Hardware Trojan Detection using ATPG and Model Checking

Jonathan Cruz[1], Farimah Farahmandi[2], Alif Ahmed[2], and Prabhat Mishra[2]
[1]Department of Electrical and Computer Engineering
[2]Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL, USA

Abstract—**The threat of hardware Trojans' existence in integrated circuits has become a major concern in System-on-Chip (SoC) design industry as well as in military/defense organizations. There is an increased emphasis on finding effective ways to detect and activate hardware Trojans in current research efforts. However, state-of-the-art approaches suffer from the lack of completeness and scalability. Moreover, most of the existing methods cannot generate efficient tests to activate the potential hidden Trojan. In this paper, we propose an effective test generation approach which is capable of activating malicious functionality hidden in large sequential designs. Automatic test pattern generation (ATPG) works well on full-scan designs, whereas model checking is suitable for logic blocks without scan chain. Due to overhead considerations, partial-scan chain insertion is the standard practice today. Unfortunately, neither ATPG nor model checking is suitable for partial-scan designs. Our proposed hardware Trojan detection technique utilizes the combination of ATPG and model checking approaches. We use model checking on a subset of non-scan elements and ATPG on scan elements to avoid common pitfalls of running the original design using any one of these techniques. Experimental results demonstrate the effectiveness of tests generated by our proposed approach to detect Trojans on Trust-hub benchmarks.**

I. Introduction

Designing today's system-on-chips (SoC) is a highly complex process that is subject to stringent time-to-market constraints. It is a common practice to integrate third-party Intellectual Property (IP) blocks in the production of SoCs in order to remain competitive in today's global market. However, interfacing with untrusted third-party IP affects a design's trustworthiness and security. An adversary can tamper with the design or insert malicious components, known as hardware Trojans, within third-party facilities. Hardware Trojans are a small modification in the design that can be triggered in an extremely rare input event. As a result, hardware Trojans are dormant during most of the run-time and can escape detection during conventional functional validation techniques (such as simulation-based validation using random or constraint random tests). The effects of hardware Trojan insertion attacks range from information leakage to complete chip malfunction [1]. Therefore, it is extremely important to find efficient validation approaches to detect and activate hardware Trojans if they exist.

Existing logic testing methodologies for Trojan detection target rare node activation while monitoring the designs observable outputs [25]. Some approaches use N-detect testing with an ATPG tool for generating test vectors leading to an increased likelihood for Trojan activation [2], [24]. Most of these techniques assume a full scan chain when dealing with IP that contain sequential elements [3], [2]. However, partial-scan chains offer an attractive alternative to designs with area or testing constraints. Therefore, with the introduction of partial-scan inserted IP, it becomes even harder to detect Trojans that rely on these verification tools alone since the depth and testable complexity of sequential parts will be increased [4], [5]. Formal methods have also been used in Trojan detection and verifying design security. With model checking, security properties can be specified by a designer to verify the security of the design [6]. However, model checking is known to suffer from state explosion which limits its effectiveness on verifying security in large designs.

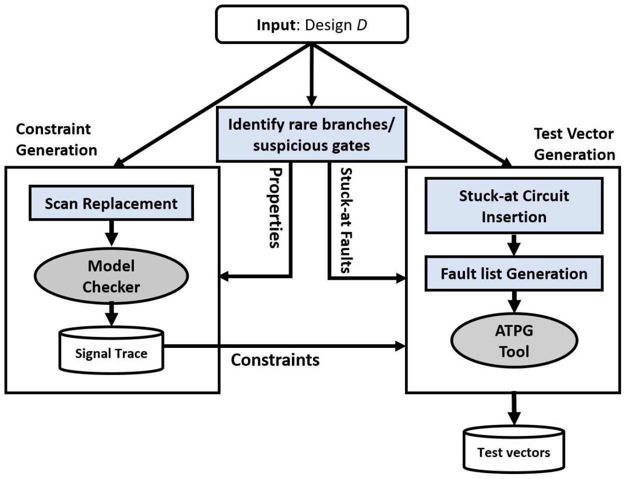


Fig. 1: The overview of our proposed approach.

