Selective Concolic Testing for Hardware Trojan Detection in Behavioral SystemC Designs

Bin Lin
Department of Computer Science
Portland State University
Portland, OR 97207, USA
linbin@cs.pdx.edu

Jinchao Chen
School of Computer Science
Northwestern Polytechnical University
Xi'an 710072, China
cjc@nwpu.edu.cn

Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207, USA
xie@cs.pdx.edu

***Abstract*—With the growing complexities of modern SoC designs and increasingly shortened time-to-market requirements, new design paradigms such as outsourced design services have emerged. Design abstraction level has also been raised from RTL to ESL. Modern SoC designs in ESL often integrate a variety of third-party behavioral intellectual properties, as well as intensively utilizing EDA tools to improve design productivity. However, this new design trend makes modern SoCs more vulnerable to hardware Trojan attacks. Although hardware Trojan detection has been studied for more than a decade in RTL and lower levels, it has only recently gained attention in ESL designs. In this paper, we present a novel approach for generating test cases by selective concolic testing to detect hardware Trojans in ESL. We have evaluated our approach on an open source benchmark that includes various types of hardware Trojans. The experimental results demonstrate that our approach is able to detect hardware Trojans effectively and efficiently.**

I. Introduction

Modern system designs involve integration of all components of a system on a single chip, namely System-on-a-Chip (SoC). Due to the growing complexities of SoCs and increasingly shortened time-to-market requirements, design abstraction level has rised from register transfer level (RTL) to electronic system level (ESL), e.g., in C/C++ or SystemC. Modern SoC designs in ESL often include a large variety of behavioral intellectual properties (IPs), such as microcontrollers, network processors, and digital signal processors. Developing and verifying all these IPs in-house is intimidating, if not impossible, due to time-to-market and budget constraints. New design paradigms such as outsourced design services and widely adoption of electronic design automation (EDA) tools, have emerged. Although this new design trend tremendously improves modern SoC design productivity, it results in partial relinquishment of the control over the SoC design process, which raises new hardware security issues such as hardware Trojan attacks in early design steps [1].

SystemC [2] is a widely adopted ESL modelling language in the semiconductor industry. It has been increasingly used for architectural exploration, functional verification, and high-level synthesis. Thus, it is critical to assure the trustworthiness of those ESL SystemC designs. If hardware Trojans are not discovered in behavioral SystemC designs, they may be translated together with normal functionalities down to RTL and lower level implementations, which makes them much more costly to fix. Existing hardware Trojan detection approaches, most of which are focused on RTL and lower levels, may be able to detect those hardware Trojans. However, detecting and fixing Trojans in RTL or lower levels is much more expensive than fixing them in ESL. Unfortunately, those low-level hardware Trojan detection approaches are not applicable to behavioral SystemC designs since they have different characteristics from RTL and lower level implementations.

Until recently, there has only been a limited amount of research on hardware Trojan detection for behavioral SystemC designs. The pioneering work [3] discusses hardware Trojan problem in behavioral designs and proposes to detect those Trojans using property checking. The subsequent work [4] uses coverage-guided fuzz testing to detect hardware Trojans in behavioral SystemC designs. Both approaches are focused on behavioral synthesizable SystemC designs, which is a subset of SystemC designs. Our approach presented in this paper does not have such a restriction so that it is applicable to any SystemC design.

In this paper, we present a novel approach to generating test cases detecting hardware Trojans in behavioral SystemC designs with selective concolic testing. Hardware Trojans are stealthy in nature and thus they can only be triggered under very rare conditions. This makes them very hard to detect during SoC validation. Concolic (a portmanteau of concrete and symbolic) testing [5] is able to generate test cases that exercise corner cases effectively. It has achieved considerable success for the software validation [6], [7] and has been used for test generation in the hardware domain recently [8]–[10]. A recent work [11] adopts concolic testing for hardware Trojan detection in RTL, which inspires us to utilize concolic testing to detect hardware Trojans in behavioral SystemC designs. Key contributions of our work are as follows.

