A Formal Framework for Gate-Level Information Leakage Using Z3

Qizhi Zhang*, Jiaji He†, Yiqiang Zhao* and Xiaolong Guo‡
*Tianjin University, †Tsinghua University, ‡Kansas State University
qizhi_zhang@tju.edu.cn, jiaji_he@mail.tsinghua.edu.cn, yq_zhao@tju.edu.cn, guoxiaolong@ksu.edu

***Abstract*—The hardware intellectual property (IP) cores from untrusted vendors are widely used, which has raised security concerns for system designers. Although formal methods provide powerful solutions for detecting malicious behaviors in hardware, the participation of manual work prevents the methods from practical applications. Information Flow Tracking (IFT) is a powerful approach to prevent sensitive information leakage. However, existing IFT solutions are either introducing overhead in hardware or lacking practical automatic working procedures. To alleviate these challenges, we propose a framework that fully automates information leakage detection in the gate level of hardware. This framework introduces Z3, an SMT solver, in checking the violation of the confidentiality automatically. On the other hand, a parser converting the gate-level hardware to the formal model is developed to further remove the manual workload. To validate the effectiveness, the proposed solution is tested on 11 gate-level netlist benchmarks. The Trojans leaking information from circuit outputs can be automatically detected. We also account for time consumption during the whole working procedure to show the efficiency of the proposed approach.**

I. Introduction

The demand for intellectual property (IP) cores has been significantly increased because of the changing landscape of the semiconductor industry. The proliferation of the IP market is affected by various factors like lowered design cost, shortened time-to-market (TTM), etc. In the meantime, the credibility of 3rd-party vendors is threatened by the hardware Trojan and design flaws, which also places high-security uncertainties on the IP end-users and customers. In a system-on-chip (SoC), a malicious IP core can bypass many existing hardware Trojan detection methods [1], [2].

In detecting hardware Trojans and vulnerabilities, formal methods have been most effective among all the existing techniques [3], [4], [5], [6], [7], [8], [9], [10], [11]. However, very few of current formal verification approaches are scalable and practical for hardware Trojan detection in the industry because of lacking automatic and efficient tools. For instance, model checking is a popular used malicious logic detection method for protecting third-party IP cores [10]. In the model checking, security properties are formalized as traces and all possible traces generated by the system are checked. The system is treated to match against the security properties is all the traces pass the checking [12]. However, checking the very large system, the model checker always runs into the state space explosion issue.

Information flow tracking (IFT) [13] is a scalable approach for detecting leakage/sneaky path of sensitive information. In the IFT, data or operations are assigned by labels standing for the trust levels. Rely on the information flow policy, the labels are propagated or updated to other data. In general, data with labels are accessed or propagated to the trust portion in the system. Some IFT based solutions on assuring hardware security are proposed like SecVerilog [14], [15], Caisson [16], Sapper [17], QIF-Verilog[18], GLIFT[19], etc. However, there is lacking IFT solution in detecting sneaky paths in the gate-level netlist. The theorem, Coq, is utilized in proving the gate-level information flow property in [20]. However, as a theorem proving method, a significant manual effort is required for constructing machine proofs. SecChisel [21] applies an automated formal verification checking using the Z3 solver [22], while the scope of the SecChisel is only on protecting high-level synthesis.

To solve those problems, we propose a framework for formalizing and checking gate-level hardware design for security purposes. In the framework, gate-level netlist data files are parsed to the formal model in the form of constraints at first. Then sensitive labels are introduced to denote secretes in the hardware design. In the end, if outputs are tainted by the labels, the information leakage is detected. Satisfiability modulo theories (SMT) solver is utilized as the checking engine to propagate the information flow and automatically check the IFT policies.

The main contributions of this paper are as follows.

- We introduce an automated formal verification framework detecting vulnerability in the gate-level netlist. The netlist data is formalized to a circuit model, based on which the security property is designed. The confidentiality is enforced to the input hardware design by applying an automatic checking engine.
- GLIFT is, for the first time, statically applied in the gate-level hardware with a fully automated working procedure. The information leakage caused by Trojans is addressed by tracking sensitive information.
- We deliver the toolchain in demonstrating the framework. It includes a parser and a Z3 SMT Solver which automate the formalization stage and property checking stage of the formal verification, respectively.

