ASAX: Automatic Security Assertion Extraction for Detecting Hardware Trojans

Chenguang Wang, Yici Cai, Qiang Zhou and Haoyi Wang
Tsinghua National Laboratory for Information Science & Technology
Department of Computer Science & Technology, Tsinghua University, Beijing, China
Email: {wang-cg13, why16}@mails.tsinghua.edu.cn, {caiyc, zhouqiang}@mail.tsinghua.edu.cn

***Abstract*—Hardware Trojans (HT) has been one of the major concerns of IC designers, and formal methods have been applied to the HT detection. In general, the assertions for detecting HT are *manually* defined, which is time-consuming and error-prone even for an expert engineer. However, there is a lack of studies on the *automatic* definition for security assertions. To fill in this gap, we propose an automatic security assertion extraction (ASAX) tool. ASAX labels the candidate signals and infers the proposed register transfer level (RTL) invariants from simulation traces. Next, the security assertions are mined from the inferred RTL invariants. By adopting a two-step invariants inferring technique, ASAX can extract high-coverage assertions with a low runtime. We validate the effectiveness and efficiency of ASAX through experiments on the benchmarks from Trust-hub, DeTrust and OpenCores. The results show that the HT can be 100% detected by model checking with the extracted security assertions.**

I. Introduction

Despite decades of research on hardware security, today's computing infrastructure remains vulnerable because the traditional design and verification do not consider or prioritize the emerging security issues. With the development of system-on-chip (SoC), IC designers integrate the intellectual property (IP) cores from third-party vendors for time-to-market and cost reasons. The rogue elements there can insert hardware Trojans (HT) in the IP cores, resulting in various malicious effects, e.g. change of functionality and information leakage [1].

To detect the existence of HT at pre-silicon stage, formal verification has been applied to checking whether the security assertions hold on the design under verification (DUV) for its exhaustive and golden-reference-free advantages. According to the types of assertion-based formal verification, the methods can be classified into theorem proving, model checking, etc.

However, there exist limitations of the methods. One of the most significant is that the definition of security assertions is time-consuming and error-prone even for an expert engineer, which is due to 1) lack of formulated security specification and 2) requirement of substantial expertise in formal languages, e.g., Linear Temporal Logic (LTL) and Property Specification Language (PSL). Besides, the procedure captures only general and high-level assertions rather than the DUV-specified ones, reflecting what the engineers think and worry about the most. As a result, the limitations hinder formal methods from coping with the increasing threat of HT.

This project is supported by National Natural Science Foundation of China (NSFC) under Grant No. 61774091.

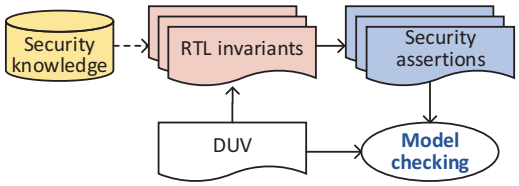

Fig. 1: Basic concept of the proposed ASAX.

To overcome the limitations, a key insight is the automatic assertion extraction in the realm of *functional verification* for software/hardware. Most of these attempts are inspired by the program invariants, i.e., the properties that hold on all the simulation traces of a program [2]. Researchers have proposed various methods to automatically extract functional assertions in static (analyzing source code), dynamic (mining simulation traces), and combinational manners [3].

We tune the program invariants to register transfer level (RTL) and define the *RTL invariants* to formulate the security domain-specified knowledge, i.e., low controllability of HT-infected signals [4]. Compared with program invariants, RTL invariants are only related to single signals with the dynamic transitions. Based on RTL invariants, we propose the tool of Automatic Security Assertion eXtraction (**ASAX**).

Fig. 1 depicts the basic concept of ASAX. We adopt a two-step inferring method for invariants: 1) The candidate signals are labeled by the HT features with a few simulation traces. 2) The RTL invariants related to the candidate signals are inferred from a larger number of traces. By this technique, ASAX can extract a complete set of RTL invariants in a reasonable runtime, which helps keep pace with the increasing scale of modern chips. Next, the corresponding security assertions are mined with predefined templates, which can also be defined by the user to achieve design-specified assertions.

