Automatic Code Converter Enhanced PCH Framework for SoC Trust Verification

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

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

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

I. Introduction

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

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

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

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

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

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

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

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

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

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

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

Digital Object Identifier 10.1109/TVLSI.2017.2751615

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

The main contributions of this paper are as follows.

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

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

II. Related Work

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

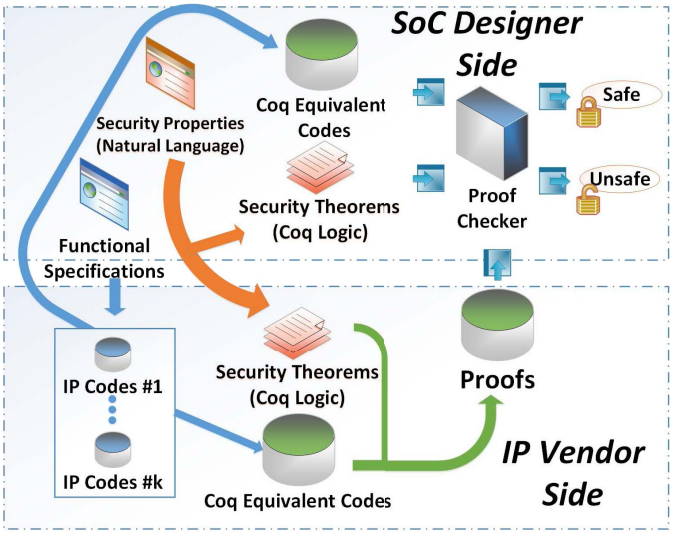

Fig. 1. Working procedure of the PCH [9].

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

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

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

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

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

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

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

III. Background

A. Attack Model and Assumptions

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

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

B. Formal Specification Languages

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

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

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

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

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

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

C. Interactive Theorem Prover

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

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

D. Model Checking

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

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

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

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

IV. Methodology

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

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

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

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

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

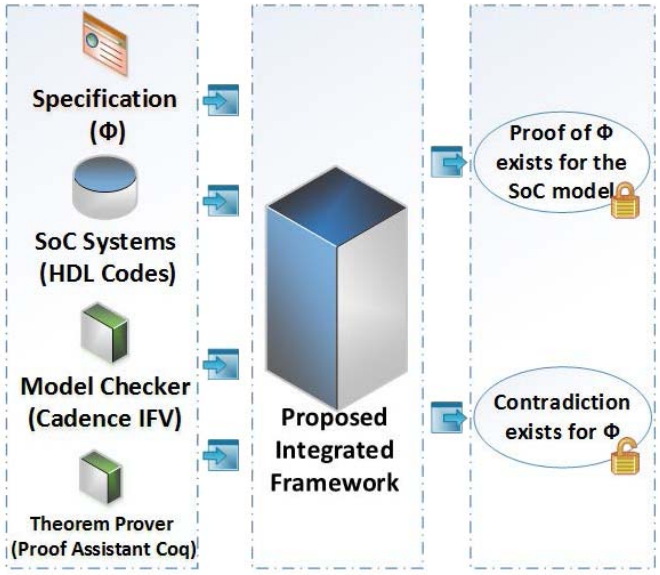

Fig. 2. Integrated formal verification framework.

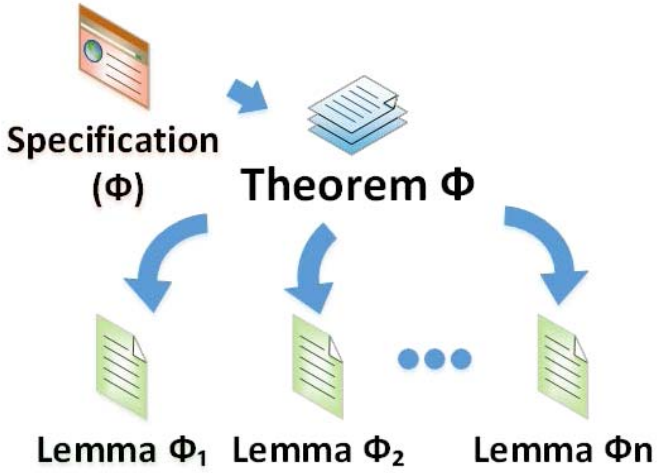

Fig. 3. Security specification (ϕ) decomposed into lemmas.

