Runtime SoC Trust Verification using Integrated Symbolic Execution and Solver

Xiaolong Guo, Jiaji He, and Yier Jin
Department of Electrical and Computer Engineering, University of Florida
yier.jin@ece.ufl.edu

Abstract*—Untrusted third-party vendors and manufacturers have raised security concerns in hardware supply chain. Among all existing solutions, formal verification methods provide powerful solutions in detection malicious behaviors at the pre-silicon stage. However, little work have been done towards built-in hardware runtime verification at the post- silicon stage. In this paper, a runtime formal verification framework is proposed to evaluate the trust of hardware during its execution. This framework combines the symbolic execution and SMT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using a SoC design with untrusted IPs. The experimentation results show that the proposed approach can provide high-level security assurance for hardware at runtime.

I. Introduction

The changing landscape of the semiconductor industry has increased the demand for third-party 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 computing system. It is designed by integrating multiple IP cores from trusted and untrusted third party vendors.

Increasing number of third-party vendors have raised security concerns in the IC industry. Due to the extremely high cost of building foundries, chip manufacturing is usually outsourced to existing foundries. Therefore, a comprehensive approach is required to protect against attacks from untrusted vendors and manufacturers. Formal methods have shown their importance in exhaustive hardware security verification [1]–[4], but few of them were designed for securing post-fabrication designs. For example, in [2]–[4], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [5], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. However, model formalization and interactive proof procedures in PCH limit the scenario into static verification for design stage in the supply chain.

In this paper, we address the runtime hardware security verification challenge by extending our Proof-Carrying Hardware (PCH) [3], [4], [6] framework from static to dynamic (aka runtime) with a SMT solver and symbolic executions. The working procedure of the developed runtime PCH framework is shown in Figure 1. The main contributions of this paper are as follows.

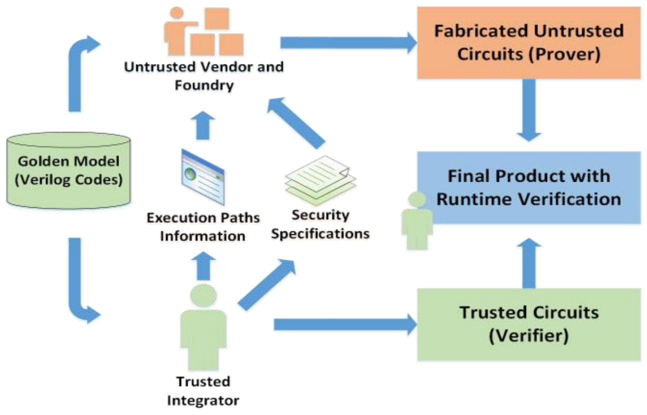
Untrusted Vendor and Foundry
Fabricated Untrusted Circuits (Prover)
Golden Model (Verilog Codes)
Execution Paths Information
Security Specifications
Final Product with Runtime Verification
Trusted Circuits (Verifier)
Trusted Integrator

Fig. 1: Working procedure of runtime PCH framework

- We combine a SMT solver with a static program analysis method for runtime checking of security of hardware.
- The work improves the study of hardware runtime verification. Our enhanced PCH framework provides comprehensive protection of hardware by complying to user specified security properties.

II. Background and Related Work

A. Attack Model

Hardware Trojans/Malicious logic can be inserted by adversaries at the different stages of the IC life-cycle. We assume that the rogue agents at the third-party IP design house and foundry can insert a hardware Trojan or backdoor to the fabricated circuit. 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.

B. Related Work

In [3], [4], [6], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [5], formal security properties were formalized and proved to ensure the trustworthiness of IP cores. However, this framework can only provide static verification on design stage of hardware other than the runtime of hardware. In [7], a SAT solver is utilized to enhance the PCH to be applicable in runtime scenario. Still, expressiveness of the SAT solver is not powerful enough so that security properties are difficult to be formalized in such framework.

Verifiable ASICs was proposed by Wahby et.al. [8] to verify the correctness of functionality of hardware system. In their paper, runtime (or dynamic) verification was performed by implementing an interactive encryption protocol between untrusted ICs and a second trusted ICs, where the untrusted ICs was called $Prover$ and trusted ICs was called $Verifier$. It was the first attempt to compute proofs of correct execution through utilizing verifiable computation. However, for secu- rity purpose, their correctness checking method would result in high computational cost and overhead. Furthermore, their method was designed for checking specific property rather than the entire set of functional properties.