Specifically, this paper makes the following contributions:

1) To formulate the features of HT, i.e. low controllability of HT-infected signals, we define the RTL invariants using the security domain-specified knowledge.
2) Based on RTL invariants, a tool (ASAX) for invariants inferring and assertion mining is proposed to automatically extract the security assertions for detecting HT.
3) The effectiveness and efficiency of ASAX are validated through experiments on the benchmarks from Trust-hub [5], DeTrust [6] and OpenCores [7].

978-1-5090-0602-1/18/$31.00 ©2018 IEEE

The rest of this paper is organized as follows. Section II summarizes the related works on assertion-based HT detection as well as automatic assertion extraction and then discusses how ASAX differs from them. Section III deliberates the RTL invariants. The details of ASAX are introduced in Section IV. Section V presents the experimental results. The conclusion and discussion are in Section VI.

II. Related Work and Motivation

A. Assertion-based HT Detection Methods

Theorem proving based methods. Love *et al.* present an acquisition technique of provably trustworthy IP in [8], which draws upon the advances in the realm of proof-carrying code (PCC). Further, Jin *et al.* introduce a proof-carrying based framework for asserting the trust of IP, particularly the microprocessor cores [9]. Recently, an automated proof-carrying IP framework for information flow policies is proposed in [10]. Guo *et al.* propose to combine an automated model checker with an interactive theorem prover for proving the security properties [11].

Model checking based methods. Zhang *et al.* present a comprehensive tool with property checking, automatic test pattern generation (ATPG), etc., to detect HT [12]. In [13], Rathmair *et al.* propose to apply various formal methods including model checking in a combinational manner to increase the introduced "Trojan Assurance Level", which is a metric for assessing the HT inexistence. Rajendran *et al.* develop the properties to detect HT which corrupt critical data [14]. Further, the properties for detecting HT which leak information are defined in [15]. In [16], Ngo *et al.* propose a synthesizable assertions module, i.e., Hardware Property Checker (HPC), which verifies the permitted and prohibited behaviors of IC at the post-silicon stage.

In general, the results of assertion-based detection methods are dominated by the quality of security assertions, which were manually defined by the IC designers. To keep pace with the increasing threat of HT and scales of designs, there is an urgent need for automatic extraction for security assertions.

B. Automatic Assertion Extraction

Fig. 2 illustrates the model checking flow with automatic assertion extraction. The assertions can be extracted statically by analyzing the DUV source codes, dynamically by mining the simulation traces of DUV, or in a hybrid manner [3].

Static methods. A system based on block-level structural analysis of the design is presented in [17]. Recently, Malburg *et al.* propose a technique by adopting dynamic dependency graphs which are extracted from the source code [18].

Dynamic methods. A well-known tool is Daikon [2], which searches for values that satisfy a set of property templates. As the first attempt to infer invariants for hardware designs, IODINE [19] hypothesizes a set of candidate invariants across one or more variables in the DUV. Dianosis [20] checks the predefined properties leaving only candidates that are valid on the traces, which are recombined until no new candidates can be created. Inferno [21] analyzes the execution data to infer

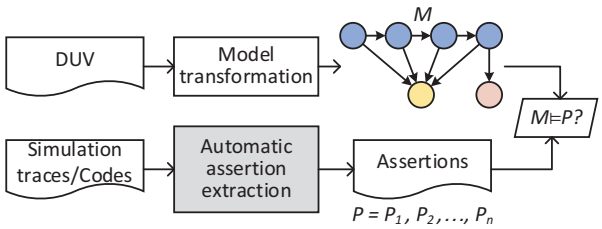

Fig. 2: Model checking with automatic assertion extraction.

transactions, i.e. sequences of signal valuations that appear several times in the execution data and start/end in some boundary valuations. Vasudevan *et al.* propose GoldMine [22] which involves a combination of static analysis and decision tree based supervised learning of the simulation data. Danese *et al.* propose a proposition and assertion miner to generate the temporal specification in the PSL format, exploiting both static and dynamic techniques [23]. Further, Danese *et al.* propose the A-TEAM tool to dynamically extracts generic LTL assertions from a set of DUV execution traces, which works for both Boolean and numeric data types [3].