- To the best of our knowledge, we first bring the concolic testing techniques into hardware Trojan detection for behavioral SystemC designs.
- We have improved the efficiency of existing concolic testing techniques with selective concolic testing and coverage-guided state search strategy.
- We have implemented the proposed approach as an effective and efficient prototype, namely SCT-HTD, which is able to detect hardware Trojans inserted by adversaries

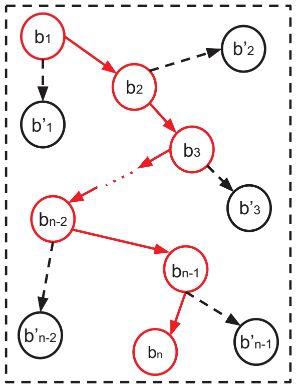

Fig. 1: Concolic test generation

in behavioral SystemC designs automatically.

- We have evaluated the performance of our approach on a benchmark suite, S3CBench [12], which includes behavioral SystemC designs with various hardware Trojans. The experimental results show that our approach is able to detect hardware Trojans effectively and efficiently.

The rest of this paper is organized as follows. Section 2 introduces background on hardware Trojan and concolic testing. Section 3 discusses the threat model to which our approach applies and overview of the proposed approach. Section 4 presents our approach in details. Section 5 elaborates on the experiments that we have conducted and the evaluation results. Section 6 reviews closely related work. Finally, we conclude this research and discuss future work in Section 7.

II. Background

A. Hardware Trojan

Hardware Trojans can be defined as any addition or modification to an electronic circuit or design with malicious intent, including functionality modification, sensitive information leakage, or denial of services. Hardware Trojans may be classified into several categories based on various characteristics such as abstraction level, insertion phase, activation mechanism, and so on. A detailed taxonomy of hardware Trojans can be found in the survey paper [13]. In general, a hardware Trojan includes two parts, the *trigger* (the activation mechanism) and the *payload* (the functional part affected by the activation mechanism). Hardware Trojans are usually stealthy and are triggered under very rare conditions so that they are hard to detect during functional validation.

B. Concolic Testing

Concolic testing [5] is a hybrid verification technique that combines concrete execution and symbolic execution [14]. It partly addresses the limitations of random testing and symbolic execution based testing [15], [16]. With concolic testing, both concrete values and symbolic values are used as inputs for a design under validation (DUV), in which case it is executed both concretely and symbolically. Figure 1 shows the idea of concolic test generation. Solid red arrows denote a concrete execution path with the concrete input values, while dashed black arrows represent possible alternative paths where new test cases may be generated. Circles denote branch points. During concolic execution, symbolic constraints are collected along the execution path guided by the concrete inputs. At each branch point, the constraints are negated and then solved, if possible, to generate a new test case which would explore an alternative path for the DUV.

III. Overview of Proposed Approach

This section presents the adversarial threat model, as well as the high level overview of our proposed approach.

A. Threat Model

With the globalization of the semiconductor industry, modern SoC designers have been driven to outsource their IPs to reduce cost. In addition, the growing complexities of modern SoCs has raised the design abstraction level from RTL to ESL. As a result, modern SoC design methodologies at ESL often involve integration of behavioral IPs supplied by third-party vendors, as well as intensive usage of EDA tools, to improve the design productivity. However, as shown in Figure 2, the trustworthiness of SoCs in ESL may be comprised. First, third-party IPs may contain malicious implants. Although a testbench with test cases is likely provided with the IPs by third-party vendors, these test cases are not able to trigger the embedded Trojans. Second, untrusted EDA tools may also insert hardware Trojans into these behavioral designs. Last but not least, in-house designers may leave back-doors when integrating SoCs, which makes the situation worse. Our approach mainly intends to detect hardware Trojans injected into behavioral SystemC designs.

