such as in the InfoShield architecture [56], which applies restrictions to operations on sensitive data. Similarly, the RIFLE architecture is developed on top of an information flow security (IFS) instruction set architecture (ISA), where all states defined by the base ISA are augmented by labels [57]. More recently, a new software–hardware architecture was developed to support more flexible security policies, either to protect sensitive data [58] or to prevent malicious operations from untrusted third-party OS kernel extensions [59].

Two formal information flow tracking methodologies were also developed, namely *static information flow tracking* [60] and *dynamic information assurance*

[9], which address the challenge of hardware Trojans, capable of leaking sensitive information. These two schemes follow the concept of proof-carrying hardware IP (PCHIP) [8] to enhance the trustworthiness of third-party IP cores.

These two formal methods are particularly geared toward secret information protection, counteracting RTL hardware Trojan attacks in hardware soft IPs, and preventing unintended design backdoors. These two methods differ in complexity and the approach they take to track information in the design. Static information flow tracking scheme is suitable for small designs, and requires less effort in proof development, while dynamic information assurance scheme considers the requirements of more complex and pipelined designs, and needs much more effort in constructing the proofs of security theorems. Designers can adopt these two methodologies based on their requirements.

The *static information flow tracking scheme* and the *dynamic information assurance scheme* are integrated with the PCH IP protection framework and they accept the data secrecy properties as the security property. Furthermore, because the target data secrecy properties are independent of circuit functional specifications, IP vendors may translate the properties from natural language to formal theorems without specifying target circuits and can store the translated formal theorems in a property library for similar designs. The development of a Coq property library and the reuse of theorem-proof contents lowers the burden for IP vendors and stimulates wider acceptance of the proposed proof-carrying based hardware IP protection method. Property formalization and proof generation of both schemes are performed using the Coq proof assistant platform [11].

10.3.2.1 Static Information Flow Tracking Scheme

For the static information flow tracking scheme [60], the IP vendor first designs the circuit based on the functional specifications provided by the IP consumer, in the form of HDL codes. Utilizing a formal semantic model and static information flow tracking rules, the IP vendor then converts the circuit from HDL code into formal logic. In parallel, the IP vendor uses the property formalization constraints to translate the agreed-upon data secrecy properties from natural language to formal theorems. The IP vendor will then try to construct proofs for the translated theorems within the context of the target circuit. Even though the IP vendor is responsible for both circuit design and theorem proving, given a set of well-defined theorems, it is not possible to prove the theorems with a Trojan-infected circuit containing the prohibited information leakage paths. Both formal theorems and their proofs are part of the final deliverable handed to the IP consumer.

Upon receiving the hardware bundle which includes the HDL code and theorem-proof pairs for data secrecy properties, the IP consumer regenerates the formal logic of the original circuit based on the same formal semantic model and static information flow tracking rules. The IP consumer also checks whether the security theorems (in formal language) accurately represent the data secrecy properties (in

natural language). The security theorems and related proofs will then be combined with the regenerated formal logic to pass through an automatic proof checker. If no exceptions are raised, then we claim that the delivered IP core fulfills the agreed-upon data secrecy properties. However, any errors during the proof checking process warn the user that malicious circuits (or design flaws) may exist in the IP core, making it violate the data secrecy properties.

10.3.2.2 Dynamic Information Assurance Scheme

The static scheme is effective in detecting data leakage caused by hardware Trojans and/or design faults. It also requires less effort for constructing proofs. However, the static scheme is limited by the fact that it can only check circuit trustworthiness statically. To overcome this shortcoming of the static scheme and to achieve high-level hardware Trojan detection capability, a dynamic information assurance scheme is later developed [9].

This dynamic scheme supports various levels of circuit architectures, ranging from low-complexity single-stage designs to large-scale deeply pipelined circuits. Similar to the static scheme, the dynamic scheme also focuses on circuits dealing with sensitive information, such as cryptographic designs, because it sets data secrecy as the primary goal and tries to prevent illegal information leakage from IP cores. Within the dynamic scheme, all signals are assigned values indicating their sensitivity levels. These values will be updated after each clock cycle according to their original values and the updating rules defined by the signal sensitivity transition model. Since the sensitivities of all circuit signals are managed in a sensitivity list, two sensitivity lists are of interests for data secrecy protection: the initial sensitivity list and the stable sensitivity list. The initial sensitivity list reflects the circuit status after initialization or powered-on mode when only some input signals contain sensitive information, such as plaintext and encryption keys. The stable sensitivity list, on the other hand, indicates the circuit status when all internal/output signals are of fixed sensitivity levels.

Similar to the static scheme, IP vendor will also translate the agreed-upon data secrecy properties from natural language to property generation functions, which can later help to generate formal theorems. Meanwhile, different from the static scheme, IP consumers will first check the contents of the initial signal sensitivity list and the stable signal sensitivity list, which represent the circuit's initial secrecy status and the stabilized status, respectively. The validity of the initial list is checked to ensure that sensitivity levels are appropriately assigned to all input/output/internal signals. The circuit's stable sensitivity status contains complete information of the distribution of sensitive information across the whole circuit, so the stable list will then be carefully evaluated to detect any backdoors that may leak sensitive information. After both signal sensitivity lists pass the initial checking, IP consumers proceed to the next step of proof checking. A "PASS" output from the automatic proof checker provides evidence that HDL codes do not contain any

malicious channels to leak information. However, a “FAIL” result is a warning that some of the data secrecy properties are breached in the delivered IP cores.

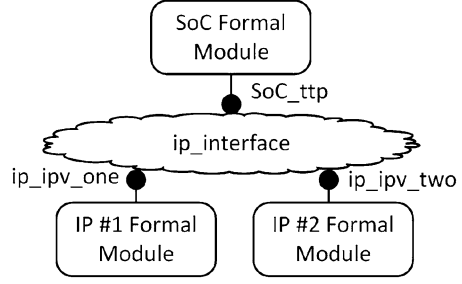
10.3.3 *Hierarchy Preserving Verification*

The above mentioned PCH frameworks treat the whole circuit design as one module and prove security properties on them [8–10, 60]. That is, the entire design is first flattened before translating the HDL code of the design into the formal language and proving it with respect to formal security theorems. Design flattening increases the complexity of translating HDL code into *Gallina*. It also adds to the risk of introducing errors during the code conversion process. Due to flattening, a verification expert has to go through the entire design in order to construct proofs of security theorems, which significantly increases the workload for design verification. Also, any updates to the HDL code will significantly change the proof for the same security property. Moreover, the PCH framework prevents proof reuse, i.e., proofs constructed for one design cannot be used in another design even though the same IP modules are used. All of these limitations prohibit a wide usage of the PCH framework in modern SoC designs.

To overcome these limitations, the *Hierarchy-preserving Formal Verification* (HiFV) framework is developed for verifying security properties on SoC designs in [53]. The HiFV framework is an extension of the PCH framework. In the HiFV framework, the design hierarchy of the SoC is preserved and a distributed approach is developed for constructing proofs of security properties. In the distributed approach, security properties are divided into sub-properties in such a way that each sub-property corresponds to an IP module of the SoC. Proofs are then constructed for these sub-properties and the security property for the SoC design is proven through the integration of all proofs from sub-properties. Similar to PCH, the HiFV framework requires semantic translation of the HDL code and informal security properties to *Gallina*. For proving the trustworthiness of the HDL code of the SoC, Hoare-logic is used. Similar to other PCH methods, the HiFV framework is carried out in Coq.

As mentioned earlier, before building the formal model for the SoC system, the syntax and semantics should be defined and then shared by any parties who need to design or check the proof. In addition, *interface* and *module* are incorporated in the *formal-HDL* to preserve the design hierarchy of the SoC. That is, in order to make distributed proof construction applicable on hierarchical designs, an *interface* is developed in the HiFV framework which makes the verification process flexible and efficient for the proof writer. To define the *interface*, information about each IP and its corresponding I/O are needed, such as the name, number, and data type. By using the *interface*, the management of the plenty of formal modules would be much easier in the verification house side. The structure of the *interface* is shown in Fig. 10.4. Through the *interface*, an IP module within an SoC can access other

Fig. 10.4 Structure of the SoC with interface



modules such as IP #1 Formal Module or IP #2 Formal Module in the figure. The *ip_ipv_one*, *ip_ipv_two*, and *SoC_ttp* are the name of the corresponding interfaces.

The distributed proof construction process uses Hoare-logic, where the trustworthiness of the SoC *formal-HDL* code is determined by ensuring that the code operates following the constraints of the pre-condition and the post-condition. The pre-condition of the *formal-HDL* code is the initial configuration of the design and the post-condition is the security theorem. Meanwhile, in order to overcome the scalability issue, a distributed proof construction approach is developed, which is dedicated for SoC designs with hierarchical structures. This approach makes the HiFV framework scalable by reducing the time required for proof construction, proof correction, and proof modification.

In the HiFV framework, the translated HDL code of the SoC, formal security theorems, and the initial configuration of the design is represented as a Hoare Triple (Eq. (10.1)).

$$(\phi)CoqEquivalentCode_SoC(\psi) \quad (10.1)$$

In this equation, ϕ is the pre-condition corresponding to the initial configuration of the design. The translated HDL code of the SoC design hierarchy in *Gallina* is given by *CoqEquivalentCode_SoC*. In the process of translation, modules in the SoC HDL code, which correspond to IPs from different vendors, are also translated. The post-condition is given by ψ which represents the formal security theorem.

The security theorem is divided into lemmas (Eq. (10.2)), which are post-conditions for individual IP modules. In Eq. (10.2), post-condition for IPs (lemmas) are represented as ψ_i ($1 \leq i \leq n$), $n = \text{maximum number of IP modules required to prove the security theorem}$ and ψ is the security theorem. These lemmas correspond to those IP modules that are required to satisfy the security theorem.

$$\psi := \psi_1 \wedge \psi_2 \cdots \wedge \psi_n \quad (10.2)$$

Similarly, the pre-condition of the SoC design (ϕ) and the translated HDL code of the SoC design (*CoqEquivalentCode_SoC*) are divided according to Eqs. (10.3) and (10.4). Here, (ϕ_i) and (*CoqEquivalentCode_IPmodule_i*) ($1 \leq i \leq n$) represent the pre-conditions and translated HDL code of each IP module of the

SoC, respectively.

$$\phi := \phi_1 \wedge \phi_2 \cdots \wedge \phi_n \quad (10.3)$$

$$\begin{aligned} \text{CoqEquivalentCode_SoC} &:= \text{CoqEquivalentCode_IPmodule_1} \\ &\wedge \text{CoqEquivalentCode_IPmodule_2} \dots \quad (10.4) \\ &\wedge \text{CoqEquivalentCode_IPmodule_n} \end{aligned}$$

The HDL code of the IP core is certified to be trustworthy only if it satisfies the pre-condition and the post-condition. When all the modules of IP cores satisfy the post-conditions (lemmas), we can state that the security theorem is proven for the SoC design.

$$(\phi_i) \text{CoqEquivalentCode_IPmodule_i}(\psi_i) \quad (10.5)$$

The distributed approach of proof construction also enables proof reuse. After certifying the trustworthiness of each IP core of the SoC, the proofs can be stored in a library and accessed by the trusted third party (TTP) verification house for verification of other SoC designs in which the same IP modules are used and similar security properties are applied. In this way the HiFV framework further reduces the time for verifying complex designs.

As a summary, in this approach, the previously developed PCH framework is extended into the SoC design flow and largely simplified the process for proving security properties through a hierarchical proof construction procedure. To reduce the workload for circuit verification, the proof of the security properties for individual IPs can be encapsulated and reused in proving security properties at the SoC level. Also, in the hierarchical framework, the amount of updates that need to be done to existing proofs when SoC designs are modified is significantly lowered. The developed HiFV framework paves the way for large-scale circuit design security verification.

10.3.4 Integrating Theorem Prover and Model Checker

Although the HiFV hierarchical approach improves scalability of the previous PCH method, it still suffers from the challenge of proof construction. Meanwhile, model checkers such as Cadence IFV cannot be used for verifying systems with large state space because of the space explosion problem. As the number of state variables (n) in the system increases, amount of space required for representing the system and the time required for checking the system increases exponentially ($T(n) = 2^{O(n)}$) (Fig. 10.5).

To further overcome the scalability issue and to verify a computer system, an *integrated formal verification framework* (see Fig. 10.6) is introduced in [61], where

Fig. 10.5 Security specification (ϕ) decomposed into lemmas

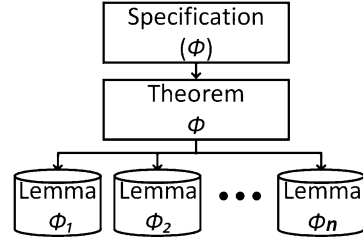
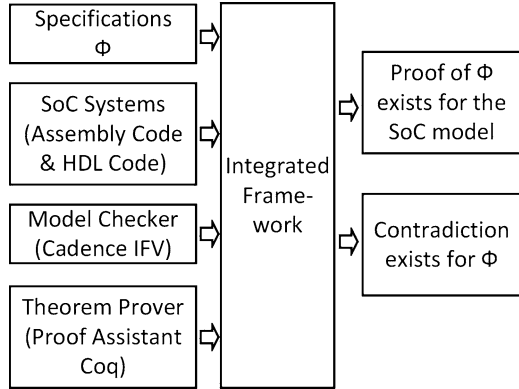


Fig. 10.6 Integrated formal verification framework



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.

Some efforts have been made to combine theorem provers with model checkers for verification of hardware and software systems [62, 63]. These methods try to overcome the scalability issue of both techniques. That is, both model checkers and theorem provers cannot scale well to formally verify large-scale circuit designs. Some of the popular theorem provers such as higher order logic (HOL Light) and prototype verification system have integrated model checkers. These tools have been used for functional verification of hardware systems. For the first time, this combined technique has been extended toward verification of security properties on third-party IP cores and SoCs [61].

In the integrated framework, the hardware design, represented in a hardware description language (HDL), and the assembly level instructions of a vulnerable program, is first translated to *Gallina*, which is similar to other PCH methods. 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. 10.5) based on sub-modules. These lemmas are then represented in the Property Specification Language (PSL) specification language and are called sub-specifications. Subsequently, the Cadence IFV verifies the sub-modules against the corresponding sub-specifications. Sub-modules 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 rare dependency relationship with each other.

The HDL code of a large design consists of many such sub-modules. If the sub-modules satisfy the sub-specifications, lemmas are considered to be proved. Checking the truth value of the sub-specifications with a model checker eliminates the effort required for proving the lemmas and translating the sub-modules to Coq. Upon proving these sub-modules, Hoare-logic is then used to combine proof of these lemmas to prove the security theorem of the entire system in Coq.

The integrated formal verification framework helps in protecting a large-scale SoC design from malicious attacks. Given that an interactive theorem prover (e.g., Coq) requires lots of effort to manually verify the design and that a model checker suffers from scalability issues, these two techniques are combined together through the decomposition of the security property as well as the design in such a way that the model checker can verify those sub-modules which have much less state variables. Consequently, the amount of effort required for translating the design from HDL to *Gallina* and proving the security theorem in Coq is reduced.

10.4 Conclusion

In this chapter, we explain our interactive theorem proving based PCH approach for security property verification of hardware IP cores. We also describe application of the framework for preventing information leakage from soft IPs. To overcome scalability and reusability issues of original PCH method, a design hierarchy preserving scheme was then introduced that incorporates both model checking and interactive theorem proving for verification.

Acknowledgements This work has been partially supported by the National Science Foundation (NSF-1319105), the Army Research Office (ARO W911NF-16-1-0124), and Cisco.

References

1. M. Banga, M. Hsiao, Trusted RTL: Trojan detection methodology in pre-silicon designs, in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2010), pp. 56–59
2. A. Waksman, M. Suozzo, S. Sethumadhavan, FANCI: identification of stealthy malicious logic using boolean functional analysis, in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS'13* (2013), pp. 697–708
3. D. Sullivan, J. Biggers, G. Zhu, S. Zhang, Y. Jin, FIGHT-metric: Functional identification of gate-level hardware trustworthiness, in *Design Automation Conference (DAC)* (2014)
4. N. Tsoutsos, C. Konstantinou, M. Maniatakis, Advanced techniques for designing stealthy hardware trojans, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)

5. M. Rudra, N. Daniel, V. Nagoorkar, D. Hoe, Designing stealthy trojans with sequential logic: A stream cipher case study, in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE* (2014)
6. S. Drzevitzky, U. Kastens, M. Platzner, Proof-carrying hardware: Towards runtime verification of reconfigurable modules, in *International Conference on Reconfigurable Computing and FPGAs* (2009), pp. 189–194
7. S. Drzevitzky, M. Platzner, Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach, in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip* (2011), pp. 1–8
8. E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: a pathway to trusted module acquisition. *IEEE Trans. Inf. Forensics Secur.* **7**(1), 25–40 (2012)
9. Y. Jin, B. Yang, 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
10. Y. Jin, Y. Makris, A proof-carrying based framework for trusted microprocessor IP, in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013), pp. 824–829
11. INRIA, The Coq proof assistant (2010), <http://coq.inria.fr/>
12. F. Wolff, C. Papachristou, S. Bhunia, R.S. Chakraborty, Towards Trojan-free trusted ICs: problem analysis and detection scheme, in *IEEE Design Automation and Test in Europe* (2008), pp. 1362–1365
13. M. Hicks, M. Finnicum, S.T. King, M.M.K. Martin, J.M. Smith, Overcoming an untrusted computing base: detecting and removing malicious hardware automatically, in *Proceedings of IEEE Symposium on Security and Privacy* (2010), pp. 159–172
14. C. Sturton, M. Hicks, D. Wagner, S. King, Defeating UCI: building stealthy and malicious hardware, in *2011 IEEE Symposium on Security and Privacy (SP)* (2011), pp. 64–77
15. X. Zhang, M. Tehranipoor, Case study: detecting hardware trojans in third-party digital ip cores, in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011), pp. 67–70
16. 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
17. X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, P. Mishra, Pre-silicon security verification and validation: a formal perspective, in *Proceedings of the 52Nd Annual Design Automation Conference, DAC'15* (2015), pp. 145:1–145:6
18. F.M. De Paula, M. Gort, A.J. Hu, S.J. Wilton, 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, New York, 2008), p. 5
19. S. Drzevitzky, Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration, in *2010 International Conference on Field Programmable Logic and Applications (FPL)* (2010), pp. 255–258
20. J. Rajendran, V. Vedula, R. Karri, Detecting malicious modifications of data in third-party intellectual property cores, in *Proceedings of the Annual Design Automation Conference, DAC '15* (ACM, New York, 2015), pp. 112:1–112:6
21. J. Harrison, Floating-point verification, in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, ed. by J. Fitzgerald, I.J. Hayes, A. Tarlecki. Lecture Notes in Computer Science, vol. 3582 (Springer, Berlin, 2005), pp. 529–532
22. S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY), ed. by D. Kapur. Lecture Notes in Artificial Intelligence, vol. 607 (Springer, Berlin, 1992), pp. 748–752
23. D. Russinoff, M. Kaufmann, E. Smith, R. Sumners, Formal verification of floating-point RTL at AMD using the ACL2 theorem prover, in *Proceedings of the 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, Paris, France* (2005)
24. J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, A. Platzer, How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *Int. J. Softw. Tools Technol. Transfer* **18**, 67–91 (2016)

25. A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant* (MIT Press, Cambridge, 2013)
26. U. Norell, Dependently typed programming in Agda, in *Advanced Functional Programming* (Springer, Berlin, 2009), pp. 230–266
27. R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, S.F. Smith, *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, Upper Saddle River, 1986)
28. L.C. Paulson, Isabelle: the next 700 theorem provers, in *Logic and Computer Science*, vol. 31 (Academic Press, London, 1990), pp. 361–386
29. E.M. Clarke, O. Grumberg, D. Peled, *Model Checking* (MIT press, Cambridge, 1999)
30. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software verification with blast, in *Model Checking Software*, (Springer, Berlin, 2003), pp. 235–239
31. J. O’Leary, X. Zhao, R. Gerth, C.-J.H. Seger, Formally verifying ieeec compliance of floating-point hardware. *Intel Technol. J.* **3**(1), 1–14 (1999)
32. M. Srivas, M. Bickford, Formal verification of a pipelined microprocessor. *IEEE Softw.* **7**(5), 52–64 (1990)
33. T. Kropf, *Introduction to Formal Hardware Verification* (Springer, Berlin, 2013)
34. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: formal verification of an os kernel, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles* (ACM, New York, 2009), pp. 207–220
35. S. Chaki, E.M. Clarke, A. Groce, S. Jha, H. Veith, Modular verification of software components in C. *IEEE Trans. Softw. Eng.* **30**(6), 388–402 (2004)
36. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M.F. Kaashoek, N. Zeldovich, Using crash hoare logic for certifying the fscq file system, in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP’15* (ACM, New York, 2015), pp. 18–37
37. M. Vijayaraghavan, A. Chlipala, N. Dave, Modular deductive verification of multiprocessor hardware designs, in *Computer Aided Verification* (Springer, Cham, 2015), pp. 109–127
38. A.A. Mir, S. Balakrishnan, S. Tahar, Modeling and verification of embedded systems using cadence SMV, in *2000 Canadian Conference on Electrical and Computer Engineering*, vol. 1 (IEEE, New York, 2000), pp. 179–183
39. M. Kwiatkowska, G. Norman, D. Parker, Prism: probabilistic symbolic model checker, in *Computer Performance Evaluation: Modelling Techniques and Tools* (Springer, Berlin, 2002), pp. 200–204
40. G.J. Holzmann, The model checker spin. *IEEE Trans. Softw. Eng.* **23**(5), 279 (1997)
41. D. Beyer, M.E. Keremoglu, Cpathchecker: a tool for configurable software verification, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 184–190
42. A. David, K. G. Larsen, A. Legay, M. Mikučionis, Z. Wang, Time for statistical model checking of real-time systems, in *Computer Aided Verification* (Springer, Berlin, 2011), pp. 349–355
43. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in *Computer Aided Verification*, (Springer, Berlin 2000), pp. 154–169
44. C. Baier, J. Katoen, *Principles of Model Checking* (MIT Press, Cambridge, 2008)
45. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using sat procedures instead of BDDs, in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference* (ACM, New York, 1999), pp. 317–320
46. R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.* **24**(3), 293–318 (1992)
47. R.E. Bryant, Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* **100**(8), 677–691 (1986)
48. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: an opensource tool for symbolic model checking, in *Computer Aided Verification* (Springer, Berlin, 2002), pp. 359–364

49. E. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.* **19**(1), 7–34 (2001)
50. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, Y. Zhu, Bounded model checking *Adv. Comput.* **58**, 117–148 (2003)
51. S. Qadeer, J. Rehof, Context-bounded model checking of concurrent software, in *Tools and Algorithms for the Construction and Analysis of Systems* (Springer, Berlin, 2005), pp. 93–107
52. 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
53. X. Guo, R.G. Dutta, Y. Jin, Hierarchy-preserving formal verification methods for pre-silicon security assurance, in *16th International Workshop on Microprocessor and SOC Test and Verification (MTV)* (2015)
54. G.E. Suh, J.W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI* (2004), pp. 85–96
55. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, R. Iyer, Defeating memory corruption attacks via pointer taintedness detection, in *Proceedings. International Conference on Dependable Systems and Networks, 2005. DSN 2005* (2005), pp. 378–387
56. W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, J. Yang, Infoshield: a security architecture for protecting information usage in memory, in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006* (2006), pp. 222–231
57. N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, D. August, RIFLE: an architectural framework for user-centric information-flow security, in *37th International Symposium on Microarchitecture, 2004. MICRO-37 2004* (2004), pp. 243–254
58. Y.-Y. Chen, P. A. Jamkhedkar, R.B. Lee, A software-hardware architecture for self-protecting data, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS'12* (2012), pp. 14–27
59. Y. Jin, D. Oliveira, Extended abstract: trustworthy SoC architecture with on-demand security policies and HW-SW cooperation, in *5th Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-5)* (2014)
60. Y. Jin, Y. Makris, Proof carrying-based information flow tracking for data secrecy protection and hardware trust, in *IEEE 30th VLSI Test Symposium (VTS)* (2012), pp. 252–257
61. X. Guo, R.G. Dutta, P. Mishra, 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.
62. S. Berezin, *Model checking and theorem proving: a unified framework*. Ph.D. Thesis, SRI International (2002)
63. P. Dybjer, Q. Haiyan, M. Takeyama, Verifying haskell programs by combining testing, model checking and interactive theorem proving. *Inf. Softw. Technol.* **46**(15), 1011–1025 (2004)

Quantifying Trust in Autonomous System Under Uncertainties

Raj Gautam Dutta, Xiaolong Guo, and Yier Jin

Department of Electrical and Computer Engineering, University of Central Florida

{rajgautamdutta, guoxiaolong}@knights.ucf.edu, yier.jin@eecs.ucf.edu

Abstract—Over the years, autonomous systems have entered almost all the facets of human life. Gradually, higher levels of autonomy are being incorporated into cyber-physical systems (CPS) and Internet-of-things (IoT) devices. However, safety and security has always been a lurking fear behind adoption of autonomous systems such as self-driving vehicles. To address these issues, we develop a framework for quantifying trust in autonomous system. This framework consist of an estimation method, which considers effect of adversarial attacks on sensor measurements. Our estimation algorithm uses a set-membership method during identification of safe states of the system. An important feature of this algorithm is that it can distinguish between adversarial noise and other disturbances. We also verify the autonomous system by first modeling it as networks of priced timed automata (NPTA) with stochastic semantics and then using statistical probabilistic model checking to verify it against probabilistic specifications. The verification process ensures that the autonomous system behave in accordance to safety specifications within a probabilistic threshold. For quantifying trust on the system, we use confidence results provided by the model checking tool. We have demonstrated our approach by using a case study of adaptive cruise control system under sensor spoofing attacks.

I. INTRODUCTION

Automation is being increasingly introduced into every man-made system. Currently, most of the day-to-day applications are *semi-autonomous* as they require human intervention in achieving desired goals. However, the thrust to achieve trustworthy autonomous systems, which can attain goals independently in the presence of significant uncertainties and for long periods of time without any human intervention, has always been enticing. Significant progress has been made in the avenues of both software and hardware for meeting these objectives. Currently, space vehicles are autonomous and there are prototypes of self-driving autonomous vehicles [1]. However, technological challenges still exist and particularly in terms of decision making under uncertainty. In an autonomous system, uncertainties can arise from the operating environment, adversarial attacks, and from within the system. As a result of these ambiguities, human-beings lack trust in these systems and hesitate to use take them for day-to-day use. For an autonomous system, *trust* is defined as the ability of the system to successfully carry out a task, at a particular time, and in a situation characterized by vulnerability and uncertainty [2]. To build trust in an autonomous system, manufacturers have to certify them. Certification is a formal means by which regulators measure expected performance of different components of an autonomous system. A generic autonomous system consist of (i) perception unit, which comprises of

sensing system and its data processing software, (ii) control and decision making unit, which decides future actions of the autonomous system, and (iii) execution unit, which carry out actions provided by the control and decision making unit [3]. To measure trust, one has to have quantitative value of following traits of the system (i) performance, which includes competence, reliability, and robustness, (ii) transparency of control and decision-making unit, and (iii) security vulnerabilities [4]. These factors should be measured both in certain and uncertain environments. Thus, design and certification of autonomous systems such as self-driving cars, which can withstand uncertain conditions is of utmost importance.

Robust control algorithms are being used to make an autonomous system resilient against environmental and in-system uncertainties. However, these methods fail to extend toward uncertainties created by adversarial attacks [5]–[12]. As such we use a statistical probabilistic formal verification method for certifying autonomous system as trustworthy and show its resiliency against attacks. In our approach, we assume that the sensors of the system are being compromised by an attacker via spoofing or jamming attacks. As a result, the control and decision making unit of the system get degraded sensor data, which makes the system behave abnormally. We also assume that the control and decision making unit, the actuators, and the diagnostics and fault management sub-system of the execution unit cannot be modified by an attacker. However, these units/sub-systems may be affected by internal and environmental disturbances. With these assumptions, we first estimate a set of safe states for the vehicle. This estimation is done at the diagnostics and fault management sub-system of the execution unit. Then, the safe states are used to monitor the outputs of the control and decision unit. When a suspicious state is detected by this sub-system, it alerts the reactive control sub-system, which is also in the execution unit. This system takes the necessary actions to keep the autonomous system within the realms of safety. As the reactive control sub-system is prone to sensor noise and environmental disturbance, we verify whether it satisfies the safety requirements with certain probability value. Based on the verification results we assign performance measurements to different components of the execution unit of the autonomous system. As 100% certification of autonomous system is nearly impossible, we adopt a probabilistic rating mechanism to assign confidence values to different units of the system. Thus, we quantify trust in the system by adding ratings of all these components. We demonstrate our approach on an autonomous ground vehicle

under sensor spoofing attacks.

II. ATTACK MODEL

Attack on an autonomous vehicle can be carried out either via physical access [5], [6] or remotely [7]–[11]. In our paper, we consider remote attacks, as attackers getting physical access of the vehicles internal operation may be an infeasible assumption. Remote exploitation is possible via a wide range of attack vectors present in the vehicle such as telematics system, Wi-Fi, Bluetooth, in-car apps, remote keyless entry, Tire Pressure Monitoring System (TPMS), radio data system, and Dedicated Short Range Communications (DSRC), developed for vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2X) communications [11]. These attack vectors uses various wireless communication wavelengths to get access of the in-vehicle network, which comprises of Controller Area Network (CAN) buses, Local Interconnect Network (LIN) buses, Media Oriented System Transports (MOST) buses, and FlexRay buses [13]. As CAN buses communicate with all the Electronic Control Units (ECUs) of the vehicle, most of the attacks are targeted toward it [5], [7], [11]. Control of the CAN bus allows the attacker to take control of most of the functionalities of the vehicle.

Another form of remote attack target data acquisition unit of the vehicle, which comprises of sensors [8]–[10], [12]. An autonomous vehicle has many sensors (radar, lidar, Global Positioning System (GPS), camera, ultrasonic, and inertial measurement unit (IMU)) for collecting data of its internal and external environment [14], [15]. Compromising the data gathered by these sensors can impact the decisions made by the motion control unit of the vehicle.

To degrade sensor data quality, an attacker has to carry out spoofing or jamming attack remotely. Following the attack models described in [8]–[10], [12], we assume that the attacker target the hardware layer (external sensors) and has limited knowledge of sensors firmware or software. We also assume that they have limited resources to carry out the remote spoofing attack. Furthermore, the attack can be mounted on the target vehicle while its stationary or in motion. By carrying out these attacks, the attacker intends to cause physical damage to the vehicle.

Based on the remote spoof attacks carried out in [8]–[10], [12], we can state that the lidar, GPS, radar, camera, and ultrasonic sensors of the vehicle can be compromised. We assume that the sensor fusion unit (shown in Figure 1), which gets processed data of compromised sensors, cannot rectify the malicious data. Other assumptions are that the adversary does not have access to the diagnostic and fault management sub-system of the vehicle and they cannot directly modify the control and decision unit.