To summarize, static methods have the advantage of accuracy, while dynamic methods can provide better scalability for large designs. In this paper, we apply the dynamic method in a two-step manner to obtain an optimal trade-off between the accuracy and scalability.

C. Motivation

Limitation of the existing methods. It is hard, if possible, to cover all test cases with the rapid growth of design scales. Besides, the generated assertions depend on the predefined templates in most cases, thus the behavior of DUV cannot be 100% covered. As a result, the assertions fail to cover the corner cases. However, the uncovered corner cases are more valuable for security verification than typical case scenarios, because HT is usually stealthy to be activated. Another limitation is that a great number of simulations are required to offset the low coverage due to insufficient data, which may result in a high computational complexity.

Novelty of our approach. Our approach differs from the existing methods in three aspects. 1) ASAX extracts the security assertions for checking if the DUV contains HT rather than if the signal values conform to the specification, and the usage of security domain-specific knowledge help precisely extract the high coverage security assertions. 2) The candidate signals for which the assertions are extracted are automatically labeled, which is usually manually done in existing works. 3) We propose a two-step dynamic techniques to infer the RTL invariants to reduce the computational complexity.

III. Preliminaries

With the key insight of inferring RTL invariants that formulate the features of HT, a question can be asked, i.e., *how to define and infer the security-related RTL invariants?* In this section, the RTL invariants will be deliberated based on the security domain-specified knowledge.

RTL sample	Signal values
module top (	1000 samples
input clk,	(2 values)
input rst,	(2 values)
input [127:0] state,	(260 values)
input [127:0] key,	(2 values)
output [127:0] out,	(150 values)
output [63:0] Capacitance);	(27 values)*
module Trojan_Trigger(input rst, state, output Tj_Trig);	1000 samples
always @(rst, state)	
if (rst == 1) begin	
Tj_Trig <= 0;	(1 value)*
end else if (state == 128'h00112233_44556677_8899aabb_ccddeeff) begin	
Tj_Trig <= 1;	(1 value)*
end	
module lfsr_counter (input rst, clk, Tj_Trig, data, output lfsr);	1000 samples
assign d0 = lfsr_stream[15] ^ lfsr_stream[11] ^ lfsr_stream[7] ^ lfsr_stream[0];	(2 values)*
always @(posedge clk)	500 samples
if (rst == 1) begin	
lfsr_stream <= data[19:0];	(38 values)*
end else	

Fig. 3: RTL sample and the corresponding signal values of AES-T1000 from Trust-hub [5] over a set of input patterns whose values are randomly generated. The HT-infected signals are labeled with "*" which have low controllability.

A. Security Domain-specified Knowledge

Typically, hardware Trojans can be divided into two categories according to the activation mechanism: *always-on* and *conditional-triggered* [1]. Moreover, a Trojan consists of two components: *trigger* and *payload*. Both the always-on and conditional-triggered HT types contain payload, while only the conditional-triggered HT type contains trigger.

To disguise the existence of HT, the trigger component of HT is usually designed to be connected to the nets with low switching activity to create a rare activation pattern, and the payload component might re-stitch the nets whose deviations are not readily observable to relieve the hardware symptoms [4]. As a result, it can be expected that the HT-infected signals present very low switching activity, i.e., low controllability, in the simulation traces of DUV, which can be adapted to detect the existence of HT.

Fig. 3 depicts the low-controllability characteristics of the HT-infected signals, which is obtained by a dynamic analysis for simulation traces that will be introduced in Section IV. To detect the HT, the features of HT, i.e. low controllability, will be formally expressed using the proposed RTL invariants.