C. Background

Satisfiability (SAT) solvers have been used in many electronic design automation fields like logic synthesis, verification, and testing. The SAT solvers are originally designed to solve the well-known Boolean Satisfiability problem, which decides whether a propositional logic formula can besatisfied given value assignments of the variables in the formula. Based on SAT solver, satisfiability modulo theories (SMT) solver is derived by including serval first-order theories, such as arithmetic, bit-vectors, quantifiers, etc. [9]. However, due to the high computational complexity, there is no hardware implementation for SMT solvers, and the software based SMT solver are not scalable to large designs.

Symbolic execution is a program analysis technique that can explore multiple paths that a program could take under different inputs [10]. In this method, execution paths that the program should take are explored systematically to avoid the space explosion problem. Specifically, inputs are represented as symbols and the solvers are used to check whether there are counter examples of the property. For each path, a Boolean formula is derived to describe the conditions of the branches, while a symbolic memory is used to map variables to symbolic expressions. The Boolean formula is updated after executing the branch and the symbolic memory is updated after each assignment. Integrating these two techniques overcome the NP-Hard computation complexity issue in SAT solver and it provides a comprehensive protection by automatically checking the customized properties.

III. Runtime Proof-Carrying Hardware

In this paper, we give a solution for hardware runtime formal verification of security properties. The proposed runtime PCH framework integrates a static program analysis method and a hardware based SMT solver, and provides a high-level protection by verifying security properties defined by users.

In detail, a trusted circuit is designed and manufactured by a trusted foundry to verify the trustworthiness of the untrusted hardware in runtime. Similar to [8], in our proposed new PCH framework, the untrusted circuit from the third-party foundry is called $Prover$, while the trusted circuit is called $Verifier$ as shown in Figure 1. If the verification of the security properties/theorems is successful, it indicates that the $Prover$ is trustworthy. Further, $Verifier$ can get all the information from $Prover$. In the case where the verification fails, the $Verifier$ can disable the $Prover$ at anytime.

There are mainly two entities - untrusted foundry and trusted integrator interacting in the developed framework (see Figure 1). At first, the untrusted foundry gets requirements of ASICs from consumer, and then fabricates the chips as part of $Prover$ depending on the functionality specifications, which is golden model in Figure 1. The other part of $Prover$ produces a conjunctive normal form (CNF), which is a combination of proof and secure specifications. The CNF will be delivered from $Prover$ to the solver, and satisfaction of the CNF will be checked. If satisfied, then the execution of circuit will be continue. If the given CNF is unsolved, then the $Verifier$ will lock the circuit. Accordingly, the trusted integrator, on the side of consumer, designs an extra trusted circuit $Verifier$ that can provide verification of $Prover$ on runtime and then combine $Verifier$ and $Prover$ together to produce the runtime verification system S. The composition of the final system S can be presented as Equation (1).

$$S \coloneqq P \wedge V \tag{1}$$

Further, the trusted integrator explores execution paths from static program analysis of the functional golden model written by hardware description language (HDL) like Verilog. In the untrusted foundry side, each execution path will be manufactured individually, and we call them individual circuit segment, marked as seg. So we define the functionality of circuits inside the P as F and then F is composed of many seg as shown in Equation (2), where $k \in$ Z is the total number of segments.

$$F \coloneqq seg_1 \wedge seg_2 \wedge \cdots \wedge seg_k \tag{2}$$

Correspondingly, security property, defined as $Prop$, would be given by the integrator and then decomposed into sub security properties, defined as $lemma$. In $Verifier$ side, satisfaction of each sub property $lemma$ will be verified for the corresponding segment seg as shown in Figure 2. So the system level security property $Prop$ is constructed as Equation (3).

$$Prop \coloneqq lemma_1 \wedge lemma_2 \wedge \cdots \wedge lemma_k \tag{3}$$

Along with the F, untrusted foundry requires to give proof to satisfy $lemma$ for each seg, and the proof is given in form of CNF, defined as cnf_{seg} in Equation (4) where $n \in$ Z stands for index number of a list, $Tseitin$ is a transformation that converts boolean circuits to CNF [11].