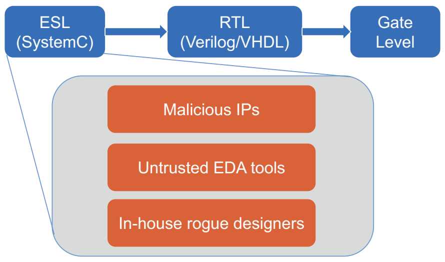

Fig. 2: Adversarial threat model targeted by SCT-HTD

B. Overview

In this work, we assume that there is a golden model for a behavioral SystemC DUV. A golden model is an executable behavioral model that is functionally correct and Trojan-free. Although it is very expensive, if not impossible, to develop an entire SoC in-house by designers, we argue that it should not be very time-consuming to develop a functionally correct golden model, which is not necessarily cycle accurate. As shown in Figure 3, our approach consists of two key components, selective concolic executor (SCE) and hardware Trojan detector (HTD). SCE generates test cases by selective concolic execution with coverage-guided state search strategy, while

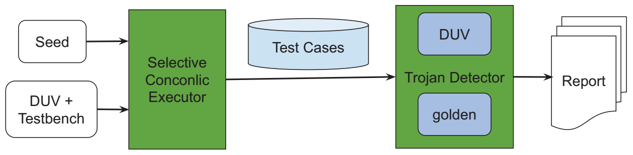

Fig. 3: Selective concolic testing for hardware Trojan detection

HTD detects hardware Trojans by simulating the generated test cases on both the DUV and its golden model. The execution of SCE includes a stack, a heap, concrete values of the inputs, path conditions (represented as symbolic expressions), a register file, and a program counter, which we define as an *execution state* or *state*.

Algorithm 1 illustrates high level steps of our proposed hardware Trojan detection approach. The algorithm takes five parameters as inputs: a design under validation *duv*, an initial test case π (called *seed*), a testbench *tb*, a golden model *golden*, and a configuration file *config*. The set TC, which contains the seed initially (line 1), saves all generated test cases. The state s_0 is the initial execution state of the DUV with the seed, and assigned to s_n which is the next state to be explored by concolic execution (line 2). The queues FPS and SPS save execution states during the process (line 3). FPS, which includes the initial state at the beginning, saves first priority states that explore new code of interest, while SPS saves second priority states that do not explore new code of interest. The variable $INTS$ stores code ranges that users are interested in for hardware Trojan detection (line 4). Those code ranges are specified in the configuration file. A time bound β is also given in the configuration file (line 5), which guarantees the termination of the Trojan detection process in case no Trojans are detected. At the beginning, since s_n is not $NULL$ and time bound has not been reached, SEL-CON-TEST-GEN is executed to explore the initial state (line 7). Upon completion, it returns a set of new states S' and a test case τ for s_n. If τ is $NULL$, no test case is generated for the explored state (line 8). Thus, another state is selected and explored (line 9–10). If τ is not $NULL$, HT-DETECTOR is invoked to detect hardware Trojans with the newly generated test case τ (line 11). If Trojans are detected, the algorithm returns all generated test cases and the test case that triggers the Trojans (line 12–13). Then, SCT-HTD terminates. Otherwise, the new test case τ is added to TC, followed by invoking STATE-SELECTOR, which selects another state s_n for concolic execution (line 14–15). The variable *time* (line 6) denotes the total time elapsed since SCT-HTD starts. If there are no more states left or the time bound has been reached, SCT-HTD returns all generated test cases and terminates (line 16). We will discuss the details of each component in the following section.

IV. Hardware Trojan Detection via Selective Concolic Testing

This section presents selective concolic testing for hardware Trojan detection in behavioral SystemC designs in details. We will first describe the core of our proposed approach, SCE, which includes our two primary optimizations, selective concolic test generation and coverage-guided state search strategy. Then, we will present HTD that detects hardware Trojans with the generated test cases by SCE.

Algorithm 1: SCT-HTD(duv, π, tb, $golden$, $config$)

```
1  $TC \leftarrow \{\pi\}$
2  $s_0 \leftarrow$ INITIALIZE($duv, \pi, tb$),   $s_n \leftarrow s_0$
3  $FPS \leftarrow \{s_0\}$, $SPS \leftarrow \emptyset$          ▷ $FPS$ and $SPS$ are queues
4  $INTS \leftarrow$ GET-INTERESTED-CODE-RANGE($config$)
5  $\beta \leftarrow$ GET-TIME-BOUND($config$)
6  while ($s_n \neq NULL$) $\wedge$ ($time < \beta$) do
7      $\{S', \tau\} \leftarrow$ SEL-CON-TEST-GEN($s_n, INTS$)
8      if $\tau == NULL$ then
9          $s_n \leftarrow$ STATE-SELECTOR($duv, tb, FPS, SPS, S'$)
10         continue
11     $ret \leftarrow$ HT-DETECTOR($duv, golden, tb, \tau$)
12     if ($ret$) then
13         return $TC$, $\tau$
14     $TC \leftarrow TC \cup \{\tau\}$
15     $s_n \leftarrow$ STATE-SELECTOR($duv, tb, FPS, SPS, S'$)
16 return $TC$
```