III. RELATED WORK

Verification of autonomous ground vehicles has been receiving increasing attention over the years [16]–[19]. Stursberg et al. [16] modeled the cruise control system of vehicle using hybrid automata with nonlinear continuous dynamics

and polyhedral guard and invariant sets. Then, they used counterexample-guided verification approach to ensure the cruise control system of two cars in one lane was operating correctly to avoid collision. However, their method could not be scaled to arbitrary number of cars. Loos et al. [17] used quantified hybrid program for modeling distributed car control system, where every car was controlled by adaptive cruise control. They used the automated theorem prover - KeYmaera - to prove that the control model satisfied the collision avoidance protocol for arbitrary number of cars on a street. Wongpiromsarn et al. [18] proposed the framework, called periodically controlled hybrid automata, for modeling control systems in which inputs to actuators were given after a fixed time unit. They verified safety and progress properties of the planner-controller sub-system of an autonomous ground vehicle using sum of square decomposition and semi-definite programming. Althoff et al. [19] used reachability analysis for safety verification of evasive maneuvers of autonomous ground vehicle with constant velocity and under uncertainties.

However, these verification methods do not consider an autonomous ground vehicle under adversarial attacks [16]–[19]. To address this scenario, many attack detection and prevention methods were developed for ground vehicles [20]–[24]. Zheng et al. [20] analyzed security and safety of in-vehicle control system and vehicle-to-vehicle communication network using hybrid modeling, formal verification, and automated synthesis techniques. They considered cooperative adaptive cruise control (CACC) as their test case and used an automated theorem prover to prove a collision avoidance property under time delay attack on vehicle communication network. Mundhenk et al. [21] used probabilistic model checker - PRISM, for quantifying security vulnerability (in terms of confidentiality, integrity and availability) of automotive architecture at design-time. Before using the model checker, they transformed the automotive communication architecture which consists of ECUs, heterogeneous bus system, and intra-vehicle networks into Continuous-Time Markov Chain with transitions given by exploitability rate and patching rate. However, they did not consider safety of an operating vehicle under adversarial attack. Fawzi et al. [23] and Tiwari et al. [22] proposed detection and prevention methods for an autonomous ground vehicle under sensor spoofing attacks. In [23], secure states were first estimated of a linear control system under sensor spoofing attacks using a decoder. Then, a linear static controller was designed based on the set of secure states. However, this method did not distinguish between noise and attack. Moreover, they assumed that not all sensors can be attacked at the same time. In [22], a learning mechanism was used to construct a set of invariants called “safety envelope”, from collected sensor data. Then, the system was monitored to check if it operated within the safety envelope. In case of a violating, an alarm was raised depending on whether it was an attack or noise. The learning algorithm did not require model of the dynamical system and could distinguish between noise and attack. Unlike the software approaches, Wolf et al. [24] proposed a vehicular hardware security module for protecting

in-vehicle ECUs and their communications.

In this paper, we use networks of priced timed automata (NPTA) with stochastic semantics for modeling an autonomous vehicle under adversarial attack. We consider all sensors of the vehicle are spoofed remotely after determining the initial state of the vehicle. Moreover, we decouple attack from environmental and system noise in our vehicular dynamics model.

IV. FRAMEWORK FOR QUANTIFYING TRUST IN AUTONOMOUS SYSTEM

In this section, we describe our framework for quantifying trust and its various components. At the beginning, we briefly describe the functional software architecture of the autonomous ground vehicle under consideration. Then, we outline the requirements for safe state estimation. Subsequently, we describe the method of our choice for modeling autonomous vehicle under adversarial attack. We verify the controller of the autonomous vehicle against probabilistic safety specification. The confidence interval and confidence percentage obtained from the verification process helps us in quantifying trust on the vehicle.

A. Functional Software Architecture of Autonomous Ground Vehicle

The software architecture of the autonomous ground vehicle (AV) under consideration (See Figure 1) is similar to the ones in [3], [15]. The AV has six on-board sensors, global positioning system (GPS), inertial measurement unit (IMU), cameras, radars, lidars, and ultrasonic. Raw data from these sensors are processed into a form which can be understood by the hierarchical sensor fusion unit. The multi-sensor data fusion algorithm of this unit uses different combination of on-board sensor data for object detection and classification. This data is further combined with information from V2V/V2X communications and on-board maps to enhance perception of the vehicle beyond the capabilities of traditional sensors. The output of the sensor fusion unit is a 3-D map of the environment, which is used by the control and decision unit along with information of the current state of the vehicle to generate an optimal obstacle free trajectory¹. Subsequently, the trajectory execution sub-system uses propulsion, steering, and braking components to execute the trajectory generated by the control and decision unit. The diagnostic and fault management (D&F) sub-system estimates safe states of the vehicle. It analyzes the generated trajectory to determine whether the vehicle is operating within the safety domain. In case of any deviation from the expected behavior, it issues command to the reactive control sub-system, which immediately responds by taking action such as braking to avoid collision. The reactive control sub-system operates in parallel with the trajectory execution unit and when a threat is identified, its output overrides normal operation of the vehicle.

¹Note: In Figure 1, the current state refer to a trajectory generated in a previous time step and future state refer to a new trajectory generated by the control and decision unit after analyzing information from sensor fusion unit.

In this paper, we consider an adaptive cruise control unit, whose function is to drive the vehicle in an intended trajectory and avoid collision with other vehicles on the road.

B. Safe State Estimation

We consider an autonomous ground vehicle in which the D&F sub-system monitors the trajectory generated by the control and decision unit to ensure the vehicle operates safely. To perform monitoring, the D&F sub-system estimate a set of safe states based on the knowledge of overall system dynamics and disturbances arising from internal components and external environment.

We first construct an abstract interval around true sensor measurements using *set-membership* method [25]. Then, we estimate the safe states of the system from these sensor values. The abstract interval is constructed by using manufacturers specifications about precision and accuracy of sensors, as well as physical limitations such as sampling jitter and synchronization error. If these information's are not present, then an approximate abstract interval is constructed based on empirical data of initial sensor measurements. Smaller the size of the interval, higher is the confidence on the sensor measurements. Anything outside this abstract interval are bad sensor values (due to adversarial attack) and the trajectory of the system should never enter the bad states (measured from the bad sensor values). Information of safe and bad states enables the D&F sub-system to monitor the system.

C. Modeling of Autonomous Vehicle under Adversarial Attacks

Modeling of systems involving interaction of discrete and continuous dynamics (such as in autonomous ground vehicles) has been successfully carried out using hybrid system [16], [17]. Over the years, several computational frameworks have been developed for representing hybrid system such as hybrid program, hybrid automata (extends the traditional finite state automata by incorporating continuous dynamics), networks of priced timed automata (NPTA), and hybrid petri net. However, these frameworks did not allow randomness in the design. Consequently, the theory of stochastic hybrid system and its modeling formalisms such as Piecewise Deterministic Markov Processes (PMDP), stochastic hybrid automata, and Switched Diffusion Processes (SDP), were developed for capturing variability/uncertainty in the system and performing probabilistic analysis. This enabled modeling and analysis of uncertain autonomous systems.

In this paper, we use NPTA with stochastic semantics [26] for modeling vehicular system dynamics and representing internal/external disturbances including adversarial attack. Using this approach we model the adaptive cruise control (ACC) system of the ground vehicle, which is shown in our case study.

D. Verification of Autonomous Ground Vehicle

When the D&F sub-system of the execution unit (See Figure 1) identifies a suspicious behavior (state), it issues command

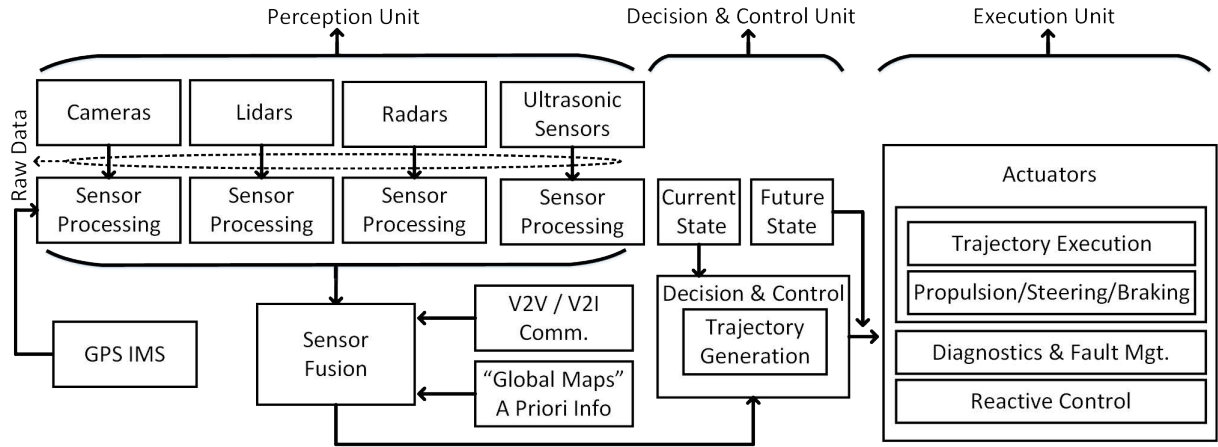


Fig. 1: Functional software architecture of autonomous ground vehicle.

to the reactive control sub-system, which then carries out the actions to prevent fatal scenarios. To ensure the reactive control sub-system behave correctly, we verify that it carries out the safe actions within a probability threshold. As specifications for a controller may be violated once in a while due to internal or external disturbances, we resort to a probabilistic method of verification. With the help of this approach, we can generate probabilistic guarantees on whether the system's state will transition to a safe state within certain time interval.

Model checking is one of the approach for verifying stochastic hybrid systems [27]. In this approach, a stochastic model $\mathcal{M}$ of a system with an initial state s_0 is expressed as a transition system, behavioral specification such as safety property ϕ is expressed using bounded linear temporal logic, $\theta \in [0, 1]$ is a probability threshold, and $\bowtie \in \{>, <, \geq, \leq\}$. Using these notations, the probabilistic model checking problem can be formally stated as $\mathcal{M}, s_0 \models P_{\bowtie\theta}(\phi)$. The underlying algorithm of this technique explores the state-space of the model to decide whether $\mathcal{M}$ satisfies ϕ with a probability $>, <, \geq$, or $\leq$ to a certain threshold θ . 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.

In our paper, we use Statistical probabilistic Model Checking (SMC) to verify the reactive control sub-system of the autonomous ground vehicle against probabilistic safety specifications (generated from knowledge of system dynamics). The specifications used in this tool are time bounded. Such a model checking approach combines randomized simulation (i.e., Monte Carlo simulation), statistical analysis, and model checking, and it is scalable to large designs. However, using large number of digital clocks (we use one clock) in the system model effects efficiency and scalability of this approach.

E. Quantification of Trust in the Autonomous Ground Vehicle

Our probabilistic approach for safety verification of the autonomous ground vehicle requires running the system for certain time units to obtain results. Based on the number of runs, we obtain an overall estimate of the correctness of the

system. The results are represented in the form of confidence interval $[\theta - \epsilon, \theta + \epsilon]$ (where, $\theta \in [0, 1]$ is the probability assigned to the safety specification and ϵ is an approximation parameter) and confidence $(1 - \delta)$ (where δ is the confidence parameter). The value of confidence represents the probability of the specification satisfying the system model within the confidence interval for the runs of the system. Higher the value of confidence within the confidence interval, more is the likelihood of the vehicle system model satisfying the safety specification. We write specifications for all the units of the autonomous ground vehicle under consideration and obtain their respective confidence values for a fixed confidence interval. Based on these values, we assign overall trust to the vehicle.

V. CASE STUDY

As a case study, we consider adaptive cruise control (ACC) of an autonomous ground vehicle. ACC relies on data from camera, lidar, or radar to measure distance (d) between a follower car (fc) and a leader car (lc) on a lane. From distance measurement, velocity and acceleration of the fc is calculated. The safety requirement for collision avoidance between fc and lc is that the distance between them should always be greater than zero i.e. $d > 0$. Under normal operation of the vehicle, the control and decision unit achieves this condition by adjusting the acceleration of fc . However, under spoofing attack, the sensors will produce wrong measurements of distance d . As a result, the control and decision unit will issue wrong commands to the execution unit, resulting in a possible vehicular collision. To mitigate this issue, diagnostics and fault management sub-system should detect this threat and alert the reactive control sub-system. Subsequently, the later unit should override normal operation and issue the brake command.

In our example, we consider safe distance d between fc and lc is 20 m. When d is < 20 m between the vehicles, the reactive controller of fc should brake within 600 msec to avoid collision. We assume the adversary spoofs the lidar signals, thereby making the sensor detect fake vehicles at a distance

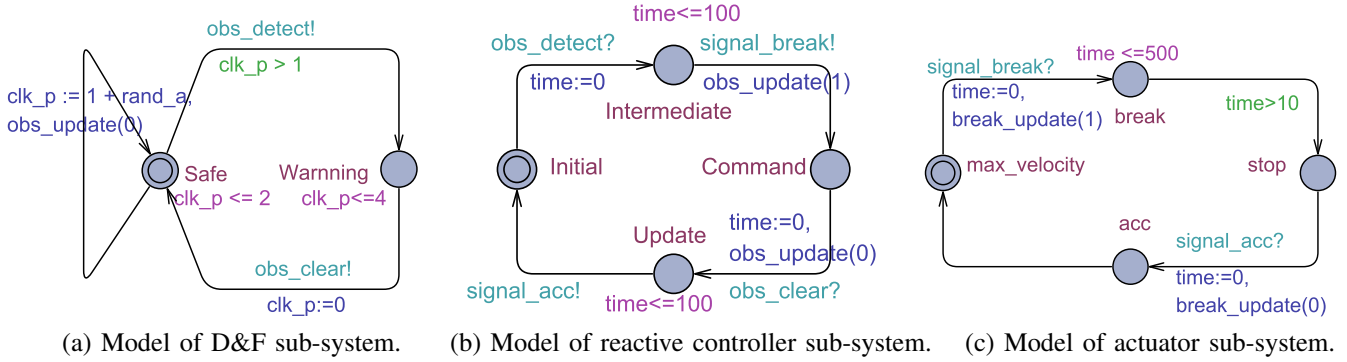


Fig. 2: Model of vehicular sub-system.

$d > 20\text{m}$, when it is actually less than the safe distance. Due to this, the reactive controller will not be able to brake within the desired time, resulting in a collision. However, our safe state estimation and monitoring method in the D&F sub-system will detect this fault and notify the reactive controller within the desired time to prevent collision.

To verify this scenario, we use the UPPAL Statistical Model Checking (SMC) tool Version 4.1. We build a very simple timed automata of *D&F sub-system* (Figure 2a) with two states (safe and warning). The safe state indicate that the future state from the control and decision unit conform with the safe state estimated by the D&F sub-system. On the contrary, the warning state indicate suspicious behavior. In our current set-up we assume that the adversary cannot modify the D&F sub-system directly, but as the estimation algorithm is not completely accurate, we assign uncertainty to the state variable of the automaton. The safe state is initialized with an uncertain variable clk_p uniformly between $[1.00, 2.00]$. The automaton transitions to the warning state when the value of clk_p is > 1 and it outputs obs_detected! , which is sent to the reactive controller sub-system (Figure 2b). From the warning state it transition back to the safe state when $\text{clk_p} \leq 4$ with an output obs_clear! and the variable clk_p resets to 0. In order to estimate the probability that the clk_p will change its value to > 1 within a simulation time bound of 1000 time units, the following specification will be checked.

$$Pr[<= 1000](\langle \rangle \text{clk_p} > 1)$$

The result of the tool for D&F automata is: confidence interval $[0.9026, 1.00]$ with 95% confidence after having generated 36 runs. After the obs_detect! signal is received by the *reactive control automata* (Figure. 2b), it transition from Initial state to Command via Intermediate state in < 100 time units. In this transition, the controller sends the command signal_break to the actuator sub-system (Figure 2c) as an obstacle has been detected. At the Initial state, internal variable obs is initialized to 0. After transitioning to Command state, the obs value changes to 1. This is shown by the method $\text{obs_update}(1)$ in Figure 2b. To estimate the probability that the internal variable obs

changes value to 1 in < 100 time units, we use the following specification.

$$Pr[<= 1000](\langle \rangle \text{time} < 100 \text{ and } \text{obs} == 1)$$

The result of the tool for reactive controller automata is: confidence interval $[0.9026, 1.00]$ with 95% confidence after having generated 36 runs. Once the obstacle is clear the automaton transitions to the Initial state via Update state and the internal variable obs value is reset to 0, shown in Figure. 2b as $\text{obs_update}(0)$. The output of this transition is the command signal_acc! , which is also sent to the actuator sub-system. When the *actuator automata* (Figure. 2c) receives the command signal_break! from the reactive control automata, it should transition from max_velocity state to the break state in < 500 time units. During the same transition, internal variable apply_break changes its value from 0 to 1. This is shown by the method $\text{break_update}(1)$ of Figure. 2c. To estimate the probability that the internal variable apply_break changes its value to 1 in < 500 time units, we use the following specification,

$$Pr[<= 1000](\langle \rangle \text{time} < 500 \text{ and } \text{apply_break} == 1)$$

The result of the tool for actuator automata is: confidence interval $[0.9026, 1.00]$ with 95% confidence after having generated 36 runs. From the break state the automata transition to stop state in $\text{time} > 10$ time units. During this transition, variable acc is initialized to an initial value of 60 with an uncertain derivative dv uniformly between $[1.00, 2.00]$. This uncertainty indicates that the car never comes to a complete halt at the stop state. After receiving the command signal_acc! from the reactive controller, the vehicle transitions back to the acc state. During this transition, the value of internal variable apply_break changes from 1 to 0 as shown by the method $\text{break_update}(0)$ of Figure. 2c. This transition indicates that the collision has been avoided and the vehicle resume its motion. All the safety specifications we considered, were satisfied in the confidence interval $[0.9026, 1.00]$ with 95% confidence after having generated 36 runs.

In Table.1 we have provided summary of trusted and untrusted signals of automaton of Figure. 2a, 2b, 2c. Now, based on the results of these automaton, trust on the autonomous ground vehicle under consideration is 95% for a confidence interval of [0.9026, 1.00].

TABLE I: Summary of Trusted and Untrusted Signals of Automaton

Signals	Untrusted	Trusted	Reason
obs_detect!	✓		Triggered on encountering bad states
obs_clear!		✓	Doesn't depend on bad states
obs_detect?	✓		Relies on obs_detect!
signal_break!		✓	Doesn't depend on bad states
obs_clear?		✓	Doesn't depend on bad states
signal_acc!		✓	Doesn't depend on bad states
signal_break?		✓	Doesn't depend on bad states
signal_acc?		✓	Doesn't depend on bad states

VI. CONCLUSION

Trust plays an important role in adoption of autonomous systems such as self-driving vehicles. In this paper, we have addressed the trust issue in autonomous systems using an estimation method and statistical model checking approach. The estimation method identifies safe states based on a set-membership method and with the help of system dynamic. The statistical model checking approach provide probabilistic guarantees on whether various sub-systems of the vehicle will satisfy safety specifications within some time interval. In our case study, we use the confidence results generated during verification process to assign trust on the adaptive cruise-control unit of the vehicle. In future, we intend to explain our estimation and verification methods in more sophisticated systems.

ACKNOWLEDGMENT

This work has been partially supported by the National Science Foundation (NSF-1319105) and Army Research Office (W911NF-16-1-0124).

REFERENCES

- [1] DARPA, "Darpa urban challenge," November, 2007.
- [2] J. D. Lee and K. A. See, "Trust in automation: Designing for appropriate reliance," *Human Factors*, vol. 46, pp. 50–80, 2004.
- [3] S. Behere and M. Torngren, "A functional reference architecture for autonomous driving," *Information & Software Technology*, vol. 73, pp. 136–150, 2016.
- [4] O. of the Chief Scientist, "Autonomous horizons: System autonomy in the air force – a path to the future," in *United States Air Force*, 2015.

- [5] K. Koscher and et.al., "Experimental security analysis of a modern automobile," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 447–462, 2010.
- [6] R. Boyle, "Proof-of-concept carshark software hacks car computers, shutting down brakes, engines, and more," in *Popular Science*, May 14, 2010.
- [7] S. Checkoway and et. al., "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.
- [8] J. Bhatti and T. E. Humphreys, "Covert control of surface vessels via counterfeit civil gps signals," *University of Texas, unpublished*, 2014.
- [9] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," *CHES'13*, (Berlin, Heidelberg), pp. 55–72, Springer-Verlag, 2013.
- [10] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and lidar," in *Black Hat Europe*, November, 2015.
- [11] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," in *DEFCON, USA*, August, 2015.
- [12] Y. Shoukry, P. Martin, Y. Yona, S. N. Diggavi, and M. B. Srivastava, "Pycra: Physical challenge-response authentication for active sensors under spoofing attacks," in *ACM CCS*, pp. 1004–1015, ACM, 2015.
- [13] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, pp. 1204–1223, June 2005.
- [14] P. Hood, "How do google's self-driving cars work?," in *Alphr*, April, 2016.
- [15] F. Mujica, "Scalable electronics driving autonomous vehicle technologies," in *Texas Instrument*, April, 2014.
- [16] O. Stursberg, A. Fehnker, Z. Han, and B. H. Krogh, "Verification of a cruise control system using counterexample-guided search," *Control Engineering Practice*, vol. 12, no. 10, pp. 1269 – 1278, 2004. Analysis and Design of Hybrid Systems.
- [17] S. M. Loos, A. Platzer, and L. Nistor, *FM 2011: Formal Methods: 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, ch. Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified, pp. 42–56. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [18] T. Wongpiromsarn, S. Mitra, A. Lamperski, and R. Murray, "Verification of periodically controlled hybrid systems: Application to an autonomous vehicle," *Special Issue of the ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. S2, 2012.
- [19] M. Althoff, D. Althoff, D. Wollherr, and M. Buss, "Safety verification of autonomous vehicles for coordinated evasive maneuvers," in *Intelligent Vehicles Symposium (IV)*, 2010 IEEE, pp. 1078–1083, June 2010.
- [20] B. Zheng, W. Li, P. Deng, L. Gérardy, Q. Zhu, and N. Shankar, "Design and verification for transportation system security," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.
- [21] P. Mundhenk, S. Steinhorst, M. Lukasiewicz, S. A. Fahmy, and S. Chakraborty, "Security analysis of automotive architectures using probabilistic model checking," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2015.
- [22] A. Tiwari, B. Dutertre, D. Jovanovic, T. de Candia, P. Lincoln, J. M. Rushby, D. Sadigh, and S. A. Seshia, "Safety envelope for security," in *Proceedings of the 3rd International Conference on High Confidence Networked Systems (HiCoNS)*, pp. 85–94, April 2014.
- [23] H. Fawzi, P. Tabuada, and S. Diggavi, "Secure estimation and control for cyber-physical systems under adversarial attacks," *IEEE Transactions on Automatic Control*, vol. 59, pp. 1454–1467, June 2014.
- [24] M. Wolf and T. Gendrullis, *Information Security and Cryptology - ICISC 2011*, ch. Design, Implementation, and Evaluation of a Vehicular Hardware Security Module, pp. 302–318. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.
- [25] M. Milanese and C. Novara, "Set membership identification of nonlinear systems," *Automatica*, vol. 40, no. 6, pp. 957–975, 2004.
- [26] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang, *Statistical Model Checking for Networks of Priced Timed Automata*, pp. 80–96. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
- [27] P. Zuliani, C. Baier, and E. M. Clarke, "Rare-event verification for stochastic hybrid systems," in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, HSCC '12*, (New York, NY, USA), pp. 217–226, ACM, 2012.

Eliminating the Hardware-Software Boundary: A Proof-Carrying Approach for Trust Evaluation on Computer Systems

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

Abstract—The wide usage of hardware intellectual property (IP) cores and software programs from untrusted third-party vendors has raised security concerns for computer system designers. The existing approaches, designed to ensure the trustworthiness of either the hardware IP cores or to verify software programs, rarely secure the entire computer system. The semantic gap between the hardware and the software lends to the challenge of securing computer systems. In this paper, we propose a new *unified framework* to represent both the hardware infrastructure and the software program in the same formal language. As a result, the semantic gap between the hardware and the software is bridged, enabling the development of system-level security properties for the entire computer system. Our *unified framework* uses a cross-domain formal verification method to protect the entire computer system within the scope of proof-carrying hardware. The working procedure of the *unified framework* is demonstrated with a sample embedded system which includes an 8051 microprocessor and an RC5 encryption program. In our demonstration, we show that the embedded system is trusted if the system level security properties are provable. Supported by the *unified framework*, the system designers/integrators will be able to formally verify the trustworthiness of the computer system integrated with hardware and software both from untrusted third-party vendors.

Index Terms—Hardware trust, proof-carrying hardware, proof-carrying code, cross-layer protection, formal verification, Coq proof assistant.

I. INTRODUCTION

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 have also helped in the proliferation of the intellectual property (IP) market. The use and reuse of existing commercial IPs has enabled improvements in TTM in addition to cost reduction. However, as a result of the globalization of the semiconductor supply chain, companies and the government 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. This has created 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 the inclusion of a Trojan circuit can derail the entire hardware industry.

Many hardware Trojan detection and protection methods have been developed for pre- and post-silicon stages [1]–[13]. However, methods for detecting Trojans at the RTL level (of the pre-silicon stage) has been lacking. Furthermore, existing Trojan detection methods for RTL designs rely on a golden circuit model. Generally, such a model is not readily available for third-party soft-IP cores. As such, it further limits the applicability of existing methods for detecting Trojans implanted in the RTL design of the hardware. Our method could detect Trojans in third-party soft-IP cores in absence of a golden model.

In fact, the security concerns raised by third-party resources are not unique to the semiconductor industry and have existed for a long time. Software developers have been combating similar issues while trying to ensure the security and integrity of software systems, often constructed on top of third-party resources. The software industry is comprised of a variety of software companies and programmers, ranging from IT giants developing operating systems (OS) and databases to freelancers developing mobile apps. This has resulted in a software market that is poorly regulated and facilitates the development and spread of large amounts of computer worms, viruses and Trojans. Cybersecurity developers are facing the additional challenge of protecting the computer system from such malware attacks.

In order to secure computer systems built from third-party resources, security researchers both in the hardware and software domains 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

Manuscript received April 24, 2016; revised August 3, 2016 and October 5, 2016; accepted October 5, 2016. Date of publication October 27, 2016; date of current version November 21, 2016. This work was supported in part by the National Science Foundation under Grant NSF-1319105 and in part by the Army Research Office under Grant ARO W911NF-16-1-0124. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ozgur Sinanoglu.

The authors are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816 USA (e-mail: guoxiaolong@knights.ucf.edu; rajgautamdutta@knights.ucf.edu; yier.jin@eecs.ucf.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/TIFS.2016.2621999

ICs [4], [5], [7], [13]–[18]. In the software domain, methods have been developed for the detection of malicious kernel extensions and kernel integrity defence [19], [20].

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 threat from the firmware or OS running on top of it. As a result, these methods fail to protect computer systems where both the hardware and the software are vulnerable to attack. The semantic gap (characterizes the difference between the operations performed by hardware and software) between the hardware and software domains has been the major obstacle for developing 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.

Our main contributions in this paper are summarized as follows:

- We propose to use deductive formal methods for trust evaluation of the computer system constructed from untrusted third-party software and hardware resources.
- Our method helps eliminate the semantic gap between the software and the hardware by representing them in a *unified framework* – where the software program and the underlying hardware infrastructure are presented in the same formal language. This enables system designers to develop system-level security properties, without worrying about cross-boundary inconsistencies.

The rest of the paper is organized as follows: Section II presents previous work on IP core trust evaluation and Trojan detection. In Section III, we provide background on proof-carrying code and its hardware counterpart, proof-carrying hardware. In Section IV, we explain the working procedure of our *unified framework*, the threat model, security theorem development process, the formal language to describe the computer system, and the proof construction process. Section V presents demonstrations of the proposed framework in preventing information flow attacks within an embedded system (in the paper we use the terms – computer system and embedded systems, interchangeably). In Section VI, we discuss the advantages of our proposed methodology for computer system protection and its limitations. Final conclusions are drawn in Section VII.

II. RELATED WORK

To overcome the threat of untrusted third-party resources, many pre-silicon trust evaluation methods have recently been developed [3], [15], [17], [21]–[26]. Among these methods, [3], [17], [21] detect malicious logic by using enhanced functional testing methods. In [3], additional “Trojan Vectors” were generated to activate hardware Trojans during functional testing. To identify suspicious circuitry, the unused circuit identification (UCI) method of [17] analyzed the RTL code to find unused lines of code. However, the methods

of [3] and [17] assume that the attacker uses rare events as Trojan triggers. This assumption was invalidated in [21], where “less-rare” events were used as hardware Trojan triggers.

Due to the limitations of enhanced functional testing methods, researchers started looking into alternative solutions. Among the possible solutions are formal methods which can exhaustively verify security properties of untrusted hardware resources [15], [22]–[27]. A multi-stage approach, which included assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and sequential Automatic Test Pattern Generation (ATPG), was used in [26] to identify suspicious signals.

Proof-Carrying Hardware (PCH) is another approach for ensuring trustworthiness of hardware [15], [22]–[25], [28], [29]. This approach is inspired from the Proof-Carrying Code (PCC) method and it has emerged as one of the most prevalent method for certifying the absence of malicious logic in soft IP cores. The method was first proposed in [22] and [23], where runtime combinational equivalence checking (CEC) was used to verify the equivalence between the design specification and the design implementation. However, the approach did not consider security property verification. To overcome this limitation, the PCH framework was expanded in [15], [24], and [25] to verify security properties on IP cores in the format of synthesizable register-transfer level (RTL) code. Hierarchical proof construction process was also proposed in [28] and [29] to reduce the workload of building proofs. In the new PCH framework, the RTL design and informal security properties are first represented in Gallina, the internal functional programming language of the Coq proof assistant. Then, Hoare-logic style reasoning is used to prove the correctness of the RTL code. The implementations are carried out using the Coq platform [30].

The *unified framework* of this paper is derived from the PCC and the new PCH methods and it can be used for verification of the computer system.

III. PROOF-CARRYING CODE AND PROOF-CARRYING HARDWARE

Various methods have been proposed in the software domain to validate the trustworthiness and genuineness of software programs. These methods protect computer systems from untrusted software programs. Most of these methods lay burden on software consumers to verify the code. However, *proof-carrying code* (PCC) switches the verification burden to software providers (software vendors/developers). During the *source code certification* stage of the PCC process, the software provider verifies the code with respect to the *security property* designed by the software consumer and encodes the formal proof of the security property with the executable code in a *PCC binary file*. In the *proof validation* stage, the software consumer determines whether the code from the potentially untrusted software provider is safe for execution by validating the PCC binary file using a proof checker [31].