$$seg_n \xrightarrow{Tseitin} cnf_{segn} \tag{4}$$

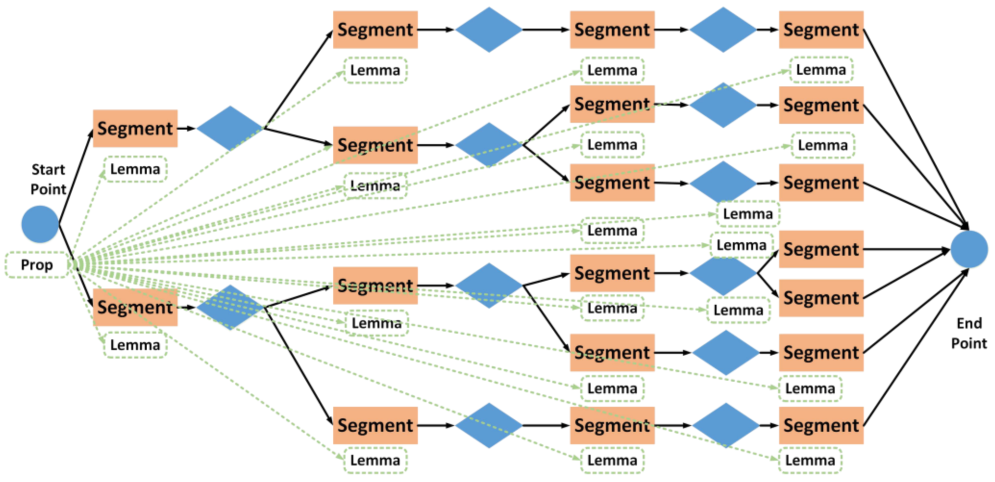

Fig. 2: Circuit segments and property decomposition

Meanwhile, $lemma$ need to be parsed to a hardware expression $lemma_{expr}$ that can be represented by using HDL. In our proposed framework, parsing is made manually in the foundry side. After that, a $Tseitin$ transformation is utilized to convert the $lemma_{expr}$ to a CNF, noted as cnf_{lan}. The procedure is presented in Equation (5).

$$lemma_n \xrightarrow{parse} lemma_{exprn} \xrightarrow{Tseitin} cnf_{lan} \tag{5}$$

Therefore, proof of sub property for segment is defined as a conjunction of cnf_{seq} and cnf_{lan} as shown in Equation (6). Furthermore, the entire proof in system level, noted as CNF, is composed of all the distributed cnf_n as described in Equation (7).

$$cnf_n \coloneqq cnf_{seg} \wedge cnf_{lan} \tag{6}$$

$$CNF \coloneqq cnf_1 \wedge cnf_2 \wedge \cdots \wedge cnf_k \tag{7}$$

Finally, in the following Equation (8), $Prover$ is constructed from functionality part F and proof part CNF. In the runtime verification process, cnf_n would be put into the DPLL SAT solver and verified individually. The verification details will be discussed in the following part.

$$P \coloneqq F \wedge CNF \tag{8}$$

Except the segment and CNF block, the rest part of Figure 3 depicts the design of the $Verifier$ which comprises a LUT and a DPLL SAT solver. The LUT in the proposed framework records information that whether the segment has been verified or not. The LUT includes two columns, where the first column contains a segment list and the second column has a binary value for each segment i.e. *1* stands for verified, *0* stands for not verified. Before the execution of a segment, the corresponding value will be checked. If the segment has been verified, then the execution continues. Otherwise, the system will be stalled and the verification of the segment is performed first.

A DPLL SAT solver is implemented based on Algorithm

Algorithm 1 DPLL Algorithm

```
Input:
 1: $F$                                          ▷ A CNF formula.
Output: $Result$ ▷ A Boolean value where $True$ stands for
     satisfaction and $False$ stands for not-satisfaction.
 2: Preprocess $F$;
 3: if $F == False$ then
 4:     $Result \leftarrow False$; return;
 5: end if
 6: Find the next unassigned variable, assign the value;
 7: Deduce based on the assignment;
 8: if $F == False$ then
 9:     $Result \leftarrow False$; return;
10: end if
11: if The conflict happened in derivation then
12:     Analyze the conflict
13:     if $F$ can be looked back upon then
14:         look back upon
15:     else
16:         $Result \leftarrow False$; return;
17:     end if
18: else
19:     return to line 6.
20: end if
```

1. A typical existing SMT solver is constructed based on the SAT solver, which is shown in the Figure 4. Specifically, in the proposed framework, the SMT solver is developed to get the extra constrains from a high level, while the CNF is input to the SAT solver directly. In the verification, Proof cnf_n is delivered from $Prover$ to the solver, and satisfaction of the input cnf_n will be checked. If satisfied, then the relevant value in LUT table will be updated as *1*. If the given cnf_n is unsolved, then the $Verifier$ will lock the segment by using an AND gate.

