Integration, the VLSI Journal 64 (2019) 143–151

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

SoC interconnection protection through formal verification

Jiaji He [a], Xiaolong Guo [b], Travis Meade [c], Raj Gautam Dutta [c], Yiqiang Zhao [a], Yier Jin [b,*]

[a] School of Microelectronics, Tianjin University, Tianjin, Tianjin, 300072, China
[b] Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA
[c] Department of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816, USA

ARTICLE INFO

Keywords:
Formal verification
Intellectual property
Hardware Trojan detection
SoC security

ABSTRACT

The wide adoption of third-party hardware Intellectual Property (IP) cores including those from untrusted vendors have raised security concerns for system designers and end-users. Existing approaches to ensure the trustworthiness of individual IPs rarely consider the entire SoC design, especially the IP interactions through SoC bus. These methods can hardly identify malicious logic (or design flaws) distributed in multiple IPs whereas individual IPs fulfill security properties and can pass the security testing/verification. One possible solution is to treat the SoC as one IP core and try to verify security properties of the entire design. This method, however, suffers from scalability issues due to the large size of SoC designs with multiple IP cores integrated. In this paper, we present a scalable SoC bus verification framework trying to verify the security properties of SoC bus implementation where the bus protocol plays the role of the golden reference. More specifically, finite state machine (FSM) models will be constructed from the bus implementation and the trustworthiness will be verified based on the property set derived from the bus protocol and potential security threats. Along with IP level formal verification solutions, the proposed framework can help ensure the security of large-scale SoCs. Experimental results on ARM AMBA Bus demonstrate that our approach is applicable and scalable to prevent information leakage and denial-of-service (DoS) attack by verifying security properties.

1. Introduction

With the rapid development of modern integrated circuit (IC) fabrication techniques, the complexity of ICs is increasing rapidly. In order to reduce the time-to-market, circuit designers try to integrate different Intellectual Property (IP) cores into a single chip to form a system-on-chip (SoC). To coordinate interactions among different IPs, bus protocols for data and control transfers are used. For example, the Advanced Micro-controller Bus Architecture (AMBA) [1] from ARM serves as a leading example among these bus protocols. However, the possibility of inserting hardware Trojans at the design stage in an IP by an untrusted third-party vendor has raised security concerns in the SoC design industry [2,3]. To ensure the trustworthiness of a SoC design, two types of methods, runtime detection and static verification solutions, have been developed targeting the SoC bus and IPs. In the runtime method, potential security vulnerabilities are identified during the SoC operation. It enables a SoC integrator to enhance security of existing bus protocols by adding additional runtime monitoring circuits or enhancing the existing protocols. The static solution uses testing and verification techniques to either validate the security of SoC design and bus protocols, or detect any design flaws that may impact the security of the SoC operations before implementation.

As shown in Fig. 1, a SoC designer builds a system using IPs from third-party vendors and some of the IPs in the SoC are untrusted. To ensure the security of the design, in this paper, we propose a comprehensive verification framework that is scalable for large SoCs. The interactions between IPs and the system bus will be considered and checked. Anomalies in normal communications between untrusted IPs and the rest of the SoC design can be detected. Our method only focuses on verification of digital circuits. If there are analog parts in the SoC system, the analog circuits will not be modeled. FSMs are firstly extracted from the SoC utilizing gate-level netlist, and the formal model is established of the design. Then security specifications are proposed based on the knowledge of possible attacks through untrusted IPs connected to the bus. Finally, the formalized model of the SoC is verified against security properties using an automated model checking tool. If a SoC

* Corresponding author.
E-mail addresses: dochejj@tju.edu.cn (J. He), guoxiaolong@ufl.edu (X. Guo), travm12@knights.ucf.edu (T. Meade), rajgautamdutta@knights.ucf.edu (R.G. Dutta), yq_zhao@tju.edu.cn (Y. Zhao), yier.jin@ece.ufl.edu (Y. Jin).

https://doi.org/10.1016/j.vlsi.2018.09.007
Received 22 May 2018; Received in revised form 30 July 2018; Accepted 20 September 2018
Available online 5 October 2018

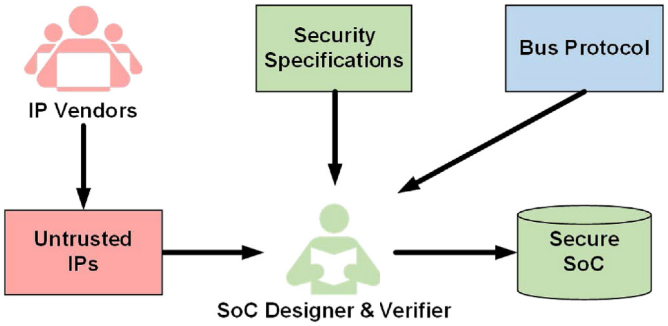

Fig. 1. Interactive of components in proposed framework.

design satisfies the security specifications, then the security of the SoC is proved.

The main contributions of this paper are listed as follows.

- Formal security specifications at the SoC level are developed with consideration of real bus operations and possible Trojans attacks.
- The finite state machine (FSM) of the whole SoC is extracted utilizing gate-level netlist. All behaviors of the circuit are modeled without performing any abstraction, and we ensure that any malicious behavior is not lost during modeling process. Our modeling strategy is scalable for large SoCs.
- We focus on securing the entire SoC from hardware Trojans considering possible attacks utilizing normal bus communications.

The rest of this paper is organized as follows. In section 2, we discuss previous work on SoC design protection and verification. Section 3 describes the threat model in our paper and provides background of the tools used in our framework. In section 4, we explain our methodology in details. In section 5, we carry out a series of experiments to demonstrate our method. The experiment results are shown in section 6. Conclusions and future work are drawn in section 7.

2. Related work

In this section, we will introduce both runtime and static solutions for hardware system protection.