The rest of the paper is organized as follows. In Section II, we introduce the threat model and discuss previous work on malicious logic detection using IFT based solutions and then present a gate-level IFT model. We explain our automated framework involving the SMT solver and code parser in Section III. Section IV presents demonstrations of our approach by testing 22 gate-level netlist benchmarks with hardware Trojan. The limitations of this work are discussed in Section V. Finally, conclusions and future works are drawn in Section VI.

II. Background

We introduce the attack model in this section at first. Then IFT based solutions are investigated in the hardware security area. The gate-level IFT model is explained with the example, based on which the formal model in the proposed framework is built.

A. Attack Model

In this paper, we assume that information leakage paths are created by either intended hardware Trojan or unintended design errors. Malicious logic can be inserted by an adversary at the design or testing stage of the supply chain. We assume that the rogue agent at the third-party IP vendor can access the RTL design or the gate-level netlist files and then insert a hardware Trojan or backdoor to create a sneaky path of the design. Lacking security knowledge, the hardware developers could produce vulnerabilities, like leakage paths, in the design stage. On the other hand, we assume that attackers are capable of accessing inputs and outputs ports of the manufactured hardware and have knowledge of the hardware functionality. Therefore, by triggering the Trojan or observing the input-output patterns, the attacker can exploit such information leakage paths to infer the sensitive/secrete information of the design. Such secrete information could be encryption keys from the hardware.

B. Related Work

Recently, IFT based security approaches for protecting confidentiality are delivered in the form of a language-based solution. Caisson [16] and Sapper [17] realize IFT isolation and separation properties and in the synthesized secure circuits. Using Caisson or Sapper, the designer labels wires and registers, which are duplicated in the generated hardware. Considerable hardware overheads are caused at the circuit level. Detecting information leakage in the compilation stage, SecVerilog avoids the hardware overheads [15]. It extends the type system of standard Verilog to enforce noninterference in the design. However, a significant complex security label system is introduced by SecVerilog to increase precision. Only with sufficient knowledge of security, the circuit designers can specify information flow policies in SecVerilog. In contrast, QIF-Verilog only extends one simple security label from the standard Verilog to reduce the cost of learning from the developers' side [18]. It quantifies the information leakage by applying the quantitative information flow tracking in the design stage. However, the QIF-Verilog is not capable of supporting IFT analysis in the gate-level netlist.

In [23], GLIFT is proposed to detect malicious logic by tracking the information flow in the runtime hardware. It models logic gates and labels individual bit at the gate-level. The information flow propagation logic is realized in hardware along with the original functional circuit [24]. Again, it introduces huge hardware overheads. A static GLIFT approach is proposed in [20] which checks security property in the gate-level netlist. It translates the property and the netlist to theorems and formal circuits, respectively. Then theorem proving is utilized to prove the satisfaction of the property against the formal circuit. Using an interactive proving approach, developers must manually construct the proofs, which increases the time required for certifying large hardware design. SecChisel is proposed in [21] to check the confidentiality and integrity of hardware design automatically using the SMT solver. Based on the Chisel hardware construction language, the SecChisel verification framework converts a higher level hardware description to the intermediate representations, FIRRTL representations, and then parses them to Z3 inputs. The information flow checking is performed in the end. Although the framework checks the IFT property automatically, it focuses on the high-level synthesis procedure rather than the gate-level netlist. Also, Chisel is not a widely used development language.

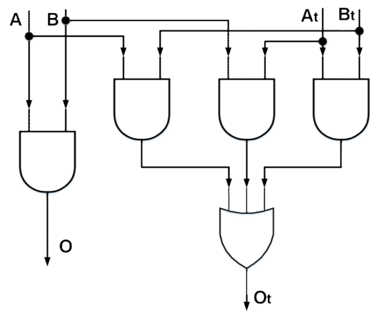

Figure 1: The AND-2 gate-level IFT model [19].

C. Modeling Gate-Level IFT

An advantage of GLIFT is that each data bit is associated with a security label, which propagates the labels more precise and reduces the false-positive rates [19]. As an example, in the Equation (1), we perform *and* operation between the secret signal and a 32-bits zero vector, then output the result.

$$Output := AND - 2(Secret, 0x00) \quad (1)$$

where **AND-2** function performs as a 32 bits two-inputs AND operation and *Secret* has been labelled as high sensitive. In the traditional IFT approach, the sensitive label would be propagated to the output port and then detected as information leakage. However, as the other signal involved in the *and* operation is a zero vector, no secret is leaked through such an operation, which causes a false-positive.