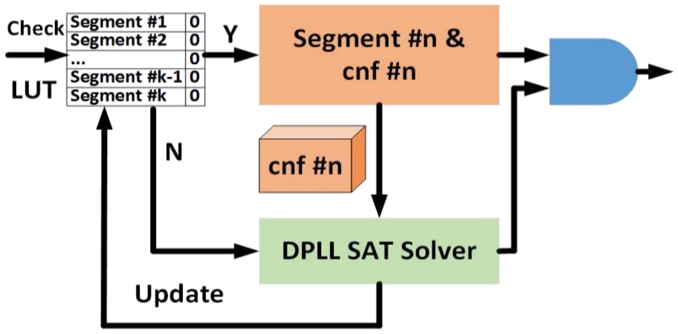

Fig. 3: Structure of *Verifier*

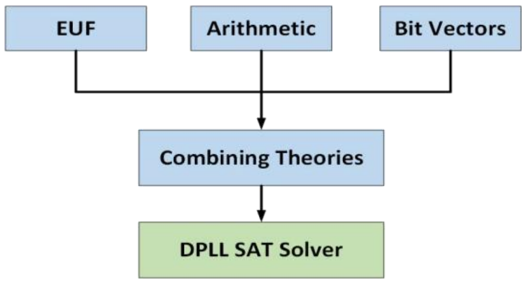

Fig. 4: Hardware based SMT solver structure

IV. Case Study

To demonstrate the effectiveness of the proposed runtime verification framework supported by the SMT solver, we utilize a FPGA platform implementing a RS232 program. Specifically, the RS232-T100, written in Verilog, is selected as the benchmark and obtained from [12]. The receiver side of this RS232, a micro-UART core, is considered for verification. In order to prove the presence/absence of hardware Trojan, we will check the important signals like in/out interfaces.

In this experiment, we consider a hardware Trojan embedded in the benchmark RS232-T100, which manipulates output data to cause the Denial-of-Service (DoS) attack. Trigger of this Trojan is detecting specific values among the control signals and output signal in the receiver part of the micro-UART core. Once the Trojan is triggered, the payload of this Trojan can stuck the output signals and as zeros.

In the above case, an example security property is formalized below:

$$\forall t \nexists t_0, t_n \in t: (t_0 < t_n) \wedge (t_0 - t_n > V_{th}) \wedge$$
$$\left(state_{t_0 \to t_n} = V_{wait}\right) \wedge \left(rec\_dataH_{t_0 \to t_n} = 0x00\right)$$

Here, t is the time parameter, $state$ means the current state of the RS232 system. rec_dataH is the output port with 8 bits length of the receiver part. Also, $V_{th\in Z}$ is the threshold that we set for the time interval. V_{wait} is a specific binary vector with value is *3'b011* which implies that the system is waiting for sampling in data transmission. The $lemma$ states that if output port generates zero values in too long consecutive time during data transmission, then there is a high risk of under DOS attack.

As a result, the SAT solver (kernel of the proposed SMT solver) took *4668406745* clock cycles or *9sec* (*2ns* per clock cycles based on our configuration) for returning an unsatisfaction conclusion for the proof/CNF of initial assignments segment, which indeed contains the Trojan. Meanwhile, the SAT solver took *7873* clock cycles or *15ms* for returning a satisfaction conclusion for the same segment without Trojan.

V. Conclusion

In this paper, we give a solution for hardware runtime formal verification of security properties. The proposed runtime PCH framework integrates a static program analysis method and a SMT solver, and provides a high-level protection by verifying security properties defined by users. The proposed method was demonstrated using FPGA and evaluated by verifying a RS232 benchmark with an embedded Trojan. Consequently, the proposed approach guarantees the security of hardware in runtime.

Acknowledgement

This work was partially supported by National Science Foundation (CNS-1812071), Army Research Office (W911NF-17-1-0477) and Cisco.

References

[1] X. Zhang and M. Tehranipoor, “Case study: Detecting hardware trojans in third-party digital ip cores,” in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, 2011, pp. 67–70.
[2] 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.
[3] 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.
[4] Y. Jin, “Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits,” in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2014, pp. 19–24.
[5] INRIA, “The coq proof assistant,” 2010, `http://coq.inria.fr/`.
[6] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, “Pre-silicon security verification and validation: A formal perspective,” in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC ’15, 2015, pp. 145:1–145:6.
[7] X. Guo, R. G. Dutta, J. He, and Y. Jin, “PCH framework for ip runtime security verification,” in *Asian Hardware Oriented Security and Trust (AsianHOST)*, 2017, (to appear).
[8] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, “Verifiable asics,” in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 759–778.
[9] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.
[10] R. Baldoni, E. Coppa, D. C. D’Elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *arXiv preprint arXiv:1610.00502*, 2016.
[11] G. Tseitin, “On the complexity ofderivation in propositional calculus,” *Studies in Constrained Mathematics and Mathematical Logic*, 1968.
[12] *https://www.trust-hub.org/*.