PCH Framework for IP Runtime Security Verification

Xiaolong Guo*, Raj Gautam Dutta‡, Jiaji He†, and Yier Jin*
*Department of Electrical and Computer Engineering, University of Florida
†School of Microelectronics, Tianjin University
‡Department of Electrical and Computer Engineering, University of Central Florida
guoxiaolong@ufl.edu, rajgautamdutta@knights.ucf.edu, dochejj@tju.edu.cn, 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 SAT solving methods to validate the user defined properties. The proposed framework has been demonstrated on an FPGA platform using an 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. Consequently, security researchers in their respective domains have started putting in considerable effort to ensure trustworthiness of third-party resources. In the hardware security industry, multiple countermeasures have been developed to protect SoC at pre- and post-silicon stages [1]–[12]. However, these methods are designed to protect the hardware only in certain scenarios [13]. 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 [3]–[6], but few of them were designed for securing post-fabrication designs. For example, in [4]–[6], the proof-carrying hardware (PCH) framework was used to verify security properties of soft IP cores. Supported by the Coq proof assistant [14], 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. Recently, Verifiable ASICs was proposed in [15] where an encryption protocol based approach was used against hardware Trojans in circuit manufacturing. However, their goal of ensuring correctness of computation incurred high costs in terms of complexity and overhead in hardware.

In this paper, we address the runtime hardware security verification challenge by extending our PCH framework from static to dynamic (aka runtime) with Satisfiability (SAT) solvers and symbolic executions.

SAT solvers have been used in many electronic design automation fields like logic synthesis, verification, and testing [16]–[18]. The SAT solvers are originally designed to solve the well-known Boolean Satisfiability problem, which decides whether a propositional logic formula can be satisfied given value assignments of the variables in the formula [19]. However, due to the high computational complexity, SAT solvers 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 [20]. 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.

The main contributions of this paper are as follows.

- We combine the SAT solver with a static program analysis method for runtime checking of security of hardware. It is the first attempt to verify security properties on runtime hardware by combining these techniques.
- We describe the method for decomposing the hardware execution and the security specification into execution paths and sub-specifications respectively. These execution paths and sub-specifications are verified using a SAT solver.
- The work improves the study of hardware runtime verification. Our enhanced PCH framework provide comprehensive protection of hardware by complying to user specified security properties.

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 SAT solver and static program analysis. We explain our integrated framework, formalization, and decomposition of the hardware and elaborate on the verification procedure in section IV.

Section V presents demonstrations of our approach and final conclusions are drawn in Section VI.

II. Related Work

Formal methods have been extensively used for verification and validation of security properties at pre- and post-silicon stages [3]–[9], [21]. In [3], 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 [4]–[6], [8], [9]. This method was inspired from the proof-carrying code (PCC) for software assurance [22]. Drzevitzky et al. proposed the first PCH framework for dynamically reconfigurable hardware platforms [9]. 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 [4]–[6], [8], [23], [24]. In this framework, the Coq proof assistant [14] was used to represent security properties, hardware designs, and formal proofs. However, this framework can only provides static verification on design stage of hardware other than the runtime of hardware.

Recently, Verifiable ASICs was proposed by Wahby et.al. [15] 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 security 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. In our paper, we follow the *Prover-Verifier* architecture to build the framework.

Meanwhile, many runtime hardware approaches were developed for information flow security, which could guarantee that all information flows satisfy the given security policy. For instance, GLIFT was proposed in [25] and could dynamically detect malicious logic through tracking the information flow in the hardware at runtime. Moreover, there are hardware description languages that enforce security policies by adding logic of information flow control in the hardware. Languages such as Caisson [26], Sapper [27] and SecVerilog [13] belong to this category. However, these information flow control based techniques could provide protections which only against information leakage. Our proposed runtime PCH framework can against any user specified security properties by verifying hardware designs.