Motivated by the limitations of these tools, we propose a test generation framework that combines the strengths of these two common design verification techniques, already a part of the normal verification design flow. Our approach increases the efficiency for Trojan detection in partial-scan designs by reducing the potential state space for model checker and removing the complexity of non-scan sequential ATPG in ATPG tools. Figure 1 shows the overview of our approach. As shown in Figure 1, our method identifies suspicious branches/gates which may be used as triggering conditions for hardware Trojans. In order to generate tests to activate rare nodes, scan replacement is done in the next step. We generate security

This work was partially supported by grants from National Science Foundation (CNS-1441667), SRC (2014-TS-2554), and Cisco.

properties that targets activation of equivalent signals/gates of rare nodes in the gate-level netlist. The scan replaced netlist as well as the security properties are used by the model checker. We generate a set of constraints using model checker to facilitate directed test generation using ATPG tool. To the best of our knowledge, there are no previous attempts to address Trojan detection particularly in partial-scan designs using a combination of model checking and ATPG tools. We have applied our methodology on Trust-Hub and custom benchmarks and have demonstrated that our approach not only detects the hidden Trojans but also activates them in an effective manner.

The rest of the paper is organized as follows. Section II introduces the background in ATPG and model checking. Section III describes the related work using these techniques. Section IV describes our hardware Trojan detection approach of using ATPG and model checking. Section V describes the experimental setup, analysis, and results. Finally, Section VI concludes the paper.

II. Background

A. Design for Test

Design for Testability (DFT) is a technique employed in designing ICs for the purposes of reducing test costs and associated time [1]. A very common technique for DFT is scan-chain insertion. The idea here is to replace regular flip-flops (FFs) in the design with scan flip-flops. Scan FFs can be chained together to form a scan-chain. The purpose of these scan-chains is to reduce the loading time when testing sequential elements of a design, increasing a circuit's overall gate controllability and observability. While including a full scan-chain is ideal, it may not be feasible due to the incurred area overhead or delays that invalidate various design constraints.

Partial-scan chain insertion is used as a cost-effective alternative [7] to achieve acceptable test coverage, while maintaining design constraints. In partial-scan designs, not all FFs are included in the scan-chain. However, these non-scan FFs introduce branches in the circuit with low controllability and observability, which can be exploited as a rare trigger condition by a malicious attacker. Controllability measures the difficulty associated with setting a particular signal to a logic value, and observability measures the difficulty of observing a signal at an observable point in the circuit.

B. Automatic Test Pattern Generation

ATPG is a test methodology used to identify faulty behavior(s) in circuits due to design defects. The goal of ATPG is to create a set of test patterns that achieve a desired test coverage, TC, and fault coverage, FC, through fault simulation. Test coverage is the ratio of detected faults over testable faults, while fault coverage is the ratio of detected faults over all faults. With the use of DFT and scan-chains, ATPG can efficiently generate test vectors by treating any design as combinational logic, consequently reducing the complexity. This is no longer the case in partial-scan designs. ATPG tools must now consider a sequential set of test vectors to activate a target fault. The sequential ATPG complexity is linearly proportinal to the sequential depth and exponentially porportional to the number of sequential cycles (more complex than feedback loops) [8]. The worst-case complexity (cyclic sequential designs) of sequential ATPG becomes 9^N, in a 9-valued logic system where N is the number of flip-flops [9].

C. Model Checking

Model checking is a formal method used to verify a design against functional properties (expected design behavior). To verify a design using model checkers, users must first either manually, or through the use of a program, translate their design into a model specification language understood by the tool. Design properties are then described in a temporal language using either computational tree logic (CTL) or linear temporal logic (LTL). Once the tool has both the design and properties to be verified, it begins to unroll the state space. A Boolean satisfiability assignment is extracted from the unrolled states and checked using an internal SAT solver. If unsatisfiable, the model checker produces a counterexample computation path [10].

Functional design verification with model checking can be extended for use in verifying design security [15]. Properties can be written and verified with security features in mind, such as monitoring the primary output for leaking secret keys [6]. However, as previously mentioned, a common problem of the model checking approach is state explosion caused by the exponential nature of exploring a designs state space. This fact limits the practicality of model checking on larger designs.

III. Related Work

ATPG and formal methods have been used for Trojan detection and security verification in recent years [15]. As with any ATPG tool, sequential test pattern generation is a complex process [9], [8]. While scan-chains are implemented to mitigate the complexity, designs with a significant amount of non-scan cells can greatly reduce ATPG performance, resulting in ATPG-untestable faults [12] that an attacker can exploit.