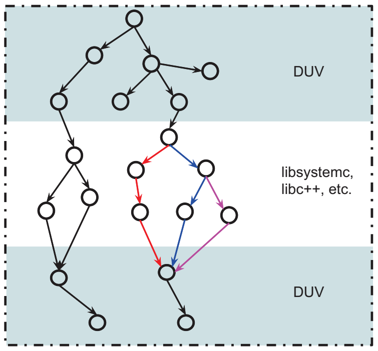

Fig. 4: Selective concolic test generation

A. Selective Concolic Test Generation

Traditional concolic test generation approaches generate test cases along an entire concrete execution path, as shown in Figure 1. However, often only a small portion of the path is from the DUV code. Most code composed of the path is from libraries, which is of no interest to users generally. It may not be beneficial to generate test cases in these libraries code for verifying the DUV. Figure 4 demonstrates the case. The circles denote branch points, while the arrows indicate the execution sequence. As the figure shown, although there are three paths indicated with red, blue, and magenta in the libraries, it is only one path from the DUV perspective. To improve the verification efficiency, concolic test generation approaches should be restricted to generate test cases for the DUV only, by which the number of generated test cases can be reduced and hence test generation time is reduced. Furthermore, the subsequent simulation time can also be reduced with fewer test cases. Unfortunately, traditional concolic test generation approaches are not able to distinguish DUV code from libraries code. Therefore, they usually generate many redundant test cases in terms of the DUV, since these test cases follow the same path from the DUV point of view.

Algorithm 2: SEL-CON-TEST-GEN($s_n, INTS$)

```
1  $S \leftarrow \emptyset$, $\tau \leftarrow NULL$
2  while HAS-NEXT-INSTRUCTION($s_n$) do
3      $I \leftarrow$ GET-NEXT-INSTRUCTION($s_n$)
4      EXECUTE-INSTRUCTION($I$)
5      if $I$ is branch then
6          $bp \leftarrow$ GET-SYMBOLIC-BRANCH-PREDICATE($I$)
7          if FIND($I$, $INTS$) then
8              $s'_n \leftarrow$ FORK($s_n, \neg bp$)
9              $S \leftarrow S \cup s'_n$
10         ADD-CONSTAINTS($s_n, bp$)
11     SET-NEXT-INSTRUCTION($s_n$)
12 $\tau \leftarrow$ CONSTRAINT-SOLVER( GET-CONSTRAINTS($s_n$) )
13 return $S, \tau$
```

Our proposed selective concolic test generation approach is able to generate test cases for a specific part of code. In this case, it is the DUV. However, our approach can also be used to generate test cases for a specific library, or a combination of multiple code segments, depending on users' interests. Algorithm 2 describes our selective concolic test generation process. The procedure SEL-CON-TEST-GEN takes two parameters, s_n and $INTS$, as inputs. The input s_n is the current execution state with the concrete values for the symbolic variables that are obtained in Algorithm 3. The input $INTS$ includes all code ranges of interest to users. The set S saves newly forked states from s_n and the test case τ will be the generated test case for s_n (line 1). If there is an instruction for execution (line 2), the instruction I is fetched (line 3) and executed (line 4). If it is a branch instruction (line 5), the symbolic predicate bp of I is computed (line 6). If I is in the ranges $INTS$, then a new state is forked with negation of bp and the state is added to the set S (line 7–9). Otherwise, no new state is forked. Afterwards, bp is added to the constraints of s_n (line 10). Then, line 11 sets the next instruction to be executed. If every instruction of s_n has been executed, a test case is generated if possible and saved to τ (line 12). Finally, the newly forked states S and the test case τ are returned.