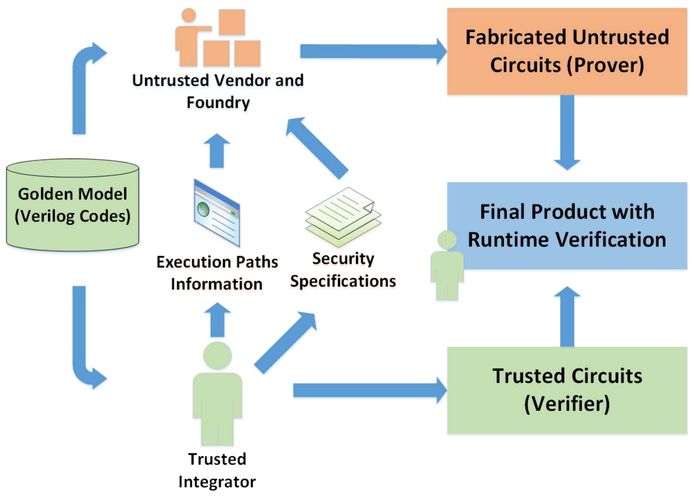
Untrusted Vendor and Foundry
Fabricated Untrusted Circuits (Prover)
Golden Model (Verilog Codes)
Execution Paths Information
Security Specifications
Final Product with Runtime Verification
Trusted Circuits (Verifier)
Trusted Integrator

Figure 1: Working procedure of runtime PCH framework

III. Background

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. Execution Paths

Symbolic execution is a popular static program analysis technique that checks whether a software program satisfies specified properties. 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. In [20], a survey is provided to show many early works on symbolic execution and its applications on software testing and security.

As to the execution paths of the hardware, a path dependency graph of Verilog-HDL (Verilog) programs was proposed in [28], which could be applied for the static Verilog program analysis. In our work, after obtaining the execution paths from the golden model, the hardware designed is decomposed into segments based on these paths. Then, the foundry manufactures the ICs depending on these segments. Accordingly, the mentioned Boolean formula will be maintained for each segment and implemented in hardware in the form of the look-up table (LUT).

C. Hardware Based SAT Solver

The SAT solvers are used in a wide range of applications such as model checking of hardware and software, circuit synthesis, and testing [19]. However, due to the excessive computation time, SAT solvers are often impractical for emerging applications [29]. Early works on hardware accelerated SAT solvers was surveyed in [30]. Solvers based on FPGA [29] and GPU [31] were discussed, and all of these hardware accelerated solvers had relative limitations. As we perform decomposition of the design, the scalability of SAT solver is not an issue.

In hardware verification using SAT solvers, a circuit is first represented in conjunctive normal form (CNF). A CNF is a conjunction of many clauses and each clause is a disjunction of literals which include variables and their logical negations [32]. SAT solver is designed to figure out the satisfaction of the given CNF, which means that all clauses must get the value *True*. Furthermore, there is at least one literal that gets *True* for each clause. For most of the modern SAT solvers, Davis-Putnam-Logemann-Loveland (DPLL), proposed in [33], is applied as the kernel algorithm. In DPLL, a depth-first search (DFS) is carried out to traverse all possible variables/literals assignments as shown in Algorithm 1. In this paper, a DPLL based SAT solver is implemented in hardware in the proposed runtime PCH framework.