In the GLIFT, both signal value and security labels are taken into consideration during the label propagation. Rather than tracking the data flow in the original design only, how the output is influenced by input value must be accounted for. To achieve this goal, extra logic gates are created to represent

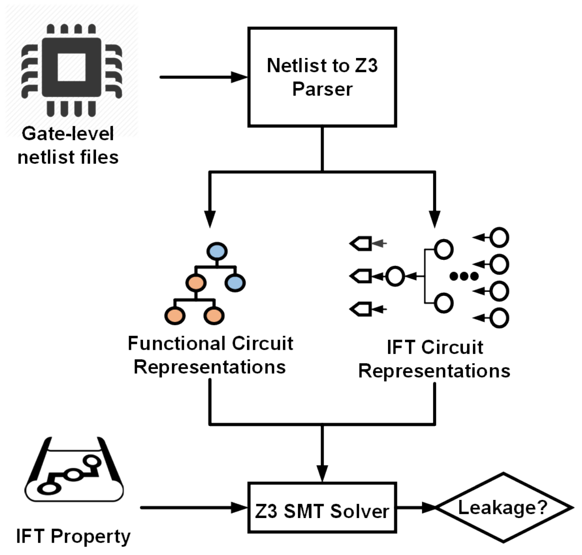

Figure 2: Working procedure of the proposed formal framework.

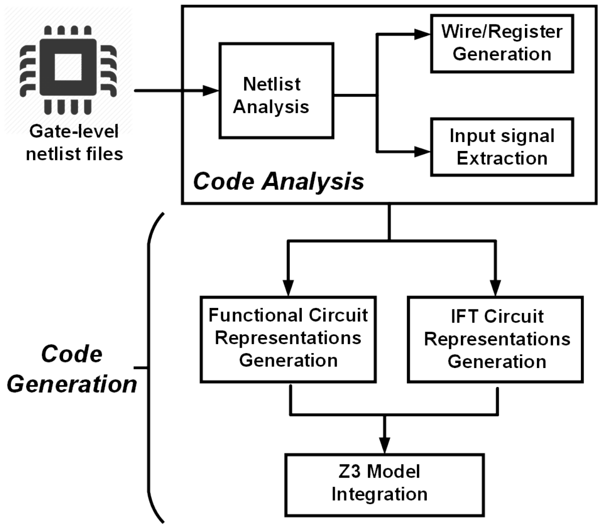

Figure 3: Block structure of the developed parser.

the influence along with the original circuit. We use the 1 bit two-input AND logic gate as the example. For each 1 bit two-input AND gate, the extra logic gates are inserted as shown in Figure 1. The A and B are 1 bit input while O is the 1 bit output. Accordingly, labels for A, B and O are denoted as A_t, B_t and O_t. Following the structure, once the low sensitive input is 0, the output label O_t keeps 0 not matter what the other high sensitive input value is. Only in the case that the low sensitive input is 1, the O_t is influenced by the high sensitive input, which means that the highly sensitive label has the potential to propagate to the output port.

D. SMT Solver

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 be satisfied 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 [22]. However, due to the high computational complexity, there is no hardware implementation for SMT solvers, and the software-based SMT solver is not scalable to large designs. From Microsoft, Z3 is a popular used SMT solver providing efficient verification and analysis applications [22]. It is assembled in the Python environment as Z3PY, which is a convenience for developing practical tools [25].

III. Methodology

The proposed framework automates the formal verification process by realizing IFT in the gate-level netlist design. It converts the whole hardware design to the Z3 constraints and adds extra logic for tracking the security labels. Label checking will be performed in the Z3 solver. Accordingly, we develop a parser to perform the netlist to Z3 program conversion and the extra logic generations.

A. The Formal Framework Overview

The working procedure of the proposed formal framework is shown in Figure 2. The gate-level netlist data is input to a parser. Then it converts the original hardware design to its formal equivalent representations, called the functional circuit representations F. In the meantime, the parser further generates extra logic gates to introduce and track security labels. We name those logic gates as the IFT circuit representations I. The IFT circuit representations are constructed as GLIFT logic. Both representations F and I are in the form of Z3 constrains. Hence we define the formal model M as Equation (2).

$$M := F \wedge I \tag{2}$$

Taking the logic gates in Figure 1 as an example, signal $\{A,\ B,\ O\}$ are composed to constrains in F while signal $\{A_t,B_t,O_t\}$ are composed to constrains in I. The corresponding procedure of deriving formal model M is shown as follows.

$$F := (O == A\&B)$$
$$I := (O_t == (A_t\&B_t)|(A_t\&B)|(A\&B_t))$$
$$\begin{aligned}M := F \wedge I := (O == A\&B)\wedge \\ (O_t == (A_t\&B_t)|(A_t\&B)|(A\&B_t))\end{aligned}$$

where $\&$ stands for the *and* operation and $|$ stands for the *or* operation.