B. Coverage-guided State Search Strategy

Algorithm 3 presents our coverage-guided state search strategy. If HT-DETECTOR does not detect hardware Trojan, then, STATE-SELECTOR is invoked to select another state for concolic execution. The state s will be the returned state, which is $NULL$ initially (line 1). For each state s_t from S' that is forked from previous concolic execution, its path constraints are sent to a solver (line 3). If it returns $NULL$, the current state is not reachable so that we do not save it (line 4–5). This prevents the state from being explored later, which reduces the overall execution time. If the solver succeeds, the concrete values cv are saved to the state for later execution (line 6). Then, coverage is analyzed with cv (line 7). If new code is covered, then s_t is added to the first priority state queue FPS (line 9). Otherwise, it is added to the second priority state queue SPS (line 11). After each state is evaluated, the first state in FPS is assigned to s (line 13) and is removed

Algorithm 3: STATE-SELECTOR(duv, tb, FPS, SPS, S')

```
1  $s \leftarrow NULL$
2  foreach $s_t \in S'$ do
3      $cv \leftarrow$ CONSTRAINT-SOLVER( GET-CONSTRAINTS($s_t$) )
4      if $cv$ == $NULL$ then
5          continue
6      SET-VALUE ($s_t$, $cv$)
7      newly_covered $\leftarrow$ COVERAGE-ANALYZER($duv$, $tb$, $cv$)
8      if (newly_covered) then
9          $FPS \leftarrow FPS \cup \{s_t\}$
10     else
11         $SPS \leftarrow SPS \cup \{s_t\}$
12 if $FPS \neq \emptyset$ then
13     $s \leftarrow$ FPS.front()           ▷ get the first state from $FPS$
14     FPS.pop()                  ▷ remove the first state from $FPS$
15 else if $SPS \neq \emptyset$ then
16     $s \leftarrow$ SPS.front()           ▷ get the first state from $SPS$
17     SPS.pop()                  ▷ remove the first state from $SPS$
18 return $s$
```

(line 14) if FPS is not empty. Otherwise, the first state in SPS is retrieved and removed (line 16–17). If both FPS and SPS are empty, then s remains $NULL$, which means no more state to be explored. Finally, the selected state s is returned.

C. Hardware Trojan Detection

Algorithm 4 demonstrates our hardware Trojan detection procedure. After a state is explored and the test case is generated, HT-DETECTOR is called with four arguments, a duv and its golden model $golden$, a testbench tb, as well as the newly generated test case τ. The test case τ is simulated on both duv and $golden$ (line 1 and line 2 respectively). If the results are not the same from both simulation, then this indicates that a Trojan is detected (line 4); otherwise, HT-DETECTOR returns false to indicate that no Trojan is discovered.

Algorithm 4: HT-DETECTOR(duv, $golden$, tb, τ)

```
1  $res1 \leftarrow$ SIMULATOR($duv$, $tb$, $\tau$)
2  $res2 \leftarrow$ SIMULATOR($golden$, $tb$, $\tau$)
3  if $res1 \neq res2$ then
4      return true                ▷ indicates that Trojan is detected
5  else
6      return false
```

V. EXPERIMENTAL RESULTS

We have implemented the proposed approach as an automated prototype, namely SCT-HTD, based on S2E [17], which is a generic platform for analyzing software systems. S2E incorporates a virtual machine and a symbolic execution engine, by which it is able to switch back and forth between concrete execution and symbolic execution. S2E consists of a path explorer that drives a target program down possible execution paths of interest and a path analyzer that checks properties of explored paths. We adapted S2E as our selective

TABLE I: SCT-HTD experimental results and comparison with the state-of-the-art approaches