2.1. Runtime methods

Many runtime hardware approaches were developed for providing secure assurances of IPs and SoC. In Ref. [4], IP sandboxing technique was utilized to defeat drone jamming-based attacks. A resilient SoC security architecture was proposed in Ref. [5], and this architecture implements certain security policies and also ensures trusted information flow within the blocks. This solution provides a comprehensive solution in the secure architecture based SoC protection. However, it still needs a central security policy checker and security wrappers. Also there are other techniques such as gate-level information flow tracking [6] to ensure the information security of the system bus. Authors in Ref. [7] gave a solution for hardware runtime formal verification of security properties on IPs depending on proof-carrying hardware (PCH) approach. A hardware based SAT solver was developed to perform validating of fabricated hardware. Still, high overhead in runtime PCH causes scalability issue in practical use.

Further, runtime methods enhance security of bus protocols with a wrapper or monitor, and require specific design modifications to achieve particular security purposes. In Ref. [8], correctness of modules was checked using an IP interface monitor, which was developed using high-level specifications. The interface monitor cannot detect system-level malicious behaviors if the modules follow the normal bus protocol. Lin et al. [9] designed a monitor for bus compliance testing and for observing bus signals in a SoC system. Although they verified the data part of the bus transfer against necessary properties systematically, their method cannot detect malicious behaviors. A security enhanced communication architecture was designed in Ref. [10] to enhance bus communications, but this architecture is not applicable to other security vulnerabilities. Kim et al. [11] developed a secure bus architecture by constructing new structures around core modules of a bus, and the architecture was able to provide runtime protection against hardware Trojans. Although the secure architecture accomplished protection without incurring high costs in terms of bus resources and performance, it can only handle security issues related with one single module.

Even if runtime methods can provide a very satisfactory security protection, these methods all require hardware modifications of original circuits or additional modules in the systems. Moreover, the modifications usually require the designers of the security solution have a well understanding of the circuits and systems. Further, implementing these methods will inevitably introduce area and power overhead of the original design and will have impact on the performance.

2.2. Static methods

Static methods, which are used prior to design deployment in the pre-layout stage, neither cause any hardware overhead nor require modifications to original designs. Among the static methods, there are many approaches targeted at pre-layout hardware Trojan detection including feature analysis and formal verification. The feature analysis Trojan detection methods focus on extracting particular features from pre-layout design files directly to find anomalies. In Ref. [12], a HTChecker was proposed to detect hardware Trojans in VLSI design based on static characteristics of gate-level netlist. Trojan features are extracted from gate-level netlist in Ref. [13], and the random forest classifier is utilized to select the best set of Trojan features. The author in Ref. [14] proposed to apply the inter-cluster distance to distinguish Trojan gates and genuine gates in a design based on the controllability and observability characteristics. In Ref. [15], a framework is proposed to combine flip-flop level and combinational logic level features for analyzing to achieve both high efficiency and accuracy. Feature analysis based methods directly detect hardware Trojans in individual IPs, however, the trustworthiness of the whole system composed of multiple IPs may not be guaranteed.

Whereas, formal methods do not directly detect HTs but try to evaluate the trustworthiness of IP cores integrated in the SoC systems. Meanwhile, formal verification can develop security properties with the consideration of potential system level threats. Formal verification with trusted models was used in the Algorithm for Resisting Trojans of [16] as part of a series of steps to provide complete protection of design from Trojan payloads, but their methods only focused on individual IPs. In Ref. [17], security critical objectives were selected to divide the system into smaller security sensitive assets, and security properties were developed and checked accordingly. Although they improved the scalability of formal verification by breaking down the SoC into subsets of security sensitive assets, some critical information may be lost during the decomposition process. A SAT-based formal verification method was proposed in Ref. [18], and the authors proposed a series of properties to detect the information-leaking Trojan. However, their methods cannot detect system level Trojans. Additionally, formal verification has been widely used for checking functional and security properties on SoC bus. An abstract model of a commercial field bus was developed in Ref. [19], and a number of imperfections in the protocol logic itself were found. An arbiter core of IBM Core Connect bus was verified by Goel et al. [20] and they found defects of the protocol itself during functional checking. In Ref. [21], an AMBA AHB protocol model was developed according to the specification of the protocol, and the non-starvation property was checked for the master. Guo et al. [22] used a hybrid approach in PCH framework to alleviate the scalability issue in formal verification of large designs. However, the above verification still only focused on the function checking.

From the discussion above, current formal verification methods usually focus on single IP verification and do not consider the interconnections among various IPs through certain bus protocols. It is important to take the bus protocols into consideration when verifying the security of the whole system. Although some methods can detect system level Trojans or security threats, they only build an abstract model of the SoC design prior to verification. As a result, some details of the design including malicious logic may be lost during the abstraction process. So a comprehensive model containing all logic operations of the system is required.

3. Background and preliminary

3.1. Threat model

Hardware Trojans can be inserted at many different stages of the IC lifecycle. In this paper, we consider the rogue agent at third-party design house who can access the hardware design, and make modifications to the RTL code and netlist files. Therefore, the availability of gate-level netlist is required. In practical, soft IPs are available in the form of RTL code, and firm IPs are in the form of gate-level netlist. While hard IPs are available in layout format, the gate-level netlist can be extracted using reverse engineering methods [23]. The proposed Trojan detection approach targets at detecting digital hardware Trojans that utilize the bus interface to monitor the triggering signals and perform malicious functions. Since there are many SoC assets that could be targeted by an attacker, it is highly possible that security threats really exist. With more third-party IPs integrated in a SoC system, system designers are having less control over the security of the SoC design. As all the IPs mounted in a SoC have access to the bus, a highly representative case is that a Trojan hidden in untrusted IPs can be triggered either by specific data on the bus or by a certain operation. Moreover, a malicious IP can typically affect system level behaviors through bus communications, control flow and data transfer. Upon activation, a Trojan may leak sensitive information from the chip, modify functionality, or cause a denial-of-service to the system. The security properties are developed by considering possible Trojan attacks and the operations of bus protocol. These properties must resort to validation of the whole system behaviors. Accordingly, the correctness of properties indicates the absence of malicious logic. Please note that the proposed method in the paper does not provide protection against attacks that are not described by the set of security properties.

3.2. Model checking

Model checking [24] is an automated method for verifying models in software and hardware applications. The model checkers are developed based on either symbolic approaches or satisfiability solving. Symbolic model checking is one of the initial approaches used for hardware system verification.