B. RTL Invariants

As mentioned above, program invariants are adopted in the software verification flows to help the engineers understand, modify, and test the software. In general, program invariants are defined as the properties that hold true on all the simulation traces of a program, which are usually formulated in the format of expressions of one or multiple variables [2]. Besides, several works have adopted the concept of invariants in the hardware functional verification flows.

Type of RTL invariants	Inferred RTL invariants (RI)
C	RI_1: Tj_Trig=0
S	RI_2: clk∈{0,1} RI_3: rst∈{0,1} RI_4: state∈{0,1,128'h3243f6a8_885a308d_313198a2_e0370734} RI_5: key∈{0,1,128'h2b7e1516_28aed2a6_abf71588_09cf4f3c} RI_6: d0∈{0,1}
R	RI_7: lfsr_stream≥0 RI_8: Capacitance≥1
U	RI_9: out

Fig. 4: RTL invariants inferred from the sample in Fig. 3.

In our approach, to express the security domain-specified knowledge, we define the RTL invariants as follows:

- **RTL invariant** is the property related to a single signal that holds true on all the simulation traces of RTL design.

RTL invariants differ from the traditional program invariants in the following aspects. 1) RTL invariant is inferred for security verification, while traditional invariant for functional verification. 2) RTL invariant is only related to a single signal (the constraint that holds on its values over time), while the traditional program invariant is related to multiple variables (the relationship among the values of multiple variables) in most cases. 3) Even for a single signal/variable, the dynamic transition is the most concerned in RTL variant, while the static valuation relationship is the most concerned in the traditional program invariant. As a result, the capability to express the dynamic transition of a single signal of RTL invariants can be used to formulate the specific characteristics of HT.

One of the principal challenges to inferring the invariants is choosing what invariants to infer [2]. Based on the security domain-specified knowledge, we check for invariants involving a single signal, which can be constant value (*C*), set of value (*S*), range limits (*R*), and uninitialized (*U*). RTL invariants over a single signal, where x is a signal, and a, b, c are computed constants from the simulation traces:

- **RTL invariant *C*** (constant value): $x = a$ indicates the signal x is a constant.
- **RTL invariant *S*** (set of value): $x \in \{a, b, c\}$ indicates x takes on only a small number of different values.
- **RTL invariant *R*** (range limits): $x \geq a, x \leq b$ or $a \leq x \leq b$ indicate the minimum and maximum values.
- **RTL invariant *U*** (uninitialized): $x = uninit$ indicates the signal is never set.

For each signal, a potential RTL invariant is inferred and tested. For example, given signal key, a potential RTL invariant is checked by examining each sample in turn; a sample is a tuple of values for the instrumented signals at an RTL point, stemming from one execution of that RTL point. Once a sample violating the invariant is encountered, the invariant is known not to hold and is not checked for any subsequent samples, while other invariants may continue to be checked.

Fig. 4 shows the RTL invariants inferred from the sample in Fig. 3, which is obtained by a dynamic inferring procedure that will be introduced in Section IV.

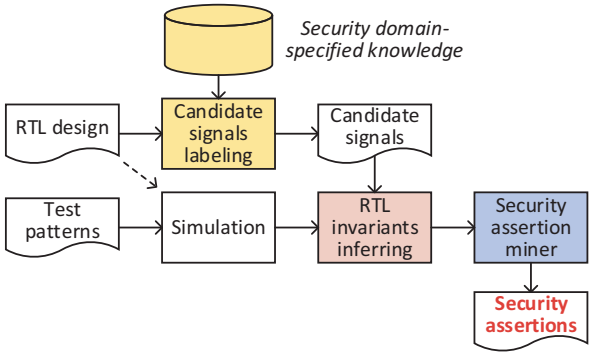

Fig. 5: Flowchart of the proposed ASAX.