Then next, M is input to the Z3 platform. In the meantime, IFT property, denoted as P, is also introduced to indicate the sensitive data bits and according to leakage output ports. Input to the Z3 solver, P is in the form of Z3 constraints as well. The final constrains C checked, in the end, is a conjunction of M

and P. Taking the above Figure 1 as one example, we assume that B is in high sensitivity while A is in low sensitivity. It leads to label value 1 in B_t and label value 0 in A_t. As O is the only port leaking secrets, O_t is set to label 1. We then derive the C as follows.

$$P := (A_t == 0) \wedge (B_t == 1) \wedge (O_t == 1)$$

$$\begin{aligned} C := P \wedge M := &(A_t == 0) \wedge (B_t == 1) \wedge (O_t == 1) \wedge \\ &(O == A \& B) \wedge \\ &(O_t == (A_t \& B_t) | (A_t \& B) | (A \& B_t)) \end{aligned}$$

Z3 SMT solver is then utilized to check C. If there is no solution, it means that whatever the inputs are, there is no way to propagate the high sensitive label to the output O_t. The design is extremely secure regarding the confidentiality property. Otherwise, by given the solution as input, the high sensitive label can be propagated to the output port and observed by the attacker. In our example, the solutions $\{A = 1, B = 0\}$ and $\{A = 1, B = 1\}$ are obtained by the Z3. Therefore, the design in Figure 1 has information leakage paths.

B. *Netlist to Z3 Parser Development*

As discussed above, the netlist data file needs to be converted to the Z3 constrains F. Also, the GLIFT logic gates I are generated depending on the hardware design. As such, we have developed an automatic parser for converting and generating Z3 constrains from gate-level netlist data. The parser is written in Python and has the structure as shown in Figure 3.

There are two parts in the parser – code analysis and code generation. In the code analysis part, netlist data is interpreted at first. We generate the wire and registers that are utilized in the functional circuit in this step. The input and output signals are extracted from those wires/registers for the property design and model integration in the following steps. Then in the code generation step, the functional circuit representations F and IFT circuit representations I are produced. In the end, rely on the extracted inputs and outputs, F and I are integrated and output to the Z3 solver.

IV. EXPERIMENT

To demonstrate the proposed Z3 based automated formal verification, we set up the experiment in the Python environment and evaluate IFT property in Verilog netlist benchmarks. Trojans are inserted into the genuine benchmarks, while properties are designed as adding labels in IFT circuit representations.

A. *Experiment Setup*

To use the proposed framework in practical applications, a developer only needs to indicate high sensitive bits in the IFT circuit representations' input signals and observable ports from the attackers. In our experiment, some specific data bits are treated as secrets and the confidentiality is checked for those labeled secrets.

The main tool we used for experimentation is Z3 SMT Solver, the automated theorem prover released by Microsoft Research. API of Z3 has been assembled in the Python environment as Z3PY. Then the Z3 solver is in the same environment as the parser, which makes the toolchain be integrated easily. We employ the Z3 to check if the tainted label of secret information can be delivered to the IFT circuit's output. All the demonstrations are executed in Windows 10 on a computing machine with Core(TM) i3-9100 CPU @3.60GHz and 8GB memory.

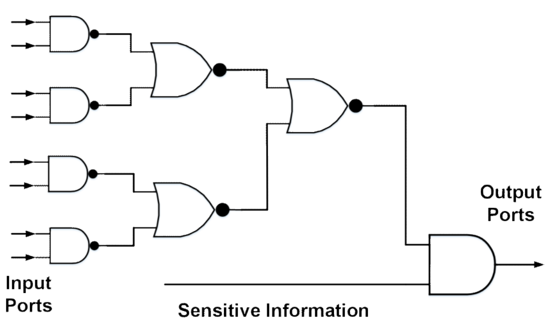

Figure 4: Design of inserted sneaky paths.