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

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

A. Semantic Translation

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

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

B. Distributed Proof Construction

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

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

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

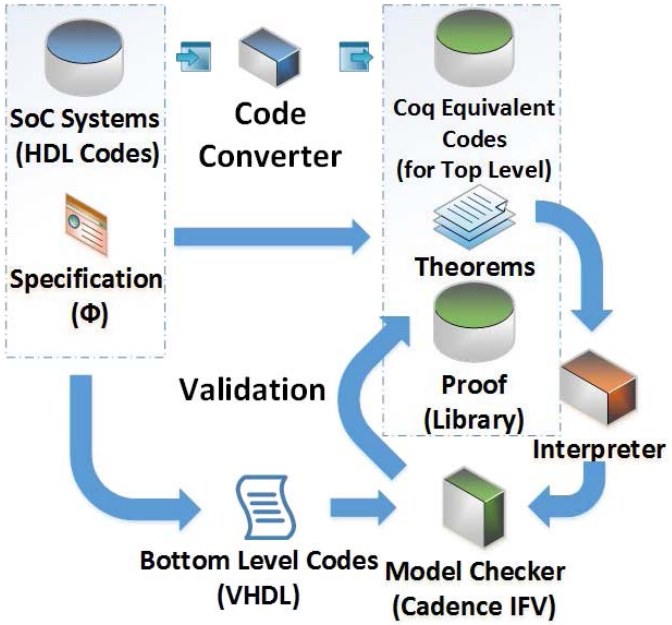

Fig. 4. Automatic PCH framework.

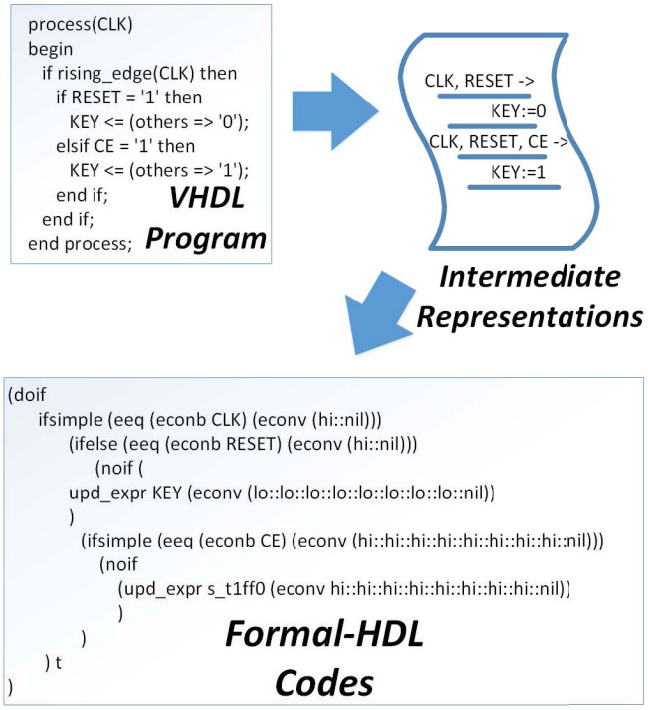

Fig. 5. Code conversion from VHDL to *Formal-HDL* through IRs.

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

C. Automatic PCH Framework

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

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

V. Automatic RTL-to-Formal Code Converter

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

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

A. VHDL to Intermediate Representations

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

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

$$c \rightarrow v := e \tag{1}$$

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 Fig. 5, most of the VHDL codes will be parsed to IRs in the form of (1).