IV. Implementation

In general, ASAX is a dynamic framework to overcome the limitations of existing assertion extraction methods. As mentioned before, we adopt a two-step technique to infer the RTL invariants. First, we label the hard-to-activate and hard-to-propagate signals as candidate signals by analyzing a small number of simulation traces. Next, a larger number of simulation traces are subjected to a dynamic analyzer to infer the RTL invariants of the labeled signals, according to a set of temporal patterns with respect to the features of HT. By this two-step technique, the computational complexity is significantly reduced because most of the genuine signals are identified by the candidate signals labeling. However, HT is stealthy and usually evades a traditional functional verification. As a result, a third step is required to map the RTL invariants to security assertions to exhaustively check if the extracted security assertions hold true on the DUV.

Fig. 5 provides the flow chart of ASAX, which is introduced as in *A*, *B*, and *C*. Furthermore, the security-related techniques will be presented in *D*.

A. Candidate Signals Labeling

To reduce the searching space of the RTL invariants inference, the candidate signals, i.e. the signals that are suspicious to be HT-infected, are labeled prior to the RTL invariants inference using dynamic analysis techniques. Based on the aforementioned security domain-specified knowledge, candidate signals are defined as:

- **Candidate signal** is the RTL signal that is hard to excite and/or propagate during the simulation.

As shown by the example in Fig.3 and the other experiments, it can be noted that all the HT-infected signals have the low-controllability and low-observability characteristics. As a result, we can label the signals by monitoring the signal values in simulation traces. By the candidate signals labeling, most of the genuine signals are removed from the candidate list of RTL invariants reference. However, a few genuine signals, e.g. the clock `clk` and reset signal `rst`, also have the low-controllability characteristics, thus are labeled as candidate signals. The candidate signals are labeled by dynamic analysis on simulation traces, which is performed by the Mentor ModelSim tool. The candidate signals can be used to provide some useful information which will guide the extraction step of the security assertions. By the candidate signals labeling, the most likely RTL invariants can be selected because it tends to eliminate the potential RTL invariants that may be coincidences due to inadequate testing.

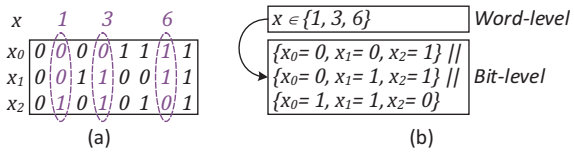

Fig. 6: Decomposition example of word-level RTL invariant to bit-level one. (a) The word-level (3-bit) signal x has a value set of $\{1,3,6\}$. (b) The obtained bit-level RTL invariant.

B. RTL Invariants Inference

We infer the RTL invariants by monitoring the simulation traces of the DUV for different input patterns and identifying properties that hold true on each execution.

Initially, the DUV is simulated using a test suite consisting of manually defined or randomly generated test patterns. If regression tests or workloads for the design, i.e. in the format of Value Change Dump (VCD), are available, they can be used to obtain the simulation traces. Otherwise, ASAX generates its own set of simulation traces using randomly generated input patterns. Besides, it is assumed that the DUV has passed the existing functional verification flows, which ensures that the RTL invariants are inferred only over the correct behavior.

Next, the generated simulation traces are analyzed by the external Daikon tool [2] to infer RTL invariants. In general, the information generated during the simulations is recorded in the format of industry-standard simulation dumps, e.g. VCD, thus a program is developed to translate the simulation dumps into formats that are supported by Daikon. As a result, this keeps ASAX agnostic to details such as the hardware description language (HDL) of the DUV and whether the DUV is modeled at RTL, making it possible to extend the ASAX to gate-level. Moreover, the user can manually specify the clocking and reset scheme for the design. At the end of inference, the analyzer emits the inferred RTL invariants into a database from which the security assertions will be mined.

C. Security Assertions Miner

In the last phase of ASAX, the inferred RTL invariants are mapped to a set of predefined temporal patterns to create the security assertions.

However, the inferred RTL invariants are at the word-level. They are required to be decomposed to the bit-level before mining the assertions in PSL because the temporal layer PSL is based on Boolean layer PSL. First, the N-bit word-level signal of x is decomposed to $\{x_1, x_2, ..., x_N\}$. Next, the value of RTL invariants is decomposed to the bit-level signals, generating the bit-level RTL invariants. An example of the RTL invariants II is presented in Fig. 6.