UPPAAL [25] is developed based on constraint-solving and on-the-fly mechanism. It provides a tool box for modeling, simulation, and verification of real-time systems. In this model checker, the real-time system is first modeled as a network of timed automata and then the property is checked using forward reachability analysis. Widely used for verification of hardware and protocol designs, UPPAAL helps developers find errors and gain confidence in a design by achieving the goal of a bug-free system. In this paper, model checker UPPAAL is utilized to verify the SoC against security properties.

3.3. REFSM

Reverse Engineering Finite State Machine (REFSM) [26] leverages gate-level netlist to extract a high level description of the control logic. REFSM performs extraction efficiently through logic registers' simulation. If a more accurate model is required, external parameters can be adjusted to take into account extra temporal logic. The state exploration is performed using a breadth first search over the state space. Once a state is selected for transition exploration, REFSM uses a modified 3-SAT algorithm to find all possible transitions. These transitions are partially compressed using a branch and bound method on the decision tree. An optional FSM decomposition process can be used to break a large FSM into several Graph Factors. The decomposition process is implemented using a polynomial time heuristic. The heuristic uses the information of the registers to make estimations on the final Graph Factors. In this paper, the decomposition function is turned off to allow the generation of FSMs that fully emulate the entire FSM interactions within the SoC design. The abstraction level of our method is down to behavior level of the design, so any malicious behavior should not be missed out. In this paper, REFSM is utilized to generate a non-simplified FSM of the SoC design from gate-level netlist.

4. Methodology

In this section, we will demonstrate how a SoC is formalized from gate-level netlist and how security properties are specified and mapped to a design. For the reason that the proposed approach follows the process of static formal verification methods and focuses on security verification of a design before the application stage, there is no overhead in area and time in runtime. Workload of the method is only caused by model formalization and property checking in design stage. The advantages of the proposed framework are as follows: 1) The SoC formalization method is able to extract behavioral models from gate-level netlist in the SoC formalization process, and all behaviors are modeled without any abstraction; 2) Security policies applied in the verification process are proposed based on the analysis of potential system security vulner-

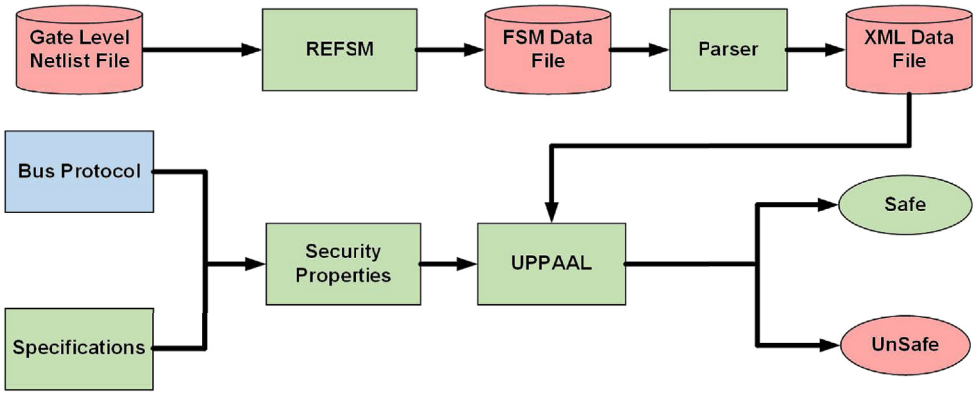

Fig. 2. The working procedure of the verification process.

abilities and interpretation of bus protocols.

As shown in Fig. 2, the framework includes three steps: 1) REFSM is used to extract the model of a SoC from gate-level netlist; 2) Security properties are developed by specifying the proper states of the FSM to state formulae and path formulae; and 3) The extracted model against security properties is verified in UPPAAL.

4.1. SoC formalization

Suppose a SoC design is in the format of gate-level netlist N, and the netlist of each individual IPs of SoC, including I/O interfaces, wires, and gates (G), is denoted as $n \in N$. The bus interconnect module [1] belongs to N and is denoted as n_{inner}. The states of I/O interfaces and wires are defined as Sig. To extract FSMs from the netlist, a transformation f is carried out of N with consideration of keeping design details. The extracted FSM is denoted as M, and the FSM of each individual IP is denoted as m. The individual FSMs have connections among each other, then m forms M. In addition, a configuration file is derived from the bus interconnect module defined as Syn, which contains the synchronization information among FSMs within M. The total states in the FSMs are denoted as S and individual state is denoted as s. The transformation of netlist to FSMs of each IPs can be given by equation (1) and transformation of the entire SoC is given by equation (2).

$$n \xrightarrow{f} m \tag{1}$$

$$N \xrightarrow{f} M \wedge Syn \tag{2}$$

Equation (3) shows that N is made up of n_{inner} and netlist of multiple individual IPs, and equation (4) shows that the FSM of the SoC is composed of many individual FSMs, where $k \in \mathbb{Z}$.

$$N \coloneqq n_{inner} \wedge n_1 \wedge n_2 \wedge \ldots \wedge n_k \tag{3}$$

$$M \coloneqq m_1 \cup m_2 \cup \cdots \cup m_k \tag{4}$$

Further, m is constructed by using states and transfers between states as shown in the following equations (5)–(7), where $l \in \mathbb{Z}$, and $cons$ represents the conditions for the transfers to happen. A transition ω is defined by equation (6), where $s_{pre} \in S$ represents the source pre-state, and $s_{post} \in S$ represents the target post-state. The corresponding transition condition of ω is shown in equation (7), where $sig \in Sig$ and $j \in \mathbb{Z}$. A transfer is driven by changes of signals in circuit.

$$m \coloneqq \omega_1 \wedge \omega_2 \wedge \ldots \wedge \omega_l \tag{5}$$

$$\omega \coloneqq s_{pre} \xrightarrow{cons} s_{post} \tag{6}$$

$$cons \coloneqq sig_1 \wedge sig_2 \wedge \ldots \wedge sig_j \tag{7}$$

Considering specific gates in the design, behavior B of a specific gate $g \in G$ is described as a collaboration of input value gin, type of gate $gtype$, and corresponding output value $gout$, and is denoted as equation (8). Compositions of B for G produces corresponding states, which is denoted as equation (9), where $i \in \mathbb{Z}$, gin and $gout$ are both binary vector, and $gtype$ belongs to an enumerable gate library.