One common approach to Trojan detection involves logic testing to trigger a set of rare nodes. These methods generate test patterns with high probability of activating Trojans. Common to most of these approaches is the use of an ATPG tool, SAT solver, or some combination for test generation. Both Wolff et al. and Zhang et al. propose Trojan detection approaches that incorporate ATPG tools for generating test patterns [13], [3]. Yet, with the introduction of partial-scan designs, the effectiveness of full-sequential ATPG for generating test patterns is greatly reduced due to the complexity of full-sequential ATPG on non-scan FFs. The method proposed in [14] utilizes N-detect full scan ATPG and SAT solver for Trojans detection. However, this approach also fails to effectively consider designs that have a significant non-scan regions which will limit the effectiveness of ATPG.

Functional design verification with model checking can be extended for use in verifying design security [15]. Properties can be written and verified with security features in mind, such as monitoring the primary output for leaking secret keys [6]. However, as previously mentioned, a common problem of the model checking approach is state explosion caused by the exponential nature of exploring a designs state space. This fact limits the practicality of model checking on larger designs. Although approaches based on symbolic algebra detect the existence of hardware Trojans, they cannot generate tests to activate it [16].

IV. TROJAN DETECTION USING ATPG AND MODEL CHECKING

Many Trojan detection techniques utilize ATPG, model checking, or both in generating test patterns for detection; however, they do not consider partial-scan instances of a third-party IP (3PIP). ATPG is expected to encounter notable execution overhead in most partial-scan designs with significant sequential depth or cyclic sequential structures due to their complexity [8]. Additionally, previously detectable faults can be unintentionally rendered undetectable with the removal of scan FFs from the scan-chain. This makes it much harder to generate tests for suspicious regions as it is no longer within the realm of combinational complexity. With model checking alone, medium to large designs are expected to suffer from state explosion, preventing efficient test generation.

We propose a framework to improve the test generation efficiency by combining the benefits of the two approaches. To the best of our knowledge, our approach is the first attempt to use both ATPG and model checking for efficient test pattern generation in partial-scan designs for Trojan detection. Algorithm 1 shows the major steps in our proposed framework shown in Figure 1. Algorithm 1 takes a design D, and outputs a set of test vectors T. In line 5, the set of rare nodes R are identified in the design which are used by the *constraintGeneration* and *testVectorGeneration* procedures. The *constraintGeneration* procedure (Algorithm 2) uses model checking to produce a set of signal traces *S*. Finally the *testVectorGeneration* method (Algorithm 3) uses ATPG with the design, rare nodes and signal traces to produce a set of test vectors for activating each rare node.

To maximize the benefits of each tool, our framework also includes parallel execution of model checking and ATPG. The final test pattern is taken from whichever execution generates results first. The remainder of this section describes the steps involved in the proposed hybrid approach: rare branch identification, constraint generation, and test vector generation.

A. Rare Branch Identification

For each IP, initial analysis is performed at the RTL level to determine suspicious gates in the design. In a design, rare branches are branches that are not covered after running random tests up to millions of cycles. Mapping the RTL branches to gate-level netlist after synthesis is done in two phases. The first phase identifies any suspicious boundary and register nets and uses the synthesis tool commands to attempt

Algorithm 1 Trojan Detection Algorithm

```
1: Input: Design $D$
2: Output: Set of testvectors $T$
3: procedure TROJANDETECTION($D$)
4:     $R$, $S$, $T$ = {}
5:     $R$ = identifyRareBranches($D$)
6:     $S$ = constraintGeneration($D$,$R$)
7:     $T$ = testVectorGeneration($D$,$R$,$S$)
8:     return $T$
```

to preserve suspicious signal nets. In these cases, identifiable naming will be preserved after synthesis. If any rare branch is not accounted for, then, the second phase constructs a structural dependency graph of the two representations and attempts to match these graphs using approximate graph matching heuristics as suggested in [17].