A similar mechanism, referred to as Proof-Carrying Hardware (PCH), was used in the hardware domain to protect third-party soft IP cores [15], [24], [25]. The PCH framework ensures trust-worthiness of soft IP cores by verifying a set of

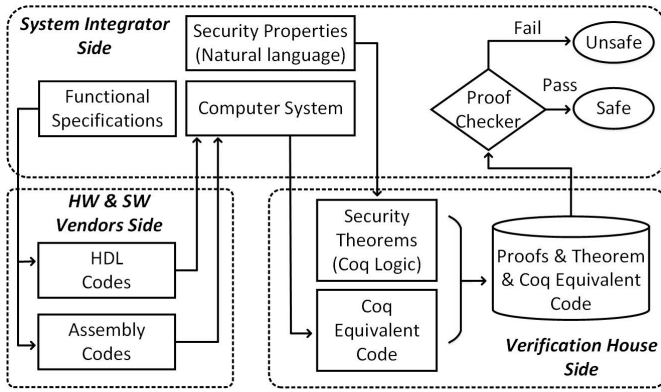


Fig. 1. Working procedure of the proposed *unified framework*.

carefully specified security properties [32]. In this approach, the IP consumer provides design specifications and informal (written in natural language) security properties to the IP vendor. Upon receiving the request, the IP vendor develops the RTL design using a hardware description language (HDL). Then, semantic translation of the HDL code and informal security properties to Gallina is carried out. Subsequently, Hoare-logic style reasoning is used for proving the correctness of the RTL code with respect to formally specified security properties in Coq. As Coq supports automatic proof checking, it can help IP customers validate proof of security properties with minimum efforts. Moreover, usage of the Coq platform by both IP vendors and IP consumers ensures that the same deductive rules will be used for validating the proof. After verification, the IP vendor provides the IP consumer with the HDL code (both original and translated versions), formalized security theorems of security properties, and proofs of these security theorems. Then, the proof checker in Coq is used by the IP consumer to quickly validate the proof of security theorems on the translated code. The proof checking process is fast, automated, and does not require extensive computational resources.

IV. UNIFIED FRAMEWORK - BRIDGING THE GAP

The PCH and the PCC frameworks share a similar working procedure including the code generation, the security property development, the proof construction, and the proof checking stage. Due to this similarity, both frameworks can be merged to build the *unified framework* as shown in Figure 1. However, the difference in representation of the hardware code and the software code causes a semantic gap, which hinder the development of the *unified framework*. The hardware code relates to the circuit logic of the hardware structure while the software code controls the data flow on the hardware structure. For representing the entire computer system in the Coq platform, the semantic gap between the hardware code and the software code needs to be eliminated.

The work in [25] provides a preliminary solution to bridge the semantic gap between the hardware code and the software code. In [25], a *formal HDL* is developed within the Coq platform for the instruction-set architecture (ISA) and supports large-scale circuit design. Compared to the PCH framework of [24] and [33], where security properties are developed

for data processing units and/or functional blocks, the *formal HDL* of [25] supports the development of security properties for both the control logic and the datapath. The security properties of the *formal HDL* outlines the trusted behaviors of each instruction when executed by the processor. The *formal HDL* also detects hardware level malicious modification for individual instructions. In this paper, the ISA-level PCH framework is expanded to support system-level security properties of assembly code running on top of third-party hardware processors. Note that the firmware and software will be compiled into assembly code before executing them.¹

Our proposed *unified framework* is the first formal framework which eliminates the software-hardware boundary and enables the construction of system-level security properties for the entire computer system. The framework supports formal reasoning to facilitate the construction of proofs of formal security theorems.

A. Threat Model and Trusted Verification House

The *unified framework* is developed to prove the presence/absence of malicious logic inserted by an adversary at the design stage of the software/hardware. We assume that there is a rogue agent in the design house who has access to both the HDL code and the software code. Such an adversary can either insert a hardware Trojan or malware in the hardware or software design. These malicious payload can be triggered either by the software or under certain physical conditions. Upon activation the Trojan/malware can cause sensitive information leakage, functionality alteration, control flow hijacking, or denial-of-service of the entire computer system.

Another assumption is that a trusted verification house exists which can use the proposed framework to guarantee the security of the whole computer system with respect to the predefined security properties provided by the system integrator. Note that the system integrator and the trusted verification house will use the same formal verification platform (in our case, Coq) to prove and validate the security properties of the computer system. According to our verification framework, proofs will be constructed for formal security theorems derived from informal security properties. The availability of theorem proofs indicates that there are no malware/Trojans in the designed computer system. On the other hand, if the verification house cannot build such proofs, it is likely that malicious modifications may be inserted in the design.

B. Working Procedure of the Unified Framework

In the *unified framework*, three entities are involved in the verification process (see Figure 2).

- **Hardware and Software Vendors:** Hardware vendors design and sell soft IP cores based on the design specifications of IP users. Similarly, software vendors develop software programs which can be compiled and run on

¹The proof-carrying based unified framework proposed in this paper primarily works on assembly-level code of the software program. For any programs written in high-level languages, we will rely on a trusted compiler to compile the programs into assembly code. The design and validation of a trusted compiler is outside the scope of this paper

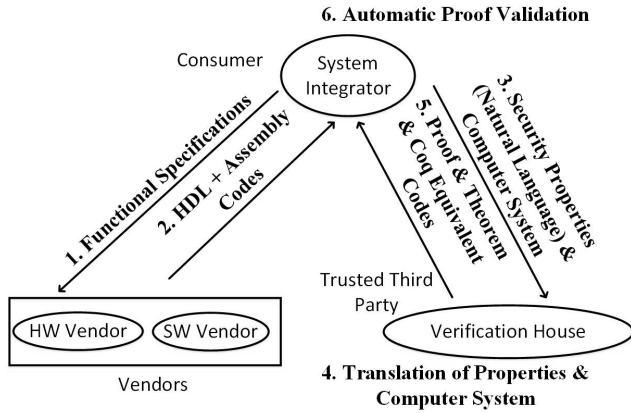


Fig. 2. Main parties in the *unified framework*.

hardware platforms. In our framework, we treat both hardware and software vendors as untrusted resource providers. We further assume that a rogue agent can manipulate both software code and hardware IP design.

- **System Integrators:** System integrators in the proposed *unified framework* can be seen as the consumer. To design the computer system, they integrate IPs and the corresponding software from the vendors. In order to ensure the trustworthiness of the developed computer system, a set of informal security properties, the HDL code of the hardware design and assembly code of the software are provided to a trusted verification house for formal verification. Upon receiving the verified system, validation is carried out by the system integrator using automated tools such as the proof checker of Coq.
- **Trusted Verification House:** The verification house in the proposed *unified framework* is treated as a trusted third party [34]. On receiving the HDL code, the assembly code, and the set of security properties from the system integrators, the verification house uses the proposed framework for semantic translation and proving the set of security properties in the Coq platform. Subsequently, they provide the system integrator with the translated Coq equivalent code, the formal security theorems, and their proofs.

In the proposed *unified framework*, we eliminate the vulnerability at the software-hardware boundary by converting the software program into a set of hardware states. During verification of security properties in the framework, all software operations, previously treated independently, become part of the hardware implementation. In other words, prior to detecting threats in the verification process, any attacks targeting the software-hardware interface will be transformed into threats targeting the hardware platform. Note that the proposed framework does not alter working process of the original embedded system, nor the format or functionality of the hardware/software. After the verification of security properties, the computer system will operate in its original way along with the assurance that the system is trustworthy. The working procedure of our proposed framework, shown in Figure 1 and figure 2, is divided into six phases.

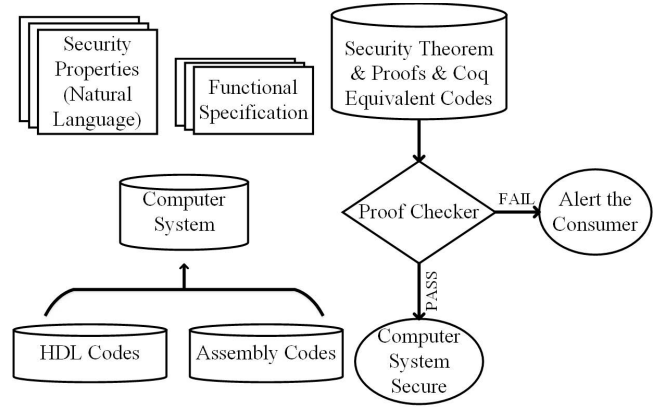


Fig. 3. Various tasks of the system integrator in the *unified framework*.

- **Phase 1: Functional Specifications:** In the first phase, the system integrator develops and sends the functional specifications of the target computer system to vendors (both hardware vendors and software vendors).
- **Phase 2: Computer System design and Functional Testing:** Based on the request of the system integrators, vendors design and deliver the soft IP core and the source code of the software (see Figure 4). The system integrators then build the computer system and perform functional testing to validate its functionality and performance. Only if passing the functional testing, the design becomes eligible for security check.
- **Phase 3: Security Properties:** In this phase, the system integrator develop a set of informal security properties for the verification house. These security properties delineate the security boundary of the computer system. Various responsibilities of the system integrators are shown in Figure 3. The entire source code of the computer system and the set of security properties are then sent to the trusted verification house.
- **Phase 4: Translation of Security Property and Source code of Computer System to Gallina:** In the trusted verification house side, the assembly code and the HDL code are translated to Gallina, which is the specification language of Coq, with the assistance of the *formal HDL*, defined in the *unified framework*. This translated code is referred to as *Coq Equivalent Code*. Translating the set of informal security properties to Gallina gives the desired formal security theorems. These translations make the security verification of the computer system possible on the Coq.
- **Phase 5: Proof Construction:** In this phase, the verification house constructs proofs for the formal security theorems of the computer system. The developed proof, the security theorems, and the Coq equivalent codes are integrated into a trusted bundle and provided to the system integrator. In order to facilitate the proof construction process and to help alleviate the scalability issue, a hierarchical proof writing method is developed and applied. Details of this hierarchical approach will be introduced shortly. Operations of the verification house is shown in Figure 5.

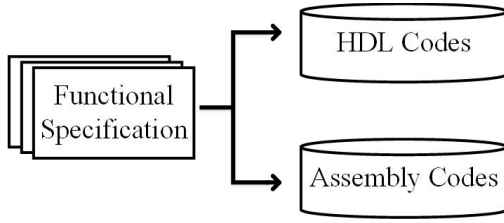


Fig. 4. Working procedure of the *unified framework* at the vendor side.

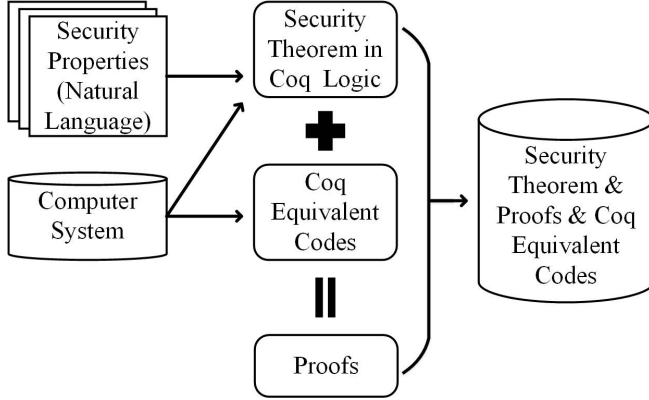


Fig. 5. Operations of the trusted verification house in the *unified framework*.

• Phase 6: Proof Validation and Computer System

Security: Upon receiving the trusted bundle from the verification house, the system integrators validate the proofs using the proof checker built in Coq platform. This step is also shown in Figure 3. The proof fails if the security properties are not satisfied. However, if the proof passes the proof checker, the system integrators are assured that the computer system from untrusted vendors is indeed trustworthy.

C. Security Theorem

As described in Figures 3 and 5, the whole unified framework follows a theorem-proving methodology. The key aspects of this methodology are elaborated below.

1) *Security Theorems for Computer Systems:* Based on the set of well-defined system-level security properties of the computer system, the security theorems within the Coq platform are constructed to formally represent the security requirements.² These theorems, if proved, will provide high-level assurance that the computer system integrated with third-party resources is trusted.

2) *Lemmas for Instructions:* Due to the manual proof construction process, directly proving system-level security theorems is infeasible. The *lemma development* method, which will be introduced in the following contents, reduces the burden of the proof construction process by using a

²Designing a set of well-defined security constraints/properties is an important step for successfully securing the computer systems. Poorly written security properties will allow attackers to include malicious code and additional functionality in the IP cores/firmware, without being detected in the verification stage. Although a sample security property is introduced, a methodology for systematically developing security property for different computer systems will be discussed in our future work.

distributed approach in which individual instructions of a software program is proved. In order to prove each individual instruction, the system-level theorems are first split into lemmas, which define the trusted behavior of each instruction used in the software program. Depending on the size of the software program, a large number of lemmas are constructed. The task of assuring the security of the whole system relies on the ability to prove the security properties of a sequence of instructions implemented on the hardware infrastructure.

3) *Proof of Lemmas:* The most time-consuming part of the proposed framework is to prove lemmas of individual instructions. However, the time required for proving all lemmas in the *lemma development* method is linear to the size of the software program. Consequently, verifying complicated and large computer systems becomes feasible due to the proof construction methods and the proposed proving process.

4) *Trusted Bundle Preparation:* System-level theorems of security properties are proved when lemmas of each instruction of the software program are verified with respect to their trusted behavior. The trusted bundle is prepared for the consumer, which includes the source code of the hardware, source code of the software, security theorems, and proof. Before executing the software program on the computer system, the consumer will regenerate the Coq equivalent code for the computer system and validate the proof of security properties using an automated proof checker. Regeneration of the Coq equivalent code by the consumer will guarantee that the source code of both the software and the hardware are not manipulated during the conversion process.

D. Formal Language for Computer System Description

The proposed proof-carrying based *unified framework* enables us to represent both the hardware code and the software code in the same formal language. The new formal language for the *unified framework* is derived from the *formal HDL* developed in [25] to support assembly-level programs with multiple instructions. Due to similarities between our proposed formal language and the *formal HDL* of [25], our language is still referred to as the *formal HDL* in rest of the paper.

Defined in [25], the previous *formal HDL* can only represent basic circuit units, combinational logic, sequential logic, and module instantiations. As shown below, a bus type is defined as a function which takes one parameter, a timing variable t , and returns a list of signal values. The keyword `assign` of the *formal HDL* is used for blocking assignment, while `update` is mainly used for nonblocking assignment. During the blocking assignment the bus value will be updated in the current clock cycle and in the nonblocking assignment the bus value will be updated in the next clock cycle. Since the *formal HDL* can be applied to only synchronous hardware, the variable t indicates the global clock cycle.

The newly proposed *formal HDL* has several unique characteristics, which make it suitable for representing the entire computer system. 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). Keywords are defined for module definitions. For example, *module_vhdl* is also used as the data type of declared modules. The *module_inst* describes the implementation detail of each module. As most digital designs are synchronous, the *formal HDL* is primarily used for synchronous designs. In software programs, the *formal HDL* represents circuit-level operations of each instruction without imposing any restrictions on the sequence of the instructions. Therefore, the *formal HDL* can deal with sophisticated information flow and support flexible function calls. The same *formal HDL* can also be applied to different ISAs with minor modifications. This is possible because most of the software programs are compiled to assembly code, which can then be described in the *formal HDL*. In Listing 1, assembly language instructions are represented by the axiom *DATA_INPUT*.

The code in Listing 1 is the representation of the ALU module. The ALU module (*module_mc8051_alu*) consist of the 8-bit adder-subtractor (*module_addsub_core*) sub-module, which itself consist of two other sub-modules (i) 4-bit adder for higher order bits (*module_addsub_cy*) and (ii) 4-bit adder for lower order bits (*module_addsub_ovcy*). Such a hierarchical design of the ALU is first flattened during implementation in Coq. In the flattening process, all the ALU modules are first declared using the the key word *Inductive* under the *module_vhdl* data-type. Two of input arguments of the *module_addsub_core* module are declared as *mymodule1* and *mymodule2*. Accordingly, to describe the *module_addsub_core* module, the sub-modules *module_addsub_cy* and *module_addsub_ovcy* will be given as these two input parameters. The same process is followed to describe the sub-module *module_addsub_core* of the top module *module_mc8051_alu*. Based on this procedure, the entire hierarchical design of the hardware can be flattened. In the demonstration section, a computer system comprising of an encryption algorithm and an 8051 microprocessor is represented using the developed procedure.

E. Proving Security Properties in Unified Framework

In the *unified framework*, security theorems are first constructed from security properties defined by the system integrators (often described in a natural language) and then proved with respect to the source code. The system integrators validate the proof of the theorem to ensure that no malicious behavior occurs during the operation of the computer system. In the event that the theorems cannot be proved, the system integrators are alerted of security property violations.

Although the PCH and PCC frameworks provide high-level security assurance to third-party resources, they both suffer from the problem of scalability. Verification of a complex IP core (or software program) increases the computational burden on vendors, resulting in a prolonged design cycle and increased development cost. For the same reason, most of the current demonstration examples of PCC and PCH are for small- to medium-scale designs where security proofs can be constructed in a reasonable time [15], [24], [33], [35], [36].

```

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

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.

Inductive aifblock :=
| anoif : assignblock->aifblock
| aifsimple : expr->aifblock->aifblock
| aifelse : expr->aifblock->aifblock->aifblock.

Inductive ifblock :=
| noif : updateblock->ifblock
| ifsimple : expr->ifblock->ifblock
| ifelse : expr->ifblock->ifblock->ifblock.

Inductive module_vhdl :=
| module_addsub_cy : bus->...->module_vhdl
| module_addsub_ovcy : bus->...->module_vhdl
| module_addsub_core : bus->...->module_vhdl->
module_vhdl->module_vhdl
| module_mc8051_alu : module_vhdl->module_vhdl->
...->module_vhdl
...

Fixpoint module_inst (m:module_vhdl) (t:nat) :=
match m with
| (module_addsub_cy opa_i ... rslt_o) =>
(assign (
(expr_assign (v_a [4,1]) (econb opa_i));
(expr_assign (v_b [4,1]) (econb opb_i))))t) /\
(adoif (
aifelse (eeq (econb addsub_i)
(econv (hi::nil))))...
| (module_addsub_ovcy opa_i ... ov_o) =>
...
| (module_addsub_core opa_i ... mymodule1 mymodule2) =>
(module_inst mymodule1 t) /\ (module_inst mymodule2 t) ...
| (module_mc8051_alu mymodule1 mymodule2 ...) =>
(module_inst mymodule1 t) /\ ...
...
end.

Axiom DATA_INPUT : rom_data_i 1 = opcode_one /\
rom_data_i 2 = data_load_one /\
...

```

Listing 1. Implementation of ALU module in Coq.

To overcome the scalability issue, a hierarchical approach for proof construction is adopted in the *unified framework*. In the hierarchical approach, instructions are proved using the *lemma development* method and the security theorem of the computer system is proved by integrating the proof of all lemmas in the *theorem development* method. Specifically, lemma development is used for security assurance of individual instructions, while theorem development is used for a program using these instructions. Hoare-logic is used for verification in our framework.

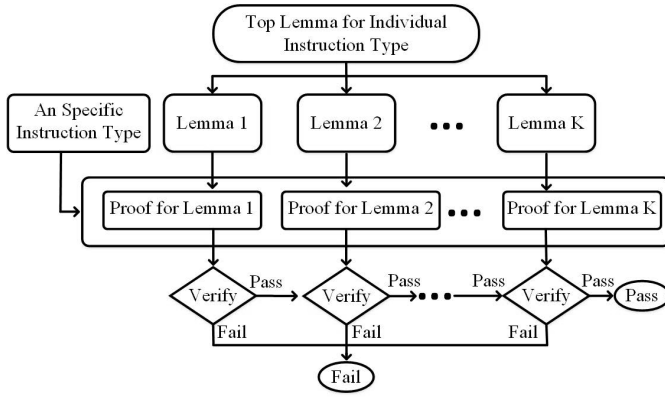


Fig. 6. Lemma development for security evaluation of individual instructions.

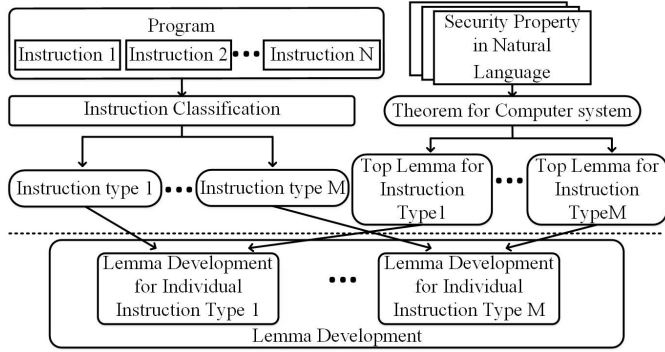


Fig. 7. Theorem development procedure for system-level security property of computer system.

The working procedure of the *lemma development* process is shown in Figure 6. This approach is applicable for security property verification of circuit-level operations of individual instructions. All security theorems of the entire computer system will be divided into separate lemmas based on different instruction types. The lemma development method focuses on building proof construction for this specific lemma, which is called the top lemma in Figure 6 and Figure 7. Specifically, in Figure 6, the top lemma can be further divided into a series of lemmas which can be categorized as (i) lemmas for *data transmission* and (ii) lemmas for *data operations*. Similar to large-scale circuits, the processor cores have a hierarchical structure with top modules representing *data transmission* and sub-modules representing functional units for *data operations*. During *data transmission*, the sub-modules communicate among themselves (e.g. signal transmission between the ALU block and the control module) whereas *data operations* represents operations within the sub-modules (e.g. updating the program counter). In Figure 6, the *lemma development* process takes advantage of the processor architecture and constructs the lemmas in the mentioned two categories.

During circuit-level verification of processor cores, the pre-conditions of security theorems are matched against the pre-condition of one of the developed lemmas. After finding a match between pre-conditions, a post-condition is obtained. The post-condition then acts as a pre-condition for another lemma and the matching of pre-conditions continues.

Following the process of connecting lemmas using pre- and post-condition matching, the security theorem is proved by using the post-condition of the last lemma. The circuit level operation of individual instructions are certified as trusted when all lemmas are proved. Due to the distributed approach of proving lemmas of *data transmission* and *data operations*, the workload of proving the security theorems for the overall behavior of each instruction is significantly reduced. Moreover, the proved lemmas especially those for *data transmission*, can be reused for other instructions mainly because many instructions share the same *data transmission* operations. For example, in Figure 6, most of the lemmas belong to either *data transmission* or *data operations*. When successfully proven, these lemmas along with their proof code can be reused in verifying other instructions on the same hardware platform.

The soundness of the *lemma development* process is demonstrated in two steps.

- 1) The developed *data transmission* lemmas prevent malicious manipulation during communication between different sub-modules.
- 2) The existence of formal proofs for lemmas of *data operations* of each instruction guarantees the generation of trusted subset of signals. That is, a sub-module will generate correct results for legitimate inputs, without performing any additional (often malicious) operations.

Moreover, modern processors often provide instructions with similar functionality. As a result, the proof construction process for security properties is similar for these instructions. Also, the lemmas developed for one instruction can be used by other instruction proof processes. Due to the constant reuse of lemmas between instructions, the development cost and time is lowered for proving instructions of a given processor.

The *theorem development* method of proof construction deals with security property verification of software programs which contain multiple instructions. This method supports the design of system-level security theorems for security properties of the computer system. The working procedure of our *theorem development* method is shown in Figure 7. To secure the computer system, the *theorem development* method integrates the proof of lemmas of individual instructions. When all the lemmas are proved, the security theorem of the computer system is stated to be proved. Also, as shown in the Figure 7, the *theorem development* process also takes advantage of hierarchical structure for complex system verification while the *lemma development* serves as the cornerstones in the whole procedure.

The *theorem development* process along with the *lemma development* method, converts the task of proving theorems on large programs to individual instruction proving. This helps in speeding up the design cycle and developing libraries of security properties for verifying different software programs.

V. DEMONSTRATION

In this section, the working procedure of the proposed system-level *unified framework* will be demonstrated in the Coq proof assistant platform [30]. For illustration, a computer system is built where the 8051 microprocessor serves as

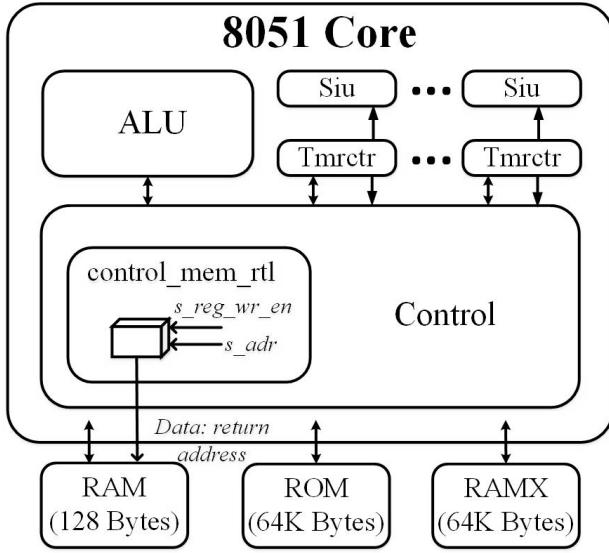


Fig. 8. Block diagram of an 8051 microprocessor.

the hardware infrastructure and the RC5 encryption algorithm performs the required functionality. Such a computer system is likely to be used in critical infrastructure where sensitive data is encrypted before transmission. The VHDL source code of the 8051 and the assembly code of the RC5 algorithm were obtained from [37] and [38], respectively. Therefore, the sample computer system is assumed to be constructed from third-party resources. We develop security properties and prove them on the sample computer system.

The block diagram of the 8051 microprocessor is shown in Figure 8, which includes 64KB on-chip program memory, 128B on-chip data RAM, and can support 64KB external memory space. The complete instruction set of the 8051 microprocessor is given in [39]. Although 8051 is relatively small compared to modern processors, the sample implementation contains instruction decoding, execution, and memory access stages, making it a good candidate for the initial investigation.

RC5 is a fast symmetric block cipher suitable for hardware or software implementations [40]. Besides variable word size (32, 64 or 128 bits), RC5 has variable number of rounds (0 to 255) and secret keys (0 to 2040 bits). The wide usage of RC5 in stream cypher and its simple implementation in software makes it suitable for our demonstration.

In our *unified framework*, no modifications are required on the original code throughout the verification procedure. As a result, there is no performance overhead on the computer system under security verification. After the security verification, the hardware code is synthesized to build ASIC logic (or FPGA bitstream) while the software code is loaded into the microprocessor memory for execution. In our case, the entire computer system is assumed to be implemented in FPGA with additional UART modules inserted to load/read memory [41].

A. Security Property

The development of security properties are required to prevent certain malicious attacks from happening. In our

```

void demo(char *str){
    char short_string[20];
    strcpy(short_string, str);
}

void main (){
    char long_string [150];
    int k;
    for( k = 0; k < 149; k++)
        large_string[k] = 'A';
    demo(long_string);
}

```

Listing 2. The example for buffer overflow.

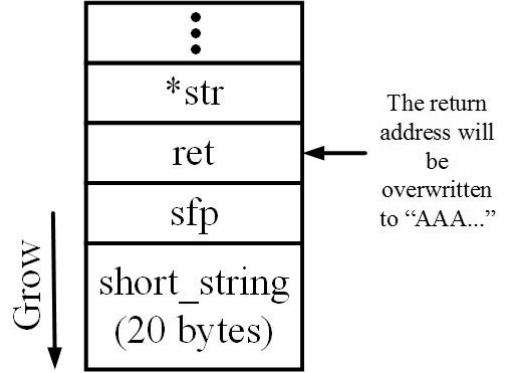


Fig. 9. The stack structure with return address.

demonstration, we try to avoid the control-flow hijacking attacks under the circumstance that both hardware infrastructure and software programs are from untrusted third-party vendors.

1) *Control-Flow Hijacking*: The control-flow hijacking attack poses a serious threat to system security and reliability [42]. This attack enables the attackers to exploit the control flow and write arbitrary data to a control structure in the presence of a vulnerability in the program [43]. During control-flow hijacking, attackers change contents of the control structure and redirect the program counter (PC) to maliciously-injected code by overwriting the return addresses.

To better illustrate the attacking procedure of control-flow hijacking, a sample program of stack-based buffer overflow is listed in Listing 2. The following program, if executed, will cause buffer overflow as the return address is overwritten maliciously.

The structure of the stack is shown in the Figure 9. The sample program pushes the argument `long_string` to the stack and then calls the function `demo`. Following the argument, the return address is saved in the stack at `ret`. After saving the return address of the function, the value of the stack pointer is pushed to the stack at `sfp`. In the function `demo`, the function `strcpy` copies the contents of character array `long_string[]` of length 150 into the smaller array `short_string[]` of length 20. Stuffing `short_string[]` with 150 bytes will overwrite the contents of the stack including the addresses stored at `sfp`, `ret` and `*str`, and results in loss of the return address. The new content in `ret` will redirect the PC to an attacker-defined address `AAA...`, causing unexpected outcomes at function return.

2) *Computer System Security Property*: Computer system security properties are used to represent trusted behaviors of the overall computer system under the assumption that both the hardware and the software provided by the third-party vendors are untrusted. The security properties impose restrictions on behaviors of the hardware, the software, and the boundary in between. Although the definition of computer system security properties may share some similarities with software level security properties, the key difference between them is the assumption on trustworthiness of the underlying hardware infrastructure. Software level security constraints are always built on the assumption that the underlying hardware is trusted and working properly. In contrast, computer system security properties do not make this assumption and treat both the hardware and the software as untrusted. For example, consider a software program which has been proven safe. In this case, control-flow hijacking attacks can still occur when executing the program on untrusted hardware. This is primarily because hardware Trojans triggered by “harmless” software code can take control over software behavior and can alter the contents of the return address resulting in a buffer overflow [41], [44].

In this paper, prevention of the entire computer system from the control-flow hijacking attack is selected as the sample computer system security property. More specifically, this security property can be described as “*when the third-party software program is executed on the third-party processor, no control-flow hijacking attacks will occur.*” In case the consumers are aware of the specific hardware and software, a more clear definition of the security property is derived and sent to third-party vendors along with functional specifications and performance requirements of the 8051 microprocessor and the RC5 software program. The refined security specification based on the knowledge of the hardware and the software will be “*when the RC5 program is executed on the 8051 microprocessor, no control-flow hijacking attacks will occur.*”

The formal security theorems are then derived from the informal security specification and are verified with respect to the instructions of RC5 program and the underlying 8051 microprocessor. The sample computer system is deemed to be secure when the security theorems are provable. Failure to prove the theorems will often indicate the presence of hardware or software Trojans in the computer system and immediately notify the customer of its presence.

B. Security Theorems and Lemmas

Using the security specifications, the third-party vendors develop security theorems for computer system protection. The vendors will initially refine security specifications according to the specific design of the processor (in the form of the HDL code) and the software program (in the form of the assembly code). Formal theorems are then developed based on the refined security properties.

In our demonstration, security theorems are developed to protect the computer system (constituting the RC5 program and 8051 microprocessor) from the control-flow hijacking attack. Note that a successful control-flow hijacking attack involves malicious modification of the program counter either

```
XOR_EQ:
  mov R2, #04H
XOR_EQ_LOOP:
  mov A, @R0
  xrl A, @R1
  mov @R0, A
  inc R0
  inc R1
  djnz R2, XOR_EQ_LOOP
  ret
```

Listing 3. Sub function XOR_EQ.

```
Main:
  mov R0, #OP_A
  mov R1, #OP_B
  acall XOR_EQ
```

Listing 4. Fragment code in main function of RC5.

with hardware malicious logic or through malicious software programs. For the case of 8051 microprocessor, `ret` is one of the instructions which can update the program counter with an arbitrary value. When `ret` is encountered, the contents stored at 08H and 09H will be written to the program counter.³ Therefore, the security property will eventually be converted to the formal theorem with two levels of consideration: first, for any instructions in the RC5 program, except the `ret`, the PC will not be overwritten by arbitrary values (including the operation to copy the contents from 08H and 09H registers to program counter); second, during any functional call, no malicious values will be written into the 08H and 09H registers.