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
```

IV. Runtime Proof-Carrying Hardware

This section introduces the runtime PCH framework and provides details of the design. A *Prover* is fabricated based on the execution paths, while a DPLL SAT solver is implemented as *Verifier*. In this section, we also discuss the decomposition of security properties and distribution of proofs, which enable the SAT solver to be used for large scale application.

A. Framework Overview and Property Decomposition

A trusted circuit is designed and manufactured by a trusted foundry to verify the trustworthiness of the untrusted hardware in runtime. Similar to [15], in our proposed new PCH framework, the untrusted circuit from the third-party foundry is called *Prover* P, while the trusted circuit is called *Verifier* V 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.

The working procedure of our proposed framework is shown in Figure. 1, and there are mainly two entities - untrusted foundry and trusted integrator interacting in the framework. Similar to the setting in [15], 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* is from security specifications, which will be introduced in details later in this section. 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).

$$S := P \wedge V \tag{1}$$

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 $k \in \mathbb{Z}$ is the total number of segments.

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

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).

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

B. Distributed Proof-Carrying in Runtime

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 $n \in \mathbb{Z}$ stands for index number of a list, $Tseitin$ is a transformation that converts boolean circuits to CNF [34].

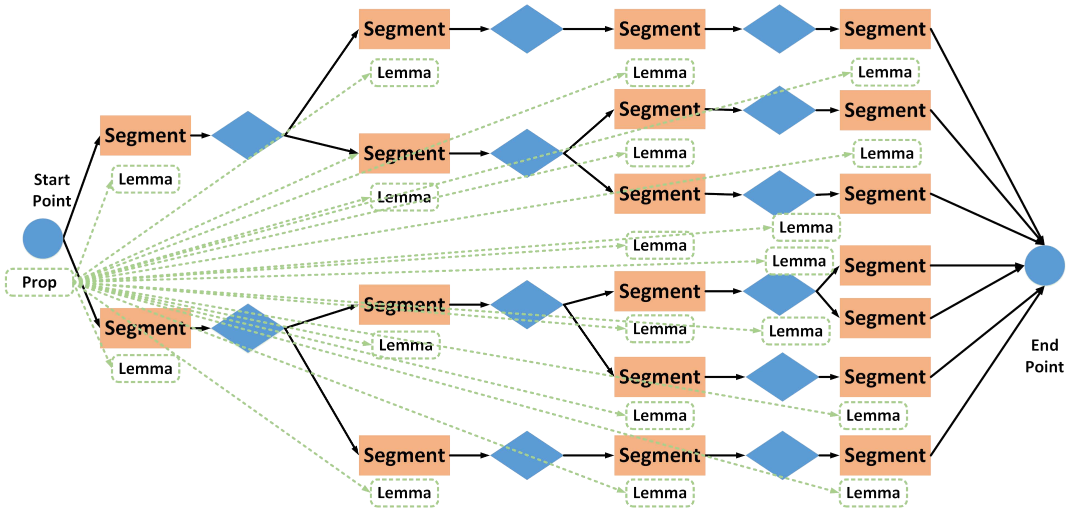

Figure 2: Circuit segments and property decomposition

$$seg_n \overset{Tseitin}{\longrightarrow} cnf_{segn} \quad (4)$$

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_{la}. The procedure is presented in Equation (5).

$$lemma_n \overset{parse}{\longrightarrow} lemma_{exprn} \overset{Tseitin}{\longrightarrow} cnf_{lan} \quad (5)$$

Therefore, proof of sub property for segment is defined as a conjunction of cnf_{seg} 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 discribed in Equation (7).

$$cnf_n := cnf_{seg} \wedge cnf_{lan} \quad (6)$$

$$CNF := cnf_1 \wedge cnf_2 \wedge \cdots \wedge cnf_k \quad (7)$$

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 sovler and verified individually. The verification details will be discussed in the following part.

$$P := F \wedge CNF \quad (8)$$

C. Design of Verifier and Runtime Verification Process

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.

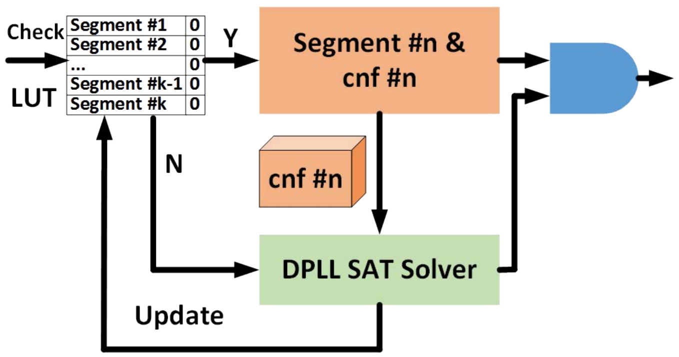

Figure 3: Structure of *Verifier*

A DPLL SAT solver is implemented based on Algorithm 1. 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. From the system viewpoint, the above runtime verification process is represented in Algorithm 2.

V. Case Study

To demonstrate the effectiveness of the proposed runtime verification framework supported by the SAT 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 [35]. The receiver side of this RS232, a micro-UART core, is considered for verification (see Figure 3). 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

Algorithm 2 Runtime Verification Process