Designs	# Test case				Time (s)				Memory (MB)			
	SCT-HTD[*]	CT[ζ]	AFL[η]	AFL-SHT[δ]	SCT-HTD	CT	AFL	AFL-SHT	SCT-HTD	CT	AFL	AFL-SHT
`adpcm-swm`	27	525	451563	423	157	T.O.	T.O.	1.71	3546	14337	N/A	N/A
`adpcm-swom`	7	503	450839	414	31	T.O.	T.O.	1.67	1341	14442	N/A	N/A
`fir-cwom`	26	76	207	41	13	26	8.51	0.07	1621	2305	N/A	N/A
`bSort-cwom`	2	2	118	39	8	10	4.82	0.05	1074	2668	N/A	N/A
`bSort-swm`	4	975	19826	108	10	T.O.	337.36	0.11	1106	10768	N/A	N/A
`uart-swm1`	3	1023	N/A	N/A	9	T.O.	N/A	N/A	1071	13011	N/A	N/A
`uart-swm2`	3	1016	172	51	9	T.O.	8.82	0.18	1070	12972	N/A	N/A
`aes-cwom`	11	11	50544	22	23	32	888.29	0.04	1386	1396	N/A	N/A

[*] Our approach [ζ] Concolic testing (CT) without our optimizations [η] Fuzzing with software-oriented mutation [δ] Coverage-guided fuzzing

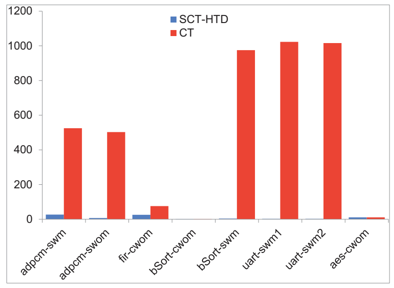

Fig. 5: Number of generated test cases

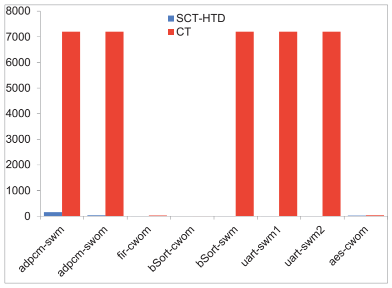

Fig. 6: Time usage

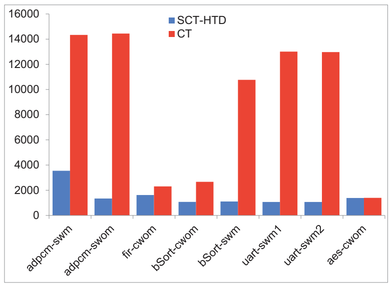

Fig. 7: Maximum memory usage

concolic executor to generate test cases for SystemC designs, which are used to detect hardware Trojans in those designs.

We have evaluated our approach on the open-source benchmark suite, S3CBench [12]. Although S3CBench only consists of behavioral synthesizable SystemC designs, our approach also works for non-synthesizable SystemC designs. S3CBench contains 10 SystemC designs that includes multiple types of hardware Trojans based on trigger mechanism, either sequential or combinational. Half of the designs are computation intensive designs, such as image processing algorithms. Each design has fixed computation procedures for any inputs. As we all know, concolic test generation is based on branch conditions, which makes it very powerful to explore deep path with complex conditions and corner cases. However, it is not good at generating test cases for a design that has fixed execution steps, since an execution path of such a design does not depend on input values. This is a known limitation of concolic testing. Thus, we conducted experiments on non-computationally intensive designs. The experiments were conducted on a laptop with a 4-core Intel(R) Core(TM) i7-4700MQ CPU, 16 GB of RAM, and running the Ubuntu Linux OS with 64-bit kernel version 4.15. Table I presents our experimental results, as well as comparison with the state-of-the-art approaches. We will discuss the table in the following.

A. *Effectiveness and Efficiency of Hardware Trojan Detection*

We developed a golden model for each design on which we experimented. The first column of Table I gives the name (before the hyphen part) of each design. The part after the hyphen denotes the type of inserted Trojan. Details about the benchmark and the Trojans can be found in S3CBench [12]. We have evaluated the performance of SCT-HTD from three perspectives, namely the number of generated test cases to trigger the Trojan (column 2), time usage (column 6) and maximum memory usage (column 10). To pursue a fair comparison, we also set a two-hour time bound during experiments as AFL-SHT [4] did. T.O. indicates that the two-hour time bound is reached and thus the hardware Trojans are not detected. The designs that we evaluated on includes the three typical hardware Trojan types in terms of payload, namely functionality modification (`adpcm`, `fir`, and `bSort`), denial of service (`uart`), and sensitive information leakage (`aes`). As demonstrated, our approach is able to detect all three types of hardware Trojans with a few test cases, short time usage and reasonable memory usage, which demonstrates the effectiveness and efficiency.

