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.ed; 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

1556-6013 © 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.

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

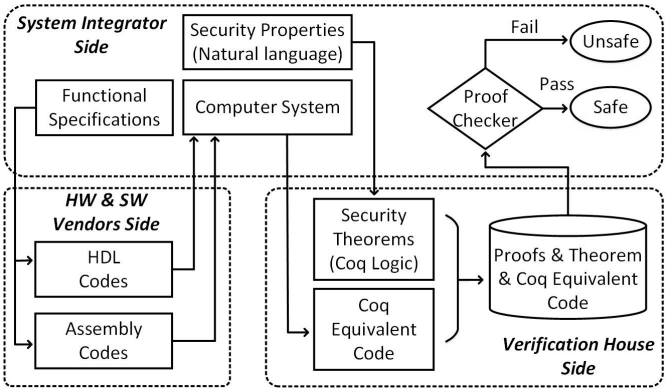

Fig. 1. Working procedure of the proposed *unified framework*.

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

[1]The 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

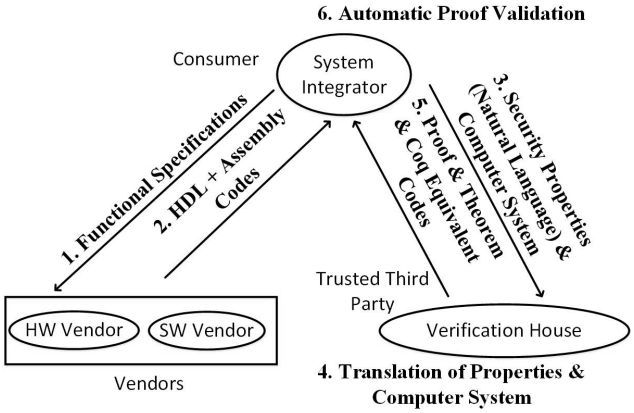

Fig. 2. Main parties in the *unified framework*.

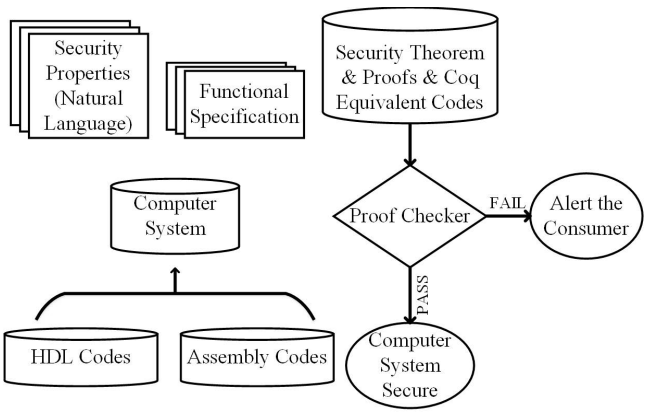

Fig. 3. Various tasks of the system integrator in the *unified framework*.

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.

- **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.

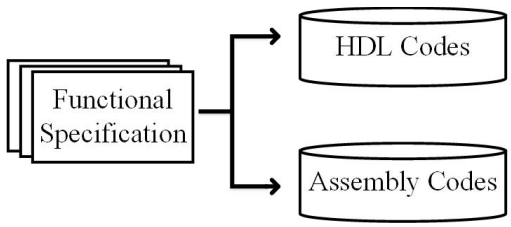

Fig. 4. Working procedure of the *unified framework* at the vendor side.

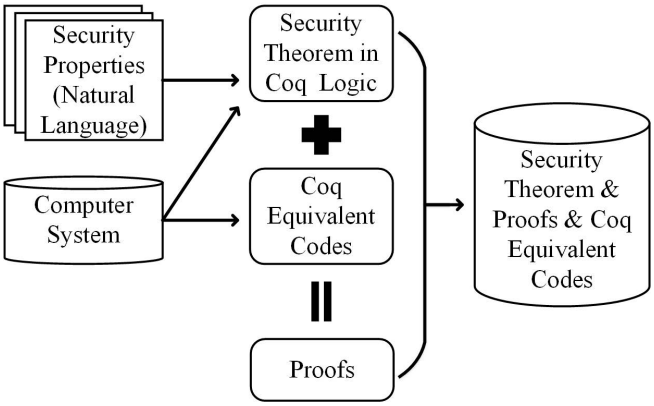

Fig. 5. Operations of the trusted verification house in the *unified framework*.

- **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 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

[2]Designing 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.

(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.