In the rest of this section, we will demonstrate the process of developing security theorems for the computer system in the proposed *unified framework*. We specifically focus on the function XOR_EQ used in the RC5 program. The XOR_EQ function implements the exclusive-or (XOR) operation in the 8051 microprocessor. The assembly code of XOR_EQ is described in Listing 3.

The RC5 program calls the function XOR_EQ using the `ACALL` instruction as illustrated in Listing 4. When executing the `ACALL` instruction, the microprocessor stores the return address to the 08H and 09H registers.

On the other hand, when the `ret` instruction within the XOR_EQ function is executed, the microprocessor loads the contents in registers 08H and 09H to the program counter to finish the execution of the XOR_EQ function. However, the control-flow hijacking attack will alter the normal program flow by overwriting the return address stored in the aforementioned registers. To secure the RC5 program running on top of the 8051 microprocessor, a security property is required which will prevent the PC from being maliciously modified during the execution of the RC5 program.

The proposed framework will prohibit the control-flow hijacking attack in the sample computer system by verifying trusted behaviors of each instruction. The system-level security property, “*when the function - XOR_EQ of the RC5 program is*

³As the first step toward computer system protection, we only consider the situations where limited amounts of nested function calls exist.

```

Theorem RegVerify_RC5 :
forall (t : nat),
t > 0 -> t < 50 ->
state 1 = FETCH ->
reset t = lo::nil ->
ie t = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
s_intpre2 t = lo::nil ->
((bv_eq (s_regs_wr_en t) (hi::lo::lo::nil) = lo /\
bv_eq (s_regs_wr_en t) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr t) (data_08H) = lo /\
bv_eq (s_adr t) (data_09H) = lo)).

```

Listing 5. Security theorem for the computer system.

executed on the 8051 microprocessor, the contents stored in the 08H and 09H registers will not be overwritten," represents the trusted behavior of the XOR_EQ function and, if it is proved, the trustworthiness of the computer system is verified.

On performing a detailed analysis of the function XOR_EQ, we gather that the XOR_EQ includes 7 different instructions and takes 49 clock cycles to operate. Accordingly, the security property is further refined to "when the function - XOR_EQ of the RC5 program is executed on the 8051 microprocessor, the contents stored in the 08H and 09H registers will not be changed during the 49 clock cycles".

An analysis of the 8051 microprocessor structure further shows that the permission to write on the 08H and 09H registers depend on the enable signals `s_regs_wr_en` and `s_adr` of the module `control_mem_rtl` (See Figure 8). The 08H register is updated when the following two conditions are satisfied in the same clock cycle: 1) the control signal `s_regs_wr_en` is equal to 100 or 101 and 2) the control signal `s_adr` is equal to 0000 or 1000. Similarly, the 09H register is updated when the following two conditions are satisfied in the same clock cycle: 1) the control signal `s_regs_wr_en` is equal to 100 or 101 and 2) the control signal `s_adr` is equal to 0000 or 1001.

Supported by the above mentioned details of the RC5 program and the 8051 architecture, the formal theorem for the security property is constructed in Listing 5.

Within the formal security theorem, several pre-conditions are explicitly specified: 1) $t > 0 \rightarrow t < 50$ means that 49 clock cycles are considered (as the function XOR_EQ takes 49 clock cycle to execute); 2) `state 1 = FETCH` indicates that the function is executed from the initial state - FETCH; and 3) `reset t = lo::nil`, `ie t = lo::...` and `s_intpre2 t = lo::nil` imply that our *unified framework* does not handle reset and interrupt during the operation of XOR_EQ. An implicit pre-condition of the theorem is the computer system itself because the proof of the theorem is built upon the sample computer system. The system-level security theorem formally specifies that no modifications on 08H and 09H registers will occur for the given pre-conditions. The variables `data_08H` and `data_09H`, of the formal theorem represent the binary code 00001000 and 00001001 respectively. The function `bv_eq` compares two binary code and return the result `lo` when there is a match between the codes and `hi` otherwise.

Following the same procedure, the theorems are also designed for nested function calls. After analyzing the

```

Theorem RegVerify_NestRC5 :
forall (t : nat),
t > 0 -> t < 50 ->
state 1 = FETCH ->
reset t = lo::nil ->
ie t = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
s_intpre2 t = lo::nil ->
((bv_eq (s_regs_wr_en t) (hi::lo::lo::nil) = lo /\
bv_eq (s_regs_wr_en t) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr t) (data_08H) = lo /\
bv_eq (s_adr t) (data_09H) = lo) /\
bv_eq (s_adr t) (data_10H) = lo) /\
bv_eq (s_adr t) (data_11H) = lo)).

```

Listing 6. Security theorem in the nested function calls.

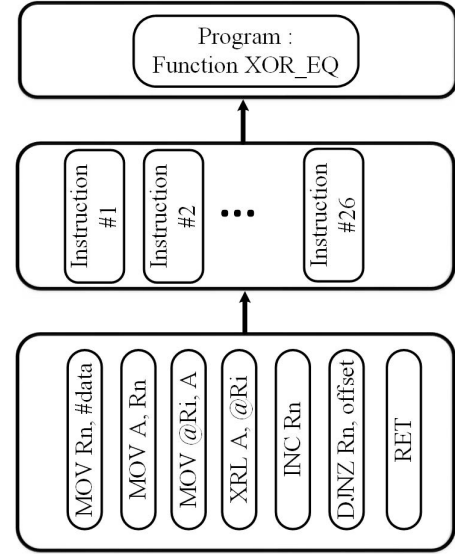


Fig. 10. Hierarchical proof construction process.

8051 microprocessor structure, the return address for the nested sub-function is stored in the 10H and 11H (using a stack to store the return address). Assuming that there are still 49 clock cycles in this nested function, the formal theorem for the security property is constructed in Listing 6.

C. Proof Construction

The proof construction process is the most time-consuming step of our framework. The computer system built from third-party resources is trusted only if the proof of the security theorem can be provided. In order to speed up the proof construction process and lower the design cost, the *lemma development* and *theorem development* methods are heavily leveraged in the *unified framework*.

According to the working procedure of *lemma development* and *theorem development*, the Theorem *RegVerify_RC5* can be seen as the *Theorem for Computer System* of Figure 7, while the Lemmas *RegVerify_mov_rr_data* can be treated as *Top Lemma* of Figure 6. As shown in Figure 10, an intermediate level is inserted to configure parameters of lemmas of the bottom layer. An example lemma in the intermediate level is *RegVerify_Instruction1*, shown in Listing 8. In the bottom level, lemmas are designed for the individual instruction type,

```

Lemma RegVerify_mov_rr_data :
forall (t : nat) (data_load : bus_value),
state t = FETCH ->
rom_data_i t = mov_r2_d ->
rom_data_i (S t) = data_load ->
reset t = lo::nil -> reset (S t) = lo::nil ->
ie t = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
ie (S t) = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
s_intpre2 t = lo::nil -> s_intpre2 (S t) = lo::nil ->
(bv_eq (s_regs_wr_en t) (hi::lo::lo::nil) = lo/\
bv_eq (s_regs_wr_en t) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr t) (data_08H) = lo/\
bv_eq (s_adr t) (data_09H) = lo) /\
(bv_eq (s_regs_wr_en (S t)) (hi::lo::lo::nil) = lo/\
bv_eq (s_regs_wr_en (S t)) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr (S t)) (data_08H) = lo/\
bv_eq (s_adr (S t)) (data_09H) = lo)).

```

Listing 7. Top lemma for mov instruction.

```

Lemma RegVerify_Instruction1:
state 1 = FETCH ->
rom_data_i 1 = mov_r2_d -> rom_data_i 2 = data_04H ->
reset 1 = lo::nil -> reset 2 = lo::nil ->
ie 1 = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
ie 2 = lo::lo::lo::lo::lo::lo::lo::lo::lo::nil ->
s_intpre2 1 = lo::nil -> s_intpre2 2 = lo::nil ->
(bv_eq (s_regs_wr_en 1) (hi::lo::lo::nil) = lo/\
bv_eq (s_regs_wr_en 1) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr 1) (data_08H) = lo/\
bv_eq (s_adr 1) (data_09H) = lo) /\
(bv_eq (s_regs_wr_en 2) (hi::lo::lo::nil) = lo/\
bv_eq (s_regs_wr_en 2) (hi::lo::hi::nil) = lo) /\
(bv_eq (s_adr 2) (data_08H) = lo/\
bv_eq (s_adr 2) (data_09H) = lo)).

```

Listing 8. Lemma for the first two clock cycles.

and the *lemma development* is used without considering any data or time inputs.

The hierarchical process of proof construction for the XOR_EQ function is shown in Figure 10. In this approach, the lemmas in the upper level access the lemmas in the lower level during the proof construction process. At the lowest level of the hierarchy, lemmas (e.g., RegVerify_mov_rr_data shown below) are proved for instructions (from the ISA of 8051) with respect to the pre- and post-condition of the security theorem, without considering the specific situations such as time and data inputs. In the upper level, the lemmas proved at the lowest level for instructions (e.g., Mov R2, #04H) are reused with the details of time (2 clock cycles for executing the Mov instruction of XOR_EQ function) and input (04H) (e.g., RegVerify_Instruction1 shown in Listing 7). At the top of the hierarchy, all the proved lemmas of the preceding stages are used to prove the security theorem of the XOR_EQ function.

In the lemma RegVerify_mov_rr_data, the variable data_load takes any binary code as input and rom_data_i takes data from the ROM of the 8051 microprocessor. The pre-conditions for the lemma are: (1) state t = FETCH indicates that the function is executed from the initial state - FETCH at the t clock-cycle, (2) rom_data_i t = mov_r2_d implies that rom_data_i takes the op-code of the mov instruction at t clock cycle, and (3) rom_data_i (S t) = data_load

```

Lemma connect_command_io_mem_fsm:
forall t:nat, command_o t = command_i t.

```

Listing 9. Example for data transmission lemma.

```

Lemma la_fsm_st_MOV_RR_DATA:
forall (t:nat), state_fsm t = EXEC1 ->
s_instr_category t = IC_MOV_RR_DATA ->
s_nextstate_fsm t = FETCH/\
s_regs_wr_en_fsm t = hi::lo::lo::nil/\
s_data_mux_fsm t = lo::hi::lo::hi::nil/\
s_adr_mux_fsm t = hi::lo::hi::lo::nil/\
s_pc_inc_en_fsm t = (lo::lo::lo::hi::nil).

```

Listing 10. Example for data operation lemma.

TABLE I
TIME CONSUMED FOR VALIDATING THE PROOFS OF THE SYSTEM

Hardware	Software	Time Consuming
8051 MCU	RC5 Algorithm	207 seconds

indicates that rom_data_i takes input data at the t+1 clock cycle.

The precondition rom_data_i 1 = mov_r2_d of the lemma RegVerify_Instruction1 implies that rom_data_i takes the op-code of the mov instruction at the first clock cycle. The next precondition of the lemma, rom_data_i 2 = data_04H signifies that rom_data_i takes the input data 0000 0100 at the second clock cycle.

The connect_command_io_mem_fsm, which connects two submodules, is an example of data transmission lemma. The command_i is an input port of one sub module, while command_o is an output port of another sub module. And the signal will be transmitted from command_o, and received at command_i. This lemma means that “the value is equal between these two ports at any time.”

The la_fsm_st_MOV_RR_DATA presents an example of data operation lemma. This lemma describes a condition statement used in the sub module for the finite state machine in the 8051 microprocessor. The variable state_fsm stands for current state of circuit. And the input op-code is stored in the variable s_instr_category. At any time, when the state_fsm is in EXEC1 status and the value of s_instr_category is IC_MOV_RR_DATA (formal expression of mov), the signals, s_nextstate_fsm, s_regs_wr_en_fsm, s_data_mux_fsm, s_adr_mux_fsm, and s_pc_inc_en_fsm, will be assigned by the specific data.

Meanwhile, the example has been tested on a desktop with 64-bit Intel i7-3370 CPU and 16GB RAM. The result of the time consumption of validation process in the system integrator side, is shown in Table I.

The proof construction process of this paper was successful in proving the security theorem for the RC5 program implemented on the 8051 microprocessor. Consequently, we can conclude that the computer system is secure from any malicious attacks within the domain of the security properties.

VI. DISCUSSION

For the first time, the proposed proof-carrying based *unified framework*, provides a solution to apply system-level security properties to both the hardware and the software domains. Most of the existing security methods assume the trustworthiness of either the hardware or the software; but this assumption is invalidated due to untrusted third-party software and hardware vendors. Our method enhances the security of computer systems by not making this assumption. Furthermore, instead of bridging the semantic gap between the hardware and the software, the *unified framework* completely eliminates the hardware-software boundary. Our *unified framework* provides a platform for improving existing security methods to incorporate system-level protection.

Although the proposed *unified framework* opens a new area for computer system protection, the current work presented in this paper is still at its infancy. Before the framework can be readily applied to computer systems – which are built from third-party hardware and software – several issues need to be addressed.

The proposed framework requires code conversion i.e. conversion of the hardware/software code to its formal counterparts, before the design of security proofs for security theorems. Thus, in our demonstration, we have manually converted a small subset of CISC instruction set of 8051 microcontroller and have automated conversion of a subset of the VHDL language to the language of Coq. This manual code conversion process and the proof construction step reduces scalability of our approach to large-scale processors and/or SoCs. These limitations have propelled us to develop automatic code conversion tool, which relies on existing hardware and software code compilers to convert entire VHDL language and different ISA's (CISC/RISC) into formal logic. The tool required development of a parser, which takes VHDL code of the design as input and produces an abstract syntax tree. Subsequently, we generate the code for the Coq theorem prover (represented in Gallina) by using the conversion rules specified in *Formal HDL* [45].⁴ On using our tool, the estimated effort for converting design of Intel 8051 core to Gallina was about 10 seconds. In contrast, an estimated manual effort of 10–14 days would have been required for converting the same VHDL code to Gallina.

Meanwhile, we have build proof rules inside the VHDL (or Verilog) so that the formal reasoning can be applied directly on computer systems without code conversion.⁵ To reduce proof construction time, we are developing tactics and proof library of frequently used lemmas in Coq.

Furthermore, the current *unified framework* performs static verification of the computer system. That is, no interrupts are considered in the static software code. Exceptions are not considered either, even though some exceptions can be statically identified through the analysis of the static software code (e.g., the divided by zero exception can be detected if

inputs are known). For these reasons, third-party vendors will be able to insert malicious code during the execution of the software program on the hardware infrastructure. To deal with these problems, we propose extending the static framework for dynamic assertions, similar to software level code assertions. However, this method will require modifications of the original computer system and can result in performance overhead.

VII. CONCLUSION

The semantic gap between hardware and software is a major obstacle in the development of system-level security properties for the entire computer system. Security properties defined in either domain are often invalid for the entire system. As a result, inconsistencies arise in the definition of security properties for the system – facilitating attackers to trigger unexpected behaviors. The security threat is worsened by the increase of third-party resources in the hardware infrastructure and software programs. A proof-carrying based *unified framework* is developed which, for the first time, bridges the semantic gap by converting the whole computer system into the same formal platform. Supported by the proposed framework, system-level security properties can be unambiguously developed and applied to the entire computer system – thus preventing attackers from triggering malicious behaviors at the software-hardware boundary.

REFERENCES

- [1] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware Trojans," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2008, pp. 632–639.
- [2] R. Rad, J. Plusquellic, and M. Tehranipoor, "Sensitivity analysis to hardware Trojans using power supply transient signals," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jun. 2008, pp. 3–7.
- [3] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards Trojan-free trusted ICs: Problem analysis and detection scheme," in *Proc. Design Autom. Test Eur.*, 2008, pp. 1362–1365.
- [4] H. Salmani, M. Tehranipoor, and J. Plusquellic, "New design strategy for improving hardware Trojan detection and reducing Trojan activation time," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jul. 2009, pp. 66–73.
- [5] 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.
- [6] L. Lin, M. Kasper, T. Guneysoy, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware Trojans through side-channel engineering," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), Berlin, Germany: Springer-Verlag, 2009, pp. 382–395.
- [7] L. Lin, W. Burleson, and C. Paar, "MOLES: Malicious off-chip leakage enabled by side-channels," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2009, pp. 117–122.
- [8] G. Bloom, R. Simha, and B. Narahari, "OS support for detecting Trojan circuit attacks," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jul. 2009, pp. 100–103.
- [9] M. Banga and M. Hsiao, "A novel sustained vector technique for the detection of hardware Trojans," in *Proc. 22nd Int. Conf. VLSI Design*, 2009, pp. 327–332.
- [10] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao, "Guided test generation for isolation and detection of embedded Trojans in ICs," in *Proc. 18th ACM Great Lakes Symp. VLSI*, 2008, pp. 363–366.
- [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—CHES* (Lecture Notes in Computer Science), vol. 5747. Berlin Germany: Springer, 2009, pp. 396–410.

⁴Details regarding development of this tool is outside the scope of this paper and will be provided in a future paper.

⁵In this case, we only need to convert software programs into a series of hardware states for unifying the whole computer system in the HDL environment.

- [12] G. Bloom, B. Narahari, R. Simha, and J. Zambreno, "Providing secure execution environments with a last line of defense against Trojan circuit attacks," *Comput. Secur.*, vol. 28, no. 7, pp. 660–669, 2009.
- [13] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan horse detection using gate-level characterization," in *Proc. 46th Annu. Design Autom. Conf. (DAC)*, 2009, pp. 688–693.
- [14] Y. Jin, N. Kupp, and M. Makris, "DFTT: Design for Trojan test," in *Proc. IEEE Int. Conf. Electron. Circuits Syst.*, Dec. 2010, pp. 1175–1178.
- [15] 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.
- [16] M. Banga and M. 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.
- [17] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 159–172.
- [18] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *Proc. IEEE Symp. Secur. Privacy*, May 2007, pp. 296–310.
- [19] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2011, pp. 1–18.
- [20] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity OS from untrusted extensions," in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2011, pp. 1–17.
- [21] C. Sturton, M. Hicks, D. Wagner, and S. King, "Defeating UCI: Building stealthy and malicious hardware," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2011, pp. 64–77.
- [22] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, 2009, pp. 189–194.
- [23] S. Drzevitzky and M. Platzner, "Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach," in *Proc. 6th Int. Workshop Reconfigurable Commun.-Centric Syst.-Chip*, 2011, pp. 1–8.
- [24] 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.
- [25] 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.
- [26] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2011, pp. 67–70.
- [27] 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.
- [28] X. Guo, R. G. Dutta, and Y. Jin, "Hierarchy-preserving formal verification methods for pre-silicon security assurance," in *Proc. 16th Int. Workshop Microprocessor SOC Test Verification (MTV)*, 2015, pp. 1–6.
- [29] 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)*, pp. 124–129, May 2016.
- [30] INRIA. (Sep. 2010). *The COQ Proof Assistant*. [Online]. Available: <http://coq.inria.fr/>
- [31] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang. (POPL)*, 1997, pp. 106–119.
- [32] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proc. Design Autom. Conf.*, 2015, pp. 1–6.
- [33] D. Yu, N. A. Hamid, and Z. Shao, "Building certified libraries for PCC: Dynamic storage allocation," in *Sci. Comput. Programming*, 2003, pp. 363–379.
- [34] Oski Technology, Inc. accessed on Nov. 7, 2016. [Online]. Available: <http://www.oskitechnology.com/>
- [35] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," *SIGPLAN Notes*, vol. 41, no. 6, pp. 401–414, 2006.
- [36] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *Proc. IEEE 30th VLSI Test Symp. (VTS)*, Apr. 2012, pp. 252–257.
- [37] *Oregano-Systems, 8051 IP Core*. accessed on Nov. 7, 2016. [Online]. Available: http://www.oreganosystems.at/?page_id=96
- [38] *Malicious Processor Design Challenge*. accessed on Nov. 7, 2016. [Online]. Available: <http://isis.poly.edu/~jv/MP.zip>
- [39] *Intel MCS51 Family User Manual*, Intel Corp., Santa Clara, CA, USA, 1981.
- [40] R. L. Rivest, "The RC5 encryption algorithm," in *Proc. 2nd Int. Workshop Fast Softw. Encryption (FSE)*, 1994, pp. 86–96.
- [41] Y. Jin, M. Maniatakis, and Y. Makris, "Exposing vulnerabilities of untrusted computing platforms," in *Proc. IEEE 30th Int. Conf. Comput. Design (ICCD)*, Sep./Oct. 2012, pp. 131–134.
- [42] E. H. Spafford, "The internet worm program: An analysis," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, 1989.
- [43] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, 1996.
- [44] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust*, Jul. 2009, pp. 50–57.
- [45] 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.



language design.

Xiaolong Guo (S'14) received double bachelor's degrees from the Beijing University of Posts and Telecoms (BUPT) and University of London (UL) in 2010, respectively, and the M.S. degree from BUPT. He is currently pursuing the Ph.D. degree in electrical engineering with the University of Central Florida, Orlando, FL.

His current research interests include design of scalable verification methods for hardware IP protection, trusted SoC verification, cyber security, formal methods, program synthesis, and secure



of attack detection and mitigation software for autonomous systems, and synthesis of controllers for multiagent systems.

Raj Gautam Dutta (S'07) received the B.Tech. degree in electronics and communication from Visvesvaraya Technological University, India, in 2007, and the M.S. degree in electrical engineering with an emphasis on control systems from the University of Central Florida, USA, in 2011, where he is currently pursuing the Ph.D. degree with the EECS Department.

His current research interests include development of security solutions for semiconductor soft IP cores by using formal verification techniques, design



Yier Jin (M'12) received the B.S. and M.S. degrees from Zhejiang University, China, in 2005 and 2007, respectively, and the Ph.D. degree from Yale University in 2012, all in electrical engineering. He is currently an Assistant Professor with the EECS Department, University of Central Florida.

His research focuses on the areas of trusted embedded systems, trusted hardware intellectual property (IP) cores, and hardware-software co-protection on computer systems. 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 IP protection scheme. He is also interested in the security analysis on Internet of Things (IoT) and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. He is a recipient of the DoE Early CAREER Award in 2016 and the best paper award of the DAC'15 and ASP-DAC'16.

Data Secrecy Protection through Information Flow Tracking in Proof-Carrying Hardware IP

Part I: Framework Fundamentals

Yier Jin, *Member, IEEE*, Xiaolong Guo, *Student Member, IEEE*, Raj Gautam Dutta, *Student Member, IEEE*, Mohammad-Mahdi Bidmeshki, *Student Member, IEEE*, Yiorgos Makris, *Senior Member, IEEE*

Abstract—Proof-carrying hardware intellectual property (PCHIP) is a previously proposed framework for ensuring trustworthiness of third-party hardware IP through the development of formal proofs for security properties designed to prevent introduction of malicious behavior. Based on this framework, we introduce new approaches for assuring that the secrecy of internal information in a hardware design is not compromised by design flaws or malicious hardware Trojans. Specifically, we devise two PCHIP-based information flow tracking (IFT) approaches which enhance the formal PCHIP framework with secrecy tags and/or sensitivity levels in order to provide mechanisms for proving that sensitive information does not reach undesired sites. To assist in the development of data secrecy properties, we also introduce the concept of theorem generation functions which enable generation of security theorems independent of the target circuit, thereby paving the way for proof automation. Additionally, we enhance the PCHIP framework with a hierarchy-preserving methodology and we show its utility in preventing malicious data modification which may indirectly result in sensitive information leakage, such as by modifying the secret key in a cryptographic core. This enhanced PCHIP framework also enables development of hybrid module libraries which contain hardware description language (HDL) code along with proofs of lemmas for these modules. These module libraries can then be used for hierarchically proving security properties in higher-level designs, thereby reducing the proof development burden in the general PCHIP framework. Efforts toward automation of the proposed methodologies, as well as evaluation of their effectiveness in identifying design flaws or hardware Trojans in various cryptographic hardware designs are presented in part II of this paper series.

Keywords—Hardware Trust, Proof-Carrying Code, Data Secrecy Protection, Dynamic Information Assurance, Information Flow Tracking

I. INTRODUCTION

With sensitive data, such as user names, passwords, or banking information routinely being stored and communicated in our everyday use of electronic devices, data secrecy protection

has become a key objective of computer security research. Encryption plays an important role in preventing the direct exposition of data to adversaries. However, culprits invest their utmost effort in getting access to such valuable data through various well-known or newly-discovered vulnerabilities of computer systems or through introduction of malicious software. Accordingly, numerous approaches have been devised toward reducing the likelihood of sensitive information leakage in such systems. Among them, information flow tracking (IFT) [1] is a powerful approach for preventing sensitive information from reaching untrusted sites. In this approach, labels representing secrecy/trust levels are assigned to data and operations on data are extended to include operations on their labels, based on predefined information flow policies. Access or propagation of data with sensitive labels is, thereby, restricted to trusted segments of the code or system and is forced to abide by the desired information flow policies.

IFT can be implemented at various levels and can approach the problem from different perspectives. Static IFT enforces information flow policies at compile time and does not introduce any overhead at runtime [2]. However, its scope of enforcement is limited. Dynamic IFT seeks to remove this limitation and implements IFT at runtime [3], at the cost of introducing performance and memory overhead in the system. To reduce this overhead and to make IFT more accurate, hardware-assisted IFT methodologies have also been introduced. For example, the authors in [4] proposed a dynamic IFT framework with all internal storage elements equipped with a security tag. Similarly, the authors in [5] focused on pointer-tainting in order to prevent both control data and non-control data attacks. Besides IFT, the hardware may also be enhanced to help with preventing information leakage, such as in the InfoShield architecture, which applies restrictions to operations on sensitive data [6]. Similarly, the RIFLE architecture is developed on top of an information flow security (IFS) instruction set architecture (ISA), where all states defined by the base ISA are augmented by labels [7]. More recently, a new software-hardware architecture was developed to support more flexible security policies, either for protecting sensitive data [8] or for preventing malicious operations by untrusted third-party OS kernel extensions [9]. Authors in [10] extended IFT to gate-level implementations by modifying the restrictive IFT operation definitions used in higher levels.

The aforementioned hardware-supported information tracking and protection schemes are based on the assumptions

Manuscript received June 26, 2016; revised January 20, 2017; accepted May 10, 2017. This work has been partially supported by the National Science Foundation (NSF-1318860, NSF-1319105) and the Army Research Office (ARO W911NF-12-1-0091, ARO W911NF-16-1-0124).

Yier Jin, Xiaolong Guo, and Raj Gautam Dutta are with the department of Electrical and Computer Engineering, The University of Central Florida, Orlando, FL 32816, USA. E-mail: yier.jin@eecs.ucf.edu, {guoxiaolong, rajgautamdutta}@knights.ucf.edu

Mohammad-Mahdi Bidmeshki and Yiorgos Makris are with the department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX 75080, USA. E-mail: {bidmeshki, yiorgos.makris}@utdallas.edu

that the underlying hardware infrastructure is trusted and that any modifications to the hardware architecture have passed functionality and security verification. Yet the recent emergence of hardware Trojans and hardware-level back-doors as a plausible threat [11], [12] has casted doubt on the validity of these assumptions and has provided attackers with alternative leakage paths, which current IFT schemes do not cover [13], [14]. Indeed, globalization of the integrated circuit (IC) supply chain has intensified the concern that unknown and potentially malicious capabilities may be inserted in a circuit or system. While inclusion of such capabilities is possible at any stage [12], modifying the hardware at the design stage is far easier than tampering with its layout mask at fabrication. Additionally, hardware designers regularly use third-party hardware IPs to expedite the design process. Consequently, flaws or malicious capabilities in hardware IPs can easily expose the end system to attacks. *As a result, besides IFT in software, hardware-level information assurance is also required to ensure that the hardware itself obeys the desired information flow policies and does not leak sensitive data, and to establish system-level trust by detecting design flaws and/or hardware Trojans.*

To this end, in Part I of this paper series we introduce several variants of a formal methodology for assessing trustworthiness of a design described in a hardware description language (HDL). These variants build upon a previously proposed proof-carrying hardware IP (PCHIP) framework [15], which we enhance in order to support IFT and hierarchy-preserving proof development. Then, in Part II, we describe our efforts towards automating the enhanced PCHIP framework and we evaluate the applicability and effectiveness of its capabilities in detecting design flaws and/or hardware Trojans in various cryptographic cores. The remainder of this paper is organized as follows. Section II introduces the threat model considered in this work, Section III reviews related efforts, and Section IV provides an overview of the proposed PCHIP-based methodologies for data secrecy protection. In Section V, we review the principles of the original PCHIP framework. In Section VI, we provide details of the two formal models devised for supporting IFT within the PCHIP framework. In Section VII, we review the existing flattened formal model in the original PCHIP, we present the new hierarchy-preserving alternative, and we describe how the enhanced PCHIP framework can be used to prevent malicious data modification. Conclusions are drawn in Section VIII.

II. THREAT MODEL

The methods proposed in this work seek to detect *sensitive information leakage*, caused either by intentional malicious modifications or by inadvertent flaws in the design of a circuit. Accordingly, our work mainly targets circuits handling sensitive information, such as cryptographic hardware. We focus on designs described as HDL code (e.g. soft hardware IP) and we assume that any leakage will occur in digital form, through *primary physical outputs* of the design, to which an adversary is expected to have access. Since the proposed methods are only able to reason in the digital domain, information leakage

through side channels (timing, power, etc.) is not directly considered in our work. However, as we discuss in Part II of this paper series, a knowledgeable user may be able to leverage the proposed methods in order to reveal suspicious internal locations in the design, where the digital information leaked through side-channels may be originating from.

Our threat model assumes that the attack surface is the HDL description of the design. Accordingly, the attacker resides either at the site of a 3rd party IP provider, if the design is sourced externally, or in-house, if the design is performed internally. Evidently, a design sourced as 3rd party IP, which is later modified in-house, also falls within the scope of our methods. We emphasize that, while a large portion of the hardware security literature focuses on the inclusion of hardware Trojans at the fabrication stage, other stages of hardware design and fabrication are not immune to such threats [12]. In fact, the threat model wherein malicious capabilities are introduced through 3rd party IP has attracted similar levels of attention and numerous methodologies targeting hardware Trojans introduced at the design stage have been developed, as we briefly review in Section III. Indeed, as the use of third-party hardware IPs has become ubiquitous in contemporary hardware design, this threat is intensified as not all IP vendors can be trusted. Even in the case of a trusted hardware IP vendor, a single rogue adversary involved in the design, acquisition or utilization of the hardware IP can introduce malicious capabilities in the design. Moreover, undiscovered design flaws and in-house attacks can also compromise the hardware at the design stage. Therefore, developing methods for detecting sensitive information leakage due to intentional or inadvertent vulnerabilities in the design of an electronic circuit has become paramount.

III. RELATED WORK

Several methodologies have been introduced for identifying hardware Trojans introduced at the HDL code of hardware designs and soft hardware IPs. For example, FANCI [16] uses statistical analysis to detect nearly unused logic in a design, which may be suspicious. Similarly, VeriTrust [17] searches for dedicated Trojan triggers in the design. While such methods are systematic, smart Trojans may still evade their checking mechanisms [18]. Alternatively, COTD [19] employs controllability and observability analysis of a gate-level netlist to mark potential Trojan gates. Through controllability- and observability-based clustering, it asserts that Trojan gates have significant inter-cluster distance from genuine gates, and therefore, may be distinguished. Other approaches employ test patterns, specifically generated and directed towards detecting hardware Trojans [20], [21]. Such methods have been gradually improving, yet are still faced with a very large space of options, which they need to intelligently prune. Furthermore, none of the above methods focuses on information leakage.