To demonstrate the practicality of our proposed framework, we evaluate 22 ISCAS'85 gate-level netlist benchmarks [26] from Github [27]. Those benchmarks are written by Verilog and have been synthesized with Cadence Genus. They provide combinational logic circuits to let users test different mythologies. For fitting to the attack model in this paper, we insert the leakage paths to simulate hardware Trojans into the design. Then, the parser translates the Trojan inserted benchmarks to the model in Z3, and the IFT logic of the benchmark is generated at the same time. After that, we establish the solver and add the constrains standing for IFT property. In the end, the model and property are checked together in the Z3 platform.

B. *Inserted Leakage Paths and Checked Properties*

In Figure 4, we show a template of inserted hardware Trojan design. All the Trojans in our experiments follow such kinds of structures and leak information of the circuit. It is a piece of the combinational circuit and composed by AND, NAND, as well as NOR gates. The trigger of the Trojan is connected with the input ports and would be activated by a specific input pattern. The payload of the Trojan enables an AND gate and passes the sensitive information to the output ports.

We specify one data bit in each benchmark input as the secret and set them according to label as high. Several output bits which are influenced by the Trojan are defined as the vulnerable output ports. The security property is represented as "Assigning the high sensitive label to a secret and low sensitive labels to the rest signals, whether there exists at least one solution causing high sensitive label appeared on vulnerable output ports." In other words, if the Z3 finds a solution, then the Trojan is detected. As a counter-example, the solution is the input vector that propagates secrets to vulnerable ports.

Benchmarks	Trigger Mode	Functional Gate	IFT Gate	Model Time (ms)	Detection Time (ms)	Total (ms)	Detected
c17	signal trigger	6	24	70	24	94	Yes
c432	signal trigger	160	775	77	143	220	Yes
c499	signal trigger	202	888	65	130	195	Yes
c880	signal trigger	383	1455	63	210	273	Yes
c1355	signal trigger	546	3315	74	275	349	Yes
c1908	signal trigger	880	2168	74	250	324	Yes
c2670	signal trigger	1193	3290	81	509	590	Yes
c3540	signal trigger	1669	4638	91	663	754	Yes
c5315	signal trigger	2307	7995	116	1112	1228	Yes
c6288	signal trigger	2416	9364	134	1085	1219	Yes
c7552	signal trigger	3512	9809	138	1361	1499	Yes
bar	signal trigger	2960	15271	1391	2553	3944	Yes
max	signal trigger	5065	14216	4486	4486	8972	Yes
sin	signal trigger	7656	25970	2634	2323	4957	Yes
arbiter	signal trigger	23189	59600	4605	7087	11692	Yes
voter	signal trigger	25993	69436	9437	8805	18242	Yes
square	signal trigger	35264	91406	7976	12017	19993	Yes
sqrt	signal trigger	36787	122891	34187	38399	72586	Yes
multiplier	signal trigger	42974	131690	8664	19698	28562	Yes
log2	signal trigger	46746	155981	27165	26367	53532	Yes
memctrl	always on	81588	224428	24932	32611	57543	Yes
div	always on	100985	274228	27908	264972	320788	Yes

Table I: Tests on Trojan insertion benchmarks.

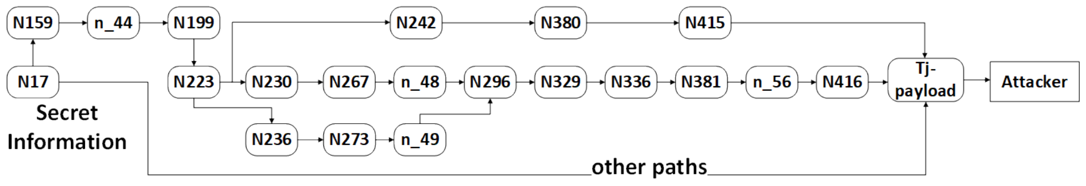

Figure 5: Detected leakage paths of benchmark c432.

C. *Results and Analysis*

Table I shows results of our experiments. Trojans are inserted into all the benchmarks for leaking information. Among those benchmarks, the Trojans in memetrl and div are always on, while the others are triggered by a signal. We account for the number of logic gates from the netlist data design files in the column of the functional gate, and the number of GLIFT gates from the formal model in the column of IFT gate. The gate number in IFT logic is 3-10 times than functional gates. It indicates the huge area overheads would be caused if we implement the GLIFT logic in the real hardware circuit.