B. Formal-HDL Representations

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

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

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

Listing 1 Date Types in Formal HDL Syntax

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

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

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

Listing 2 Assignment in Formal HDL Syntax

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

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

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

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

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

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

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

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

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

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

C. Automatic Code Converter Development

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

Listing 4 Circuit Module Definition in Formal HDL

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

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

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

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

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

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

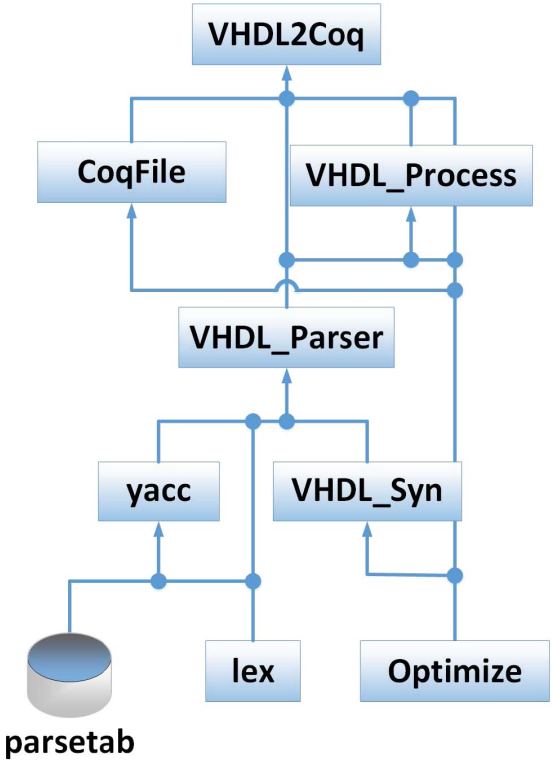

Fig. 6. Code conversion from VHDL to Formal-HDL through IRs.

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

VI. Case Study

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

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

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

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

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

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

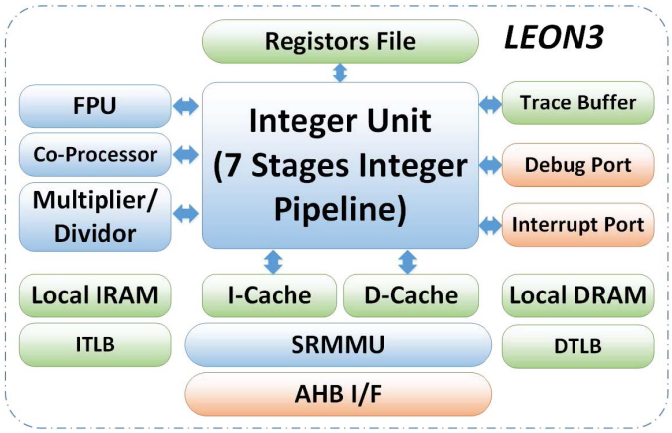

Fig. 7. Block diagram of IU of LEON3 core [37].

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

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

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

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

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

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

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

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

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

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

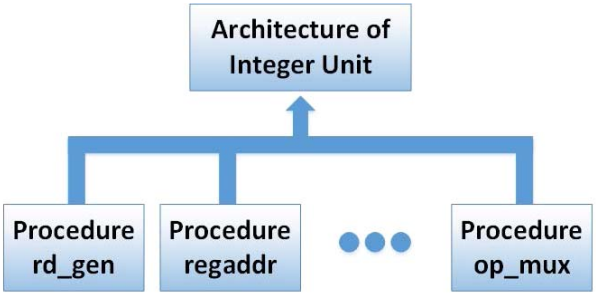

Fig. 8. Submodules in the IU of LEON3.

as follows:

$$\begin{aligned}\textbf{assert}(\{(cwp\_ifv\,[1:3] = \text{“111”})\\ \wedge\ (\text{reg\_ifv}\,[1:5] = \text{“01111”})\}\\ \mapsto\ (\text{rao\_ifv}\,[1:8] = \text{“01111111”})).\end{aligned}$$

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

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

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

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

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

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

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

VII. Conclusion and Discussion

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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