Several language-level approaches have also been introduced, seeking to enforce information flow policies on hardware designs. Caisson [22] is a hardware description language with static information flow verification capabilities at design

TABLE I
SUMMARY OF PROPOSED PCHIP-BASED METHODOLOGIES

Method	Type	Complexity	Expressiveness	Ease of use
Static information flow tracking	IFT	Low	Low	Easy
Dynamic information assurance	IFT	Moderate	Moderate	Moderate
Flattened functional model	General	High	High	Hard
Hierarchical functional model	General	High	High	Moderate

time. Sapper [23] provides a language which employs static analysis at compile-time but inserts dynamic checks in the resulting hardware in order to enforce information flow policies. SecVerilog [24] enforces information flow policies by introducing a type system. It is essentially Verilog, which has been extended with type annotations. SecVerilog is powerful and provides a very accurate information flow model by supporting dependent security types (i.e., labels defined as a function of signal values). Also, while such methodologies are very useful in designing secure hardware, such as microprocessors, their main focus is on storage and timing channels, not on possible hardware Trojans. Moreover, as explicitly stated in the threat model considered in [24], these methods assume a software-level adversary, as opposed to the hardware-level adversary considered in our work.

Finally, a method which detects hardware Trojans leaking sensitive information based on gate-level IFT is introduced in [25]. While this method takes advantage of precise IFT at the gate-level, it requires exploration of the entire signal/value space using a SAT solver and/or a model checker, which results in state explosion and limits its scope. Moreover, to evade detection in this approach, it is sufficient for an adversary to have only one operation which propagates the key to the output (e.g. eXclusive-OR), while hijacking all the encryption rounds in a cryptographic core.

IV. PROPOSED DATA SECRECY PROTECTION METHODS

The proof-carrying hardware IP (PCHIP) framework [15] introduced a formal methodology for ensuring trustworthiness of soft hardware IPs and designs delivered as code in HDL. PCHIP augments the hardware IP with machine-checkable formal proofs of security properties which are agreed upon by the IP developer and the IP consumer and which are crafted in a way that prevent inclusion of malicious capabilities violating these properties in the design. The IP consumer may, then, use an automatic proof checker to verify the proofs and ensure that the security properties are obeyed by the design.

The original PCHIP framework does not provide any means for enforcing information flow policies, which are particularly important for ensuring trustworthiness of cryptographic cores and hardware IPs that process sensitive data. Nevertheless, PCHIP is quite flexible and can serve as a foundation for developing an array of IFT methodologies, each of different complexity, expressiveness and ease of use. Accordingly, in this paper we describe four such PCHIP-based methodologies seeking to protect data secrecy in hardware designs. Table I contrasts these four methods, enabling a user to select the most appropriate one, based on the design requirements and the security properties that need to be proven.

The first two of these methods, which we introduce in Section VI, are specifically crafted to incorporate IFT in the PCHIP framework and aim at preventing sensitive information leakage. The *static information flow tracking* [26] method provides a starting point which demonstrates how the HDL code of a design can be converted to a formal theorem proving language representation wherein IFT is supported. This initial option is straightforward and easy to apply, but its scope is limited and it can only handle simple hardware designs. In contrast, the *dynamic information assurance* methodology [27] which we introduce next, adds support for more complex and pipeline designs, at the cost of somewhat increasing complexity and requiring further effort to apply. While early versions of these two approaches were presented as part of our initial efforts [26], [27], herein we extend them and present them in more detail, emphasizing their integration within the generalized PCHIP framework. Additionally, we introduce the notion of *security property theorem generation functions*. These functions can be used to generate security theorems for information flow policies, independent of the target circuit, and are an essential step towards automating the IFT framework. It is worth noting that these two models are geared towards manual conversion of the design to a formal theorem proving language representation and enforcement of information flow policies, and they intentionally rely on specific procedures for making such conversion simpler while retaining the essential requirements for performing IFT in the formal representation. Moreover, as compared to the original PCHIP framework which retains the entire functionality, these methods preserve the structure but only partially retain the functionality of the hardware design. For example, all binary operations are represented in the same way in the formal representation. While efforts towards automating these methods are presented in Part II of this paper series, we consider it essential to first describe the manual conversion-based models in order to help in grasping the basics of PCHIP-based IFT.

The last two of these methods, which we introduce in Section VII, are motivated by the observation that information leakage may occur not only through manipulation of information flow but also through manipulation of sensitive data, such as the secret key in a cryptographic core, which may not be directly detectable through our IFT methods. Therefore, in order to prevent such malicious modification of sensitive data and ensure integrity, we resort to a solution employing the general PCHIP framework which, as opposed to the two IFT methodologies described above, converts the entire functionality of the design to the formal representation. The first of these two methods utilizes the *flattened functional model* first introduced by the general PCHIP framework. The second method introduces a *hierarchical functional model*, which is based on the flattened model but preserves the hierarchy of the hardware design in the formal representation. For each of these two options, we describe how this framework can be utilized to develop security properties which can prevent malicious modification of sensitive data in hardware designs. While either the flattened or the hierarchical functional model can be utilized for this purpose, the hierarchical approach makes proof development and reuse of previously proved

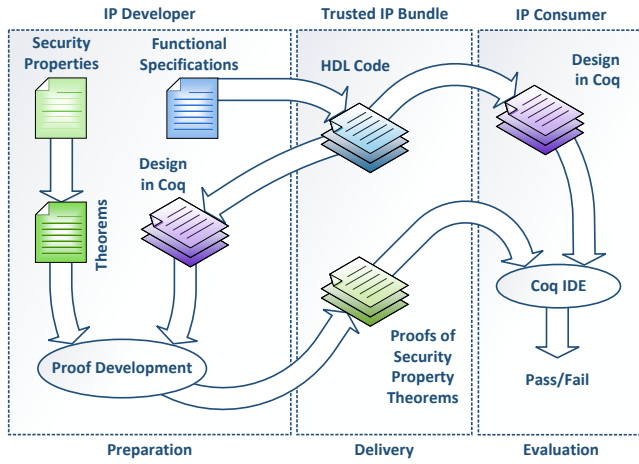


Fig. 1. PCHIP framework

lemmas significantly easier and is more appropriate for large designs. However, the flattened model may also provide some advantages. As we will demonstrate in Part II of this paper series, the flattened model allows us to build a fully automated PCHIP-based IFT framework. Also, flattened and hierarchical functional models can be mixed together, creating a combined model. In this model, larger modules are converted hierarchically while the modules instantiated inside them use the flattened approach. This provides a degree of flexibility to developers who may prefer to develop lemmas only for larger modules and do not want to overuse modules in the formal representation.

V. PROOF-CARRYING HARDWARE IP (PCHIP) OVERVIEW

In this section, we briefly review the PCHIP framework, which is depicted in Fig. 1. In this framework, along with the HDL code for a design, IP developers are required to develop and deliver another essential piece: formal proofs that the code abides by a set of security properties that are agreed upon by both the IP developer and the IP consumer. These properties do not necessarily impose restrictions on the details of implementation. Rather, they institute a high-level boundary of trusted functionality, which prevents misbehavior or unsolicited actions. For example, a security property for a microprocessor IP could be defined as follows: Each instruction is only allowed to access memory locations which are specified in the corresponding fields of its op-code [28]. This property prevents stealthy information leakage. However, it does not restrict the details of instruction implementation.

Mechanized proof development and checking requires a theorem-proving language and a proof-checking environment, such as Coq and CoqIDE [29], respectively. Therefore, in order to be applicable and leverage the rich collection of hardware IPs developed in HDLs such as Verilog and VHDL, PCHIP defines conversion rules from HDLs to a Coq representation. Consequently, PCHIP does not intervene in the current hardware IP design and test methodology, as is the case when introducing a new formal HDL [30]. Rather, it adds extra steps in parallel to the current design methodology, namely converting to Coq, stating security properties

as theorems in Coq, constructing proofs for such theorems based on the hardware design, and delivering those proofs along with the HDL code to the IP consumer. PCHIP does not inflict IP consumers with much extra burden. Along with the IP developers, they need to agree on the desired security properties. The onerous task of proof development is, then, the responsibility of the IP developers. PCHIP can be employed in various types of hardware and can be adaptively modified to fit the requirements of the design and the IP consumer.

While PCHIP establishes a very powerful formal framework for proving security properties of an IP, it does not provide any means for enforcing information flow policies. Nevertheless, the underlying representation of a hardware design in the Coq formal language can be flexibly varied to develop such a provision. In the following section, we describe how IFT capabilities can be added to the original PCHIP framework.

VI. INFORMATION FLOW TRACKING IN PCHIP

In this section, we enhance the PCHIP framework to support IFT in the HDL description of an IP, with the intention of using this capability for analyzing the security of cryptographic hardware cores. We start with a static IFT methodology, targeting small and simple designs, and we then evolve it into a dynamic IFT methodology which can enforce information flow policies for larger, more complicated and pipelined hardware designs. For each methodology, we describe how to express the hardware design in the formal representation required for the verification of information flow policies.

A. Static Information Flow Tracking

The first formal model we introduce, named *Coq formal semantic model*, is used in the static IFT scheme and is developed in such a way that it can precisely describe the structure of the circuit but has loose restrictions on the functionality of any operators. In other words, the architecture of the circuit is accurately described in the formal semantic model but proof writers have flexibility in defining the functionality of the Coq circuit. This formal semantic model, as we will demonstrate later, is quite effective in statically tracking information flow inside the circuit in the PCHIP framework, supporting the target objective of data secrecy property verification. The formal semantic model includes preliminary definitions of signals, syntax of complex expressions, and semantics of operators.

1) *Signal Definition*: Values of signals are defined in an inductive set with two constructors, *hi* and *lo*, indicating high voltage level and low voltage level, respectively. Instead of defining one-bit signals and multi-bit buses separately, we unified both definitions under the bus scope, i.e., a one-bit signal is treated as a one-bit wide bus. The bus is then defined as a mapping of time, specified in clock cycles and given as a natural number, onto a *bus_value*, which is composed of a list of signal values. The natural number *t* defines an important property in temporal logic, namely that values of buses vary according to the system clock cycles. We also define a function to obtain the width of the bus, *bus_length*. These definitions are written in Coq as follows:

```

Inductive value := lo | hi.
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition bus_length (b : bus) :=
  fun t : nat => length (b t).

```

2) *Signal Operations*: We construct bus handling methods in the formal semantic model which include logic operations such as and, or, xor, etc., as well as bus comparisons such as checking for bus equality, bus_eq, less-than comparison, bus_lt, etc. Conditional statements of Register Transfer Level (RTL) code, such as **if**...**else**, check whether signals are on or off. To incorporate this functionality in our formal semantic model, we add a special function, bus_eq_0, which compares the bus value to hi or lo. Note that **Fixpoint** defines a recursive function in Coq.

```

Fixpoint bv_bit_and (a b : bus_value)
{struct a} : bus_value :=
  match a with
  | nil => nil
  | la :: a' =>
    match b with
    | nil => nil
    | lb :: b' => (v_and la lb)::(bv_bit_and a' b')
  end
end.

```

```

Definition bus_bit_and (a b : bus) : bus :=
  fun t:nat => bv_bit_and (a t) (b t).

```

```

Fixpoint bv_eq_0 (a : bus_value)
{struct a} : value :=
  match a with
  | hi :: lt => lo
  | lo :: lt => bv_eq_0 lt
  | nil => hi
  end.

```

```

Definition bus_eq_0 (a : bus) (t : nat) : value :=
  bv_eq_0 (a t).

```

3) *Bus Slicing*: In a circuit, it is often the case that operations are performed on certain bits of the bus, but not the entire bus. Most hardware description languages, therefore, provide quite flexible syntax to define bus length and bus bit-sequence. In order to support similar flexibility in our Coq semantic model, we developed two bus-slicing operations to shuffle data bits from lower bit positions to higher bit positions and vice-versa. Bit selection notations are also proposed to simplify code writing, such as [,] for the sliceD function defined below. The firstn and skipn functions used in these definitions get a number n and a list l, and return a list containing the first n elements of l, or a list skipping the first n elements of l, respectively.

```

Definition sliceA (b : bus) (p1 p2 : nat) : bus :=
  fun t : nat => firstn (p2-p1+1) (skipn (p1-1) b).

```

```

Definition sliceD (b : bus) (p1 p2 : nat) : bus :=
  fun t : nat =>
    rev (firstn (p1-p2+1) (skipn p2 (rev b))).

```

```

Notation "b [ m , n ]" := (sliceD b m n)
  (at level 50, left associativity).

```

```

Notation "b @ [ m , n ]" := (sliceA b m n)
  (at level 50, left associativity).

```

4) *Expressions*: On top of signal definitions and operation rules, we build expressions to represent more complicated circuit logic. An expression is defined as an inductive set with operators to construct new expressions or combine expressions. Plenty of operators are supported, varying from basic logic operations (AND, OR, etc.) to sophisticated data manipulation (S-box mapping, permutation, etc.). The expression definition shown below is an excerpt from the complete

expression definition, wherein operators have been chosen with particular attention to common tasks performed in cryptographic IP cores, since it is highly likely that data secrecy properties will have to be proven for such designs. A constant value list and a bus can be directly converted to expressions using the econv and econb constructors, respectively. The eand, eor and exor constructors connect two expressions to form a new expression, by performing logical AND, OR and eXclusive-OR operations, respectively. The exor_key also performs the eXclusive-OR operation with keys (or sub-keys) as one input (details of this operation will be elaborated on in Section VI-A8 and Section VI-B2). The perm and sbbox constructors are used to indicate permutation and S-box mapping operations. In this Coq semantic model, it is unnecessary to specify how the permutation and/or S-box mapping is actually performed. These structural constructors liberate the proof writers from tedious functional conversion, which may be unnecessary for data secrecy property checking and enforcement of information flow policies.

```

Inductive expr :=
  | econv : bus_value -> expr
  | econb : bus -> expr
  | eand : expr -> expr -> expr
  | eor : expr -> expr -> expr
  | exor : expr -> expr -> expr
  | exor_key : expr -> expr -> expr
  | enot : expr -> expr
  | cond : expr -> expr -> expr -> expr
  | perm : expr -> expr
  | sbbox : bus -> expr
  (* ... *)

```

Evaluation of expressions is recursively defined to calculate the value of an expression at a specified time (denoted by the t parameter) and return data of type bus_value, a list of values whose length depends on the width of the underlying bus. A close look at the eval function supports our claim that some expressions, such as perm, only denote that a permutation operation will be performed on the underlying bus, but do not specify the exact nature of the permutation. Of course, other expressions, such as eand which performs a logical AND on two sub-expressions, result in a case where both functionality and structure are fully specified.

```

Fixpoint eval (e : expr) (t : nat)
{struct e} : bus_value :=
  match e with
  | econv v => v
  | econb b => b t
  | eand ex1 ex2 =>
    bv_bit_and (eval ex1 t) (eval ex2 t)
  | eor ex1 ex2 =>
    bv_bit_or (eval ex1 t) (eval ex2 t)
  | enot ex =>
    bv_bit_not (eval ex t)
  | cond cex ex1 ex2 =>
    match (bv_eq_0 (eval cex t)) with
    | hi => eval ex1 t
    | lo => eval ex2 t end
  | perm ex => eval ex
  | sbbox b => b t
  (* ... *)

```

5) *Coq Representation*: The definition of signals, expressions and their semantic models paves the way for converting RTL circuits into Coq representation. When choosing code constructors, we sought to make the new semantic model user-friendly. The constructor outb is used to denote output signals of the module. Similarly, inb means input signals; wireb represents internal wire signals; and regb denotes the internal registers (note that, similar to HDLs, the reg

type does not necessarily result in actual registers in the synthesized model). Two assignment constructors are also defined, namely `assign_*`, which works for combinational logic, and `nonblock_assign_*`, which is appropriate for non-blocking assignment in sequential logic. An extra notation is added to pile the code through the ‘;’ symbol. The selection of the ‘;’ mark is consistent with the syntax of HDLs.

```
Inductive code :=
| outb : bus -> code
| inb : bus -> code
| wireb : bus -> code
| regb : bus -> code
| assign_ex : bus -> expr -> code
| assign_b : bus -> bus -> code
| assign_case3 : bus -> expr -> code
| nonblock_assign_ex : bus -> expr -> code
| nonblock_assign_b : bus -> bus -> code
| codepile : code -> code -> code.
```

Notation " `c1` ; `c2` " := (codepile `c1` `c2`)
(at level 50, **left** associativity).

6) *Verilog-Coq Conversion Rules*: Although multiple HDLs are available, we chose Verilog as the sample HDL to compose IP cores. The fundamental Verilog-to-Coq conversion rule which we need to obey is to keep the original code and destination code structurally the same. This forms the basis for the Verilog-to-Coq conversion methodology that we developed. For example, a combinational assign logic is mapped to an `assign_ex` statement and module instantiation is mapped to a `module_inst` statement as shown below.

Verilog code:

```
assign Lout = (roundSel == 0) ? IP[33:64] : R;
```

Converted Coq representation:

```
assign_ex Lout (cond (eq (econb roundSel)
  (econv (lo::lo::lo::lo::nil)))
  (econv (IP @ [33, 64])) (econv R));
```

Verilog code:

```
crp u0 (.P(out), .R(Lout), .K_sub(K_sub));
```

Converted Coq representation:

```
module_inst2in out Lout K_sub;
```

7) *Tracking the Information Flow*: In itself, the new formal semantic model still cannot achieve the goal of facilitating theorem proving for information flow policies, since it is simply an alternative HDL to represent the circuit structure. Information-leaking hardware Trojans can still use signal bypassing strategies, which propagate sensitive internal data to primary outputs [13] or disseminate it through Trojan side channels [14], with little modification of the original circuit. However, the circuit description is now in a language that lends itself to formal reasoning. Therefore, by adding appropriate elements (i.e., secrecy tags) and logic for formally reasoning on these elements, we can now support IFT.

To facilitate tracking of internal sensitive data and proving of secrecy properties on this data in the new semantic model, we enhance circuit signals with an additional property, namely *sensitivity*. This property is akin to the existing *bus_value* property and it allows us to formally examine and prove signal integrity (from a security point of view) within the entire design. The chosen Coq platform for our formal semantic model comes in handy, since it is easy to enhance the Coq formal logic to support information flow tracking. The only significant change is that we need to extend the bus definition so that it will return a `bus_value*sensitivity` pair at a

specified time `t` instead of just a `bus_value`. Sensitivity is defined as an inductive set with two constructors, *sensitive* and *non_sensitive*, indicating whether the signals are sensitive and need protection or not. Any output signals of the target circuit should be of *non_sensitive* secrecy tags at any time `t`. Otherwise we claim that data leakage channels exist in the circuit and the data secrecy properties are violated.

Inductive *sensitivity* := sensitive | non-sensitive.
Definition *bus* := nat -> (bus_value * sensitivity).

8) *Security Policies*: A security policy defines the legitimate propagation rules for data secrecy tags as the corresponding signals travel from inputs through bus operations to outputs. Three such rules are defined in the static IFT scheme: (i) *Signal assignment*: When a bus is the operand of a unary operator, the unary operator keeps the secrecy tag of the bus; (ii) *Logic/Functional operations*: For most of the logic/functional operations, if any of the input signals are of sensitive secrecy tag, then the operational outputs need protection and are assigned as *sensitive*; (iii) *Secrecy declassification*: In order to prevent leakage of sensitive internal data by illegitimate “declassification”, it is important to restrict the ability of removing the *sensitive* tag to a few well-controlled operators. As a general rule, in cryptographic cores, eXclusive-OR operations between round keys and intermediate results are considered as sensitivity reducers. Similarly, since internal modules are verified independently, module instantiations are also considered as sensitivity reducers. Examples are provided in the context of the cryptographic cores on which IFT is demonstrated in part II of this paper series.

B. Dynamic Information Assurance

The methodology of Section VI-A provides a static method for reasoning about information flow policies in a hardware design. However, it has some restrictions. Specifically, reasoning about multiple rounds of cryptographic or declassifying operations in a design, such as in a pipelined implementation, is cumbersome and may not be accurate. In this section, we enhance this static methodology to be able to dynamically track the flow of information in a hardware design and eliminate such restrictions. Moreover, as we will describe in Part II of this paper series, PCHIP with dynamic IFT is a more suitable framework for automation. For this purpose, a second formal model, called *structural Coq formal logic*, is developed to represent circuit logic in the Coq platform and to dynamically track information flow within the circuit.

1) *Basic Definitions*: The structural Coq formal logic is defined in such a way that it can accurately map the data secrecy-related structure of the original circuit to its Coq representation, while leaving the circuit functionality unspecified. In contrast to the Coq formal logic of the static scheme, the Coq representations are significantly simplified in the new structural Coq formal logic, since circuit functionality does not need to be explicitly specified. As defined in the structural Coq formal model, signal values in Coq circuits represent their sensitivity levels, not electronic values. Furthermore, circuit signals are not just treated qualitatively as *non_sensitive* or *sensitive*, as in the static scheme, but are also quantitatively allocated integers: A number 0

means `non_sensitive` while a positive number indicates sensitive. A larger number relates to a higher level of sensitivity, implying that the underlying signal requires higher level of protection. For example, non-critical control/data signals, such as input clock signal, loading control, etc., are set to value 0, whereas encryption/decryption key and plaintext are assigned positive integers. As cryptographic algorithms usually consist of multiple rounds, this enables easier handling of pipeline designs. However, in simpler designs, two-state labels may also be employed for the same purpose. As we will introduce shortly, all signal sensitivities are managed in a central way (i.e., through a signal sensitivity list) and the definition of bus is only a number indicating its position in the list. Other definitions, such as bus slicing, expressions, etc., are similar to the formal model introduced in Section VI-A for the static scheme, with the exception that the calculation will only reflect updating of the bus sensitivity levels rather than the electronic values.

(* The definition of bus is only a number indicating the position of the bus in sensitivity tag list *)
Definition `bus` := nat.

2) *Signal Sensitivity Transition Model*: We, then, need a mechanism to depict the way in which signal sensitivity levels dynamically evolve when signals pass through circuit logic. This task is performed by a newly developed signal sensitivity transition model that involves a set of rules which put restrictions on how to upgrade/downgrade (or increase/decrease) signal sensitivity levels.

To ensure the integrity of the transition model and to prevent sensitive information leakage from illegal signal sensitivity downgrading operations, the signal sensitivity transition model is made conservative, i.e., only a small set of Coq circuit operations are allowed to downgrade signal sensitivity levels. The set of sensitivity downgrading operations can be further divided into two groups discussed below, *sensitivity downgrading expressions* and *module instantiation*.

Sensitivity downgrading expressions: These expressions are defined with similar syntax to other expressions but often perform special operations with sensitive data involved. In case of cryptographic circuits, the general rule for downgrading sensitivity of signals is to perform the eXclusive-OR operation between sensitive data and the key (or sub-key). For example, eXclusive-OR of round keys with sensitive data is the only sensitivity downgrading expression for an advanced encryption standard (AES) core. We note that not all eXclusive-OR operations are sensitivity reducers. Also, sensitivity downgrading operations are design-specific and are determined based on a clean high-level architecture (or block diagram), independent of the IP vendor implementation. The IP consumer and the IP vendor should agree on these definitions beforehand, while the IP consumer should verify them in the final design. Extension and generalization of sensitivity downgrading operations through evaluation of a broader set of designs remains as an interesting topic of further investigation.

The recursive function `expr_sen_eval` provides detailed information on how to evaluate the sensitivity level of an expression. Specifically, for most operations the sensitivity level of an entire expression is equal to the highest sensitivity level among all operands and is obtained by calling the

maximum number selection function `boptag`, with a few exceptions of sensitivity downgrading expressions.

Definition `boptag` (a b : nat) : nat := max a b.

Fixpoint `expr_sen_eval` (e : expr) (sl : code_sen)
{**struct** e} : bus_expr_sen :=
match e **with**
| `econv` v => 0
| `econst` b => nth b sl 0
| `eand` ex1 ex2 => boptag (expr_sen_eval ex1 sl)
 (expr_sen_eval ex2 sl)
| `eor` ex1 ex2 => boptag (expr_sen_eval ex1 sl)
 (expr_sen_eval ex2 sl)
| `exor` ex1 ex2 => boptag (expr_sen_eval ex1 sl)
 (expr_sen_eval ex2 sl)
| `eand_bit` b => nth b sl 0
| `eor_bit` b => nth b sl 0
| `exor_bit` b => nth b sl 0
| `eplus` ex1 ex2 => boptag (expr_sen_eval ex1 sl)
 (expr_sen_eval ex2 sl)
| `eminus` ex1 ex2 => boptag (expr_sen_eval ex1 sl)
 (expr_sen_eval ex2 sl)
| `enot` ex => expr_sen_eval ex sl
| `eapp` b1 b2 => boptag
 (nth b1 (sl) 0) (nth b2 (sl) 0)
| `perm` ex => expr_sen_eval ex sl
| `exor_key` b key => lowertag
 (boptag (nth b sl 0) (nth key sl 0))
(* ... *)

As may be observed in the definition of the `boptag` function, it selects the maximum of two sensitivity levels. As an example, consider the evaluation of the logical AND operation, as defined in line 6 of the `expr_sen_eval` function, where each operand is recursively evaluated and the maximum of the two sensitivities is selected as the result of the evaluation. The `lowertag` function reduces the sensitivity level by 1 and is used for the evaluation of sensitivity downgrading operations, e.g. `exor_key` in the above definition, as opposed to the regular `xor` which is evaluated similar to other binary operations. For the evaluation of a constant and a bus, 0 and the current sensitivity level of the bus in the sensitivity list is returned, respectively, as seen in lines 4-5 of this function.

Module instantiation: Almost all modern designs are of hierarchical structure with submodules instantiated to perform various functionalities, from round key generation to memory control. Unless the entire design is flattened, this hierarchy makes it quite difficult to track information flow in and out of submodules in our formal environment. To prevent attacks targeting the interface between higher level modules and their submodules, we propose a sensitivity reshuffling strategy under which output signals from submodules are denoted as input signals of the top module. These signals (outputs from submodules) are called *endogenous* inputs, as opposed to primary inputs which, hereafter, are called *exogenous* inputs. All sensitivity assigning and transition rules that apply to exogenous inputs are also valid for endogenous inputs. This reshuffling strategy eliminates the relation between inputs and outputs of submodules, so that submodules can adjust signal sensitivity levels independently, including sensitivity level downgrading operations. This method of handling submodules and their outputs makes the manual conversion of the design to its Coq representation simpler. A flattened, automated method for handling module instantiations, which does not require defining endogenous and exogenous inputs, will also be presented in part II of this paper series where we discuss framework automation efforts.

3) *Implicit Information Flow*: Implicit information flow is also considered in our sensitivity transition model as attackers may leverage conditional statements to indirectly leak internal information. To solve this problem, a conservative signal sensitivity transition rule is applied, such that the output of the conditional statement maintains the sensitivity of any of the inputs, including the condition clause. In our dynamic scheme, this rule is further specified by the `expr_sen_eval` function in Coq as follows:

```
Fixpoint max_list (l : list nat) : nat :=
  match l with
  | nil => 0
  | a :: rl => max a (max_list rl)
  end.

Fixpoint expr_sen_eval (e : expr) (sl : code_sen)
{struct e} : bus_expr_sen :=
  (* ... *)
  | cond cex ex1 ex2 => max_list
    ((expr_sen_eval cex sl) ::
     (expr_sen_eval ex1 sl) ::
     (expr_sen_eval ex2 sl) :: nil)
  (* ... *)
```

The `max_list` function selects the maximum in a list of numbers. Using this function, the `expr_sen_eval` function selects the maximum sensitivity among the conditional clause and the branches when evaluating conditional statements.

This conservative information flow approach may result in false positives, i.e., the design is secure, but the proof for security theorems cannot be constructed and our framework marks it as insecure. To resolve this limitation, further research and development of advanced and more accurate PCHIP-based information flow models is required.

4) *Signal Sensitivity List*: To facilitate the operation of the signal sensitivity transition model, all signal sensitivity levels in the target circuit are managed in a centralized way. The entire sensitivity status of a circuit at a specified time t is stored in one signal sensitivity list, wherein each element represents the sensitivity level of one input, output, or internal signal. IP consumers can easily check the validity of the data secrecy property by defining the initial status of the sensitivity list, i.e., the starting point from which the sensitivity information spreads across the whole circuit, and then monitoring sensitivity levels of all output signals. Although the data secrecy properties, which serve as the basis for the proposed scheme, are independent of the circuit functionality and architecture, generation of the initial signal sensitivity list is closely related to the circuit structure and its functional specification. Guidelines are developed below for both IP vendors and IP consumers to initialize sensitivities for all signals, including inputs, outputs, and internal signals.

Input signals: The assignment of input signal sensitivity levels, including both exogenous inputs (primary inputs) and endogenous inputs (submodule outputs), can be divided into two steps: (i) deciding whether input signals contain secret information, and (ii) computing the sensitivity level of input signals if they contain secret data.

The first task is mostly completed upon analysis of circuit functionality and is relatively easy according to the circuit specification. For example, a DES encryption core would have plaintext, key, clock signal, round count, and reset signal as exogenous inputs and round keys as endogenous inputs. From the DES specification, IP vendors/consumers can recognize

that exogenous inputs (plaintext, key, round count) and endogenous inputs (round keys) contain sensitive information so that their sensitivity levels should be positive integers requiring protection against information leakage attacks. Other inputs, such as the clock signal, do not contain sensitive information, so their sensitivity levels are set to 0.

After categorizing input signals into sensitive or non-sensitive, we proceed to the second task, which is to decide the actual sensitivity levels for sensitive signals, with larger numbers indicating higher sensitivity. The calculation process is closely related to the circuit architecture designed by IP vendors, particularly dominated by (pipeline) stages of the circuit implementation. While we considered more complex sensitivity level determination algorithms, we opted for a simple sensitivity-level downgrading counting method which is proven very effective in our later demonstrations. According to this method, after producing the HDL code and constructing the circuit architecture, the IP vendor will check all paths from sensitive inputs to primary outputs and will count the number of sensitivity downgrading operations along each route. The sensitivity levels of input signals are then set to the smallest count of sensitivity downgrading operations among all paths. Choosing the smallest count ensures that the output sensitivity becomes zero upon evaluation of a legitimate design in Coq representation.

Upon receiving the HDL code and the description of the circuit architecture, the IP consumer will check the validity of input signal sensitivity levels using the same method. Because the IP vendor may sabotage the circuit by adding extra sensitivity downgrading logic to “bleach” sensitive signals, and make the outputs non-sensitive, all downgrading operations must be clearly annotated with notes explaining why and how these operations are performed. We would like to emphasize that the IP consumer should perform this validation on a high-level architecture or block diagram of the intended implementation. In other words, IP consumers compute the initial sensitivity values using a clean block diagram of the design, and utilize these values in the HDL code delivered by the IP vendor to verify the proofs. Checking the validity of sensitivity reducing operations is also performed similarly, based on a clean block diagram of the design. This enables an independent review of initial sensitivity values and sensitivity reducing operations.