$$B \coloneqq gin \xrightarrow{gtype} gout \tag{8}$$

$$B_1 \wedge B_2 \wedge \ldots \wedge B_i \longrightarrow s \tag{9}$$

4.2. Security specification

The main purpose of a model checker is to verify whether a model satisfies required specifications. The informal specifications, which are written in natural language, should be translated to a formal language prior to verification. For UPPAAL, the properties should be specified in Timed Computation Tree Logic, which consists of state formulae and path formulae. State formulae describe individual states, and path formulae quantify over paths or traces of the model [27]. As shown in Figs. 1 and 2, after receiving the required information, the SoC designer and verifier can agree upon a set of security properties that the SoC system and IPs should satisfy.

All masters and slave IPs integrated on bus must obey the bus protocol and follow certain sequence of operations. The composition of states is given by equation (10), where $s_1 \in m_1$, $s_2 \in m_2$, …, $s_k \in m_k$ and $k \in \mathbb{Z}$.

$$s \coloneqq s_1 \wedge s_2 \wedge \ldots \wedge s_k \tag{10}$$

Hardware Trojans in untrusted IPs may pass the functional testing and verification, and can get activated through interactions among IPs when mounted on the bus. Thus a malicious state set $S_{bad} \in S$ exists in the formal model accordingly. The components in the state set S_{bad} are states related with the Trojans' trigger and malicious payload. Then the security specification is designed as whether there exits at least one $Path$, which is denoted as equation (11), that makes S_{bad} reachable as shown in equation (12), where $p \in \mathbb{Z}$. The states in the state set S_{bad} should be reached in a particular order following the $Path$ to activate the Trojans and then perform certain malicious function.

$$Path \coloneqq cons_1 \wedge cons_2 \wedge \ldots \wedge cons_p \tag{11}$$

$$s \xrightarrow{path} S_{bad} \tag{12}$$

5. Implementations

In this section, the effectiveness of our framework for SoC system protection is validated using practical cases.

5.1. Attack vectors

Typically, a SoC design can suffer from different types of attacks, such as information leakage and DoS. While some of the threats come from one single malicious Trojan-infected IP, other threats arise as a collaboration of several system-level malicious IPs [5]. These attacks are very hard to detect because the Trojans hidden in untrusted IPs all act like non-malicious circuits and follow the normal bus operation. Following are two typical cases for a representative SoC implementation. The first is the information leakage attack performed by one single malicious Trojan-infected IP, and the second is the DoS attack performed by several system level malicious IPs.

5.1.1. Information leakage attack

Information leakage attack is one of the most common attacks for SoC designs. A hardware Trojan can be inserted in one of the slave modules. The Trojan is triggered by monitoring normal bus operation and the trigger condition is a specific set of data transmitted on bus. Once triggered, the Trojan can leak data which belongs to other modules. This kind of attack is very hard to detect using normal method because the Trojan's behavior follows the normal bus protocol and leaks information anonymously.

5.1.2. Denial of service attack

Denial of service attack can cause system failure at key points. A typical case is that a hardware Trojan can be inserted in one of the master

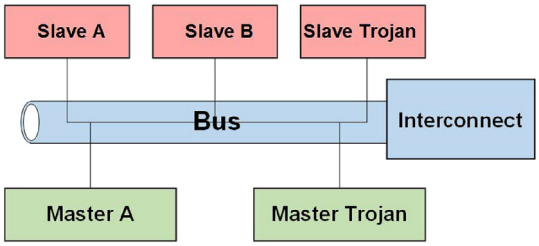

Fig. 3. Untrusted IPs mounted on bus.

Table 1
Model parameter counts for the bus system.

Modules	Gates	States	Transitions
Master A	1926	1244	14141
Master Trojan	2102	1244	14141
AXI Interconnect	1359	81	290
Slave A	1542	768	66880
Slave B	1513	768	66880
Slave Trojan	1512	768	66880

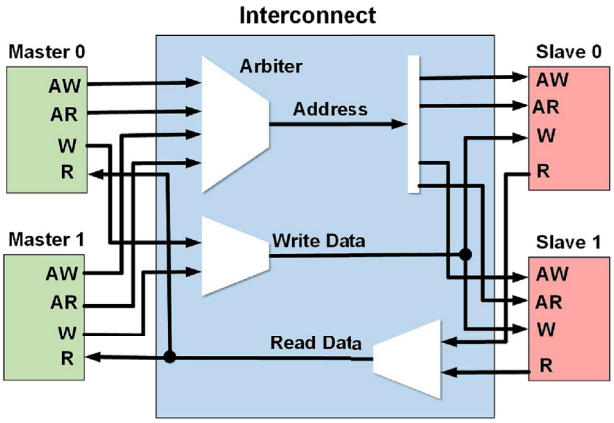

Fig. 4. Shared access mode of AXI interconnect.

modules. The Trojan's trigger can be hidden in one of the slave modules. Once the Trojan is activated, it will continuously occupy the system bus and block other master modules from using the system bus, thus causing the system's malfunction or denial of service. Although there are some strategies that can help preventing this type of DoS attack, such as round-robin arbitration [28] in AMBA bus systems, the Trojan can still cause denial of service in masters with lower priority settings [29].

5.2. Modeling process

A SoC design testbed is built upon six IPs, which are two masters, AXI interconnect, and three slaves. As shown in Fig. 3, one of the slaves, whose name is *Slave Trojan*, is infected with a hardware Trojan and this Trojan will leak secret information after activated. Also one of the masters, whose name is *Master Trojan*, is infected with a hardware Trojan and this Trojan will block other masters from using the bus once triggered. The bus protocol in the SoC design is the Advanced eXtensible Interface (AXI) which is one of the most widespread AMBA interface [1], and AXI4 bus protocol is the latest version of AXI bus. A bus system with third-party IPs based on the AXI4-Lite interface is established and the AXI interconnect is configured in the shared access mode [29], as shown in Fig. 4. In the shared access mode, the write channel and the read channel are shared and connected to all slaves so that slaves can access all data on the bus channel. Also, addresses provided by the masters are broadcast to all slaves.