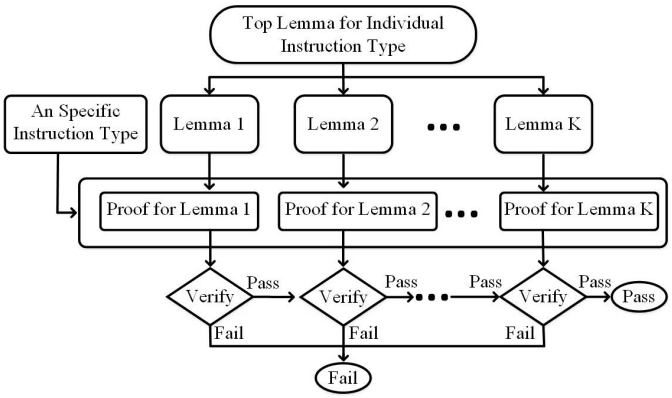

Fig. 6. *Lemma development* for security evaluation of individual instructions.

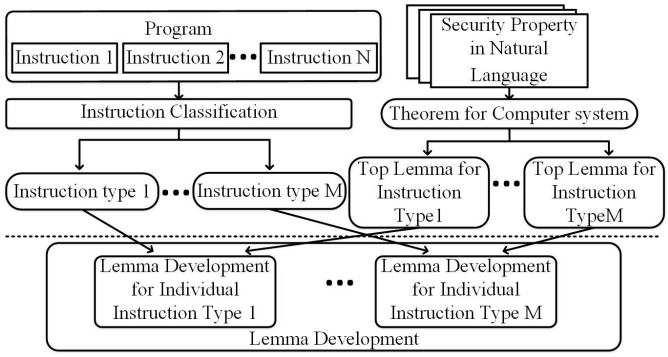

Fig. 7. *Theorem development* procedure for system-level security property of computer system.

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

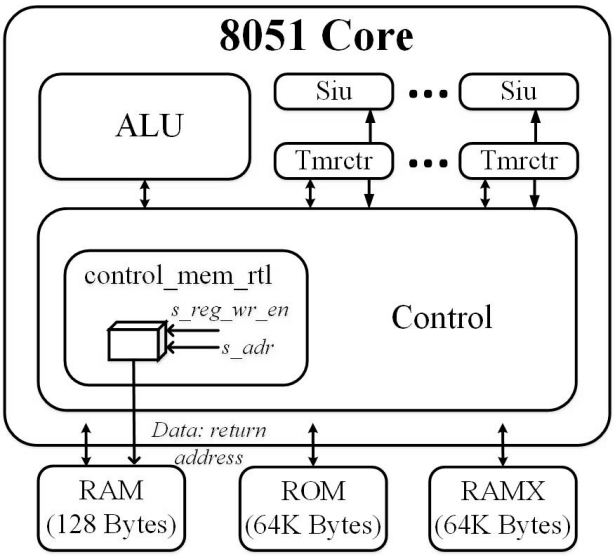

Fig. 8. Block diagram of an 8051 microprocessor.

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 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.

```
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.

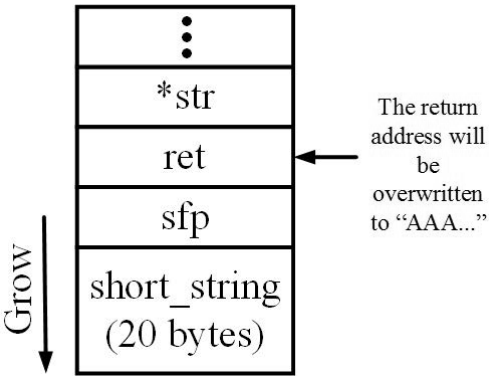

Fig. 9. The stack structure with return address.

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 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.

```
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.

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*

[3] As 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::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.

```
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::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.

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 -> 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 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.

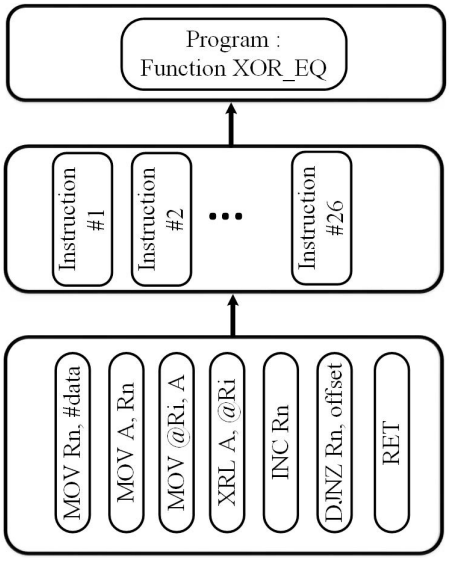

Fig. 10. Hierarchical proof construction process.

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::nil->
ie (S t) =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::nil ->
ie 2 = 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` indicates that `rom_data_i` takes input data at the `t+1` clock cycle.

```
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

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 languge 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.

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

[5]In 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.

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. Guneysu, 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.

[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 verifiation 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. Maniatakos, 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.

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 language design.

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

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.