Internal signals and output signals: All signals other than input signals are treated as non-sensitive with level 0. This is because all internal and output signals have preset (or random) values when the circuit is reset or powered off, since in this case they do not contain any sensitive information. Exceptions may occur for storage elements. For example, non-volatile memory can keep stored values at power-off mode and may already contain sensitive information at the moment the circuit is powered on. This problem is solved because memory is always treated as a submodule with all outputs categorized as endogenous inputs under the reshuffling strategy.

As an example application of the above guidelines, an initial signal sensitivity list for a DES core is shown below, where the plaintext input, the key, the round count, and the internally generated sub-key are located at the zeroth, the first, the third,

and the fifth position, respectively.

```
Definition des_sen_initial : code_sen :=
  1::1::0::1::0::1::0::0::0::0::0::0::0::0::0::0::nil.
(*
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14 *)
```

The initial signal sensitivity list, combined with the signal sensitivity transition model, helps both IP vendors and IP consumers track the progress of how sensitive information is propagated and finally absorbed inside the chip.

5) *Theorem Generation Functions*: To facilitate conversion of data secrecy properties from natural language to Coq theorems and to assist with automation of the framework, the dynamic information flow tracking scheme introduces a new concept, namely *theorem generation functions*.

A theorem generation function takes Coq circuits and sensitivity lists as parameters and generates data secrecy theorems in the Coq platform. The use of theorem generation functions is a breakthrough in the field of proof-carrying based hardware IP protection. It simplifies the theorem generation process for the IP vendor and it provides a safety control capability to the IP consumer, who may later check the validity of the initial sensitivity list. Theorem generation functions also separate the tasks of circuit design and theorem generation, a major step toward EDA tool development for theorem and proof auto-generation, and makes it possible to integrate the proof construction task into the standard IC supply chain.

Before delving into the details of theorem generation functions, we need to introduce a special sensitivity list that represents circuit security status in a stable mode, if one exists.

Definition 1. Stable sensitivity list: A stable sensitivity list is a special sensitivity list containing circuit secrecy status at a specified time t , with the key characteristic of stability, meaning that further evaluations of the code (at larger timestamps t) do not change the sensitivity levels of the signals. Denoting `coq_circuit` as the converted Coq circuit and `stable_list` as the stable sensitivity list, if `stable_list` represents the current circuit secrecy status, then the circuit status will not change until the circuit is reset (or is powered off). In the Coq platform, if the signal sensitivity evolution function is `update_sensitivity`, the stability characteristic is presented in the form:

```
update_sensitivity coq_circuit stable_list
= stable_list.
```

Described in natural language, in the scope of the dynamic scheme, the data secrecy property means that “no sensitive data has leaked through primary outputs”. Supported by the signal sensitivity transition model, the “no leakage” property can be further elaborated into three subproperties: (i) there exists a stable sensitivity list for the target circuit implementation, (ii) the stable sensitivity list is achievable from a legitimate initial sensitivity list, and (iii) the circuit secrecy status defined by a stable sensitivity list is trusted. As long as the target circuit and the initial sensitivity lists are both specified, these three subproperties will, then, be translated into Coq theorems using three theorem generation functions.

Theorem Generation Function I. Existence: The existence of a stable sensitivity list is a prerequisite before proving any data secrecy property for the target circuit. If we cannot find

one or more stable sensitivity lists, we believe that the target circuit is untrusted because sensitive data can be leaked freely. In the Coq platform, given the Coq circuit `coq_circuit`, the signal sensitivity list `sen_list` and the signal sensitivity evolution function `update_sensitivity`, we define the theorem generation function for the stable sensitivity list existence property as:

```
Theorem stable_list_existence:
  update_sensitivity coq_circuit sen_list = sen_list.
```

After plugging in the actual Coq circuit and the corresponding sensitivity list, we can generate and prove the existence theorem.

Theorem Generation Function II. Accessibility: The existence property demonstrates the availability of a stable sensitivity list for the target circuit, but it does not provide evidence that the stable sensitivity list is accessible from the circuit’s initial secrecy status. The second theorem seeks to solve this problem by proving that, given an initial sensitivity list, the circuit will finally achieve stable status after a finite number of clock cycles. Having the Coq circuit `coq_circuit`, the initial signal sensitivity list `ini_list`, a finite natural number n , the multiple-step evolution function `check_sensitivity`, and a known stable sensitivity list `stable_list`, we define the theorem generation function for the accessibility property as follows:

```
Theorem stable_list_accessibility:
  forall t : nat, t > n ->
    (check_sensitivity t coq_circuit ini_list)
    = stable_list.
```

The selection of the number n is arbitrary, as long as it is a finite number; typically this number is bounded by the count of circuit stages. It is also possible that $n=0$, such that the initial signal sensitivity list is itself a stable sensitivity list, a case which sometimes exists in small-scale circuits with simple architecture.

Theorem Generation Function III. Trustworthiness: Evaluating the trustworthiness of the derived stable sensitivity list is the most critical step when validating a data secrecy property. Because the stable sensitivity list contains the complete secrecy status of the target circuit, the goal of the trustworthiness theorem is to ensure that no sensitive information is leaked through primary outputs when the target circuit has reached the stable status. The evaluation of the trustworthiness process is shown below:

```
Theorem no_leaking_output:
  nth output stable_list = 0. (* For all outputs *)
```

6) *Data Protection of Intermediate Status*: The above developed three theorems do not fully cover the entire working status of target circuits because the intermediate secrecy status between the initial status and the stable status are left unprotected. Although the intermediate status may only last a few clock cycles, the attackers may still be able to leak internal information during this short transition period. To overcome this limitation, a fourth theorem generation function is developed to formally prove that no sensitive information is leaked before the circuit reaches its stable status.

Theorem Generation Function IV. Trustworthiness of Intermediate Status: The intermediate status during the transition procedure which evolves from the initial status to

the stable status also needs protection to prevent information leakage. In the Coq platform, given the Coq circuit `coq_circuit` and the intermediate signal sensitivity list `intermediate_list`, we define the theorem generation function for the intermediate sensitivity list security property as:

```
Theorem no_leakage_intermediate:
  nth output intermediate_list=0. (*For all outputs*)
```

In the next section, we demonstrate how the general PCHIP framework can be utilized to prevent malicious modification of sensitive data which may leak secrets indirectly.

VII. PREVENTING MALICIOUS MODIFICATION OF SENSITIVE DATA

Even if the data secrecy properties generated through the aforementioned formal theorems are proven to hold, attackers may still be able to indirectly compromise security. Specifically, this can be achieved by inserting hardware Trojans which replace sensitive data, such as the encryption/decryption keys (or sub-keys), with a value known only to the attacker, or which create a new key with much less entropy than the original key, thereby reducing the effort required for cryptanalysis of the ciphertext. Such Trojans may rely on a rare trigger [13] for activation; hence, functional and structural testing may not detect their existence.

The PCHIP-based IFT capabilities introduced in Section VI cannot fully detect such attacks. The underlying reason for this shortcoming is the conservative information flow model employed in our approach for addressing implicit data flows. Another reason is that only the structure of the circuit is precisely considered in the conversion of the HDL code to its corresponding Coq representation, while the functionality of the operations and signal values are abstracted away. This is done to make proof development for information flow policy theorems simpler, as the exact functionality is not crucial in ensuring that sensitive data does not reach primary outputs.

While one possible approach would be to label critical data as non-sensitive and all other signals as sensitive, and then verify that the critical data remains non-sensitive [25], our conservative information flow model prevents us from using it. For example, consider loading the key register based on a load signal in the following code:

```
always @(posedge clk) begin
  if (load == 1'b1)
    key_reg <= key_in;
end
```

In this case, `load` should be sensitive and `key_reg` non-sensitive. During code evaluation, the implicit flow from `load` to `key_reg` results in the the key register to also become sensitive. While this is a legitimate operation, it violates the above rule. This is not a problem in [25], since flip-flops in the gate level model only propagate tags from the input to the output [31] and do not consider the implicit data flow from the load signal.

In order to address this limitation and develop theorems to prevent malicious modification of sensitive data using the PCHIP framework, we resort to a general PCHIP-based solution wherein the entire functionality and structure of the HDL code needs to be converted to the Coq representation.

For this purpose, the newly-introduced IFT capabilities need to be combined with the scope of the original PCHIP framework [15], [28]. Therein, however, the conversion process, which we review for the sake of completeness below, flattens the entire design hierarchy, thereby making proof development very cumbersome. Therefore, we also introduce a new conversion methodology which preserves the design hierarchy and reduces the burden of proof construction. While we chose to convert the entire functionality of the design to the Coq representation, the possible trade-off between the amount of functionality preserved in the conversion process and the level of protection which can be provided is an interesting topic for further investigation.

A. Flattened Functional Model

This model flattens the design hierarchy in a way that the precise functionality of the hardware design is converted to its corresponding Coq representation.

1) *Basic Circuit Elements*: In the flattened functional model, circuit elements are defined in the same way as what we presented in Section VI-A1. In other words, all signals in the design are considered as `bus` in Coq representation.

2) *Module Definitions*: In this model, we flatten the design hierarchy and convert the module definitions in the Verilog source code to an inductive type in the Coq representation. We create a constructor for the module and we consider module inputs and outputs as parameters of this constructor. The body of the module is created in a function named `module_inst`, which is described in the following.

3) *Local Signals*: Local signals are widely defined and used inside modules in HDLs such as Verilog. However, Coq does not provide a flexible way of defining local variables inside functions. Therefore, in this conversion method, local signals are passed as additional arguments to the module type and instance definitions. Consequently, when instantiating a module, we need to define global variables in the Coq representation and pass them to the `module_inst` function.

4) *Module Instantiations*: Due to the flattened hierarchy in the Coq representation, we consider sub-modules as parameters of the `module` type definition, similar to the local signals. Therefore, when instantiating a module in the Coq representation, we need to appropriately instantiate its sub-modules too, and pass them as parameters to the `module_inst` function.

5) *Expressions and Verilog Operations*: We build expressions using the same method as presented in Section VI-A4. However, a difference here is that an expression covers only the basic operations built into the Verilog language, and does not include higher level operations such as permutation or `sbox`, as was the case for the formal model of Section VI-A. Such higher level operations, which are mainly implemented as sub-modules, are handled by module definition and instantiation. We also define the `eval` function to precisely implement the operations in the Coq representation for evaluating the expressions.

6) *Conditional, Combinational and Sequential Statements*: The models in Sections VI-A and VI-B, which are devised for information flow tracking, consider combinational and

sequential statements in the same way. In contrast, when converting the entire functionality into a flattened model, we need to distinguish between them in the Coq representation. Therefore, we define two distinct inductive types for conditions in a combinational and a sequential block. To simplify working with the converted code, unconditional statements are considered a special case of conditionals without any condition. For sequential blocks, `noif`, `ifsimple` and `ifelse` constructors are used for no condition, `if`, and `if-else` statements, respectively. Constructors `anoif`, `aifsimple` and `aifelse` are used similarly to represent combinational blocks. Constructors `ifcons` (with `\` notation) and `aifcons` (with `;` notation) are used to link such statements together and create Coq code blocks in sequential and combinational cases. Since these constructors expect corresponding `if` blocks as their action, nested conditional statements can be seamlessly converted to Coq. These conditional constructors constitute the base structure of the code in its Coq representation. We also unroll `case` statements and treat them as consecutive `if..else` statements.

```
Inductive ifblock :=
| noif : updateblock->ifblock
| ifsimple : expr->ifblock->ifblock
| ifelse : expr->ifblock->ifblock->ifblock
| ifcons : ifblock->ifblock->ifblock.
```

```
Inductive aifblock :=
| anoif : assignblock->aifblock
| aifsimple : expr->aifblock->aifblock
| aifelse : expr->aifblock->aifblock->aifblock
| aifcons : aifblock->aifblock->aifblock.
```

To distinguish between combinational and sequential assignments, we define two inductive types through the `expr_assign` and `upd_expr` constructors, respectively. The difference between these two is that, in a combinational assignment, the computed result affects the left side in the current clock cycle, while in a sequential one, the result is computed for the next clock cycle.

```
Inductive assignblock :=
| expr_assign : bus->expr->assignblock.
```

```
Inductive updateblock :=
| upd_expr : bus->expr->updateblock.
```

To evaluate combinational and sequential conditional blocks we define `adoif` and `doif` functions which are used in the Coq representation. The `assign` and `update` functions evaluate the expression and assign the result to the corresponding `bus` in the current or the next clock cycle, accordingly.

```
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
| ifcons if1 if2 => doif if1 t /\ doif if2 t
end.
```

```
Fixpoint adoif (ai : aifblock) (t : nat) {struct ai} :=
match ai with
| (anoif a_s) => assign a_s t
| (aifsimple exp ifb) =>
  match (eval exp t) with
```

```
| hi::nil => (adoif ifb t)
| lo::nil => True
| x::nil => True
| _ => True end
| (aifelse exp ifb1 ifb2) =>
  match (eval exp t) with
  | hi::nil => (adoif ifb1 t)
  | lo::nil => (adoif ifb2 t)
  | x::nil => (adoif ifb2 t)
  | _ => True end
| aifcons aif1 aif2 => adoif aif1 t /\ adoif aif2 t
end.
```

For further clarification of the conversion process, we list below an excerpt from the Verilog code of a DES core and its partial conversion to the Coq representation.

Verilog code:

```
module des(desOut, desIn, key, decrypt,
  roundSel, clk);
// ...
assign Lout = (roundSel == 0) ? IP[33:64] : R;
assign Xin = (roundSel == 0) ? IP[01:32] : L;
// ...
always @(posedge clk)
  L <= #1 Lout;
always @(posedge clk)
  R <= #1 Rout;
```

```
crp u0(.P(out), .R(Lout), .K_sub(K_sub));
```

```
key_sel u1(.K_sub(K_sub), .K(key),
  .roundSel(roundSel), .decrypt(decrypt));
endmodule
```

Converted Coq representation:

```
Inductive module :=
| module_des : bus->bus->bus->bus->bus->bus->bus->
  bus->bus->bus->bus->bus->bus->module->
  module->module
| module_key_sel : bus->bus->bus->bus->bus->bus->
  bus->bus->bus->bus->bus->bus->bus->bus->module
(* Other modules ... *)
.

Fixpoint module_inst (m:module) (t:nat) :=
match m with
| (module_des desOut desIn key decrypt roundSel clk
  Lout R IP Xin L Rout out FP module_key_sel_u1
  module_crp_u0) =>
  (adoif (
    (anoif (expr_assign Lout
      (cond (eeq (econj roundSel)
        (econj (lo::nil))) (econj (IP [33, 64]))
        (econj R)))));
    (anoif (expr_assign Xin
      (cond (eeq (econj roundSel)
        (econj (lo::nil))) (econj (IP [1, 32]))
        (econj L)))));
    (* ... *)
  ) t)
/\
  (doif (
    (noif (upd_expr L (econj Lout)))
    (noif (upd_expr R (econj Rout)))
  ) t)
/\
  (module_inst module_key_sel_u1 t) /\
  (module_inst module_crp_u0 t)
| (* Other modules ... *)
end.
```

As we mentioned, in this conversion local signals and sub-modules are considered as parameters of the module definition. That is why `module_des` gets so many bus parameters and two module parameters. The appropriate constructors are used to convert combinational and sequential statements to the Coq representation.

In principle, this flattened model, which is employed by the original PCHIP framework, can be utilized to prove general security properties including the ones geared towards preventing malicious data modification in a hardware design. However, the corresponding Coq representation can quickly become very complicated due to the flattened hierarchy, as

seen in the DES code excerpt example. Furthermore, this approach limits reusability of theorems proven for a module, thereby making proof development much more difficult, as the entire design needs to be reasoned upon monolithically. To overcome these limitations, we developed a hierarchy-preserving methodology which we present in the next section.

B. Hierarchical Functional Model

This new hierarchy-preserving model follows the general principles for flattened functional model which was described in Section VII-A. However, it takes advantage of the **Module Type** constructs in Coq to preserve the hardware design hierarchy.

For each module in the Verilog code of the hardware design we define its corresponding module in the Coq representation using the **Module Type** construct. Sub-modules instantiated inside this module are defined using **Declare Module** inside the **Module Type** definition of the Coq representation. We define local signals of the module as **Parameters** inside this **Module Type**. We also define a function named **instantiate** and we create the body of the module using similar constructs as those defined in Section VII-A. The following shows the DES core of Section VII-A, this time converted to the Coq representation through this hierarchy-preserving model.

```
Module Type module_key_sel.
  (* module definition ... *)
End module_key_sel.

(* Other module definitions ... *)

Module Type module_des.

Declare Module crp_u0 : module_crp.
Declare Module key_sel_uk : module_key_sel.

Parameters K_sub IP FP L R Xin Lout Rout out : bus.

Definition instantiate (desOut desIn key decrypt
  roundSel clk : bus) (t:nat) :=
  (adoif (
    (anoif (expr_assign Lout
      (cond (eeg (econb roundSel)
        (econv (lo::nil))) (econb (IP [33, 64]))
        (econb R)))));
    (anoif (expr_assign Xin
      (cond (eeg (econb roundSel)
        (econv (lo::nil))) (econb (IP [1, 32]))
        (econb L)))));
    (* ... *)
  ) t)
  /\
  (doif (
    (noif (upd_expr L (econb Lout)))
    (noif (upd_expr R (econb Rout)))
  ) t)
  /\
  (key_sel_uk.instantiate
    K_sub key roundSel decrypt t) /\
  (crp_u0.instantiate out Lout K_sub t)
End module_des.
```

C. Theorems Preventing Malicious Modification of Data

By maintaining design hierarchy, this new formal model introduced in this work has the advantage that it eliminates the need for keeping track of local signals and module instantiations inside modules. By comparing the Coq code excerpts of the flattened and hierarchical models, it is evident that the hierarchical model is much more readable and easier

to follow. Using this model, we can develop various lemmas inside each **Module Type** in the Coq representation. These lemmas can, then, be applied hierarchically to prove theorems in higher-level modules. This procedure paves the way for developing hybrid module libraries containing both the HDL code and various lemmas usable for proving security theorems in higher-level modules. Such libraries greatly reduce the proof development effort required in the PCHIP framework and make PCHIP far more appealing to the hardware design community.

Moreover, based on this model, we can now also simplify the development of security properties which aim to protect the design against malicious modification of sensitive data. This is achieved by introducing theorems which track legitimate operations on such data, according to the design specification, as we present using a cryptographic core example in part II of this paper series. In conjunction with the IFT methodologies described in Section VI, this capability provides a comprehensive solution for evaluating and assessing data secrecy protection in hardware IPs.

VIII. CONCLUSION

While the majority of contemporary research in hardware Trojan prevention and detection focuses on post-tapeout threats, the problem of untrusted, potentially Trojan-infected RTL code is becoming equally important. Indeed, the increasing reliance on third-party IP cores for hierarchical designs calls for solutions to protect IP cores from inclusion of malicious functionality. The problem is particularly critical in the cryptography domain, where IP cores are used to run encryption/decryption operations and serve as the basis of system-level security. To address this problem, we augmented the previously proposed proof-carrying hardware IP (PCHIP) framework through the introduction of various information flow tracking approaches aiming to protect data secrecy in hardware IPs. To this end, new formal models have been developed for expressing an RTL circuit description in the Coq theorem-proving language through a set of HDL-to-Coq conversion rules, supporting formal reasoning on data secrecy properties and, thereby, increasing hardware trustworthiness. Theorem generation functions and proof libraries have also been introduced to reduce the burden of proof preparation and to pave the way towards construction of a security property library, a key step towards automating proof generation. In addition, we presented a hierarchy-preserving model of the original PCHIP framework, facilitating formal reasoning on security properties to prevent indirect leakage of sensitive information through malicious data modification. In Part II of this paper series, we focus on automation of the extended PCHIP framework and we demonstrate its application and effectiveness in detecting hardware Trojans and/or design flaws in the RTL description of third party IP cores.

REFERENCES

- [1] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1997, pp. 129–142.

- [2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.
- [3] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 463–472.
- [4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 85–96.
- [5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 378–387.
- [6] W. Shi, J. Fryman, G. Gu, H.-H. Lee, Y. Zhang, and J. Yang, "Infoshield: a security architecture for protecting information usage in memory," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 222–231.
- [7] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August, "RIFLE: An architectural framework for user-centric information-flow security," in *Proceedings of Int. Symposium on Microarchitecture (MICRO)*, 2004, pp. 243–254.
- [8] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A software-hardware architecture for self-protecting data," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 14–27.
- [9] Y. Jin and D. Oliveira, "Extended abstract: Trustworthy SoC architecture with on-demand security policies and HW-SW cooperation," in *Workshop on SoCs, Heterogeneous Architectures and Workloads*, 2014.
- [10] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 109–120.
- [11] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware Trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 6:1–6:23, 2016.
- [12] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan attacks: threat analysis and countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.
- [13] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *Proceedings of IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2009, pp. 50–57.
- [14] L. Lin, M. Kasper, T. Guneysoy, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware Trojans through side-channel engineering," in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 382–395.
- [15] 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.
- [16] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of ACM Conference on Computer & Communications Security (CCS)*, 2013, pp. 697–708.
- [17] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "Veritrust: verification for hardware trust," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 61:1–61:8.
- [18] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 153–166.
- [19] H. Salmani, "COTD: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 2, pp. 338–350, Feb 2017.
- [20] S. Saha, R. S. Chakraborty, S. S. Nuthakki, D. Mukhopadhyay *et al.*, "Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability," in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2015, pp. 577–596.
- [21] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 396–410.
- [22] 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 *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 109–120.
- [23] 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 *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 97–112.
- [24] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 503–516.
- [25] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.
- [26] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *Proceedings of IEEE VLSI Test Symposium (VTS)*, 2012, pp. 252–257.
- [27] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *Proceedings of IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 99–106.
- [28] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.
- [29] INRIA. The coq proof assistant. [Online]. Available: <http://coq.inria.fr/>
- [30] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *Computer Aided Verification*. Springer, 2013, pp. 213–228.
- [31] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, p. 2, 2014.



Yier Jin is currently an assistant professor in the ECE Department at the University of Central Florida. He received his PhD degree in Electrical Engineering in 2012 from Yale University after he got his B.S. and M.S. degrees in Electrical Engineering from Zhejiang University, China, in 2005 and 2007, respectively.

His research focuses on the areas of trusted embedded systems, trusted hardware intellectual property (IP) cores and hardware-software co-protection on computer systems. He proposed various approaches in the area of hardware security, including the hardware Trojan detection methodology relying on local side-channel information, the post-deployment hardware trust assessment framework, and the proof-carrying hardware IP protection scheme. He is also interested in the security analysis on Internet of Things (IoT) and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. He is awarded the DoE Early CAREER Award in 2016 and is the best paper award recipient of DAC'15, ASP-DAC'16, and HOST'17.



Xiaolong Guo received his double Bachelor's degrees from Beijing University of Posts and Telecoms (BUPT), and University of London (UL) in 2010, respectively. In 2013, he received his M.S. degree from BUPT. He is currently working toward the Ph.D. degree in electrical engineering at University of Central Florida, Orlando, FL. His current research interest includes design of scalable verification methods for hardware IP protection, Trusted SoC Verification, Cyber Security, Formal Methods, Program Synthesis, and Secure Language Design.



Raj Gautam Dutta is a Ph.D student with the EECS department at University of Central Florida. He obtained his B.Tech in Electronics and Communication from Visvesvaraya Technological University, India in 2007 and MS in Electrical Engineering with an emphasis on control systems from the University of Central Florida, USA in 2011. His current research interest includes development of security solutions for semiconductor soft IP cores by using formal verification techniques, design of attack detection and mitigation software for autonomous systems, and synthesis of robust controllers for cyber-physical systems.



Mohammad-Mahdi Bidmeshki (S'11) received the B.Sc. and M.Sc. degrees in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 2004 and 2006 respectively. He is currently a Ph.D. candidate in Computer Engineering at the University of Texas at Dallas. His current research includes hardware based security, trusted hardware design, formal methods in security and verification, and the applications of machine learning in computer security.



Yiorgos Makris (SM'08) received the Diploma of Computer Engineering from the University of Patras, Greece, in 1995 and the M.S. and Ph.D. degrees in Computer Engineering from the University of California, San Diego, in 1998 and 2001, respectively. After spending a decade on the faculty of Yale University, he joined UT Dallas where he is now a Professor of Electrical Engineering, leading the Trusted and RELiable Architectures (TRELA) Research Laboratory. His research focuses on applications of machine learning and statistical analysis in the development of trusted and reliable integrated circuits and systems, with particular emphasis in the analog/RF domain. Prof. Makris serves as an Associate Editor of the IEEE Transactions on Information Forensics and Security and the IEEE Design & Test of Computers Periodical and he has also served as a guest editor for the IEEE Transactions on Computers and the IEEE Transactions on Computer-Aided Design of VLSI Circuits and Systems. He is a recipient of the 2006 Sheffield Distinguished Teaching Award, as well as Best Paper Awards from the 2013 Design Automation and Test in Europe (DATE'13) conference and the 2015 VLSI Test Symposium (VTS'15).

Data Secrecy Protection through Information Flow Tracking in Proof-Carrying Hardware IP

Part II: Framework Automation

Mohammad-Mahdi Bidmeshki, *Student Member, IEEE*, Xiaolong Guo, *Student Member, IEEE*, Raj Gautam Dutta, *Student Member, IEEE*, Yier Jin, *Member, IEEE*, Yiorgos Makris, *Senior Member, IEEE*

Abstract—Part II of this paper series focuses on automation of the extended proof-carrying hardware intellectual property (PCHIP) framework for data secrecy protection in third-party IPs, which was presented in part I. Specifically, we introduce (1) *VeriCoq-IFT*, an automated PCHIP framework for information flow policies, and (2) *VeriCoq-H*, a hierarchy-preserving Verilog-to-Coq converter. *VeriCoq-IFT* aims to (i) automate the process of converting designs from HDL to the Coq formal language, (ii) generate security property theorems ensuring compliance with information flow policies, (iii) construct proofs for such theorems, and (iv) check their validity in a design, with minimal user intervention. *VeriCoq-H*, on the other hand, seeks to convert the entire functionality of a Verilog design to its Coq representation while preserving design hierarchy. It facilitates development of hierarchical proofs and enables construction of hybrid module libraries containing the HDL code and the corresponding reusable lemmas for each module. Applicability of our automated *VeriCoq-IFT* framework is demonstrated by evaluating trustworthiness of two DES encryption circuits and several genuine and Trojan-infested AES designs, along with the utility of *VeriCoq-H* in preventing malicious modification of sensitive data, such as the secret key of an encryption circuit.

Index Terms—Hardware Trust, Proof-Carrying Code, Proof-Carrying Hardware IP, Information Flow Tracking

I. INTRODUCTION

In part I of this paper series, we reviewed the fundamentals of the proof-carrying hardware IP (PCHIP) methodology and we developed frameworks for information flow tracking (IFT) to ensure that a hardware design is secure in terms of information flow policies. We also presented a new hierarchy-preserving methodology for the PCHIP framework, on which our solution for protecting the hardware design against malicious modification of data is based. Despite their tremendous potential, broad deployment of PCHIP-based solutions faces

certain challenges. Converting HDL code to a formal representation, such as the Coq [1] language used in PCHIP, and developing proofs for security properties, requires additional knowledge of formal methods, theorem proving environments, and proof writing. Even for someone who has this expertise, the process is tedious and time-consuming, making IP developers hesitant to adopt PCHIP. In this paper, we present our efforts toward automation of the PCHIP framework and we demonstrate the applicability of our methods in detecting design flaws and hardware Trojans in cryptographic cores.

The dynamic information flow tracking method presented in part I of this paper series offers the opportunity to develop a fully automated PCHIP framework, specifically geared towards enforcing information flow policies. An initial attempt in this direction was described in our earlier work [2]. *VeriCoq-IFT*, shown in Fig. 1, is an automated PCHIP framework for enforcing information flow policies, covering various tasks such as conversion of the design from HDL to the Coq formal language, generating security property theorems that ensure information flow policies, constructing proofs for such theorems, and checking their validity for the design, with minimal user intervention. Herein, we extend the capabilities of *VeriCoq-IFT* so that it is able to protect not only outputs but also intermediate signals in the design. Additionally, we provide a procedure for automatically establishing the initial sensitivity levels of all signals.

Applicability of *VeriCoq-IFT* is demonstrated on two different implementations of the data encryption standard (DES) [3], as well as on the genuine and several Trojan-infested versions of the advanced encryption standard (AES) [4]. Based on minimal input provided by the user, trustworthiness of these circuits with respect to information flow policies is automatically checked. *VeriCoq-IFT* gathers the required information through special comments (pragmas) inserted in the HDL code by the IP designer. IP consumers, on the other hand, only need to check the validity of these special comments and utilize *VeriCoq-IFT* to assess the trustworthiness of the acquired design. Any alteration of circuit description against information flow policies causes proofs to fail. This methodology is a first essential step towards establishing PCHIP as a valuable method for assessing credibility of third party IPs with minimal extra effort. We note that, while both a static and a dynamic IFT method were introduced in part I, *VeriCoq-IFT* is established atop the dynamic one, which supersedes its static counterpart.

Manuscript received June 26, 2016; revised January 20, 2017; accepted May 10, 2017. This work has been partially supported by the National Science Foundation (NSF-1318860, NSF-1319105) and the Army Research Office (ARO W911NF-12-1-0091, ARO W911NF-16-1-0124).

Mohammad-Mahdi Bidmeshki and Yiorgos Makris are with the department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX 75080, USA. E-mail: {bidmeshki, yiorgos.makris}@utdallas.edu

Xiaolong Guo, Raj Gautam Dutta, and Yier Jin are with the department of Electrical and Computer Engineering, The University of Central Florida, Orlando, FL 32816, USA. E-mail: {guoxiaolong, rajgautamdutta}@knights.ucf.edu, yier.jin@eecs.ucf.edu

This paper has supplementary downloadable material available at <http://ieeexplore.ieee.org>, provided by the authors. The material includes all RTL source code, the Coq representation, the data secrecy theorems and their proofs. Contact the authors for further questions about this work.