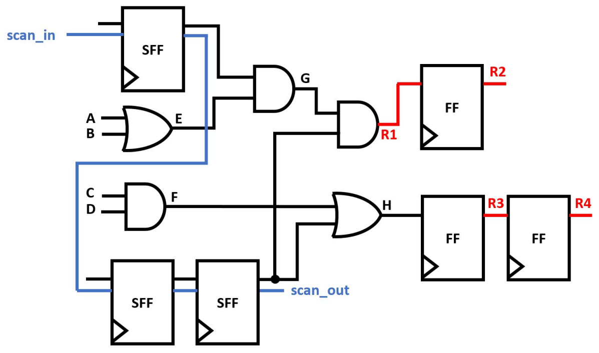
Other statistical or functional methods for determining rare nodes at RTL or gate-level such as FANCI [18] and MERS [19] are equally applicable. These circuit branches identified as rare will be used in model checking property generation and ATPG stuck-at faults/ node justification. The rationale here is that a Trojan can be activated as a result of a rare sequence of inputs and/or state transitions; otherwise, the malicious insertion becomes a triviality that can be detected during traditional design testing and verification [13]. By focusing on activating hard-to-trigger or rare nodes, we are increasing the likelihood of Trojan detection. With partial-scan designs, the non-scan FFs can be ideal candidates for embedding Trojans due to the low controllability and observability values. These Trojan instances are generally much harder to detect and will be the threat model we use for the remainder of the paper.

Algorithm 2 Constraint Generation Algorithm

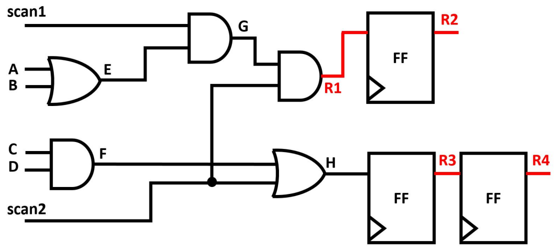
```
1: Input: Design $D$, set of rare nodes $R$
2: Output: set of signal traces $S$
3: procedure CONSTRAINTGENERATION($D$, $R$)
4:     $S$, $P$ = {}
5:     replace scan FFs with pseudo-primary inputs
6:     for each $r \in R$ do
7:         $P_i = assert\ G\ !(r\_cond)$
8:         $S_i$ = modelChecker($P_i$,$D$)
9:     return $S$
```

B. Constraint Generation

Because ATPG performance generally suffers in the presence of non-scan sequential elements, we use model checking to generate traces transformed into constraint structures for the ATPG tool and facilitate test generation for rare nodes with non-scan FFs in their fan-in. Suppose a design *D*, has *m* scan elements and *n* non-scan elements. The potential state space is 2^{m+n}. To reduce the state space for use in model checking, we create that has all scan FFs replaced with pseudo-primary inputs. This is equivalent to breaking up a scan-chain of *m* elements into *m* separate scan-chains. As a result of this



(a)



(b)

Fig. 2: (a) Design before scan replacement. (b) Design after scan replacement with pseudo-primary inputs scan1 and scan2

replacement, the model checking tool now has an effective state space of 2^n mitigating the state explosion issue. It is important to note that design can still have a significant amount of state space after the scan replacement. In such cases, the design can be fed to the model checking tool at the component level, helping reduce the state space even further.

Example 1: Let us consider a sample design A given in Figure 2 (a). After simulating the design for millions of cycles with random input, the rare branches are identified and the corresponding nodes are marked in red R1-R4. All scan FFs are then replaced with pseudo-primary inputs as shown in Figure 2 (b) for use in constraint generation.

The synthesized design is automatically converted to the tool's intermediate representation. For each rare node in the design identified from the first step, we generate properties (P_i) expressed in LTL as:

$$P_i : \ assert\ G\ !(ActivationCond)$$

The negation of the rare node activation is specified as properties in order to output the trigger condition as a counter-example trace from the model checker tool.

Algorithm 2 generates a set of signal traces to be used in ATPG. The algorithm takes the design, D, and replaces the scan FFs with pseudo-primary inputs. A property specified as the negation of the activation is generated for each rare node $r \in R$. Model checker then outputs a signal trace for each property, which is used in Algorithm 3. Note, if the traces from this algorithm are invalid, ATPG will not generate a valid test vector due to justification conflicts.