Given the RTL invariants related to signal x, the considered patterns are the following[1], where $[0:T]$ indicates the clock cycles in the 0 to T interval:

- **Security assertion *C*** (constant value, $x = a$):
  $always\ (rst- > next\_a[0:T]\ (x_1))\ (if\ a_1 == 1)$;
  $always\ (rst- > next\_a[0:T]\ (!x_1))\ (if\ a_1 == 0)$;
- **Security assertion *S*** (set of value, $x \in \{a, b, ..., z\}$):
  $always\ (rst- > next\_a[0:T]\ (x_0 == a_0\ \&\&\ x_1 == a_1\ \&\&\ x_2 == a_2))$;
  $always\ (rst- > next\_a[0:T]\ (x_0 == b_0\ \&\&\ x_1 == b_1\ \&\&\ x_2 == b_2))$;
  .....
  $always\ (rst- > next\_a[0:T]\ (x_0 == z_0\ \&\&\ x_1 == z_1\ \&\&\ x_2 == z_2))$;
- **Security assertion *R*** (range limits, $a \leq x \leq b$):
  $always\ (rst- > next\_a[0:T]\ (x_0 == a_0\ \&\&\ x_1 == a_1\ \&\&\ x_2 == a_2))$;
  $always\ (rst- > next\_a[0:T]\ (x_0 == (a+1)_0\ \&\&\ x_1 == (a+1)_1\ \&\&\ x_2 == (a+1)_2))$;
  .....
  $always\ (rst- > next\_a[0:T]\ (x_0 == b_0\ \&\&\ x_1 == b_1\ \&\&\ x_2 == b_2))$;
- **Security assertion *U*** (uninitialized, $x = uninit$):
  $always\ (x_0)\ (if\ x_0 == 1)$;
  $always\ (!x_0)\ (if\ x_0 == 0)$;

For example, the signal `Tj_Trig` (in Fig.3 and Fig.4) is identified by RTL invariant ***C***, which is mapped to assertion by Security assertion ***C*** as follows. The extracted security assertion can check exhaustively if `Tj_Trig` has a constant value which indicates the low controllability.

```
property P_LOW_CNTROLLABILITY_RTL_INVARIANT_C =
always aes_start -> next_a[0:1000](!Tj_Trig);
assert P_LOW_CNTROLLABILITY_RTL_INVARIANT_C;
```

D. Security-related Techniques

The used security-related techniques are shown as follows.

Threshold of HT features. A significant difference between ASAX and the existing works is that we infer the security-related RTL invariants for a single signal with the HT features considered. It has been noted that the HT-infected signals have low controllability. However, it is hard to set the threshold of controllability to identify the HT-infected signals. In our approach, the threshold is set empirically by comparing the switching activities of HT-infected and the genuine signals in the simulation traces of the same benchmark.

Quantity of simulation traces. It can be noted that more simulation traces indicate higher coverage security assertions yet higher computation complexity. Fig.7 presents the trade-off between the simulation traces and the coverage of the generated assertions on the benchmarks. It can be noted that 1000 simulations are reasonable in most cases.

[1]Due to the limited space, the temporal patterns related to $x_2, ..., x_N$ can be easily obtained by modifying the presented pattern related to x_1. In addition, it is assumed that signal x and values $a, b, ..., z$ are 3-bit.

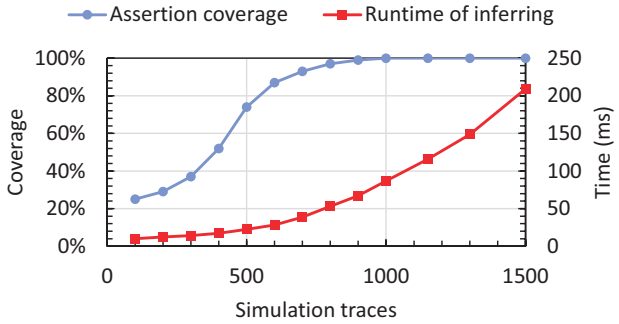

Fig. 7: Assertion coverage and runtime vary with the number of simulation traces. The coverage is 100% within 1000 traces, motivating the use of 1000 traces for our experiments.