```
Input:
 1: $P$                                              ▷ Prover
 2: $V$                                              ▷ Verifier
Output: $null$
 3: $list_{next}, list_{cnf}$;
 4: $ExePaths \leftarrow P$                  ▷ Get all the execution paths
 5: $SAT() \leftarrow V$                          ▷ Get the SAT solver
 6: $checkTable() \leftarrow V$                ▷ Get the look-up table
 7: $list_{next} \leftarrow checkPath(ExePaths)$;     ▷ Get the next
    execution paths
 8: $list_{cnf} \leftarrow checkTable(list_{next})$;   ▷ Whether the next
    execution paths are verified
 9: if $list_{cnf}$ == null then          ▷ All next execution paths
    verified
10:     Go to line 7;
11: else
12:     For each $cnf$ in $list_{cnf}$:
13:     $SAT(cnf)$;
14:     if All $list_{cnf}$ have solutions then
15:         Go to line 7;
16:     else
17:         Lock the circuit, return;
18:     end if
19: end if
```

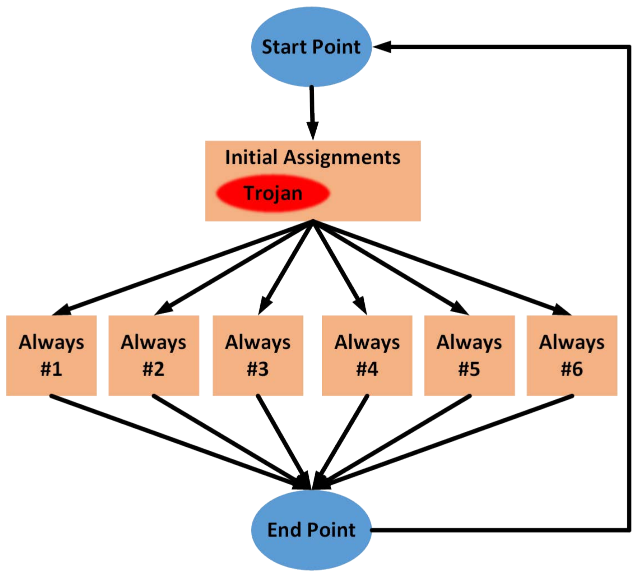

Figure 4: Decomposition of receiver part in RS232-T100

this Trojan is detecting specific values among the control signals $state$, $bitCell\_cntrH$, $recd\_bitCntrH$ and output signal rec_dataH in the receiver part of the micro-UART core. Once the Trojan is triggered, the payload of this Trojan can stuck the output signals rec_dataH and rec_readyH as zeros.

To detect such a DoS vulnerability, we observe the output signal in consecutive time. A heuristic property is that the output data should not always be 0 during data transmission. In our case study, we decompose and formalize the above heuristic property into $lemmas$ depending on the segments of the benchmark. Figure 4 indicates the execution paths of the receiver in micro-UART core and segments were set based on the paths. Similar to the Verilog program control flow graph in [28], the circuit is divided into an initial assignment part followed by parallel always parts as shown in Figure 4. Additionally, for RS232-T100 benchmark, Trojan is embedded into the initial assignment part.

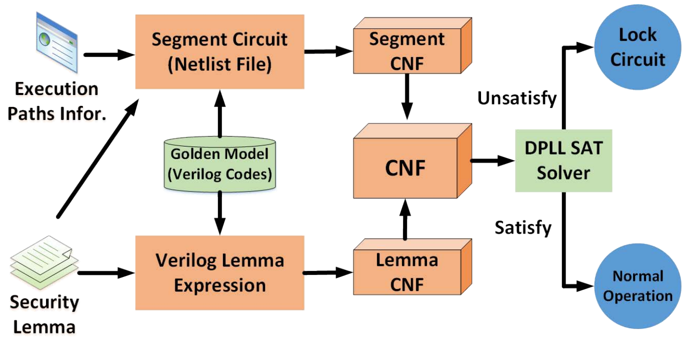

Figure 5: Working procedure of verification of a segment

Then, depending on the working procedure in Figure 5, verification is carried out on the segment which is going to be executed. In the FPGA platform, gate level netlist file is used to simulate the ASICs. The netlist file of each segment is designed and synthesized by considering both golden model and security $lemma$. Proof of the segment is in the form of CNF, and part of the CNF is from the the netlist file. Meanwhile, security $lemma$ is translated to the expressions written in Verilog. After that the expressions are converted to the rest of the CNF. Finally, a DPLL SAT solver, implemented in FPGA, is applied to check whether the solution of the input CNF exists. If the CNF is satisfied, then SAT solver return a signal to continue the execution or the system will stall.

In our case, an example security $lemma$ for the initial assignments segment in Figure 4 is formalized below:

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

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, $V_{th \in \mathbb{Z}}$ 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.

When the SAT solver gets a solution from the given CNF, we can be assured that the proof of the $lemma$ exist. Otherwise, system will be locked for protection. In the above case, the SAT solver we developed 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.

VI. 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 SAT solver, and provides a high-level protection by verifying security properties defined by users. Furthermore, decomposition of property and distributed proofs of segments significantly reduces the computation complexity undertaken by SAT solver. 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.