The details of modeling process are shown in Fig. 5. The first step is to get gate-level netlist for the entire SoC design. The system is developed utilizing RTL code according to AXI4-Lite interface specifications. The SoC design is synthesized to gate-level netlist using Design Compiler and the SAED_EDK3228_CORE Digital Standard Cell Library from Synopsys. During the synthesis process, the original hierarchy of the design is retained. The hardware Trojan showed in Fig. 3 is inserted at RTL and also synthesized into gate-level netlist. The synthesis results and gate counts for these modules are shown in Table 1. The Master A interface has 1926 gates, while the Master Trojan interface has 176 more gates than Master A. The difference in the gate count is caused by the Trojan in the master module, because the function of the two masters is identical except for the address range and Trojan insertion. Slave A interface has 1542 gates and Slave B interface has 1513 gates, while Slave Trojan interface has 1512 gates. The difference in the gate count between Slave B and Slave Trojan is also caused by the Trojan in the slave module.

After the synthesis process, FSMs are extracted from the netlist using REFSM tool [26], the details of this process are shown in Fig. 5. The data structure of a state transition is composed of three parts: *pre–state*, *post–state* and *conditions*. *pre–state* stands for the source state of a transition, while *post–state* stands for the target state. Each state in FSMs are mapped to corresponding states of SoC circuit and *conditions* imply the conditions which trigger transitions between different states. In the extracted FSMs, a state is actually a reflection of one specific circuit status, and the running hardware will lead to state changes in FSMs. Accordingly, circuit's behaviors, such as receiving data or triggering the Trojan, are mapped to one or a sequence of state transitions. For each slave module, there are over 700 states and 60,000 transitions. Also there are over 1000 states and 10,000 transitions for the master modules. The interconnect module is designed to be a combination of connections between masters and slaves. The interconnect module has the arbiter integrated inside in the experiment.

The next stage is to model FSMs as the input of UPPAAL. In general, manually drawing schematic through graphical user interface is used to facilitate modeling in UPPAAL. However, this manual modeling approach suffers from scalability problems when there are too many states and transitions, so a parser is developed to map all FSMs into XML files which UPPAAL can read. Note that a stand-alone command line verifier of UPPAAL *verifyta* [27] is leveraged to improve the data

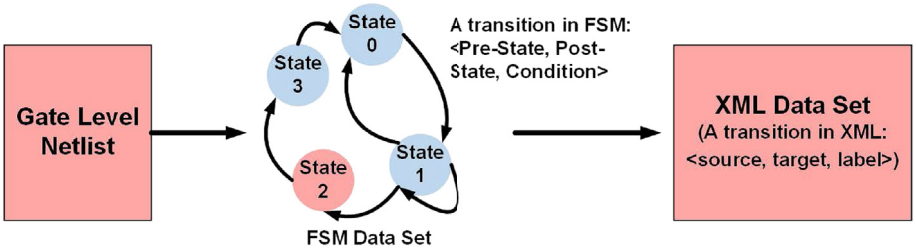

Fig. 5. Data structure and parsing in formalization process.

processing capability. UPPAAL can take XML file as input with specific data format as shown in Fig. 5. In the format, *template* indicates the module, which is one formal model of an IP, and a *transition* in the *template* means a FSM state transition. Furthermore, the *pre–states* are mapped as *source*, the *post–states* are mapped as *target*, and the *conditions* are mapped as transitions in UPPAAL and tagged with different *labels*. These *labels* help to distinguish different kinds of transitions, such as guard, synchronization, update or broadcast [25]. Finally, all the modules are assembled together as a group using the broadcast feature of UPPAAL.

5.3. Property development

According to the formal models, security properties are developed in natural language first, then formal specifications are constructed with the knowledge of FSMs of the SoC and AXI4-Lite bus protocol.

5.3.1. Information leakage Trojan detection

For the information leakage attack, Master A first reads data from Slave A and then writes the data to Slave B, and the data contains sensitive information which should not be leaked to other slaves. The Trojan is triggered by a specific write address, and Slave Trojan steals the data that belongs to Slave B and will not give out any response signal. Properties are developed to prevent information leakage attack. In natural language, the property is stated as "*Information written to Slave B should never be leaked to other IPs*". During the property development and verification process, the first verification should be carried out to check whether there is any risk in the design, and the second step is supposed to be locating where the risk comes from. If there are more than one Trojan infected slave modules in the system, the source of the risk can still be located by printing out the search paths, denoted as equation (11), by UPPAAL, because the Trojans are not identical on either of triggering conditions or of payload functions.

Slave B first recognizes that the address in the address channel belongs to its own address range, then Slave B begins to receive data from the data channel. During this operation, Slave B reaches a data receiving related state in its FSM. At the same time, Slave Trojan also recognizes the address in the address channel and the Trojan in Slave Trojan gets activated, then the Trojan steals the data from the bus that belongs to Slave B. Because Slave Trojan can pass the normal functional checking, then it should follow the normal bus protocol, thus Slave Trojan also reaches a similar data receiving state in its FSM. However, in order to prevent information leakage, the above two states cannot be reached at the same time. More specifically, only one slave is permitted to receive data from the data channel when one write address is sent to the address channel. Therefore, the above security property can be constructed as equation (13). In this expression, state formula *slaveX_selected* represents the state of slave *X* that is ready to receive data from the data channel. The path formula $E <> \phi$ represents that there at least exits a path that satisfies ϕ.

$$E <> \ (slaveT\_selected \ and \ slaveB\_selected) \tag{13}$$

In the SoC model of UPPAAL, states are labeled with the name of modules and binary values. More specifically, before the binary values, *Par*, *Pm*, *Pmt*, *Pa*, *Pb* and *Pt* represent Arbiter, Master A, Master Trojan, Slave A, Slave B and Slave Trojan modules in UPPAAL, respectively. Further, in the established model, the state relevant to data receiving is numbered as 00111100000. As all the modules in the system follow the same AXI4-Lite interface, the same binary value *id*00111100000 is utilized to represent the ready state of receiving data for all three slaves. Then the security property discussed in (13) is elaborated as the following equation (14).

