Inter-IP Malicious Modification Detection through Static Information Flow Tracking

Zhaoxiang Liu, *Kansas State University*
Orlando Arias, *University of Florida*
Weimin Fu, *Kansas State University*
Yier Jin, *University of Florida*
Xiaolong Guo, *Kansas State University*

***Abstract*—To help expand the usage of formal methods in the hardware security domain. We propose a static register-transfer level (RTL) security analysis framework and an electronic design automation (EDA) tool named If-Tracker to support the proposed framework. Through this framework, a data-flow model will be automatically extracted from the RTL description of the SoC. Information flow security properties will then be generated. The tool checks all possible inter-IP paths to verify whether any property violations exist. The effectiveness of the proposed framework is demonstrated on customized SoC designs using AMBA bus where malicious modifications are inserted across multiple IPs. Existing IP level security analysis tools cannot detect such Trojans. Compared to commercial formal tools such as Cadence JasperGold and Synopsys VC-Formal, our framework provides a much simpler user interface and can identify more types of malicious modifications.**

***Index Terms*—SoC Protection, AMBA Bus System, Hardware Trojan Detection, Formal Method**

I. INTRODUCTION

The semiconductor industry has an urgent demand for automated methods to detect and evaluate vulnerabilities in System-on-chip (SoC) at the design stage. Interestingly, while most of the hardware designs are crafted using HDL, security verification tools that work at the HDL level is lacking [1]. For verification purposes, designers must convert their SoCs into gate-level netlists for analysis purposes [2]. Unfortunately, there are limitations to these approaches. High-level circuit structures are lost in post-synthesis netlists as well as in high-level software descriptions. The recovery of high-level circuit structures is proven to be NP-hard and is an ongoing research problem.

Another factor affecting the scalability of formal verification at the SoC layer is the lack of well-defined security properties. Commercially available platforms such as Cadence JasperGold Security Path Verification (SPV) [3], Synopsys VC Formal Connectivity Checking, and Synopsys Formal Security Verification (FSV) [4] provide means of verifying SoC infrastructure. These tools are developed with bug/fault detection in mind, and rarely consider maliciously inserted logic.

To address these issues, in this paper we propose If-Tracker, an automatic and efficient formal verification framework. If-Tracker works at HDL level and is capable of checking security properties of SoC-scale hardware designs. Our framework leverages static information-flow tracking (IFT) to analyze the interaction between IP cores that are attached to the same SoC bus, trying to identify anomalies that can bypass individual security checks.

In summary, the main contributions of this paper are:

- We present a scalable framework to statically identify threats in SoC platforms at RT-Level. By parsing HDL into an AST, we extract more information that can be used for tracking, identification, and localization of anomalies.
- The introduction of If-Tracker, a CAD tool which can parse Verilog HDL and automates security checks on the design. If-Tracker models an SoC into a data-flow graph and automatically infers interactions between IP cores.
- We demonstrate the efficacy of If-Tracker using an SoC benchmark equipped with AMBA-AXI bus with a series of concealed Trojans. We also compare the If-Tracker to commercial tools from Cadence and Synopsys.

II. BACKGROUND

A. Security Formal Verification on SoC

Information-flow tracking (IFT) is a powerful approach to protect confidentiality in a hardware system by detecting the sneaky path of sensitive information leakage. IFT associates data or operations with labels/taint indicating the security levels. Various secure RTL programming languages, such as Caisson [5], Sapper [6], and SecVerilog [7], have been developed to check the noninterference property. When applying these solutions, users must learn the complex tag system and manually denote labels standing for the trust level to specify the information flow policy. QIF-Verilog is later proposed to protect the confidentiality with only one tag type [8]. It extends one security label from standard Verilog. However, QIF-Verilog can only protect confidentiality and validate IFT properties at IP level.