B. *Evaluation of Two Optimization Strategies*

To illustrate the advantages of our selective concolic testing and coverage-guided state search strategy, we have conducted experiments with traditional concolic testing approach (without the two optimization), denoted as CT. Column 3, column 7 and column 11 show the number of generated test cases, time usage and maximum memory usage with CT, respectively. Five out of eight Trojans are not detected within the two-hour time bound, although many test cases are generated. Figure 5, Figure 6, and Figure 7 demonstrate the advantage of SCT-HTD in the three aspects graphically, compared with traditional concolic testing approach. As shown in figures, our optimization strategies reduce the number of generated test cases, time usage, and memory usage tremendously for more than half of the designs. For other designs, our approach is also able to detect hardware Trojans with fewer or equal number of generated test cases, less time and memory usage.

C. *Comparison with State-of-the-Art Approaches*

Two existing approaches target hardware Trojan detection for behavioral SystemC designs. One [3] uses property checking and the other, AFL-SHT [4], adopts coverage-guided fuzzing. Although we do not have access to the commercial formal tools to conduct experiments, formal approaches on Trojan detection for behavioral SystemC designs are not as promising as AFL-SHT. Since the source code of AFL-SHT is not available, we take the results obtained by both AFL-SHT and AFL [18] from its paper for comparison. Some data are not presented in the paper, such as memory usage, which are denoted as N/A. As illustrated, our approach is able to detect the Trojans with far fewer test cases than AFL and AFL-SHT. Time usage of SCT-HTD is much less than AFL for half of the designs. For other designs, SCT-HTD uses a little longer time than AFL and AFL-SHT. There are two possible reasons. First, power of machines running experiments are different. We conducted experiments on a laptop which might be less powerful. Second, concolic testing involves constraint solver to generate test cases, which is a known time-consuming operation. However, the increase of time usage is moderate.

VI. Related Work

Until recently, most of computer security research was devoted to software security. The underlying hardware was expected to be secure. However, this is no longer the case with the globalization of modern SoC design and manufacturing processes, and the emergence of new design paradigms such as outsourced design services and intensive usage of EDA tools. Thus, hardware vulnerabilities such as hardware Trojan attacks have raised serious concerns. As a result, a variety of hardware Trojan detection approaches have been developed.

So far, most hardware Trojan detection approaches are focused on RTL or lower level designs. There are a handful of Trojan detection approaches in RTL [11], [19] . There are also various Trojan detection approaches that are focused on the gate level [20]–[25]. Post-silicon Trojan detection has also been studied [26]–[28]. There has only been limited research on hardware Trojan detection for behavioral designs. The pioneering work [3] detects hardware Trojans in behavioral SystemC designs using C++ control flow constructs and the property checker provided by a commercial behavioral synthesis tool. The subsequent work [4] uses coverage-guided fuzz testing to detect hardware Trojans in behavioral SystemC designs. Both approaches are focused on behavioral synthesizable SystemC designs which is a subset of SystemC, while our approach presented in this paper is not restricted to the behavioral synthesizable SystemC designs.

VII. Conclusion

In this paper, we have presented a novel approach for detecting hardware Trojans in behavioral SystemC designs with selective concolic testing. We have also proposed an algorithm to improve efficiency of the approach with coverage-guided state search strategy. We have implemented the proposed approach as a prototype, SCT-HTD. To show the effectiveness and efficiency of the proposed approach, we have conducted experiments on an open source benchmark. The results demonstrate that our approach is very promising on hardware Trojan detection for behavioral SystemC designs. In the future, we will extend the S3CBench with non-synthesizable SystemC designs and conduct experiments on them using SCT-HTD. Moreover, we will combine other techniques, such as mutation testing and fuzzing testing, with SCT-HTD to overcome the limitations of concolic testing, which is not suitable for computationally intensive designs.

Acknowledgement

This research received financial support in part from National Science Foundation (Grant #: 1908571).

References

[1] I. Polian, G. T. Becker, and F. Regazzoni, “Trojans in Early Design Steps—An Emerging Threat,” in *Proc. of TRUDEVICE*, 2016.