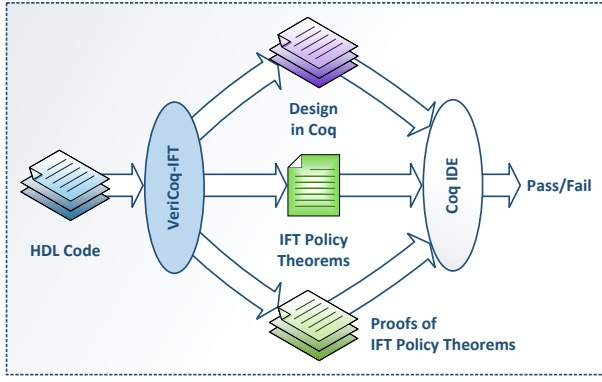


Fig. 1. Automated PCHIP framework for information flow policies

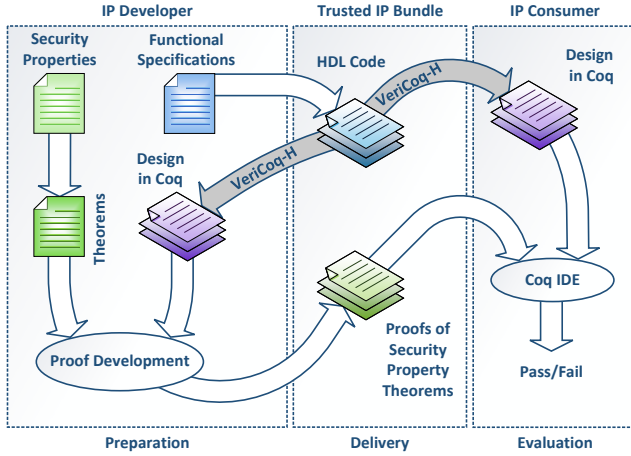


Fig. 2. General PCHIP framework and VeriCoq-H

An alternative way through which hardware Trojans may introduce harmful actions, and which may not always be detectable through the enforcement of information flow policies using *VeriCoq-IFT*, is the malicious modification of data, such as the key in a cryptographic core. To prevent such attacks, in part I of this paper series we proposed a solution based on the general PCHIP framework, which involves manual development of security theorems and their proofs. To assist in the process, in our earlier work we introduced *VeriCoq* [5], which automates conversion of the design from Verilog to its Coq representation. However, *VeriCoq* performs the flattened functional conversion presented in part I, removing design hierarchy and, therefore, making manual proof development very challenging. Therefore, herein we introduce *VeriCoq-H*, an enhanced version of *VeriCoq*, which automatically converts the Verilog design to its Coq representation following the hierarchy-preserving methodology presented in part I. *VeriCoq-H*, whose role in the general PCHIP framework is shown in Fig. 2, enables hierarchical proof development. Additionally, hybrid libraries of hardware modules including proofs of various reusable lemmas can be developed to expedite the proof development for higher-level designs. Consequently, the hierarchy-preserving methodology reduces the proof-writing burden in the PCHIP framework.

Application of *VeriCoq-H* towards facilitating proof de-

velopment for security properties which prevent malicious modification of data in cryptographic cores is detailed through an example. We note that this approach is different from the methodology for detecting malicious modification of data in third-party hardware IPs which was introduced in [6]. The latter employs a formal state-space exploration approach for this purpose, which is limited in verifiable clock cycles and cannot guarantee the trustworthiness of the design beyond a given time-stamp. In contrast, the PCHIP-based solution does not have such a restriction.

The rest of this paper is organized as follows. In Section II, we illustrate the details of the *VeriCoq-IFT* framework including conversion to Coq, generation of security property theorems and construction of proofs. In Section III, we depict the hierarchical conversion of the entire functionality of a Verilog design to Coq representation within the general PCHIP framework, as performed by *VeriCoq-H*. Section IV provides a case study of the static IFT method presented in part I of this paper series and demonstrates the capabilities of *VeriCoq-IFT* on two implementations of the DES encryption algorithm, and several AES designs including a genuine and various Trojan-infested versions. Additionally, it details the application of *VeriCoq-H* in preventing malicious modification of the secret key in a DES core. A discussion is provided in Section V and conclusion are drawn in Section VI.

II. VeriCoq-IFT IN DETAIL

VeriCoq-IFT is a comprehensive solution for enforcing information flow policies on hardware designs, as depicted in Fig. 1. It *automates* the three intricate tasks of (i) converting the HDL code to Coq representation, (ii) generating theorems enforcing information flow policies, and (iii) providing proofs of such theorems. Then, these three essential pieces are delivered to Coq-IDE, which evaluates the validity of the proofs for the design. While information flow policies appear to be simple in principle (e.g., no sensitive data should leak through untrusted channels), enforcing them in a design is complicated. In the following sections, we explain how *VeriCoq-IFT* automates these three main tasks in the PCHIP framework to ensure trustworthiness with respect to information flow policies.

A. Conversion To Coq Representation

VeriCoq-IFT builds upon the HDL-to-Coq conversion rules defined in the original PCHIP framework [7]–[10] and in part I of this paper series and enhances them with additional rules which consider not only precise tracking of signals in the design but also automation requirements. In this section, we illustrate the conversion of Verilog constructs to their Coq representation, as performed by *VeriCoq-IFT*. For this purpose, we utilize the partial Verilog source code of a DES encryption core [11], shown in Fig. 3, and its Coq representation generated by *VeriCoq-IFT*, as it appears in Fig. 4.

Sensitivity (Secrecy) Tags: To enforce information flow policies, functionality and result of operations are not important. However, a mechanism is required whereby each signal is

1556-6013 (c) 2016 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.

to index 0, as seen in lines 18 and 20 of the converted code in Fig. 4.

Module Definitions: *VeriCoq-IFT* converts module definitions in the Verilog source code to an inductive type in the Coq representation, similar to what we described for the flattened functional model in part I of this paper series. It creates a constructor for the module and considers module inputs and outputs as parameters of this constructor. The body of the module is created in a function named `module_inst`. For example, lines 3-11 in Fig. 4 show the created module type definition for the modules defined in the Verilog source code of Fig. 3. Then, as shown in lines 13-35 of Fig. 4, *VeriCoq-IFT* also creates the `module_inst` function which constitutes the body of the modules. More details on the structure of this function are provided below.

Local Signals: *VeriCoq-IFT* traces all the signals in a module and, whenever a local signal is needed, it adds it to the parameter list of the module in the Coq representation, even though such signals are not present in the port list of the module in Verilog. For example, consider the local signals defined in lines 12-17 of the `des` module in Fig. 3. As lines 15-16 of Fig. 4 show, these signals are considered as parameters for this module in its Coq representation. However, if a local signal is used only to connect module instantiations and is not assigned or read directly inside a module (e.g. `K_sub` in `des` module of Fig. 3), there is no need to treat it as a module parameter. *VeriCoq-IFT* can correctly identify such local signals and accurately create their equivalent Coq description.

Parameters: *VeriCoq-IFT* supports Verilog numeric parameters which are often defined within modules. Since such parameters can be modified by each module instance, *VeriCoq-IFT* considers them as additional parameters when defining the module in its Coq representation. It also tracks the parameter definitions in the Verilog source code and passes the correct values for these parameters when creating module instances.

Module Instantiations: To support hierarchy, *VeriCoq-IFT* tracks module instantiations inside a module and defines them as parameters of the module definition in the Coq representation. For example, the `des` module in Fig. 3 instantiates two modules named `crp` and `key_sel`. As lines 5 and 16 in Fig. 4 show, these modules are added to the definition of the `des` module in its Coq representation.

VeriCoq-IFT automatically creates a definition for the top module of the Verilog source code, representing the top module instantiation. For this purpose, *VeriCoq-IFT* creates the appropriate variables, parameters and module instantiations. As lines 37-52 in Fig. 4 show, to instantiate `des`, *VeriCoq-IFT* defined the required buses with their corresponding position in the sensitivity list. It also created two module instances required for the `des` module, namely `module_key_sel` and `module_crp`, in order to instantiate `module_des`. Each of these two module instances require other module instantiations which *VeriCoq-IFT* creates recursively and are not shown in Fig. 4 due to space limitations.

Verilog Operations and Expressions: *VeriCoq-IFT* defines an inductive type `expr` in order to build expressions based on basic mathematical and logical operations of Verilog as

follows.

```
Inductive expr :=
| econb : bus -> expr (*bus to expr*)
| euop : expr -> expr (*unary operator*)
| euop_bit : expr -> expr (*unary op.-1 bit result*)
| ebinop : expr -> expr -> expr (*binary operator*)
| ebinop_bit : expr -> expr -> expr (*bin op.-1-b res*)
| ereduce : expr -> expr (*sensitivity reduction*)
| ecats : expr -> expr -> expr (*concatenation*)
| cond : expr -> expr -> expr -> expr (*query(? : )*)
```

Note that this definition of `expr` is meant to work on the sensitivity tags in Coq representation, not the binary values of the signals, which we omitted in the conversion to Coq for information flow tracking purposes, and covers almost all basic Verilog operations. For example, addition, subtraction, or logical AND operation are all considered as binary operations built by the `ebinop` constructor. Such definition for `expr` reduces the effort required for proof development yet accurately tracks the flow of information in the hardware design. *VeriCoq-IFT* identifies the type of operators and creates their corresponding Coq representation accordingly, using the appropriate constructors. For example, we point out the eXclusive-OR operation in line 23 of Fig. 3, which is converted as line 21 in Fig. 4 using `ebinop`.

In the PCHIP framework for information flow tracking presented in part I, certain operations of each design are considered as sensitivity level reducers (or declassifying operations). In order to identify such sensitivity reducing operations, *VeriCoq-IFT* defines a special comment (pragma) as `/* vericoq sensitivity_reducer */`, and whenever it finds this comment in the Verilog source code, it considers the next operation a sensitivity reducer. For cryptographic hardware, we consider the eXclusive-OR operation on secret input and sub-keys as a sensitivity reducer. Consequently, notice the eXclusive-OR operation of line 23 in Fig. 3 which is preceded by this special comment. This operation is contained in the `ereduce` constructor, as shown in line 21 of its Coq representation in Fig. 4. We note that not all eXclusive-OR operations are considered sensitivity reducers. Sensitivity reducing operations should be individually marked and these decisions should be verified by the IP consumers using a clean high-level architecture or block diagram of the design.

VeriCoq-IFT also defines the `eval` function to evaluate expressions and produce the corresponding sensitivity levels. Evaluation of the query operator requires special consideration, which we illustrate next, along with the conditional statements.

Assignments and Conditional Statements: To build the code block in Coq representation, *VeriCoq-IFT* defines inductive type code as follows.

```
Inductive code :=
| assign : bus -> expr -> code
| ifsimpl : expr -> code -> code
| ifelse : expr -> code -> code -> code
| code_cons : code -> code -> code.
```

`assign`, `ifsimpl`, and `ifelse` are constructors used for Verilog assignments, if, and if-else statements, respectively. The current definition of `assign` restricts the left hand side to `bus`. Nevertheless, this definition does not impose actual restrictions on the Verilog code, since assignments which do not follow this rule (e.g. concatenation on the left hand side) can be broken into several assignments by the

designer. `code_cons` connects code together and *VeriCoq-IFT* uses `;` notation for its representation. For further details, notice the body of the `des` module in lines 19-36 of Fig. 3, which are converted to lines 17-27 of Fig. 4 as their Coq representation. One point to elaborate on in this conversion is that *VeriCoq-IFT* treats sequential and combinational blocks in the same way. Actually, when evaluating the code to update the sensitivity levels of the signals, all statements are treated sequentially. This does not create any problems for information flow tracking purposes. It may only delay the evolution of the sensitivity levels by a few clock cycles.

The importance of maintaining a time stamp together with the sensitivity levels of signals reveals itself in the evaluation of conditional statements. Since we omit the functionality of the operations in the conversion to Coq representation, there is no way to find out which branch of the condition is taken upon evaluation. Therefore, *VeriCoq-IFT* evaluates all branches in Coq representation; when updating the sensitivity levels of buses in the assign statements, it uses the maximum value of the sensitivity level produced for a signal on each (virtual) clock cycle. To prevent implicit information leakage through conditions, when evaluating statements inside conditional statements *VeriCoq-IFT* considers the sensitivity level produced by the condition expression together with each branch.

Initial Sensitivity List: To create the initial sensitivity list for the hardware design, *VeriCoq-IFT* requires knowing which signals carry sensitive information and their sensitivity levels. Thus, to make gathering of such information easier, *VeriCoq-IFT* defines a special Verilog comment (pragma) in the form of `/* vericoq init_sensitivity_level_2 */`. This special comment should appear before signal definition in the Verilog source code. Its numeric value, which appears last, is considered as the initial sensitivity level of the signal. Designers determine this value by considering the number of sensitivity reducing operations that the signal experiences through the design. For signals that *VeriCoq-IFT* finds no such information, it assumes the initial sensitivity to be **None**. Note that the initial sensitivity list is comprehensive, containing information for all signals in the design, including inputs, outputs and local signals. When instantiating the top module, *VeriCoq-IFT* also creates the initial sensitivity list using the information gathered through these comments. To elaborate further, note that the definition of `desIn` and `key` in lines 5 and 7 of the `des` module in Fig. 3 is preceded by such special comments. As seen in lines 54-61 of Fig. 4, *VeriCoq-IFT* assumes an initial sensitivity of 1 for `desIn` and 2 for `key` in the `init_state` created in Coq representation.

Finding the exact sensitivity level of sensitive signals requires an understanding of the overall flow of information in the design, and as mentioned before, determining how many sensitivity reducing operations act on those signals before reaching the outputs. To facilitate this task, we can take advantage of *VeriCoq-IFT* auxiliary capabilities to help the users. For this purpose, designers need to mark the sensitivity reducing operations in the design as explained before. They can use the following procedure to get recommendations on the initial sensitivity levels.

For each sensitive signal bit b :

- 1) Set the initial sensitivity level of b to a high value, e.g., 100, in the initial sensitivity list.
- 2) Set the sensitivity level of all other signal bits in the initial sensitivity list to zero.
- 3) Evaluate the code using the updated initial sensitivity list until a stable list is reached.
- 4) Check all output bits in the resulting stable sensitivity list and consider the minimum reduction in the sensitivity of the signal seen in the output as the initial sensitivity level of the signal bit b .

We note that this procedure should be performed on a clean architecture of the design which only comprises high level blocks and operations, without detailed implementation information. This ensures that any design flaws or malicious design modifications will not interfere with the task of finding the initial sensitivity levels.

As we explained in this section, the conversion procedure from Verilog code to its Coq representation is completely automated in the *VeriCoq-IFT* framework, without any user intervention. IP developers are only required to provide necessary information by inserting special comments defined in the *VeriCoq-IFT* framework into the HDL code. Similarly, IP consumers only need to check the validity of those comments for the corresponding signals and operations.

B. Security Property Theorems

VeriCoq-IFT generates functions and theorems which are required to ensure the trustworthiness of the design in terms of information flow policies. For this purpose, it has a function which is used to evaluate the statements in the code and to update the sensitivity level based on a time-stamp parameter, named `check_code_sen`. *VeriCoq-IFT* generates theorems for all the outputs of the top module to ensure that their sensitivity level remains safe at all times. To elaborate further, notice the theorem starting at line 7 of Fig. 5, which is generated as part of the conversion from Verilog to Coq representation in the *VeriCoq-IFT* framework. Since the `des` module in Fig. 3 has only one output, namely `desOut`, *VeriCoq-IFT* generates a theorem stating that the sensitivity level of `desOut` remains safe at all times. This theorem covers all 4 requirements presented as theorem generation functions in part I. Assuming that the initial sensitivity level information and sensitivity reducing operations are provided accurately, proving this theorem ensures that no sensitive information is leaked through this primary output of the design. We describe the details of these theorems in the next section, which illustrates the proof.

Some Trojans may not leak sensitive information to an output of the hardware IP design. Instead, they establish a side-channel inside the design, e.g., by manipulating the power consumption or the leakage current, to facilitate leaking sensitive information indirectly [12]. While *VeriCoq-IFT* cannot automatically detect such side-channels, it provides a special comment as `/* vericoq force_nonsensitive */` for the designers or, more importantly, the IP consumers, to mark specific signals which they might be suspicious of leaking

```

1 Definition check_sensitivity t := check_code_sen des init_state t.
2 Definition stable := find_stable_list des init_state 20.
3
4 Definition is_safe_bef_stable_desOut :=
5   is_safe_bef_stable desOut des init_state (fst stable).
6
7 Theorem desOut_secret : forall (t : nat),
8   ((fst stable) < t) =>
9   is_safe_op_bus_sensitivity (read desOut (check_sensitivity t))
10  /\ is_safe_bef_stable_desOut.
11 Proof.
12   intros. split.
13   assert (get_sen_val_sen_list (check_sensitivity t) =
14     get_sen_val_sen_list (check_sensitivity (fst stable))).
15   apply check_code_sen_eq_st.
16   vm_compute. reflexivity.
17   apply H. simpl. omega.
18   assert (get_sen_val_op_bus (read desOut (check_sensitivity t)) =
19     get_sen_val_op_bus (read desOut
20       (check_sensitivity (fst stable)))).
21   apply read_sen_eq_st. apply H0.
22   assert (is_safe_op_bus_sensitivity (read desOut
23     (check_sensitivity t)) =
24     is_safe_op_bus_sensitivity (read desOut
25       (check_sensitivity (fst stable)))).
26   apply op_bus_same_sen_val_is_safe. apply H1. rewrite H2.
27   vm_compute. tauto. vm_compute. tauto.
28 Qed.

```

Fig. 5. *VeriCoq-IFT* generated theorem and proof for the DES core

sensitive information. *VeriCoq-IFT* also generates security theorems for these specific internal signals, in order to ensure that no sensitive information reaches them.

C. Proofs of Security Theorems

Proofs of security theorems generated by *VeriCoq-IFT* are constituted in two parts. Since the sensitivity level of the primary inputs does not change and we do not have any sensitivity enhancing operator, the sensitivity list should reach a stable condition in which further code evaluations do not change the sensitivity values in the list. To find the clock cycle at which the sensitivity list becomes stable, *VeriCoq-IFT* defines a function named `find_stable_list`. Using this function, *VeriCoq-IFT* can automatically find the stable clock cycle, and its usage is found in line 2 of Fig. 5. This function evaluates the code cycle-by-cycle and returns the clock cycle at which the resulting sensitivity list evaluated in two consecutive clock cycles is equal. Before the sensitivity list stabilizes, *VeriCoq-IFT* evaluates the code cycle-by-cycle and checks whether the sensitivity of the target signals is safe. This is performed using a function `is_safe_bef_stable` called specifically for `desOut` in lines 4-5 of Fig. 5. To prove this part, *VeriCoq-IFT* evaluates this function by computation.

After the sensitivity list becomes stable, *VeriCoq-IFT* uses proof-by-induction. *VeriCoq-IFT* proves a lemma named `check_code_sen_eq_st`, which states that after stabilization of the sensitivity list, further evaluation of the code does not change the sensitivity values, and its application is shown in lines 13-17 of the proof in Fig. 5. Using this lemma, the theorem for not leaking sensitive information is proven. Since the lemma is proven generally for all codes, proofs of all generated security property theorems can be constructed similarly and automatically, which is a big advantage of the *VeriCoq-IFT* framework.

III. *VeriCoq-H*: HIERARCHICAL VERILOG TO COQ CONVERTER

Although information flow policies, enforced automatically by *VeriCoq-IFT*, are able to capture sensitive information

leakage through the outputs or suspicious signals, they cannot thoroughly prevent or capture malicious modifications of sensitive data in the design. As an example of such malicious modifications, consider a hardware Trojan which, upon activation, sets the key to a constant or a value created from the plaintext or the original key through a transformation known to the attacker. Such action might not be detected by functional tests due to its rare activation, yet can disrupt the core functionality and leak sensitive information because the attacker obtains the key.

In order to protect the hardware design against such attacks, in part I of this paper series we presented a solution based on the general PCHIP framework. As opposed to the PCHIP framework for information flow policies implemented by *VeriCoq-IFT*, which does not consider the functionality of operations and only tracks the flow of information, the general PCHIP methodology converts the exact functionality of the circuit to the Coq representation. Using this general framework, security properties which prevent malicious modifications of data can be developed and proven. We note that, besides conversion of the design to its corresponding Coq representation, generating these security properties and proving them is design-dependent, still requires the developer's effort, and is not yet automated.

Earlier, we introduced *VeriCoq* [5] for automated conversion of the exact functionality of a hardware design in Verilog to its corresponding Coq representation based on the hierarchy flattening methodology presented in Part I. A key limitation in *VeriCoq* is, that similar to *VeriCoq-IFT*, it flattens the design hierarchy. For information flow policies, flattening the hierarchy helps to automate the entire framework including the proofs. In case of the general PCHIP, however, it makes the manual development of proofs for security property theorems complicated. In this section, we introduce an enhanced version of *VeriCoq*, namely *VeriCoq-H*, which converts the design to its Coq representation according to the hierarchy-preserving methodology of part I. As we mentioned earlier, this conversion approach enables the development of hierarchical proofs and proof reuse. Lemmas proven in sub-modules can be applied to prove theorems in higher level modules. A similar hierarchical security theorem proving process has also been demonstrated on a large-scale SoC design [13]. Moreover, a library of modules can be created, including the lemmas and proofs for each module, and utilized for the development of new designs as well as the proof of their security properties. The evolution of such a module library will extensively reduce the burden of proof development for IP designers who choose to utilize the PCHIP framework in production.

Fig. 6 shows the partial hierarchical Coq representation of the DES core of Fig. 3 generated by *VeriCoq-H*, following the hierarchical formal model presented in Part I. To assist with understanding it, a few details or additional capabilities of *VeriCoq-H* are described below.

Parameters: *VeriCoq-H* handles Verilog numeric parameters similar to *VeriCoq-IFT* and considers them as additional arguments when defining the module through the `instantiate` function in its Coq representation. It also tracks the parameter definitions in the Verilog source code and passes the correct

```

1 Require Import Vericoq.
2
3 Module Type module_sbox1.
4
5 Definition instantiate (addr dout : bus) (t:nat) :=
6   (* ... *)
7
8 End module_sbox1.
9
10 (* Other sub-modules *)
11
12 Module Type module_crp.
13
14 Parameters E X Ss : wire.
15
16 Declare Module sbox1_u0 : module_sbox1.
17 Declare Module sbox2_u1 : module_sbox2.
18 (* ... *)
19
20 Definition instantiate (P R K_sub : bus) (t:nat) :=
21   (adoif (
22     (* ... *)
23     t)
24     /\
25     (* ... *)
26     (sbox2_u1.instantiate (X [(48 - 7), (48 - 12)])
27       (Ss [(32 - 5), (32 - 8)]) t) /\
28     (sbox1_u0.instantiate (X [(48 - 1), (48 - 6)])
29       (Ss [(32 - 1), (32 - 4)]) t)
30   ).
31 End module_crp.
32 (* ... *)
33
34 Module Type module_des.
35
36 Parameters K_sub IP FP L R Xin Lout Rout out : wire.
37
38 Declare Module crp_u0 : module_crp.
39 Declare Module key_sel_uk : module_key_sel.
40
41 Definition instantiate (desOut desIn key decrypt roundSel clk
42   : bus) (t:nat) :=
43   (adoif (
44     (anoif (expr_assign Lout (cond (eeq (econb roundSel)
45       (econv (lo:nil)))
46       (econb IP [(64 - 33), (64 - 64)]))
47       (econb R)))));
48     (anoif (expr_assign Xin (cond (eeq (econb roundSel)
49       (econv (lo:nil)))
50       (econb IP [(64 - 1), (64 - 32)]))
51       (econb L)))));
52     (* ... *)
53   ) t)
54   /\
55   (doif (
56     (noif (upd_expr L (econb Lout)))
57     (noif (upd_expr R (econb Rout)))
58   ) t)
59   /\
60   (crp_u0.instantiate out Lout K_sub t) /\
61   (key_sel_uk.instantiate K_sub key roundSel decrypt t)
62 ).
63 End module_des.
64
65 Module Type module_des_top.
66
67 Parameters desOut desIn key decrypt roundSel clk : bus.
68
69 Declare Module des_top : module_des.
70
71 Axiom des: forall (t:nat),
72   des_top.instantiate desOut desIn key decrypt roundSel clk t.
73
74 End module_des_top.

```

Fig. 6. Partial *VeriCoq-H* generated Coq code for the DES core

values for these parameters when creating module instances.

Top Module Instantiation: *VeriCoq-H* automatically creates an axiom for the top module of the Verilog source code, representing the top module instantiation. For this purpose, *VeriCoq-H* creates the appropriate variables, parameters and module instantiation. As lines 65-74 in Fig. 6 show, a **Module Type** is defined for the top module and an axiom is created to instantiate *des*, corresponding to *des* module as the top module in the Verilog source code of Fig. 3.

Part Selection: *VeriCoq-H* handles part selection similar to *VeriCoq-IFT* and uses the same $[,]$ notation. Likewise, it normalizes indices in part selection of buses such that the least significant bit (LSB) of a bus is always referred to by index 0. As an example, note the part selection of signal **IP** in lines 46 and 50 of the Coq representation in Fig. 6, corresponding to lines 19-20 of the Verilog source code of Fig. 3.

The code generated by *VeriCoq-H*, as partially shown in Fig. 6, is directly usable in the Coq environment to develop proofs for the desired security properties, such as theorems which prevent malicious modification of data in cryptographic

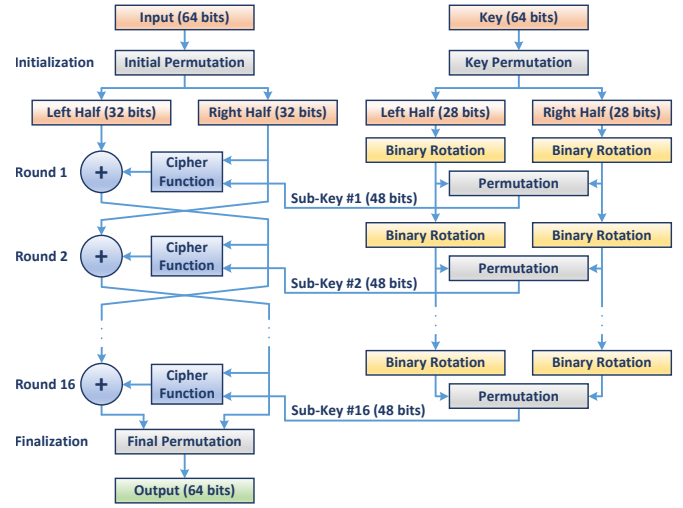


Fig. 7. DES block diagram [3], [15]

cores, as we show in Section IV-C. The first line of Fig. 6 imports a Coq library containing the general PCHIP definitions.

IV. DEMONSTRATION ON CRYPTOGRAPHIC HARDWARE

In this section, we demonstrate the capabilities of the extended PCHIP frameworks by evaluating trustworthiness of two DES cores provided in [11] and several genuine and Trojan-infested AES cores offered in [14]. We first demonstrate the static IFT approach on an area efficient DES implementation. We then show how the dynamic IFT methodology implemented by *VeriCoq-IFT* can be used to evaluate trustworthiness of various genuine and Trojan-infested cryptographic cores. Finally, using two Trojan-infested DES cores, we showcase the effectiveness of the PCHIP framework, as enhanced through the hierarchy-preserving capabilities of *VeriCoq-H*, in detecting malicious data modifications. Details of the Verilog source codes, the corresponding Coq representations and proofs can be found in the supplementary material.

A. Evaluation of DES Cores

A block diagram of the DES [3] algorithm is shown in Fig. 7. It includes 16 rounds, preceded and succeeded by initial and final permutations. We mark the eXclusive-OR operations of each round and the one inside the cipher function as sensitivity reducing operations. Since permutations and rotations are deterministic, we do not consider them as sensitivity reducing operations. Input text and key are considered sensitive and their initial sensitivity levels are assigned accordingly.

1) *Area Efficient DES Core:* Since encryption rounds in the DES algorithm are similar, the first DES core, of which part of the Verilog code is shown in Fig. 3, implements a single round. Several iterations are then invoked to perform the entire encryption. The iteration number, which is determined by the *roundSel* input, is also considered sensitive.

Static Methodology: First, we show how the static IFT approach can be used for evaluating this core. The *des* module of Fig. 3, instantiates two sub-modules, the Feistel function (*crp.v*) and the key generator (*key_sel.v*) to perform

```

1 Definition des : code :=
2   outb desOut;
3   inb desIn;
4   inb key;
5   inb decrypt;
6   inb roundSel;
7   inb clk;
8   wireb K_sub;
9   wireb IP;
10  wireb FP;
11  regb L;
12  regb R;
13  wireb Xin;
14  wireb Lout;
15  wireb Rout;
16  wireb out;
17
18  assign_ex Lout (cond (eq (econb roundSel)
19    (econv (lo::lo::lo::lo::nil))) (econb (IP @ [33, 64])) (econb R));
20  assign_ex Xin (cond (eq (econb roundSel)
21    (econv (lo::lo::lo::lo::nil))) (econb (IP @ [1, 32])) (econb L));
22  assign_ex Rout (exor_key (econb Xin) (econb out));
23  assign_ex FP (econb (bus_app Rout Lout));
24
25  module_inst2in out Lout K_sub;
26
27  nonblock_assign_ex L (econb Lout);
28  nonblock_assign_ex R (econb Rout);
29
30  module_inst3in K_sub key roundSel decrypt;
31
32  assign_ex IP (perm (econb desIn));
33  assign_ex desOut (perm (econb FP)).

```

Fig. 8. Coq representation of area efficient DES core for static information flow tracking

round encryption/decryption and round key generation, respectively. Detailed analysis of the code reveals a potential sensitive information leakage for this DES core, which might occur in the first round. As is evident in Fig. 7, the right half of the input text, after the initial permutation, does not go through a sensitivity reduction operation (eXclusive-OR). Rather, it goes directly to the second round. Analysis of the partial Verilog source code of Fig. 3, focusing on `desIn`, `IP`, `Lout`, `FP` and `desOut`, reveals the same observation. Since the permutation operations are deterministic, at least part of the sensitive input `desIn` can leak to `desOut`. In essence, this area-efficient design only implements one round of the algorithm and has a potential risk in leaking information, which can be caught by PCHIP-based static IFT, as we show below.

Based on the Verilog-to-Coq conversion rules and the *formal semantic model* of part I, this design is converted to Coq. The complete version of the converted Coq representation for `des.v` is shown in Fig. 8.

Static Property Proofs: For each module, we need to denote the signal secrecy tags and prove the data secrecy properties. For the `des.v` module, among all input, output and internal signals, the input key (`key`), input plaintext (`desIn`), input round count (`RoundSel`), and internal generated round keys (`K_sub`) require protection. Reflected in Coq formal logic, three axioms are added to indicate the semantics that the `key`, `desIn`, `RoundSel`, and `K_sub` have sensitive tags in *all* clock cycles. The separation of signal property denotation axioms and circuit code constitutes a key characteristic of the Coq platform. These axioms act as *preconditions* for all security properties extracted from the Coq circuit.

```

Axiom secret_key : forall (t : nat),
  bus_sen key t = sensitive.
Axiom secret_desIn : forall (t : nat),
  bus_sen desIn t = sensitive.
Axiom secret_K_sub : forall (t : nat),
  bus_sen K_sub t = sensitive.
Axiom secret_RoundSel : forall (t : nat),
  bus_sen roundSel t = sensitive.

```

With both the preconditions and the DES circuit itself available in Coq representation, the next step is to construct

the data secrecy property that we wish to prove and express it also in Coq. Our requirement that “no internal sensitive information is leaked through a primary output” is formalized into the following `no_leaking_des` theorem in Coq.

```

Theorem no_leaking_des : forall (t : nat),
  chk_code_sen des t = non_sensitive.

```

If the proof can be successfully constructed for the `no_leaking_des` theorem given the Coq circuit and the preconditions, we can then declare that the delivered HDL code is trusted *with respect to data secrecy protection policy*. However, the theorem cannot be proven. In fact, we can prove the opposite conclusion, i.e. the DES output is sensitive. This vulnerability stems from the single-round implementation of DES, which is repeatedly used to perform the entire encryption. As illustrated in partial Verilog source code of Fig. 3, in this implementation the intermediate results are visible at the primary outputs; hence, the right half of the plaintext, which does not go through a sensitivity reducing operation in the first round, becomes exposed and causes the security theorem proof to fail. From an IP customer perspective, this implies that an IP vendor cannot provide a trusted bundle for this DES core since a proof cannot be constructed. While this static IFT approach is effective in successfully revealing design flaws, such as the one mentioned above, it involves a tedious manual process and is more suitable for single-stage or small designs.