TABLE I: Detection capability of ASAX.

Benchmark	Security Assertions				Cov.	Runtime	
	C	*S*	*R*	*U*		Ext. (s)	Ass. (ms)
AES-T100	0	6	0	2	100%	5.079	31.669
AES-T200	1	2	3	0	100%	4.995	25.298
AES-T300	0	9	1	2	100%	3.209	80.240
AES-T400	2	8	5	1	100%	4.425	43.496
AES-T600	3	0	3	3	100%	5.523	30.570
AES-T700	2	5	2	1	100%	2.278	30.891
AES-T1300	1	2	4	0	100%	4.886	30.250
AES-T2000	2	3	0	0	100%	3.607	27.834
BasicRSA-T100	4	10	3	2	100%	6.164	110.931
AES-T800	4	5	6	2	100%	3.491	70.289
AES-T900	1	7	6	3	100%	3.597	51.326
AES-T1600	3	17	9	0	100%	3.700	100.324
BasicRSA-T300	4	7	9	1	100%	4.465	86.597
PIC16F84	2	4	9	2	100%	3.059	68.789
Or1200_ctrl	0	6	3	1	100%	4.142	40.166

V. Experimental Results

To verify the effectiveness and efficiency of ASAX, we check both HT-infected and genuine benchmarks using the extracted security assertions by the model checker, Cadence Incisive Formal Verifier (IFV). The off-the-shelf benchmarks are from Trust-hub [5] and DeTrust [6] that have successfully defeated several HT detection methods. We also update the genuine RTL designs from OpenCores [7] with the strategy of DeTrust in mind. ASAX is implemented in C++ language and executed on a Linux server with Intel Xeon E5604 CPU @ 2.67GHz and 64GB RAM. A set of execution traces for 1,000 simulation instants has been generated for all benchmarks.

A. Detection Capability

Table I reports the results. The first column lists the name of benchmarks, and the next four columns present the number of extracted security assertions, which have been divided among the constant value (*C*), set of value (*S*), range limits (*R*), and uninitialized (*U*). The next column lists the assertion coverage. Finally, the last two columns report the time spent by external tool (Ext.) and assertion miner (Ass.), respectively.

It can be seen that the most time-expensive step of ASAX is the inferring for RTL invariants performed by Daikon, which is dominated by the initialization of internal data structure [23]. Thus, ASAX can scale to the future and larger designs.

TABLE II: Comparisons with the existing works. For the limitation of reported results, AES-T700 is selected as the reference benchmark because it is the most commonly used in the works. "N/A" indicates that no reported result is available.

Works	Assertion Extraction	AES-T700	
		Memory (MB)	Time (s)
Guo *et al.* [11]	Manual	N/A	N/A
Zhang *et al.* [12]	Manual	N/A	N/A
Rajendran *et al.* [14]	Manual	2058.24	40.600
Rajendran *et al.* [15]	Manual	4043.56	96.330
This paper	**Automatic**	**12.63**	**2.309**

Assertion coverage. The coverage of extracted assertion presents a major short-coming for program invariants as the assertions depend on the simulation traces which cannot 100% cover the behavior space. However, it can be seen that ASAX obtains a set of high coverage assertions. Due to the exhaustive capability to express the dynamic transition of a single signal, RTL invariants can capture the complete features of HT.

B. Comparison with Related Works

As is inspired by the theory of program invariants, ASAX differs from existing works in the realm of HT detection. Besides, a unified metric for HT detection capability is lacking. Thus, a fair and elaborate comparison is not currently possible. We conclude a rough comparison based on the reported results, as shown in Table II. It can be seen that the manual work in developing assertions is significantly relieved. Moreover, our technique incurs a lower memory and time consumption, which can be contributed to the proposed two-step technique for inferring the RTL invariants.

