Scalable SoC Trust Verification using Integrated Theorem Proving and Model Checking

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 and software programs from untrusted vendors have raised security concerns for system designers. Existing solutions for detecting and preventing software attacks do not usually consider the presence of malicious logic in hardware. Similarly, hardware solutions for detecting Trojans and/or design backdoors do not consider the software running on it. Formal methods provide powerful solutions in detecting malicious behaviors in both hardware and software. However, they suffer from scalability issues and cannot be easily used for large-scale computer systems. To alleviate the scalability challenge, we propose a new integrated formal verification framework to evaluate the trust of computer systems constructed from untrusted third-party software and hardware resources. This framework combines an automated model checker with an interactive theorem prover for proving system-level security properties. We evaluate a vulnerable program executed on a bare metal LEON3 SPARC V8 processor and prove system security with considerable reduction in effort. Our method systematically reduces the effort required for verifying the program running on the System-on-Chip (SoC) compared to traditional interactive theorem proving methods.**

I. Introduction

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

Increasing number of third-party vendors have raised security concerns in both the hardware and the software industry. Consequently, security researchers in their respective domains have started putting in considerable effort to ensure trustworthiness of third-party resources. In the software domain, methods have been developed for the detection of malicious kernel extensions and kernel integrity defense. In the hardware security industry, multiple countermeasures have been developed for verification and validation of SoC's at pre- and post-silicon level [1]–[11].

Among all the existing techniques, formal methods (both automated and deductive) have been most effective in detecting vulnerabilities in both software and hardware [4]–[12]. For example, model checking is used for detecting malicious logic that corrupts data in critical registers of third-party IP cores [11]. In a model checker, security properties such as integrity (related to safety) and availability (related to liveliness) are represented as *traces* and it checks all the possible traces generated by the system (software or hardware). If all the traces are good then the system is said to satisfy the security properties. However, not all security properties can be expressed as traces. Moreover, model checkers run into the state space explosion problem when the system under consideration is very large. Due to these limitations of model checkers, theorem provers are being mostly used for verification of large-scale hardware designs [5]–[7], [10].

Although these 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. Moreover, existing formal verification frameworks such as proof-carrying hardware (PCH), which rely on an interactive theorem prover for evaluating trustworthiness of IP cores, are not scalable to SoC designs [5]–[7]. The reason behind the scalability problem is the lack of efficient methods for constructing machine proofs. As the size of the design increases, time required for proving security properties on the design grows exponentially. To solve these problems, we propose an integrated formal verification framework where we combine a model checker with an interactive theorem prover for proving security properties on entire computer systems. Integrating these two techniques overcomes the state-space explosion problem in model checking approaches and it reduces the time required for constructing machine proofs of the properties in the theorem prover.

The main contributions of this paper are as follows.

- We combine the interactive theorem prover, Coq, with the Cadence IFV model checker for verifying system-level security on SoC designs. It is the first attempt to verify security properties on large-scale hardware designs through the combination of both techniques.
- We describe the method for decomposing the hardware design and the security specification into sub-modules and sub-specifications, respectively. These sub-modules and sub-specifications are verified using the model checker. In Coq, we combine the sub-specifications to prove the security property. Following this strategy, our approach can verify large systems and, thus, help alleviate the scalability issue.

The rest of the paper is organized as follows: In Section II, we discuss previous work on malicious logic detection using formal techniques. In Section III, we introduce the threat model and provide some relevant background on formal languages for specifying security properties, theorem prover, and model checker. We explain our integrated framework, semantic translation of VHDL language, and elaborate on the proof construction procedure in section IV. Section V presents demonstrations of our approach and final conclusions are drawn in Section VI.