B. Formal Verification Tools for Security Property Checking

Existing commercial tools provide security property checking based on the IFT approach, such as JasperGold Secruity Path Verification (SPV), Synopsys VC Formal Connectivity Checking (CC) and Synopsys Formal Security Verification (FSV) [3], [4]. Several of recent formal verification frameworks were proposed leveraging these commercial tools [9], [10]. However, these commercial tools may not be effective nor efficient in detecting different types of security threats. Taking JasperGold SPV as an example, the taint propagation performed in SPV relies on simulation. It results in the issue that malicious modifications with special structures may either bypass the verification or significantly increase the time consumption for property proving. These limitations associated with the commercial tools are discussed at Section IV in detail.

978-3-9819263-6-1/DATE22/©2022 EDAA

III. Methodology

A. Adversarial Model

We assume that the adversary is capable of inserting malicious logic at the design phase of the IC lifecycle. For example, a rogue agent in a third-party design house may manipulate RT-Level hardware descriptions and insert hardware Trojans or backdoors [11]. Such Trojans or vulnerabilities can bypass security checks at IP level since they do not perform malicious behaviors as individual IPs. Instead, malicious behaviors will be triggered when these IPs are integrated into an SoC platform. Signals for triggering Trojans may come from a counter, input vectors, or certain physical conditions. Once activated, the inserted malicious logic may sniffer information in the bus, cause denial-of-service (DoS) to hardware IPs, or block communication channels. Furthermore, vulnerabilities may be introduced by designers unintentionally due to the lack of security knowledge during the development and integration of an SoC [12], [13], [14].

B. Workflow

The basic workflow of If-Tracker is shown in Figure 1. The HDL code of an SoC design is parsed into an abstract syntax tree (AST). Then we navigate the AST tracking assignments, as well as the scope and conditions of those assignments. The assignments and their dependencies are then merged into a single, directed data-flow graph (DFG).

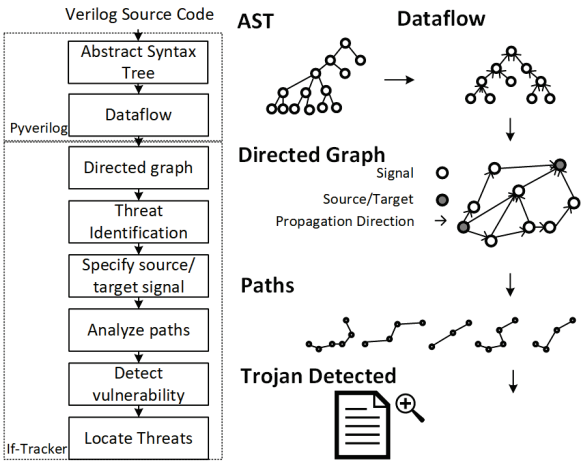

Fig. 1: Workflow of the proposed If-Tracker framework.

The directed graph consists of signal nodes and temporary nodes. Signal nodes represent variables declared in the HDL code. Temporary nodes are auto-generated by the developed If-Tracker, which represent the semantics of statements affecting the interaction of signals, such as Boolean expressions, if-statements, decomposition assignments, etc. A propagation path including these nodes reflects the relationship between the target node and the source node. Furthermore, the propagation conditions can be auto-analysed through If-Tracker.

Getting the directed DFG, security properties or check rules are provided by specifying the taint sources and targets in the IFT. Analyzing all those constrains in a path, the security rules can then be verified.

$$\begin{bmatrix} \textbf{Condition}_1: & \texttt{WSTRB[byte\_index]} == 1 & = & \textbf{true} \\ \textbf{Condition}_2: & \texttt{slv\_reg\_wren} & = & \textbf{true} \\ \textbf{Condition}_3: & \texttt{RESET} == \texttt{1'b0} & = & \textbf{false} \end{bmatrix}$$

Fig. 2: Propagation conditions of SHA256

$$\begin{bmatrix} \textbf{Condition}_1: & \texttt{WSTRB[byte\_index]} == 1 & = & \textbf{true} \\ \textbf{Condition}_2: & \texttt{slv\_reg\_wren || 1} & = & \textbf{true} \\ \textbf{Condition}_3: & \texttt{RESET} == \texttt{1'b0} & = & \textbf{false} \end{bmatrix}$$

Fig. 3: Propagation conditions of RAM

In this paper, we design and apply four types of information leakage properties. These four properties with vulnerabilities described in natural language are listed in Table I to address various threats/vulnerabilities. For these properties, we apply them in detecting the threats demonstrated in the Section IV, where Property 1 is for case study 1, Properties 3 and 4 are for case study 2, and Property 2 is for case study 3.