[2] IEEE Standards Association, *Standard SystemC Language Reference Manual*. IEEE Std. 1666-2011, 2011.

[3] N. Veeranna and B. C. Schafer, “Hardware Trojan Detection in Behavioral Intellectual Properties (IP’s) Using Property Checking Techniques,” *IEEE Transactions on Emerging Topics in Computing*, 2017.

[4] H. M. Le, D. Große, N. Bruns, and R. Drechsler, “Detection of Hardware Trojans in SystemC HLS Designs via Coverage-guided Fuzzing,” in *Proc. of DATE*, 2019.

[5] K. Sen, “Concolic Testing,” in *Proc. of ASE*, 2007.

[6] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” *SIGPLAN Not.*, 2005.

[7] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C,” in *Proc. of ESEC*, 2005.

[8] L. Liu and S. Vasudevan, “Efficient Validation Input Generation in RTL by Hybridized Source Code Analysis,” in *Proc. of DATE*, 2011.

[9] Y. Zhou, T. Wang, H. Li, T. Lv, and X. Li, “Functional Test Generation for Hard-to-Reach States Using Path Constraint Solving,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[10] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie, “Concolic Testing of SystemC Designs,” in *Proc. of ISQED*, 2018.

[11] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, “Scalable Hardware Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution,” in *Proc. of ITC*, 2018.

[12] N. Veeranna and B. C. Schafer, “S3CBench: Synthesizable Security SystemC Benchmarks for High-Level Synthesis,” *Journal of Hardware and Systems Security*, 2017.

[13] M. Tehranipoor and F. Koushanfar, “A Survey of Hardware Trojan Taxonomy and Detection,” *IEEE Design Test of Computers*, 2010.

[14] J. C. King, “Symbolic Execution and Program Testing,” *Communications of the ACM*, 1976.

[15] B. Lin and D. Qian, “Regression Testing of Virtual Prototypes Using Symbolic Execution,” *International Journal of Computer Science and Software Engineering*, 2015.

[16] B. Lin, Z. Yang, K. Cong, and F. Xie, “Generating High Coverage Tests for SystemC Designs Using Symbolic Execution,” in *Proc. of ASP-DAC*, 2016.

[17] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-vivo Multi-path Analysis of Software Systems,” in *Proc. of ASPLOS*, 2011.

[18] M. Zalewski, “Technical Whitepaper,” http://lcamtuf.coredump.cx/afl/technical_details.txt.

[19] M. Banga and M. S. Hsiao, “Trusted RTL: Trojan Detection Methodology in Pre-Silicon Designs,” in *Proc. of HOST*, 2010.

[20] D. Ismari, J. Plusquellic, C. Lamech, S. Bhunia, and F. Saqib, “On Detecting Delay Anomalies Introduced by Hardware Trojans,” in *Proc. of ICCAD*, 2016.

[21] F. Koushanfar and A. Mirhoseini, “A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection,” *IEEE Transaction on Information Forensics Security*, 2011.

[22] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa, “A Score-based Classification Method for Identifying Hardware-trojans at Gate-level Netlists,” in *Proc. of DATE*, 2015.

[23] J. Rajendran, V. Vedula, and R. Karri, “Detecting Malicious Modifications of Data in Third-party Intellectual Property Cores,” in *Proc. of DAC*, 2015.

[24] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, “VeriTrust: Verification for Hardware Trust,” in *Proc. of DAC*, 2013.

[25] B. Çakir and S. Malik, “Hardware Trojan Detection for Gate-level ICs Using Signal Correlation Based Clustering,” in *Proc. of DATE*, 2015.

[26] K. Hu, A. N. Nowroz, S. Reda, and F. Koushanfar, “High-sensitivity Hardware Trojan Detection Using Multimodal Characterization,” in *Proc. of DATE*, 2013.

[27] S. Narasimhan, D. Du, R. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia, “Hardware Trojan Detection by Multiple-Parameter Side-Channel Analysis,” *IEEE Transactions on Computers*, 2013.

[28] E. Love, Y. Jin, and Y. Makris, “Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition,” *IEEE Transaction on Information Forensics and Security*, 2012.