In future, we plan to use our approach for protecting large scale designs such as processor. Also, automated tool will be developed such as for auto generation of CNF. Furthermore, more sophisticated Trojans or attacks will be tested in different benchmarks using our runtime verification framework.

Acknowledgement

This work was partially supported by National Science Foundation (CNS-1319105), Army Research Office (W911NF-17-1-0477), Cisco and National Natural Science Foundation of China (61376032).

References

[1] 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.

[2] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, 2013, pp. 697–708.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 255–258.

[10] S. Wei, S. Meguerdichian, and M. Potkonjak, "Malicious circuitry detection using thermal conditioning," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1136–1145, 2011.

[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*, ser. Lecture Notes in Computer Science, vol. 5747, 2009, pp. 396–410.

[12] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 49–63.

[13] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 503–516.

[14] INRIA, "The coq proof assistant," 2010, `http://coq.inria.fr/`.

[15] 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.

[16] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[17] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.

[18] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[19] J. Marques-Silva, "Practical applications of boolean satisfiability," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*. IEEE, 2008, pp. 74–80.

[20] 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.

[21] 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.

[22] 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.

[23] X. Guo, R. G. Dutta, and Y. Jin, "Eliminating the hardware-software boundary: A proof-carrying approach for trust evaluation on computer systems," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 12, no. 2, pp. 405–417, 2017.

[24] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable soc trust verification using integrated theorem proving and model checking," in *IEEE Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 124–129.

[25] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 109–120.

[26] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 109–120.

[27] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 97–112.

[28] M. Zaki, Y. Mokhtari, and S. Tahar, "A path dependency graph for verilog program analysis," in *Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03), Montreal*, 2003, pp. 109–112.

[29] J. Thong and N. Nicolici, "Sat solving using fpga-based heterogeneous computing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 232–239.

[30] A. A. Sohanghpurwala, M. W. Hassan, and P. Athanas, "Hardware accelerated sat solvers—a survey," *Journal of Parallel and Distributed Computing*, vol. 106, pp. 170–184, 2017.

[31] A. Dal Palu, A. Dovier, A. Formisano, and E. Pontelli, "Cud@ sat: Sat solving on gpus," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 27, no. 3, pp. 293–316, 2015.

[32] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[33] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[34] G. Tseitin, "On the complexity ofderivation in propositional calculus," *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[35] Trust-Hub, "Benchmarks," [Online]. https://www.trust-hub.org/benchmarks.php.