IV. Experimental Results

A. Case Study 1: Eavesdropping Attack

1) AXI4-Lite SoC: Our experimental benchmark employs an AXI4-Lite based SoC with a 32 bit shared bus interconnection. The SoC integrates an open-source pico RISC-V CPU as the master IP core and two slave IP cores, i.e., a RAM and a SHA256 crypto core. Data transactions on shared bus based SoC face the risk of data being snooped by malicious third-party IPs as the shared bus is connected to all IPs. It is assumed that the RAM module contains malicious logic – a Trojan is inserted into the RAM's slave interface. As a result, the malicious RAM module constantly snoops the shared bus data and copies the data inside the module.

2) Experiment on AXI4-Lite Benchmark: The hardware Trojan in RAM's slave interface eavesdrops the bus communication, which means that data needs to propagate from the CPU to both SHA256 and RAM modules at the same time. By extracting the propagation conditions and checking their conjunctions, our framework can detect this attack.

Since the data leakage potentially occurs on the write data channel, our experiment specifies the CPU's `WDATA` as the taint source to track the signal propagation. When a write transaction is initiated, `WDATA` will propagate taint to the SHA256 registers and RAM. Using If-Tracker, four paths are detected from CPU write data port to each IP module. These four paths are similar because the slave IP follows the AXI protocol to read data from the bus byte by byte and each path represents one byte signal propagation. Furthermore, AXI interconnect is designed with the purpose of connecting CPU to slave IPs. The detected paths just prove this functionality. Namely, the malicious behavior cannot be addressed by only checking the path existence.

In this case study, malicious modification is detected by analyzing the dependency of propagation conditions using the developed If-Tracker. Figure 2 lists three conditions of the taint propagation from the bus to SHA256 module.

TABLE I: Security properties for addressing threats. In the table, BM: Bus Manipulation

Vulnerability	Malicious Modification	Security Property (Prop)
1. Bus eavesdropping	When a master device writes to a specific IP, another malicious IP eavesdrops information illegally through the bus	1. When a master device is writing data to a specific IP through SoC bus, no other IPs should be in the status of accepting data from the bus
2. Sneaky path	When triggered, a sneaky path will leak information or breach the signal integrity	2. Reachability is checked from the master processor write output port to assets under protection
3. BM with signal replacement	Replace the sensitive signal by a constant value in the middle of the data transaction via AXI bus	3. Taints from all sources must be propagated to the targets
4. BM with extra logic insertion	Insert extra logic between the sensitive signal and the transmission target inter AXI bus	4. Taint propagation starts from targets and then propagates in an inverse way. Finally, the taints arrive in areas outside of the sources

Condition$_1$ ensures that the byte on the bus has valid information; *Condition*$_2$ ensures that the write enable signal for the peripheral register is asserted; and *Condition*$_3$ requires that the reset signal is deactivated. If all conditions are met data can be written to the memory mapped SHA256 register.

Note that signal `WSTRB` is sent from CPU. Two slave IPs receive the same signals from the CPU all the time.The `SLV_REG_WREN` signal acts as a write enable for the memory mapped register and must be asserted to enable a write transaction. When routing traffic, the interconnect asserts the `SLV_REG_WREN` for the IP being accessed, selecting the desired IP. The corresponding path from bus to RAM is shown in Figure 3 and only the *Condition*$_2$ is different from the SHA256 path.