II. Related Work

Currently, formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [4]–[11]. In [4], a multi-stage approach was adopted for identifying suspicious signals using assertion based verification, code coverage analysis, redundant circuit removal, equivalence analysis, and sequential Automatic Test Pattern Generation.The PCH framework has been effective in ensuring trustworthiness of soft IP cores [5]–[7], [9], [10]. This method was inspired from the proof-carrying code (PCC) approach of Necula [13]. Drzevitzky et al. proposed the first PCH framework for dynamically reconfigurable hardware platforms [10]. They used runtime combinational equivalence checkingto verify the equivalence between the design specification and the design implementation. However, instead of using security properties, the approach verified safety policies on the design. Another PCH framework was proposed for security property verification on soft-IP cores [5]–[7], [9]. In this framework, the Coq proof assistant [14] was used to represent security properties, hardware designs, and formal proofs. However, this framework was not scalable to large SoC designs because of the extremely high conversion and verification efforts in proving security properties on large designs.

In semiconductor industry, automated tools like equivalence checker and model checker have been consistently used for functional verification of hardware designs [15]. Using these tools, a model represented as a transition system is verified against a set of behavioral specification stated in a temporal logic. Recently, model checkers have been used for detecting malicious signals in third-party IP cores [4], [11]. However, these tools suffer from the state space explosion problem and hence cannot be used exclusively for verifying large designs.

Some efforts have been made to combine theorem provers with model checkers for verification of hardware and software systems [16], [17]. These methods try to overcome the limitations of both techniques. Some of the popular theorem provers such as higher order logic (HOL Light) and prototype verification system (PVS) have integrated model checkers. These tools have been used for functional verification of hardware systems. However, to the best of our knowledge, this combined technique has not been extended toward verification of security properties on third-party soft IP cores.

In this paper, we have combined a model checker with an interactive theorem prover for verifying security properties on SoCs. Through the proposed method, we are able to significantly reduce the time required for security verifications on SoCs.

III. Background

A. Attack Model and Assumptions

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

We assume that the verification tools (e.g., Coq and Cadence IFV) used in our integrated framework produce correct results. The existence of proofs for the security theorems indicates the genuineness of the design 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. Formal Specification Languages

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

In the Coq proof assistant [14], behavioral specifications are written using the *Gallina* specification language. This language can also be used to represent the hardware design. In case of an automated tool such as a model checker, specification language such as the Property Specification Language (PSL) is used for specifying properties or assertion of hardware designs. The directives of the PSL language, *assert*, *assume*, and *cover*, are understood by a verification tool such as the Cadence IFV. By using the *assert* construct a user can check at run time or at simulation time if a certain condition holds and reports a warning or an error if it does not hold. To put constrains on inputs of the design, *assume* is used and *cover* is used for specifying scenarios.

The PSL language is divided into four layers: (i) Boolean layer, (ii) temporal layer, (iii) verification layer, and (iv) modeling layer [18]. The Boolean layer is composed of Boolean expressions that either hold or not hold over a given clock cycle. The temporal layer allows to relate the Boolean expression with time. This layer is further divided into (i) foundation language (FL) and (ii) optional branching extension (OBE). The FL is used to describe linear properties in which there is only a single successor for a current state. Therefore, FL is often used to describe traces/path. In the FL, linear temporal logic (LTL) and the sequential extended regular expression

(SERE) are used to represent behavioral specifications of the system. Alternatively, OBE is based on computational tree logic (CTL) and can describe multiple traces (i.e., successor states) at a time. The verification layer consists of directives, which describe how the temporal properties should be used by the verification tool. That is, the verification layer specifies the semantics for PSL directives and operators in the temporal layer. It also helps the verification tool to understand the difference between properties which use *assert*, *assume*, and *cover* directives. The modeling layer provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables.