$$E <> \ (Pt.id00111100000 \ and \ Pb.id00111100000) \tag{14}$$

If the property given in equation (13) is violated, another set of security properties is utilized to identify the source of attacks. From the description of attacks and the stealthy nature of Trojans, Slave B provides response after receiving data from Master A while Slave Trojan does not. This response given by a slave *X* is mapped to a state, *slaveX_response*, of the FSM. When the state *slaveX_selected* is reached, the behavior of response is equivalent to the judgment "*the state slaveX_response is reachable from slaveX_selected*". Hence, the above security property is formally stated as equation (15).

$$\begin{aligned} &slaveT\_selected \ \rightarrow \ slaveT\_response \\ &slaveB\_selected \ \rightarrow \ slaveB\_response \end{aligned} \tag{15}$$

In this expression, the path formula $\phi \rightarrow \psi$ represents that when the state formula ϕ is true, then the state formula ψ will also be true. Also, the states *slaveX_selected* and *slaveX_response* are numbered as *id*00111100000 and *id*00011100001, respectively. The corresponding security properties discussed in equation (15) are elaborated as equation (16).

$$\begin{aligned} &Pt.id00111100000 \ \rightarrow \ Pt.id00011100001 \\ &Pb.id00111100000 \ \rightarrow \ Pb.id00011100001 \end{aligned} \tag{16}$$

5.3.2. Denial of service attack detection

In the normal operations of a SoC system, the master modules are granted permission by arbiter only when the masters need to access the bus. During a DoS attack, Master Trojan keeps requesting the permission from Arbiter, and blocks other masters from using the bus for an indefinite period of time. However, there are possibilities that the normal operated master uses the bus for a certain period of time, which could cause a high false positive rate if only one master's behavior is considered. To address this issue, all master modules in the system are taken into account in developing the system-level security properties. In natural language, the property is that "*The bus permission should never be given to only one specific master continuously while other masters are requesting permission*". During the property development and SoC verification, the first step is to check whether there exists a master that keeps requesting permission from the arbiter, and the next step is to check whether other masters require the permission simultaneously.

As a normal operation, a master first requests permission from the arbiter and then the arbiter decides whether to give the master access to the bus. During functional testing, the Trojan is not activated, and Master Trojan only requests permission when needed according to the normal function. However, when the Trojan in Master Trojan module is activated, Master Trojan module keeps requesting permission from Arbiter. If Master Trojan has a high priority settings, Arbiter will not permit the requests from other master modules, like Master A, thus causing a DoS or malfunction of the entire SoC. In order to prevent such attack, security properties needs to be developed to check whether there are infinite loops that involve between the request-permission pair of Master Trojan module and Arbiter. The above security property is constructed as equation (17). In this expression, the state formula *masterX_request* represents the state that master X is requesting permission from Arbiter, and *masterX_permission* represents that Arbiter gives the permission to master X. The path formula $E[]\phi$ represents that there is at least an execution path in which ϕ holds for all states in the path. The *deadlock* indicates that whether there is any infinite loop in the model.

$$\begin{aligned} &E[] \ deadlock \ imply \\ &(masterX\_request \ and \ masterX\_grant) \end{aligned} \tag{17}$$

Restricted by the expression ability of the query language [25] of UPPAAL, the property demonstrated in equation (17) is decomposed into several specific formal properties. In the established model, the state when Arbiter gives out permission signal is denoted as *idar*11010010. The permission signal receiving states in Master A and Master Trojan are denoted as *Pm.idm*10010000000 and

*Pmt.idmt*10010000000, respectively. The decomposed security properties are given in the equation (18). The reachability of each permission signal receiving state is checked first, then the loop between permission request and response is checked to determine the trustworthiness of the modules.

$$\begin{aligned} Par.idar11010010 &\rightarrow Pm.idm10010000000 \\ Pm.idm10010000000 &\rightarrow Pm.idm10010000000 \\ Par.idar11010010 &\rightarrow Pmt.idmt10010000000 \\ Pmt.idmt10010000000 &\rightarrow Pmt.idmt10010000000 \end{aligned} \tag{18}$$

If the property given in equation (17) is violated, it means that there exists a situation where one master keeps occupying Arbiter. The next step is to check whether there are any other masters trying to request the permission at the same time. The supplementary property is demonstrated in equation (19). In this expression, the state formula *master_request* represents the state that a master is requesting the permission from Arbiter, and $A <> \phi$ represents that in the whole system, the state ϕ is reachable. After verifying this property in UPPAAL, traces for state transitions are printed out for checking according to the bus protocol, and the relevant states can also be verified.

$$A <> \ master\_request \tag{19}$$

6. Experimental results

After a design is formalized and properties are proposed, the model checker verifier of UPPAAL *verifyta* [27] is utilized for verification on a server. The server is equipped with 4 CPU cores and 8 GB memory, using Linux Red Hat distribution as the operating system. Totally, 5 properties are designed for information leakage Trojan detection, 4 properties are designed for DoS attack detection. The results are shown in Table 2. Note that in Table 2, Slave T represents Slave Trojan and Master T represents Master Trojan.

6.1. Information leakage Trojan detection results

Three properties are designed for identifying the Trojan's existence and two properties are designed for Trojan localization. Property details and experimental results are given in Table 2. For Trojan detection properties, satisfaction of property in the first line of Table 2 means that the data receiving state can be reached at the same time for Slave Trojan and Slave B. In the second and third line, properties are not satisfied, which implies that the data receiving state cannot be reached at the same time for Slave A and Slave B, and for Slave A and Slave Trojan. These results show that: 1) an attack occurred, and 2) either Slave Trojan or Slave B is the malicious IP. To validate the satisfaction of these properties, the system states given in Table 3 are checked. During the information leakage attack, Slave Trojan's state is the same with Slave B, meaning that both Slave Trojan and Slave B are in the data receiving state. While Slave A's state proves that Slave A is indeed not in the data receiving state.

Table 3
System states.