In our design, the propagation condition in RAM path was changed to `SLV_REG_WREN || 1'b1`, which leaves the RAM writable regardless of how the interconnect routes the write request. This results in the two paths propagation constraints being satisfied at the same time. Therefore, all propagation constraints of the two paths can be satisfied at the same time, which means that there is a data leakage from the bus.

B. Case Study 2: TrustZone Attack

1) Malicious Interconnection in TrustZone: Benhani et al. demonstrate three different attacks to maliciously modify the interconnection parts by tampering with the second bit of the `ARPROT` signal in [14]. The first attack cuts the connection of `ARPROT[1]` between the master and slave interfaces. The slave side receives a permanent `1'b0` on `ARPROT[1]`. This causes all read transactions to the core to be considered as secure. The data would be leaked from the slave side due to this non-secure visit. The other two attacks utilize combinational logic on the `ARPROT[1]` by inserting an AND gate or a multiplexer. These serve as a trigger condition to arbitrarily make transactions act as if they are secure.

2) Experiment on AXI-Interconnect of TrustZone: The developed If-Tracker can detect malicious modifications to `ARPROT` in all three of the scenarios mentioned above. If no hardware Trojan is inserted, our tool finds one direct path in the $3\,\mathrm{bit}$ signal. However, if we tie a bit of `ARPROT` to ground to force transactions to be recognized as secure, then If-Tracker detects a second source, indicating the presence of malicious logic. If-Tracker further recognizes that a $2\,\mathrm{bit}$ signal is propagated from the source but a total of $3\,\mathrm{bit}$ reach the destination, providing yet another indication of the conflict. For the event where extra logic is added to the `AWPROT` signal through an extra gate or a multiplexer, If-Tracker detects four source nodes instead of the expected single node, indicating unintended or malicious logic. By backtracking the path, If-Tracker can determine the exact source of the modifications, showing what operations are taking place as well as their locations in HDL source files.

C. Case Study 3: Counter-based Trojan Attack

1) AXI4-Lite Delayed attack: Hardware Trojans tend not to be triggered immediately, and satisfying the trigger of a Trojan may take hundreds or thousands of clock cycles. It is a concern in the dynamic analysis of circuits, where one clock cycle of circuit simulation often requires minutes or even hours. The time cost to detect such a Trojan may be unacceptable. To mimic this situation, we build a three clock cycles delay based Trojan trigger on the AXI4-Lite SoC as shown in Trojan design 1. Sensitive information is leaked after a few clock cycles and a covert channel is created for the secret transmission. Our framework would be tested on this Trojan.

Trojan Design 1 Information leaked in 3000 clock cycles

```
1: Every three clock cycle count++
2: if count == 10000 then
3:     out ← sensitive_data
4: end if
```

2) Experiment on AXI4-Lite Delayed attack: Delay-based Trojans set up a covert path for the transmission of sensitive data. Our method checks for the presence of a transformation channel to help determine the data leakage opportunities provided by the illegal path. The framework sets sensitive data as the source signal and an output port as the target. The framework verifies the reachability between the source and the target and the collection of constraints on the information flowing through the node is extracted.

V. Comparison with Commercial Tools

Our experiments are conducted on a dual AMD EPYC 7401 processor with $264\,\mathrm{GiB}$ memory. Table II summarizes how our tool performs versus commercially available tools.

In the first column, we list the benchmarks mainly including AXI4-Lite SoC and TrustZone interconnection. Taint source and target are in columns 2 and 3. The malicious modification for each benchmark is shown in the fourth column. The Property column includes the utilized security properties designed in Table I. The detecting results of different applications are put in the rest of columns where time consumed and performance

TABLE II: Detection results and comparison with commercial tools. In the table, PE: path exists, CE: conditions extracted, PNE: path not exists, CC: Connectivity Checking, FSV: Formal Security Verification, SPV: Security Path Verification. All times given in seconds, RL: Reach Time Limit.