Then the time consumption of parsing Verilog netlist to Z3 constrains is listed as model time. Accordingly, time cost in Z3 solving is listed as the detection time. The column of total time includes the time consumption from taking in benchmarks to detecting hardware Trojans. Taking the benchmark c6288 as an example, the c6288 includes 2416 gates in the design, from which 9364 GLIFT logic gates are generated by the parser. The time consumption of code parsing and generation is 134 ms. In the Z3 solving, 1085 ms is taken to detect the hardware Trojan. Assume that the security property has already been designed, then the total time cost for detecting Trojan in c6288 is 1219 ms. All the Trojans in those benchmarks are detected successfully.

The largest benchmark in this experiment is div which includes $100,985$ functional gates. The total time spent on the security verification is $320,778$ ms or around 5 minutes. From the results, the evaluation can be finished in minutes. The proposed formal framework is efficient for protecting the confidentiality of the gate-level netlist design.

Further, the leakage paths can be obtained by analyzing results, which provides a guide to help developers improve their designs. In Figure 5, we demonstrate the leakage paths detected in the benchmark c432. The signal N17 is the secret input signal that is tainted and the signal Tj-payload is the output of the Trojan. In this example, we detected 167 leakage paths in the gate-level netlist while 3 of them are shown in the figure. Developers could consider to improve the secure level, for instance, by adding obfuscation on those paths.

V. Limitations and Discussions

Although the proposed framework has an excellent performance in detecting sneaky paths of information leakage, there are still 2 major limitations – proof of genuine benchmark and lacking sequential logic support. For SMT solving, it's efficient to get the result if a solution exists. However, it becomes an NP-hard problem once there is no solution to the given problem/constrains. Mapping to our framework, it has a significant high performance in detecting sneaky paths in the condition that the Trojan or vulnerability exists. The solution searching strategy can be optimized to further improve efficiency. In contrast, if there is no such paths leaking secrets, the solver must check all possible cases before termination, which leads to intense computation complexity. To address this issue, we will set a threshold according to the size of the netlist file. The SMT solving would be terminated in the threshold and report a compromised secure checking to users.

In the meantime, lacking sequential logic is the other limitation of this work. By now, our parser only supports combinational circuit logic parsing and generation. Therefore, the sequential logic needs to be manually converted to Z3 constrains. We will add the function of support such logic by interpreting and parsing circuit like latch, flip-flop, etc.

VI. Conclusion

In this paper, a formal framework is proposed to protect the confidentiality of hardware design at the gate-level. By delivering a parser, the formal model is generated and composed of the functional circuit and GLIFT logic circuit. The Z3 solver validates the model with IFT property in the end. Therefore, the framework provides a fully automatic static formal verification from the input netlist file to the IFT property checking.

In the future, supporting of the sequential logic will be added to the framework. Accordingly, larger benchmarks with hardware Trojans will be tested. Also, we will extend the framework to cover more properties and features. The integrity property will be considered to identify malicious modifications.

Acknowledgments

Portions of this work were funded by the National Science Foundation (CNS-2019310).

References

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *IEEE Symposium on Security and Privacy*, 2007, pp. 296–310.

[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] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *HOST*, 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*. ACM, 2015, p. 145.

[9] 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.

[10] 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.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software*. Springer, 2003, pp. 235–239.

[12] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Automatic code converter enhanced pch framework for soc trust verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3390–3400, 2017.

[13] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1997, pp. 129–142.

[14] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 99–110, 2012.

[15] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[16] 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.

[17] 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.

[18] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin, "Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019, pp. 91–100.

[19] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

[20] M. Qin, W. Hu, X. Wang, D. Mu, and B. Mao, "Theorem proof based gate level information flow tracking for hardware security verification," *Computers & Security*, vol. 85, pp. 225–239, 2019.

[21] S. Deng, D. Gümüsoglu, W. Xiong, Y. S. Gener, O. Demir, and J. Szefer, "Secchisel: Language and tool for practical and scalable security verification of security-aware hardware architectures." *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 193, 2017.

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[23] 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.

[24] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the complexity of generating gate level information flow tracking logic," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 1067–1080, 2012.

[25] Microsoft Research, "Z3 api in python," https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

[26] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[27] EPFL and ISCAS85, "Epfl and iscas85 combinational benchmark circuits in generic gate verilog," https://github.com/jpsety/verilog_benchmark_circuits.