Dynamic methodology implemented by VeriCoq-IFT: We also applied the *VeriCoq-IFT* framework to evaluate the trustworthiness of this DES core. This automatically converted the design to Coq representation and generated the security theorems and proofs. When provided to Coq-IDE for checking, however, we observed that the checking of proofs fails. This is expected, since as we mentioned in the evaluation by the static approach, this area efficient DES core has the potential of leaking sensitive information. *VeriCoq-IFT* exposes such inherent potential problems as well as deliberate violation of information flow policies by malicious modifications, automatically and accurately, and is therefore capable of handling larger and more complex designs, as we discuss next.

2) Performance Optimized DES Core: This DES core is implemented in a 16-stage pipeline, optimized for high performance requirements. Evaluation of this pipelined core using the static methodology is difficult, since we need to verify every round of encryption implemented by each pipeline stage separately. For each round, we need to consider its eXclusive-OR operations as sensitivity reducers while considering these operations in other stages as normal and verify the data secrecy property. Using the dynamic IFT method, however, which can automatically propagate the sensitivity levels, is much easier and accurate for this purpose. Therefore, we evaluate this core using only the dynamic methodology.

We converted the design to Coq representation utilizing *VeriCoq-IFT* and used its automatically generated theorems and proofs to evaluate the trustworthiness of this DES core with respect to information flow policies. Coq-IDE successfully passes the proofs of the security property theorems for this design, thereby guaranteeing that, indeed, it does not violate any information flow policies.

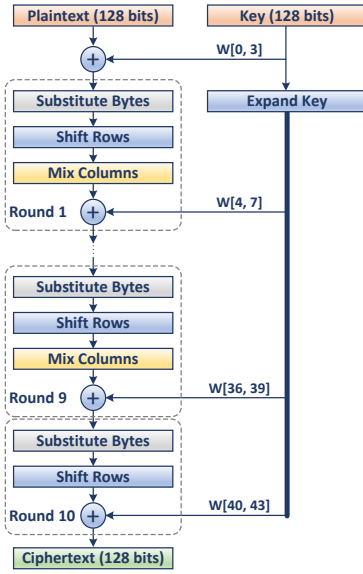


Fig. 9. AES-128 block diagram [4]

B. Evaluation of AES Cores

AES [4] is a more sophisticated encryption algorithm as compared to DES, and can be performed using 128-, 192-, and 256-bit keys. Fig. 9 shows the block diagram of AES-128 which requires 10 rounds. Along with the genuine AES-128, the Trust-Hub website [14] provides various Trojan-infested AES cores. Given the complexity and pipelined structure of these designs, we only use the dynamic methodology implemented by *VeriCoq-IFT* to evaluate these AES cores. Also, since the Trojan-infested AES designs combine several leaking mechanisms with various trigger conditions, we group them by their leaking mechanism for a clearer presentation. Table I summarizes the Trojan infected AES designs which we evaluated using *VeriCoq-IFT*. For each of these designs, on a Windows 7 computer with 8 GB of RAM and an Intel Core i7 processor, the conversion process takes less than a second, while proof verification completes in about 30 minutes.

1) *Genuine AES Core*: The genuine AES-128 core is a 10-stage pipelined design based on the block diagram shown in Fig. 9. We marked adding round key operations (+ operations) as sensitivity reducers and we defined appropriate sensitivity levels for the inputs. Then, we used *VeriCoq-IFT* to convert this design to its Coq representation and we evaluated the automatically generated proofs for the security property theorems in Coq-IDE. The checking of proofs passes, confirming that this design abides by the information flow policies.

2) *Group 1 of Trojan-Infested AES Designs*: These Trojan-infested AES designs have an additional output in which the Trojan tries to leak 8 bits of the key using a CDMA like modulation [12], [16]; their block diagrams are shown in Fig. 10. The intention is to connect this output, named *load* in Fig. 10, to a large capacitive load, and create a power side channel. The Trust-Hub website provides this Trojan with various trigger conditions. As Fig. 10 shows, AES-T100 and AES-T200 are always on. AES-T700 and AES-T1000 are triggered by a specific plaintext input, while AES-T800 and AES-T1100 require a predefined sequence of plaintext values

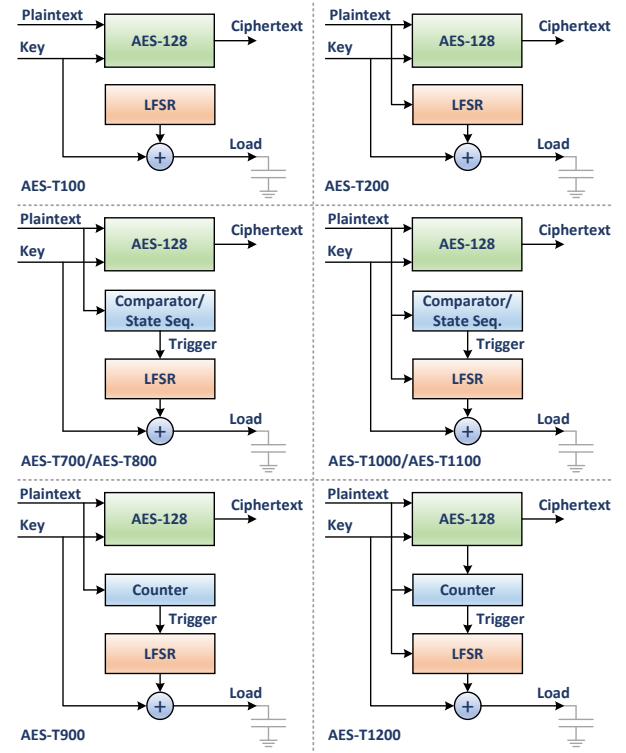


Fig. 10. First group of Trojan-infested AES designs with different triggers and LFSR initializations

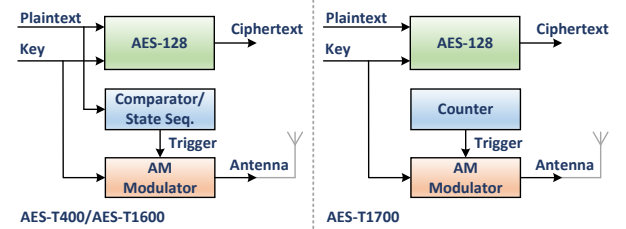


Fig. 11. Second group of Trojan-infested AES designs which leak the key by AM radio

to be activated. AES-T900 and AES-T1200 are triggered after a predefined number of encryptions. Another variation in these Trojans is the value used to initialize the LFSR for generating pseudorandom numbers. In Trojans shown on the left column of Fig. 10, the LFSR is initialized using a predefined constant value. Trojans on the right column use a subset of bits from the plaintext input to initialize the LFSR instead.

We used *VeriCoq-IFT* to convert these designs to their Coq representation and we evaluated the automatically generated proofs of the security property theorems in Coq-IDE. Thereby, we observe that checking of these proofs fails for the *load* output in all of these designs, indicating that *VeriCoq-IFT* can successfully detect the sensitive information leakage by these Trojans. Also, as expected, Coq successfully verifies the proofs of the security property theorems for *Ciphertext* outputs, as these do not leak any sensitive information.

3) *Group 2 of Trojan-Infested AES Designs*: This group of Trojan-infested AES designs leak the key by generating a radio-frequency (RF) signal on an output pin, which is intended to work as an antenna. They leak the key through amplitude modulation (AM), easily received and interpreted by a regular AM radio as meaningful beeps [17]. The difference

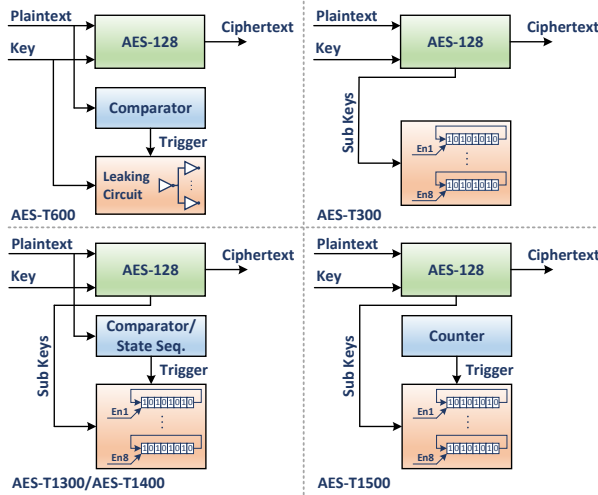


Fig. 12. Third group of Trojan-infested AES designs which leak the key by establishing an internal side channel

between the Trojans in this group is the trigger mechanism, as shown in Fig. 11. AES-T400 is activated by a predefined plaintext value. AES-T1600 is triggered by a specific sequence of plaintext input and AES-T1700 is activated after a predefined number of encryptions.

Evaluation of these Trojan-infested AES designs using *VeriCoq-IFT* shows that checking the proofs of the security theorems for the *Antenna* output fails for these designs. Therefore, *VeriCoq-IFT* can successfully detect this group of Trojans. Since these Trojans do not modify the AES core itself, checking the proofs of security theorems for the *Ciphertext* output in these designs passes in Coq-IDE, showing that no sensitive information is leaked by these output ports.

4) *Group 3 of Trojan-Infested AES Designs*: The third group of Trojan-infested AES designs that the Trust-Hub website provides, shown in Fig. 12, do not have a dedicated output to create the side channel for leaking the key. AES-T600, one of the Trojan-infested designs in this group, creates an internal load using a few inverters to leak the entire key and is activated by a predefined plaintext input. Another type of Trojan-infested design in this group generates 8 enable signals as a function of the plaintext input and 8 sub-keys. Each enable signal is used to activate rotation of the value in an 8 bit shift register, initialized with “10101010”, to create a power side channel [12]. As Fig. 12 shows, this type of Trojans are provided with four trigger conditions similar to Trojans in the previous groups. AES-T300 is always on, AES-T1300 is activated by a predefined plaintext, AES-T1400 is triggered by a specific sequence of plaintext input, and AES-T1500 is activated after a predefined number of encryptions.

Since these Trojans do not use an explicit output, *VeriCoq-IFT* cannot automatically generate the information flow policy theorems for the internal load signals and shift registers. However, by marking the suspicious signals and registers as explained in Section II-B, we can instruct *VeriCoq-IFT* to generate such theorems for these signals. Following this step, checking the automatically generated proofs for the security theorems on these signals does not pass in Coq-IDE. This reveals that these signals obtain sensitive values, thereby

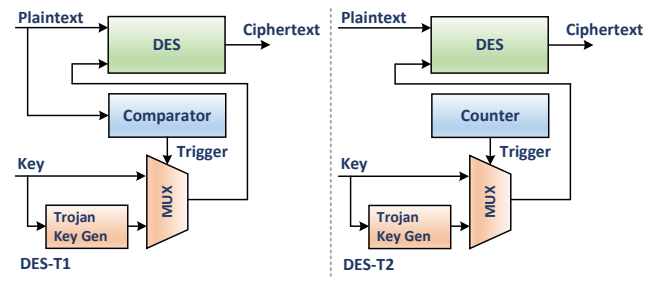


Fig. 13. DES Trojans which modify the secret key

alerting designers to perform further evaluations in order to either justify existence or remove such assignments.

5) *Group 4 of Trojan-Infested AES Designs*: The last group of Trojan-infested AES designs that the Trust-Hub website provides consists of Trojans which do not leak sensitive information. These Trojans increase the power consumption of the design, with the intention of depleting the energy source (battery) more quickly, and use three different Trigger conditions: predefined plaintext (AES-T1800), specific sequence of plaintext input (AES-T500) and predefined number of encryptions (AES-T1900). Since these Trojans do not leak sensitive information, we exclude them from evaluation by *VeriCoq-IFT*. However, since the Trigger condition in two of these Trojans depends on the plaintext input, which is sensitive, we can use the same procedure as in Section IV-B4 to check their safety. Such checking, however, requires design review by the IP consumers and cannot be performed automatically by *VeriCoq-IFT*.

C. Preventing Malicious Modification of Data

To evaluate our methodology for preventing malicious modification of data, we developed two Trojan-infested DES cores by slightly modifying the pipelined DES implementation, shown in Fig. 13. These Trojans set the secret key to a constant value once they are triggered and have a simple structure.

The first DES Trojan is activated by a specific plaintext value. Using *VeriCoq-IFT*, we converted this design to the Coq representation and verified the information flow policy theorem. Since assignment of a malicious (modified) key is triggered by the plaintext input, this Trojan alters the sensitivity level of the key (due to a conditional assignment based on the trigger value), which then propagates in the design and violates the information flow policies. Therefore, the proof of theorems generated by *VeriCoq-IFT* fails Coq-IDE evaluation and the Trojan is detected.

Unfortunately, *VeriCoq-IFT* cannot always detect a malicious modification of sensitive data. As an example, the second DES Trojan is activated after a predefined number of clock cycles and does not change the sensitivity level of the secret key, because the trigger in this case has a sensitivity level of zero. Therefore, checking the proof of the information flow policy theorem passes for this design and the Trojan evades detection. Other methods are also possible to hide these Trojans. Hence, malicious modification of data cannot always be prevented by enforcing information flow policies. To prevent such meddling, we resort to the general PCHIP

TABLE I
SUMMARY OF TROJAN INFECTED AES DESIGNS

Group	Trojan	Trigger	Leaking mechanism	Detected by <i>VeriCoq-IFT</i> ?
1	T100	Always on	CDMA like modulation, LFSR initialized by a constant	Yes
	T200	Always on	CDMA like modulation, LFSR initialized by plaintext	Yes
	T700	Specific plaintext value	CDMA like modulation, LFSR initialized by a constant	Yes
	T800	Specific plaintext sequence	CDMA like modulation, LFSR initialized by a constant	Yes
	T1000	Specific plaintext value	CDMA like modulation, LFSR initialized by plaintext	Yes
	T1100	Specific plaintext sequence	CDMA like modulation, LFSR initialized by plaintext	Yes
	T900	Specific number of encryptions	CDMA like modulation, LFSR initialized by a constant	Yes
	T1200	Specific number of encryptions	CDMA like modulation, LFSR initialized by plaintext	Yes
2	T400	Specific plaintext value	AM modulation	Yes
	T1600	Specific plaintext sequence	AM modulation	Yes
	T1700	Specific number of encryptions	AM modulation	Yes
3	T600	Specific plaintext value	Internal load for side-channel	Yes (with guidance)
	T300	Always on	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1300	Specific plaintext value	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1400	Specific plaintext sequence	Enabling internal shift registers for side-channel	Yes (with guidance)
	T1500	Specific number of encryptions	Enabling internal shift registers for side-channel	Yes (with guidance)
4	T1800	Specific plaintext value	No leak - deplete battery	Not evaluated
	T500	Specific plaintext sequence	No leak - deplete battery	Not evaluated
	T1900	Specific number of encryptions	No leak - deplete battery	Not evaluated

framework. In this case, we utilized *VeriCoq-H* to convert the genuine pipelined DES core to the corresponding Coq representation and developed theorems ensuring the authenticity of key bits throughout the design. Proving these theorems was performed hierarchically, wherein lemmas are first proven in sub-modules and then applied in theorems of higher-level modules to construct proofs.

As an example, lines 65-69 in the partial Coq representation of Fig. 14 define a theorem named `key_sub1_genuine` stating that the first round of encryption is done by an eXclusive-OR operation on the first sub-key (**K1**) and the corresponding signal created from the plaintext input (**E**). Also, this theorem ensures that the LSB of the first sub-key is indeed coming from the corresponding key input, according to the specifications. To prove this theorem, we used theorems `crp_u0_X_eq_E_xor_k1` and `k1_bit48_decrypt0_genuine` which are proved in `module_des`, as seen in lines 27-50 of Fig. 14. To prove `crp_u0_X_eq_E_xor_k1` in `module_des`, we applied a theorem proved in `module_crp` which is a sub-module instantiated in `module_des`. Similarly, the proof of theorem `k1_bit48_decrypt0_genuine` uses theorem `k1_bit48_decrypt0_key_r` which subsequently applies a theorem proven in `module_key_sel`, a sub-module of `module_des`. For Trojan-infested DES designs which modify the secret key, evaluation of proofs of these theorems in Coq-IDE results in failure, thereby enabling their detection by this generalized PCHIP-based framework. Crucially, this success relies on the hierarchical approach, facilitated by *VeriCoq-H*, which makes proof construction more streamlined and facilitates the creation of design libraries consisting of modules and corresponding reusable lemmas.

In order to compare the *hierarchical* to the *flattened* functional model, we also converted the DES core to the Coq representation using the flattened option and we developed the corresponding proof for the same theorem. The details of the converted code can be found in the supplementary material. While proof development style varies across developers (e.g., by proving different intermediate lemmas or employing

different Coq tactics), the key advantage of the hierarchical flattened model is that it provides separate **Module Type** definitions for each Verilog module. This modularity enables development of lemmas inside a module, independent of the higher-level module that instantiates it and without the need to track and globally define local module signals, as required by the flattened model. In turn, this supports lemma reusability, which can significantly shorten proof development time.

As an additional experiment to corroborate the advantages of the hierarchical over the flattened functional model, we also developed theorems to protect the key in a basic RSA core. A clean and Trojan-infested version of an RSA core, which replaces the secret key based on a trigger condition, can be found on TrustHub [14] in VHDL format. After converting both versions to Verilog and applying the proposed method, we concluded that checking the corresponding proof passes for the clean design but fails for the Trojan-infested one. The codes for both the hierarchical and the flattened functional model, as well as the pertinent theorems and proofs, are provided in the supplementary material.

V. DISCUSSION

VeriCoq-IFT provides an automated PCHIP framework for information flow tracking in hardware cores, which is highly effective in preventing infestation of malicious capabilities seeking to leak sensitive data. However, similar to any other information flow tracking methodology, *VeriCoq-IFT* relies on trusted and accurate labeling of the initial sensitivity values and declassifying operations, in order to effectively track the flow of sensitive information in the design. In Section II, we described a procedure to find the initial sensitivity values of sensitive signals. Nevertheless, to make the best use of *VeriCoq-IFT*, designers and IP consumers should carefully review the *VeriCoq-IFT* labels in the design and should be able to justify them with precise reasoning based on a clean high-level architecture or block diagram of the design.

The scope of protection provided by PCHIP is limited by what is covered by the agreed upon security properties. Therefore, a carefully selected set of security properties is

```

1 Require Import Vericoq_ift.
2 (* ... *)
3 Module Type module_des.
4 Parameters key_r desIn_r (* ... *) : wire.
5 (* ... *)
6
7 Declare Module crp_u0 : module_crp.
8 Declare Module crp_u1 : module_crp.
9 (* ... *)
10 Declare Module key_sel_uk : module_key_sel.
11
12 Definition instantiate
13   (desOut desIn key decrypt clk : bus) (t:nat) :=
14   (* ... *)
15   .
16
17 Theorem k1_bit48_decrypt0_key_r:
18   forall (t : nat) (desOut desIn key decrypt clk : bus),
19   module_des.instantiate desOut desIn key decrypt clk t ->
20   decrypt t = lo::nil ->
21   K1 [(48 - 48), (48 - 48)] t = key_r [28, 28] t.
22 Proof.
23   (* ... *)
24   apply key_sel_uk.k1_bit48_decrypt0_genuine with (* ... *)
25   Qed.
26
27 Theorem k1_bit48_decrypt0_genuine:
28   forall (t : nat) (desOut desIn key decrypt clk : bus),
29   module_des.instantiate desOut desIn key decrypt clk t ->
30   module_des.instantiate desOut desIn key decrypt clk (S t) ->
31   decrypt t = lo::nil ->
32   decrypt (S t) = lo::nil ->
33   K1 [(48 - 48), (48 - 48)] (S t) = key [28, 28] t.
34 Proof.
35   (* ... *)
36   apply k1_bit48_decrypt0_key_r
37   with desOut desIn key decrypt clk.
38   apply H0. apply H2.
39   Qed.
40
41 Theorem crp_u0_X_eq_E_xor_k1:
42   forall (t : nat) (desOut desIn key decrypt clk : bus),
43   module_des.instantiate desOut desIn key decrypt clk t ->
44   crp_u0.X t = bv_xor (crp_u0.E t) (K1 t).
45 Proof.
46   intros.
47   apply crp_u0.X_eq_E_xor_K_sub
48   with out0 (IP [(64 - 33), (64 - 64)]).
49   (* ... *)
50   Qed.
51
52 End module_des.
53
54 Module Type module_des_top.
55
56 Parameters desOut desIn key decrypt clk : bus.
57 Declare Module des_top : module_des.
58
59 Axiom des: forall (t:nat),
60   des_top.instantiate desOut desIn key decrypt clk t.
61
62 Axiom decrypt0: forall (t: nat),
63   decrypt t = lo::nil.
64
65 Theorem key_sub1_genuine: forall (t : nat),
66   des_top.crp_u0.X (S t) =
67   bv_xor (des_top.crp_u0.E (S t)) (des_top.K1 (S t)) /\
68   des_top.K1 [(48 - 48), (48 - 48)] (S t) = key [28, 28] t.
69   (* /\ statements for authenticity of other key bits ... *)
70 Proof.
71   intros.
72   split.
73   apply des_top.crp_u0_X_eq_E_xor_k1
74   with desOut desIn key decrypt clk.
75   apply des.
76   apply des_top.k1_bit48_decrypt0_genuine
77   with desOut desIn decrypt clk.
78   apply des. apply des.
79   apply decrypt0. apply decrypt0.
80   Qed.
81
82 End module_des_top.

```

Fig. 14. Partial hierarchical proof construction to prevent malicious modification of data in the pipelined DES core

required, commensurate with the protection level sought by each application. Additionally, while enforcement of information flow policies can effectively protect the design against leakage of sensitive information, the scope of these security properties pertains only to information flow. For example, such security properties and, consequently, *VeriCoq-IFT*, are not geared towards guaranteeing functional correctness of the hardware design. Moreover, tampering with sensitive information cannot always be captured by enforcing information flow policies. Nevertheless, the general PCHIP framework is extensive enough to address such aspects of security requirements in a cryptographic hardware IP design by combining *VeriCoq-IFT* with the original PCHIP method, enhanced with *VeriCoq-H* to convert the functionality of a design to a hierarchy-preserving Coq representation whereon formal reasoning on functionality can be hierarchically performed.

Moreover, the generalized PCHIP-based framework is mainly geared towards design-level attacks which use primary outputs to digitally leak sensitive information. Stealing information through other means, such as side-channels (e.g. power, timing, etc.), is not directly addressed through the proposed methods. Nevertheless, such side channel-based Trojans may still be detected indirectly, as we presented through a few examples in Sections IV-B2 to IV-B4. This happens when a primary output, for which *VeriCoq-IFT* automatically generates security theorems, is proven by the proposed method to carry sensitive information due to malicious design modifications. If this signal is the source from which the side channel-based hardware Trojan obtains the sensitive data, such as in the MOLES [16] side-channel attack, the Trojan is indirectly detected. The same holds true when an internal signal is used as the source of leakage for the side-channel attack. In this case, however, the user needs to annotate the internal signals which are deemed suspicious, so that *VeriCoq-IFT* will automatically generate the corresponding security theorems. Such user guidance is not inconsistent with the practice required by many other hardware Trojan detection methodologies such as FANCI [18], VeriTrust [19] or ATPG-based methods for Trojan detection [20], [21]. Moreover, these methods focus on identifying nearly unused circuits or rare triggers as signatures for hardware Trojans and do not incorporate a notion of information flow. Hence, detection of a trigger is the only mechanism through which these methods may detect a side channel-based hardware Trojan. However, as summarized in Table I, several among the hardware Trojans that we studied in our experiments are always on and, therefore, evade these Trojan detection methodologies, while still being detected by the PCHIP-based solutions proposed herein.

Finally, our conservative information flow model implemented by *VeriCoq-IFT* is less accurate as compared to recently introduced methodologies such as SecVerilog [22] and the gate-level based information flow tracking for hardware Trojan detection [23]. Therefore, even though we have not encountered this situation in our experiments, it is possible that it may introduce false positives during circuit evaluation. SecVerilog, on the other hand, is not geared towards hardware Trojans, which is specifically considered in our methods, while hardware Trojan detection based on gate-level information flow tracking relies on model checking which is limited due to the state explosion problem in larger designs. Overall, the proposed work constitutes a powerful alternative approach for addressing data secrecy protection based on PCHIP-based information flow tracking, whose limitations may be further mitigated through further research and development.

VI. CONCLUSION

In part II of this paper series we described our efforts towards automating the extended PCHIP frameworks whose foundations we presented in part I. We first introduced *VeriCoq-IFT*, which provides a fully automated PCHIP framework for enforcing dynamic information flow policies, wherein designers only need to annotate the HDL code with minimal required information, such as the initial sensitivity levels

and the declassifying operations. Based on this information, *VeriCoq-IFT* automatically generates the Coq representation, theorems and proofs required for enforcing information flow policies. In our experiments, *VeriCoq-IFT* successfully revealed design flaws and/or malicious capabilities hidden inside various genuine and Trojan-infested DES and AES implementations. We then presented *VeriCoq-H*, a method for converting the exact functionality of a design from HDL to its Coq representation, based on the hierarchy-preserving functional model described in part I. We demonstrated how it can be used to hierarchically develop proofs of security properties for a design and we detailed its utility through an example of preventing and/or capturing malicious modification of sensitive data in hardware cores. Notably, *VeriCoq-H* facilitates proof construction and development of hybrid hardware module libraries containing HDL code and proof of reusable lemmas for the module, thereby significantly reducing the burden of proof writing.

In our ongoing research, we plan to expand the capabilities of *VeriCoq-IFT* and *VeriCoq-H* by adding support for a few additional Verilog constructs, such as **task** and **generate**. Enhancing the information flow model and reducing its limitations by improving its accuracy, as well as studying the trade-off between the degree of functionality preserved in the converted code in the Coq representation and the protection level achieved, is another direction of our current efforts. We also plan to explore further automation possibilities in the general PCHIP framework for various hardware designs, such as microprocessor IPs and communication cores. Expansion of the automated PCHIP framework will help its wide adoption by the hardware design community, resulting in more secure and trustworthy third-party IP acquisition transactions.

REFERENCES

- [1] INRIA. The coq proof assistant. [Online]. Available: <http://coq.inria.fr/>
- [2] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware IP," in *Proceedings of IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2015, pp. 163–168.
- [3] NIST, "Data encryption standard (DES)," *Federal Information Processing Standards Publication*, 1999.
- [4] N.-F. Standard, "Announcing the advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, pp. 1–51, 2001.
- [5] M.-M. Bidmeshki and Y. Makris, "VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015.
- [6] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proceedings of ACM Design Automation Conference (DAC)*, 2015, pp. 112:1–112:6.
- [7] 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.
- [8] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 824–829.
- [9] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *Proceedings of IEEE VLSI Test Symposium (VTS)*, 2012, pp. 252–257.
- [10] Y. Jin, B. Yang, and Y. Makris, "Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing," in *Proceedings of IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 99–106.
- [11] OpenCores. [Online]. Available: <http://opencores.org/>

- [12] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, "Trojan side-channels: lightweight hardware trojans through side-channel engineering," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 382–395.
- [13] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable SoC trust verification using integrated theorem proving and model checking," in *Proceedings of IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2016.
- [14] Trust-Hub. [Online]. Available: <https://www.trust-hub.org/>
- [15] D. Rudolf. Development and analysis of block ciphers and the DES system. [Online]. Available: <http://homepage.usask.ca/~dtr467/400/>
- [16] L. Lin, W. Burleson, and C. Paar, "Moles: Malicious off-chip leakage enabled by side-channels," in *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2009, pp. 117–122.
- [17] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware trojan design and implementation," *International Journal of Information Security*, vol. 10, no. 1, pp. 1–14, 2011.
- [18] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of ACM Conference on Computer & Communications Security (CCS)*, 2013, pp. 697–708.
- [19] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "Veritrust: verification for hardware trust," in *Proceedings of ACM Design Automation Conference (DAC)*, 2013, pp. 61:1–61:8.
- [20] S. Saha, R. S. Chakraborty, S. S. Nuthakki, D. Mukhopadhyay *et al.*, "Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2015, pp. 577–596.
- [21] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2009, pp. 396–410.
- [22] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 503–516.
- [23] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.



Mohammad-Mahdi Bidmeshki (S'11) received the B.Sc. and M.Sc. degrees in Computer Engineering from Sharif University of Technology, Tehran, Iran, in 2004 and 2006 respectively. He is currently a Ph.D. candidate in Computer Engineering at the University of Texas at Dallas. His current research includes hardware based security, trusted hardware design, formal methods in security and verification, and the applications of machine learning in computer security.



Xiaolong Guo received his double Bachelor's degrees from Beijing University of Posts and Telecoms (BUPT), and University of London (UL) in 2010, respectively. In 2013, he received his M.S. degree from BUPT. He is currently working toward the Ph.D. degree in electrical engineering at University of Central Florida, Orlando, FL. His current research interest includes design of scalable verification methods for hardware IP protection, Trusted SoC Verification, Cyber Security, Formal Methods, Program Synthesis, and Secure Language Design.



Raj Gautam Dutta is a Ph.D student with the EECS department at University of Central Florida. He obtained his B.Tech in Electronics and Communication from Visvesvaraya Technological University, India in 2007 and MS in Electrical Engineering with an emphasis on control systems from the University of Central Florida, USA in 2011. His current research interest includes development of security solutions for semiconductor soft IP cores by using formal verification techniques, design of attack detection and mitigation software for autonomous systems, and synthesis of robust controllers for cyber-physical systems.



Yier Jin is currently an assistant professor in the ECE Department at the University of Central Florida. He received his PhD degree in Electrical Engineering in 2012 from Yale University after he got his B.S. and M.S. degrees in Electrical Engineering from Zhejiang University, China, in 2005 and 2007, respectively.

His research focuses on the areas of trusted embedded systems, trusted hardware intellectual property (IP) cores and hardware-software co-protection on computer systems. He proposed various approaches in the area of hardware security, including the hardware Trojan detection methodology relying on local side-channel information, the post-deployment hardware trust assessment framework, and the proof-carrying hardware IP protection scheme. He is also interested in the security analysis on Internet of Things (IoT) and wearable devices with particular emphasis on information integrity and privacy protection in the IoT era. He is awarded the DoE Early CAREER Award in 2016 and is the best paper award recipient of DAC'15, ASP-DAC'16, and HOST'17.



Yiorgos Makris (SM'08) received the Diploma of Computer Engineering from the University of Patras, Greece, in 1995 and the M.S. and Ph.D. degrees in Computer Engineering from the University of California, San Diego, in 1998 and 2001, respectively. After spending a decade on the faculty of Yale University, he joined UT Dallas where he is now a Professor of Electrical Engineering, leading the Trusted and RELiable Architectures (TRELA) Research Laboratory. His research focuses on applications of machine learning and statistical analysis

in the development of trusted and reliable integrated circuits and systems, with particular emphasis in the analog/RF domain. Prof. Makris serves as an Associate Editor of the IEEE Transactions on Information Forensics and Security and the IEEE Design & Test of Computers Periodical and he has also served as a guest editor for the IEEE Transactions on Computers and the IEEE Transactions on Computer-Aided Design of VLSI Circuits and Systems. He is a recipient of the 2006 Sheffield Distinguished Teaching Award, as well as Best Paper Awards from the 2013 Design Automation and Test in Europe (DATE'13) conference and the 2015 VLSI Test Symposium (VTS'15).