Benchmark	Source	Target	Behavior	Property	If-Tracker		CC		FSV		SPV	
					Time)	Perf	Time	Perf	Time	Perf	Time	Perf
AXI4-Lite	CPU WDATA	RAM	Eavesdropping	Prop1	0.43 s	PE CE	1.00 s	PE	102 s	PE	26.2 s	PE
AXI4-Lite	Asset	Covert output	Sneaky path	Prop2	0.16 s	PE CE	1.00 s	PE	RL	N/A	RL	N/A
TrustZone	Master ARPROT	Slave ARPROT	Cut ARPROT	Prop3	0.01 s	PE CE	1.00 s	PE CE	13.0 s	PE	N/A	N/A
TrustZone	Master ARPROT	Slave ARPROT	And ARPROT	Prop4	0.01 s	PE CE	1.00 s	PE CE	13.0 s	PE	0.02 s	PE
TrustZone	Master ARPROT	Slave ARPROT	Mux ARPROT	Prop4	0.01 s	PE CE	1.00 s	PE CE	13.0 s	PE	0.02 s	PE

are displayed. All threats are detected by our framework with higher efficiency.

In the eavesdropping attack from case study 1, Synopsys CC can locate the path but cannot extract the conditions. The limitation of CC is that conditions cannot be derived when the path involves flip-flops [4]. Synopsys FSV and Cadence SPV take longer to formally prove the reachability from the source node to target node. Since little information is involved in these paths, sophisticated malicious behavior like eavesdropping cannot be addressed by FSV and SPV. That is, only if both path exists (PE) and conditions extracted (CE) are obtained from the results, the threat can be handled.

At the second row, sneaky path attack from case study 3 is a typical taint propagation model, where Property 2 would be formally proved if taint source reaches the target. This counter-based Trojan will only be triggered after ten thousand clock cycles. As a result, both FSV and SPV reach the time/cycle limit during the proving process (we set one hour as the maximum running time). Again, due to the flip-flop issue, CC can only prove the path existence and list the path. In the developed If-Tracker, the conditions are extracted as symbols and then formally analyzed. Thus, the time consumed is not influenced by the number of counters and clock cycles.

As the the rest rows, for detecting TrustZone attack from case study 2, both If-Tracker and CC can identify the modification. CC is also able to extract conditions since the malicious modification is the combinational logic. Even though SPV and FSV can validate the reachability or PE in most cases, for identifying the anomaly, both of them need the user to do further waveform inspection.

The results presented in this paper indicate that malicious behaviors in SoC are sophisticated so that various kinds of attacks cannot be addressed by only checking reachability or path existence. Considering the propagation conditions makes the IFT be more precise and efficient by associating multiple propagation paths and avoiding unnecessary computation burden. Therefore, If-Tracker discovers the potential malicious tampering through extracting the propagation path topology and the conditions with a much higher performance.

VI. Conclusion

To secure SoC designs, we present If-Tracker, an HDL level automatic IFT-based security verification framework to discovers the potential malicious tampering through extracting the propagation path topology and the conditions with a much higher performance. We compare If-Tracker with the state of the art tools and prove our tool can better address different types SoC level attacks.

Acknowledgments

Portions of this work were supported by the National Science Foundation (CCF-2019310, CCF-2028910).

References

[1] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying rtl verilog," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 27, no. 2, pp. 366–379, 2008.

[2] C. Wolf, "Yosys open synthesis suite," 2016.

[3] Cadence, *Security Path Verification App User Guide*, December 2020.

[4] Synopsys, *VC Formal Verification User Guide*, December 2019, Version P-2019.06-SP2.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*. IEEE, 2017, pp. 1–10.

[10] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Soc security verification using property checking," in *2019 IEEE International Test Conference*. IEEE, 2019, pp. 1–10.

[11] 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.

[12] K. Xiao, A. Nahiyan, and M. Tehranipoor, "Security rule checking in ic design," *Computer*, vol. 49, no. 8, pp. 54–61, 2016.

[13] J. He, X. Guo, T. Meade, R. G. Dutta, Y. Zhao, and Y. Jin, "Soc interconnection protection through formal verification," *Integration*, vol. 64, pp. 143–151, 2019.

[14] E. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019.