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 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.

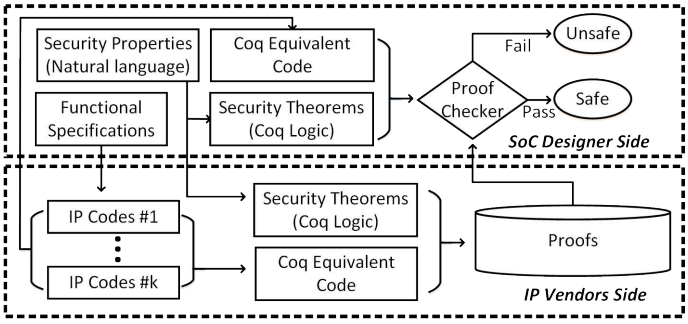

Figure 1: Working procedure of the PCH [13]

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 process[1].

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*.

[1]Note 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.

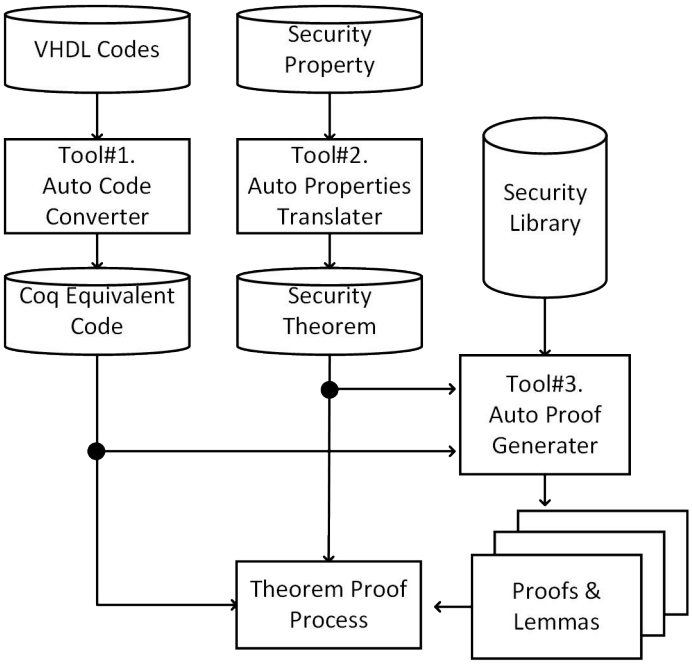

Figure 2: Automatic PCH Framework

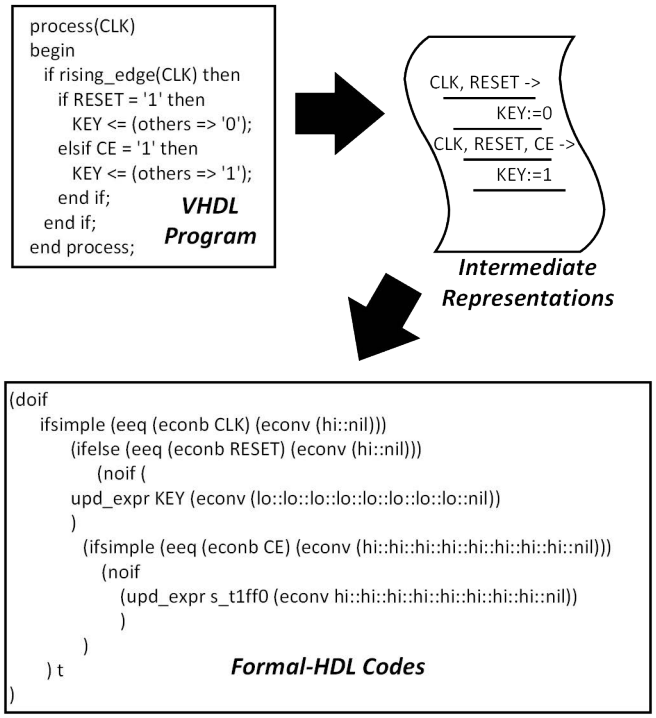

Figure 3: Code conversion from VHDL to *Formal-HDL* through IRs

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 ($\neg$, $\bigvee$, $\bigwedge$). Let C be the subset of E containing all boolean valued expressions. Then, a behavioural rule $b \in B$ can be written as:

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

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 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 a 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 scalablity 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. Maniatakos, "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.