VI. Conclusion

In this paper, we explore the automatic security assertion extraction by the concept of program invariants. We investigate the security domain-specified knowledge and define the RTL invariants. Further, ASAX is proposed to infer the RTL invariants and mine the security assertions. Experimental results show that ASAX can extract a set of high coverage security assertions in a reasonable runtime, and the HT-infected and genuine RTL benchmarks can be successfully checked with 0 false positives and false negatives.

The ASAX technique has the advantages as follows. 1) It is a fully-automatic extraction tool for the security assertions, which significantly relieves the manual burden. 2) It has at least 2 orders of magnitude lower runtime, which can be contributed to the two-step RTL invariants inferring method. 3) The extracted assertions have a high coverage for the HT-infected designs with 0 false positives and false negatives.

Any attempt to automate the extraction of security assertions would result in improved productivity and quality with the rapid development of SoC and IP cores. In the future, we plan to investigate the RTL invariants with new and stealthier HT considered. Another future research would be to theoretically quantify the capability that each RTL invariant can distinguish HT-infected with genuine samples to provide an accurate score-based trust verification of IP cores.

References

[1] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 99–123, 2001.

[3] A. Danese, N. D. Riva, and G. Pravadelli, "A-TEAM: automatic template-based assertion miner," in *54th Annual Design Automation Conference, DAC*, 2017, pp. 37:1–37:6.

[4] H. Salmani, "COTD: reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist," *IEEE Trans. Information Forensics and Security*, vol. 12, no. 2, pp. 338–350, 2017.

[5] Trust-hub. http://www.trust-hub.org.

[6] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2014, pp. 153–166.

[7] OpenCores. http://www.opencores.org/.

[8] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[9] Y. Jin and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," in *IEEE/ACM International Conference on Computer-Aided Design, ICCAD*, 2013, pp. 824–829.

[10] Y. Jin, X. Guo, R. G. Dutta, M. Bidmeshki, and Y. Makris, "Data secrecy protection through information flow tracking in proof-carrying hardware IP - part I: framework fundamentals," *IEEE Trans. Information Forensics and Security*, vol. 12, no. 10, pp. 2416–2429, 2017.

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

[12] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital IP cores," in *IEEE International Symposium on Hardware-Oriented Security and Trust, HOST*, 2011, pp. 67–70.

[13] M. Rathmair, F. Schupfer, and C. Krieg, "Applied formal methods for hardware trojan detection," in *IEEE International Symposium on Circuits and Systemss, ISCAS*, 2014, pp. 169–172.

[14] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *52nd Annual Design Automation Conference, DAC*, 2015, pp. 112:1–112:6.

[15] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal security verification of third party intellectual property cores for information leakage," in *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID*, 2016, pp. 547–552.

[16] X. T. Ngo, J. Danger, S. Guilley, Z. Najm, and O. Emery, "Hardware property checker for run-time hardware trojan detection," in *European Conference on Circuit Theory and Design, ECCTD*, 2015, pp. 1–4.

[17] A. Hekmatpour and A. Salehi, "Block-based schema-driven assertion generation for functional verification," in *14th Asian Test Symposium, ATS*, 2005, pp. 34–39.

[18] J. Malburg, T. Flenker, and G. Fey, "Property mining using dynamic dependency graphs," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC*, 2017, pp. 244–250.

[19] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: a tool to automatically infer dynamic invariants for hardware designs," in *42nd Design Automation Conference, DAC*, 2005, pp. 775–778.

[20] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, "Automatic generation of complex properties for hardware designs," in *Design, Automation & Test in Europe, DATE*, 2008, pp. 545–548.

[21] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen, "Inferno: Streamlining verification with inferred semantics," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 728–741, 2009.

[22] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, W. Tuohy, and D. R. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation & Test in Europe, DATE*, 2010, pp. 626–629.

[23] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Design, Automation & Test in Europe, DATE*, 2015, pp. 67–72.