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

$$\phi ::= true \,|\, v \,|\, (\phi_1 \wedge \phi_2) \,|\, \neg\phi$$

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

C. Interactive Theorem Prover

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

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

D. Model Checking

Model checking [19] is a method for verifying and validating models in software and hardware applications [12], [15]. In this approach, a model (Verilog/VHDL code of hardware) $\mathcal{M}$ with an initial state s_0 is expressed as a transition system and its behavioral specification (assertion) ϕ is represented in a temporal logic. The underlying algorithm of this technique explores the state-space of the model to find whether the specification is satisfied. This can be formally stated as, $\mathcal{M}$, $s_0 \models \phi$. If a case exists where the model does not satisfy the specification, a counterexample in the form of a trace is produced by the model checker [20]. The application of model checking techniques, including symbolic approaches based on reduced order binary decision diagrams (ROBDD) and satisfiability (SAT) solving, to SoC has had limited success due to the state-space explosion problem. For example, a model with *n* Boolean variables can have as many as 2^n states, a typical soft IP core with 1000 32-bit integer variables has billions of states.

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

Another technique called bounded-model checking (BMC), replaces BDDs in symbolic checking with SAT solving [21]. In this approach, a propositional formula is first constructed using a model of the system, the temporal logic specification, and a bound. Then, the formula is given to a SAT solver to either obtain a satisfying assignment or to prove there is none. Although BMC outperforms BDD based model checking in some cases, the method cannot be used to test properties (specification) when bound is large or cannot be determined.

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

IV. Methodology

Existing PCH framework uses an interactive theorem prover for verifying security properties on soft IP cores which triggers a large design overhead [5], [6], [22]. Moreover, PCH requires flattening of the hardware design before translation of the HDL code to the formal language. Design flattening increases the verification effort and adds to the risk of introducing errors during the code conversion process. Also, this framework cannot be used to verify a computer system which consists of both software and hardware.

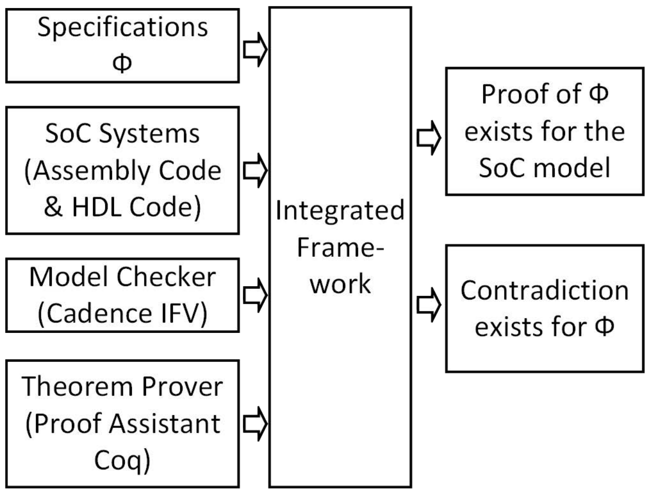
Specifications Φ
SoC Systems (Assembly Code & HDL Code)
Model Checker (Cadence IFV)
Theorem Prover (Proof Assistant Coq)
Integrated Frame-work
Proof of Φ exists for the SoC model
Contradiction exists for Φ

Figure 1: Integrated formal verification framework

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

To overcome the scalability issue and to verify a computer system, we introduce the *integrated formal verification framework* (see Figure 1), where the security properties are checked against SoC designs. In this framework, the theorem prover is combined with a model checker for proving formal security properties (specifications). Moreover, the hierarchical structure of the SoC is leveraged to reduce the verification effort.

In the integrated framework, the hardware design, represented in a hardware description language (HDL), and the assembly level instructions of a vulnerable program, is first translated to *Gallina*. Then, the security specification is stated as a formal theorem in Coq. In the following step, this theorem is decomposed into disjoint lemmas (see Figure 2) based on sub modules. These lemmas are then represented in the PSL specification language and are called sub-specifications. Subsequently, the Cadence IFV verifies the sub-modules against the corresponding sub-specifications. Sub-modules are functions, which have less number of state variables and are connected to primary output of the design. These functions are always from the bottom level of SoC and have no dependency relationship with each other.