Attacks	Modules	States ID
Information leakage	Slave A	Pa.id00000000000
	Slave B	Pb.id00111100000
	Slave Trojan	Pt.id00111100000
Denial of service	Arbiter	Par.idar11010010
	Master A	Pm.idm10010000000
	Master Trojan	Pmt.idmt10010000000

Next, two Trojan localization properties are checked after Slave B and Slave Trojan reach the data receiving state at the same time. The satisfaction result implies that the response state is reached in Slave B while the dissatisfaction result demonstrates the response state is unreachable in Slave Trojan. So the conclusion is that the attack is caused by Slave Trojan because of Slave Trojan's malicious behavior. Our method can also print out the states of signals under certain conditions, so to validate the results of Trojan localization, the relevant signals when the attack occurred are checked. *s*00*_axi_awvalid* indicates that Master A is signaling a valid write address and control information. *slave_axi_wvalid* indicates that valid write data is available. *s*00*_axi_awaddr* represents the write address. Additional, the write address allocated to Slave B in bus system is 0111. By analyzing the value of those signals in equation (20), it is confirmed that the data was supposed to be written to Slave B legally when the attack occurred. In other words, information of Slave B is leaked under attack.

$$\begin{aligned} s00\_axi\_awvalid &= 1 \\ slave\_axi\_wvalid &= 1 \\ s00\_axi\_awaddr &= 0111 \end{aligned} \tag{20}$$

6.2. Denial of service attack detection

Four properties are proposed for denial of service attack detection, and one manual check is performed. Property details and experimental results are given in Table 2. Two sets of properties are designed to check whether there exists an infinite loop that Master A or Master Trojan keeps occupying the bus. Within each set of the properties, first we check the normal operation that the arbiter gives permission to either masters, then we check the loop between the master request and arbiter permission. In the experiments, all paths that involve the master request and arbiter permission states are checked. From the results, it is possible that Master Trojan constantly requests permission from Arbiter, and it is confirmed that the Trojan's trigger is involved in this situation

Table 2
Security Property Checking results.

Attacks	Properties	Formal Property	Results	Time(sec)
Information leakage	Concurrent data read by Slave T and B	E <> (Pt.id00111100000 and Pb.id00111100000)	Satisfied	97.82
	Concurrent data read by Slave A and B	E <> (Pa.id00111100000 and Pb.id00111100000)	Not Satisfied	100.04
	Concurrent data read by Slave T and A	E <> (Pa.id00111100000 and Pt.id00111100000)	Not Satisfied	98.42
	Slave T's response exists after data receiving	Pt.id00111100000 – > Pt.id00011100001	Not Satisfied	99.43
	Slave B's response exists after data receiving	Pb.id00111100000 – > Pb.id00011100001	Satisfied	101.69
Denial of service	Loop between Master A request and Arbiter permission	Par.idar11010010 – > Pm.idm10010000000	Satisfied	88.70
		Pm.idm10010000000 – > Pm.idm10010000000	Non-loop	N/A[a]
	Loop between Master T request and Arbiter permission	Par.idar11010010 – > Pmt.idmt10010000000	Satisfied	87.07
		Pmt.idmt10010000000 – > Pmt.idmt10010000000	Loop	N/A[a]
	Master A permission state reachability	A <> Pm.idm10010000000[b]	Not Satisfied	N/A[a]

[a] Checking time not applicable.
[b] Under the condition where the DoS attack happens.

after printing out the searching path. These results show that: 1) a DoS attack happened, and 2) the normal operation of Master A module is not influenced when Master Trojan is not triggered. The system states when the DoS attack happens are shown in Table 3. The Arbiter is in the state granting the permissions to master modules, while both Master A and Master Trojan are in the state waiting for permission.

Then we apply the same conditions in our model and check whether Master A can get the permission from Arbiter when the DoS attack happens. To validate the results of DoS attack, the relevant signals when the attack happens are checked. More specifically, *txn_done*1 and *txn_done*2 indicate that Master A and Master Trojan are permitted, respectively. *m_axi_awvalid*1 and *m_axi_awvalid*2 represent the permission request in Master A and Master Trojan, respectively. The states are demonstrated in equation (21). By analyzing the value of those signals, there exist conflict between Master A and Master Trojan, and the permission is only issued to Master Trojan. More specifically, during normal bus operations, if the DoS attack happens, Master A will never enter a bus permission state during the attack. A DoS Trojan is detected in the SoC system.

$$\begin{aligned} &txn\_done1 = 0 \\ &txn\_done2 = 1 \\ &m_a xi_a wvalid1 = 1 \\ &m_a xi_a wvalid2 = 1 \end{aligned} \tag{21}$$

6.3. Discussions

Our method focuses on verification of the SoC system before the implementation stage and no hardware modification of the original circuit is needed. Compared with the state-of-the-art formal verification based techniques, our method performs better in several aspects: scalability, flexibility, and physical overhead. The proposed approach is scalable for large SoC designs, and the scalability is guaranteed by the model methods and verification strategies. In future work, we may explore proper algorithms that are able to distinguish and remove normal states before the verification process, thus alleviating potential state explosion issue. As the interactions between different IPs at system level are considered and all vulnerabilities along with hardware designs are mapped to states in our model, this paper provides a flexible solution to address a wide range of threats in design stage. When facing other types of potential hardware Trojan threats, the behaviors of Trojans' trigger conditions and payload functions will be analyzed first according to the bus protocols. Security properties will then be developed to specify the Trojans' malicious logic utilizing the states in the formalized model. Finally, the security properties are verified for Trojan detection.

At the meantime, there is still room for further improvements in our framework. In detail, the success in detecting the security vulnerabilities relies on several aspects. First is the integrity of the security properties. For a security threat, there exist several aspects that can be utilized for property development. Theoretically, any security properties that have been verified should be comprehensive enough for judging whether there are any threats. However, for threats that are not included in the properties, the method may not find the vulnerabilities. Second is the checking engine and verification strategy, the UPPAAL verification tool resorts to a width first strategy, so every time when the property satisfies, the tool returns a path for the results. The success of Trojan detection is based on the verification of well defined security properties. In our paper, we have covered almost every aspects of the mentioned vulnerabilities. We also set the tool to search and verify every possible paths until all paths that satisfies the property are returned. Third is that some of the properties can not be fully expressed using query language, we should exploit other formal language in the future for better expression ability.