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

A. Semantic Translation

The developed semantic translation method is based on the *formal HDL* in [22]. Using this method, the HDL code of the

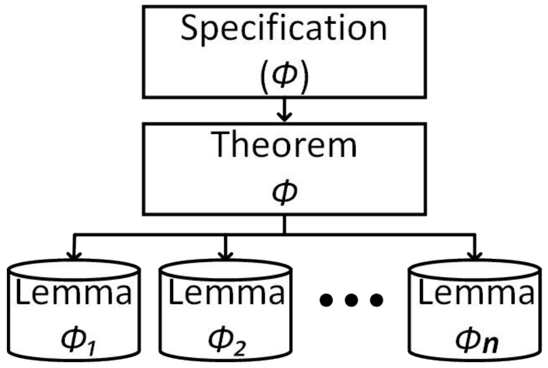

Figure 2: Security specification (ϕ) decomposed into lemmas

SoC and the informal security properties are translated to *Gallina*. During the translation process, syntax and semantics of the HDL are represented in Coq. To preserve the hierarchical design of the SoC, we use the *module* functionality of Coq. We also translate the syntax and the semantics of the assembly level program to *Gallina*. In this paper, semantic translation is manual. We are in the process of developing an automated framework to support code conversion.

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

B. Distributed Proof Construction

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

V. Case Study

For demonstration purpose, we consider a 32-bit LEON3 processor implementing the SPARC V8 architecture. This processor core is written in the VHDL. The integer-unit of the core, a 7-stage pipeline, is considered for verification (see

Figure 3). In order to prove the presence/absence of malicious logic that can trigger buffer overflow, we will check the signals connecting the integer unit to the register file.

In order to perform stack based buffer-overflow attacks, we consider the following assembly code of the subroutine, `vulnerable_function`.

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

This code is assembled and executed on a bare metal LEON3 processor. We only consider those scenarios where the `call` instruction is followed by a corresponding `return` instruction. Due to this constraint, if a return address is overwritten, then the callee will not be able to return to the caller. When a callee is invoked using the `call` instruction by a caller in LEON3, the return address of the caller is saved in the `i7` register of callee's register window (we consider the default setting of 8 register windows from `w0-w7`).

In the examined subroutine, the vulnerable instruction is `call 0x206b4 <strcpy@plt>`, which corresponds to the `strcpy()` function of the C library. This function is used to copy input to a buffer. When the input is longer than the size of the stack allocated buffer, the space reserved for a register window on the stack is overwritten. Upon returning from the function, if this portion of memory is loaded into the register file, the return address is corrupted.

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

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

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

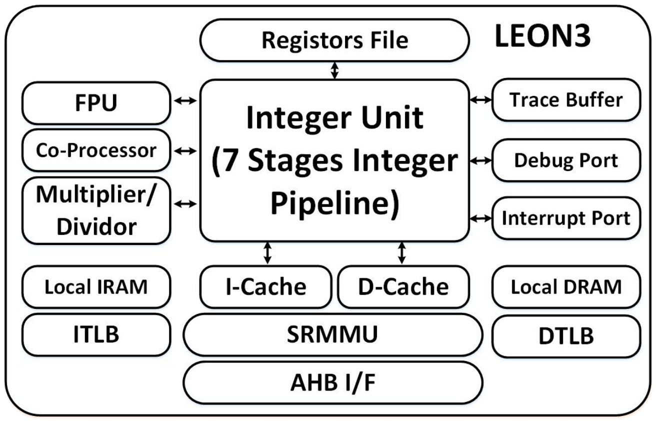

Figure 3: Block diagram of integer unit of LEON3 core [23]

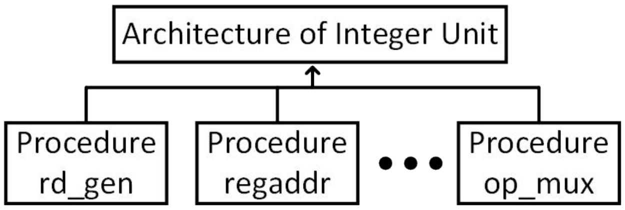

Figure 4: Sub-modules in the Integer Unit of LEON3

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

```
Theorem BufferOverflow_Cycle_1:
forall (t : nat),
t = 1 ->
ico.data_0 t = sethi_0_g0 ->
rstn t = hi::nil ->
holdn t = lo::nil ->
irqi.run t = hi::nil->
irqi.rst t = hi::nil->
 (bv_eq (rfi.wren t) (hi::nil)=lo\/
  bv_eq (rfi.waddr t) (lo::hi::hi::hi::hi::hi::hi::hi::nil)=lo).
```