7. Conclusion and future work

In this paper, a formal verification approach is proposed to protect the entire SoC system from threats of untrusted third-party IPs. FSMs are firstly extracted from gate-level netlist of a SoC, then security properties are constructed by analyzing possible attacks from Trojans in IPs and bus protocols. Finally, a model checker is used to verify whether these properties are satisfied. For demonstrating the effectiveness of our method, we consider a SoC with an AMBA bus and Trojans in some of the slave and master modules. Results show that our framework can detect and localize information leakage attack and DoS attack launched by hardware Trojans hidden in IPs of a SoC. As future work we intend to build a security property library to detect assorted hardware Trojans in various bus protocols.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China under Grant No. 61832018 and the China Scholarship Council under Grant No. 201606250061.

References

[1] Amba specifications, https://www.arm.com/products/system-ip/amba-specifications, (Accessed 21 May 2018).
[2] G. Mosensoson, Practical approaches to soc verification, in: Proceedings of DATE User Forum, Citeseer, 2002, pp. 05–08.
[3] H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level, in: Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on, IEEE, 2013, pp. 190–195.
[4] J. Mead, C. Bobda, T.J. Whitaker, Defeating drone jamming with hardware sandboxing, in: Hardware-Oriented Security and Trust (AsianHOST), IEEE Asian, IEEE, 2016, pp. 1–6.
[5] A. Basak, S. Bhunia, T. Tkacik, S. Ray, Security assurance for system-on-chip designs with untrusted ips, IEEE Trans. Inf. Forensics Secur. 12 (7) (2017) 1515–1528.
[6] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, R. Kastner, Information flow isolation in i2c and usb, in: Proceedings of the 48th Design Automation Conference, ACM, 2011, pp. 254–259.
[7] X. Guo, R.G. Dutta, J. He, Y. Jin, Pch framework for ip runtime security verification, in: Hardware Oriented Security and Trust Symposium (AsianHOST), 2017 Asian, IEEE, 2017, pp. 79–84.
[8] M.T. Oliveira, A.J. Hu, High-level specification and automatic generation of ip interface monitors, in: Proceedings of the 39th Annual Design Automation Conference, ACM, 2002, pp. 129–134.
[9] H. Lin, C. Yen, C. Shih, J. Jou, On compliance test of on-chip bus for soc, in: Proceedings of the 2004 Asia and South Pacific Design Automation Conference, IEEE Press, 2004, pp. 328–333.
[10] J. Coburn, S. Ravi, A. Raghunathan, S. Chakradhar, Seca: security-enhanced communication architecture, in: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM, 2005, pp. 78–89.
[11] L.W. Kim, J.D. Villasenor, et al., A trojan-resistant system-on-chip bus architecture, in: Military Communications Conference, 2009. IEEE, IEEE, 2009, pp. 1–6.
[12] H. Shen, Y. Zhao, Htchecker: detecting hardware trojans based on static characteristics, in: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), 2017, pp. 1–4, https://doi.org/10.1109/ISCAS.2017.8050674.
[13] K. Hasegawa, M. Yanagisawa, N. Togawa, Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier, in: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), 2017, pp. 1–4, https://doi.org/10.1109/ISCAS.2017.8050827.
[14] H. Salmani, Cotd: reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist, IEEE Trans. Inf. Forensics Secur. 12 (2) (2017) 338–350, https://doi.org/10.1109/TIFS.2016.2613842.
[15] X. Chen, Q. Liu, S. Yao, J. Wang, Q. Xu, Y. Wang, Y. Liu, H. Yang, Hardware trojan detection in third-party digital intellectual property cores by multilevel feature analysis, IEEE Trans. Comput. Aided Des. Integrated Circ. Syst. 37 (7) (2018) 1370–1383, https://doi.org/10.1109/TCAD.2017.2748021.
[16] A. Waksman, S. Sethumadhavan, J. Eum, Practical, lightweight secure inclusion of third-party intellectual property, IEEE Design Test 30 (2) (2013) 8–16.
[17] J. Portillo, E. John, S. Narasimhan, Building trust in 3pip using asset-based security property verification, in: VLSI Test Symposium (VTS), 2016 IEEE 34th, IEEE, 2016, pp. 1–6.
[18] J. Rajendran, A.M. Dhandayuthapany, V. Vedula, R. Karri, Formal security verification of third party intellectual property cores for information leakage, in: VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), 2016 29th International Conference on, IEEE, 2016, pp. 547–552.

[19] A. David, W. Yi, Modelling and analysis of a commercial field bus protocol, in: Real-time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on, IEEE, 2000, pp. 165–172.
[20] A. Goel, W.R. Lee, Formal verification of an ibm coreconnect processor local bus arbiter core, in: Proceedings of the 37th Annual Design Automation Conference, ACM, 2000, pp. 196–200.
[21] A. Roychoudhury, T. Mitra, S. Karri, Using formal techniques to debug the AMBA system-on-chip bus protocol, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2003, pp. 828–833.
[22] X. Guo, R.G. Dutta, P. Mishra, Y. Jin, Scalable soc trust verification using integrated theorem proving and model checking, in: Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on, IEEE, 2016, pp. 124–129.
[23] R. Torrance, D. James, The state-of-the-art in ic reverse engineering, in: CHES, vol. 5747, Springer, 2009, pp. 363–381.
[24] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT press, 1999.
[25] Uppaal, http://www.uppaal.org/, (Accessed 21 May 2018).
[26] T. Meade, S. Zhang, Y. Jin, Netlist reverse engineering for high-level functionality reconstruction, in: Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific, IEEE, 2016, pp. 655–660.
[27] A tutorial on uppaal 4.0 (2006), http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, (Accessed 21 May 2018).
[28] E.S. Shin, V.J. Mooney III, G.F. Riley, Round-robin arbiter design and generation, in: Proceedings of the 15th International Symposium on System Synthesis, ACM, 2002, pp. 243–248.
[29] Axi interconnect, https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf, (Accessed 21 May 2018).