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

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

As the VHDL code of the integer unit has a lot of `procedures` (shown in Figure 4) and `functions`, we allocate their verification task to the Cadence IFV. We verify the procedure, `regaddr`, using the model checker for the corresponding informal specification - *when the input signals* `cwp` *equals to "111", and* `reg` *equals to "01111", the output signal* `rao` *will be "01111111"*. An example specification (assertion) in the PSL language is shown below.

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

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

The above PSL specification in the VHDL language is given below.

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

We state the above PSL specification as the following lemma.

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

When the model checker proves that the code satisfies the specification, we can be assured that the proof of the lemma exist. By combining proofs of all the lemmas, we were able to prove the theorem, `BufferOverflow_Cycle_1`. Following this procedure, we were able to reduce the effort required for proving the security theorem in Coq. For instance, the Cadence IFV took only 0.03 seconds of CPU time for verifying the `Procedure regaddr` against the `ASSERT_SubModule_regaddr` specification. If this `Procedure regaddr` is verified in interactive theorem prover, e.g. Coq, it will take much longer to build models and prove lemmas manually.

VI. Conclusion

In this paper, an integrated formal verification framework is proposed to protect a large-scale SoC design from malicious attacks. Given that an interactive theorem prover (e.g. Coq) requires lot of effort to manually verify the design and that a model checker suffers from scalablity issues, we combine these two techniques together through the decomposition of the security property as well as the design in such a way that the model checker can verify those sub-modules which have much less state variables. Consequently, we reduced the amount of effort required for translating the design from HDL to *Gallina* and proving the security theorem in Coq. In future, we plan to use our approach for detecting sophisticated hardware Trojans with the assistance of automatic semantic translation tool.

Acknowledgment

This work was partially supported by NSF grants (CNS-1319105 and CNS-1441667), SRC(2014-TS-2554) 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] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.
[3] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," ser. CCS '13, 2013, pp. 697–708.
[4] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *HOST*, 2011, pp. 67–70.
[5] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.
[6] 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.
[7] Y. Jin, "Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2014.
[8] F. M. De Paula, M. Gort, A. J. Hu, S. J. Wilton, and J. Yang, "Backspace: formal analysis for post-silicon debug," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008, p. 5.
[9] X. Guo, R. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, June 2015, pp. 1–6.
[10] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *International Conference on Field Programmable Logic and Applications*, 2010, pp. 255–258.
[11] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," ser. DAC '15, New York, NY, USA, 2015, pp. 112:1–112:6.
[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software*. Springer, 2003, pp. 235–239.
[13] 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.
[14] INRIA, "The coq proof assistant," 2010, `http://coq.inria.fr/`.
[15] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying ieee compliance of floating-point hardware," *Intel Technology Journal*, vol. 3, no. 1, pp. 1–14, 1999.
[16] S. Berezin, "Model checking and theorem proving: a unified framework," Ph.D. dissertation, SRI International, 2002.
[17] P. Dybjer, Q. Haiyan, and M. Takeyama, "Verifying haskell programs by combining testing, model checking and interactive theorem proving," *Information and software technology*, vol. 46, no. 15, pp. 1011–1025, 2004.
[18] C. Eisner and D. Fisman, *A Practical Introduction to PSL*, ser. Series on Integrated Circuits and Systems. Springer, 2006.
[19] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.
[20] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.
[21] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.
[22] 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.
[23] Gaisler Research. LEON3 synthesizable processor. http://www.gaisler.com.