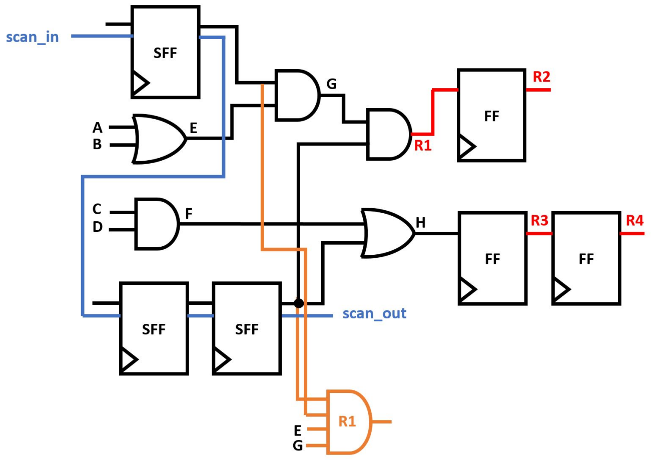


Fig. 3: Rare node R1 Counter-example trace from model checker captured in stuck-at circuit

Example 2: Suppose we want to generate a trace for rare node R1 (low 1 signal probability). From Algorithm 2, the property $assert\ G\ !(R1)$ (G == always temporal operator) along with the scan replaced design from Figure 2 (b) are run through the model checker. One possible signal trace for the activation levels would be {1,1,1,X,1,1} ("X" = don't care) for signals {scan1, scan2, A, B, E, G}.

Algorithm 3 Test Vector Generation Algorithm

```
1:  Input: Design $D$, Rare nodes $R$, signal traces $S$
2:  Output: set of test vectors $T$
3:  procedure TESTVECTORGENERATION($D, R, S$)
4:      $T$, $F$ = {}
5:      add ATPG AND primitive from $S$
6:      for each $r \in R$ do
7:          $F_i$ = addStuckAtFault($r_i$)
8:          $T_i$ = runFullSequentialATPG($F_i$)
9:          if $F_i$ == AU then
10:             $T_i$ = runJustification($r_i$)
11:             $T = T \cup T_i$
12:     return $T$
```

C. Test Vector Generation

We cannot expect random test patterns to reliably activate rare or suspicious nodes in a circuit. Therefore, we use ATPG with N-detect testing to generate directed tests patterns for the remaining circuit. The activation levels of all relevant internal signals from the suspicious node's fan-in cone and scan replacements are extracted from the trace and combined together with an ATPG primitive AND gate referred to as stuck-at circuit. The addition of these primitives are for test generation purposes only and have no effect on the design functionality. A stuck-at 0 fault is added to the tool's fault list for each stuck-at circuit. The ATPG tool is then run using full-sequential ATPG to generate test vectors that trigger each fault. If the stuck-at faults from the modified design are undetectable by the ATPG tool, we then attempt to justify the rare node trigger condition to generate a pattern. With our framework, the ATPG tool experiences a much faster execution time because the complexity of non-scan sequential ATPG is removed by utilizing a model checker for that portion shown in the results. In the event that no test pattern can be generated for a rare node due to justification conflicts, we cannot say anything about the existence of a Trojan in the design.

In Algorithm 3, we use the design D, rare nodes R, and signal trace S from model checking as input and output a set of test vectors T. The test vectors and fault list, F, are initially empty sets. We build the design in the ATPG tool then create stuck-at circuits using ATPG primitives and the traces generated in the previous step. Faults for each rare node r stuck-at circuit are added to the ATPG's fault list F. Full-sequential ATPG is run with the current fault list to generate patterns. If the ATPG tool returns AU (ATPG untestable) no test pattern is generated. We then attempt to run justification to generate a test pattern.

Example 3: Let us consider the design after the constraints are generated from model checking. We provide the design from Figure 1, the list of rare nodes, and signal traces to the ATPG tool. The relevant traces are transformed into a stuck-at circuit with ATPG AND primitives as shown in Figure 3. A stuck-at 0 fault is added for each stuck-at circuit and the ATPG tool is run. An example test vector from ATPG for a stuck-at 0 fault at stuck-at circuit R1 would be:
scan-in: $\{1, 1, 1\}$ *primary inputs* (A,B,C,D) : $\{1,1,1,1\}$

V. Experiments

A. Experimental Setup

To evaluate our approach, we implemented the framework described above and applied it on two AES-128 and RS232 benchmarks from Trust-Hub benchmark suite. More information on the Trojan circuits and their implementation can be found on Trust-Hub [20], [21]. Additionally, two modified AES-128 benchmarks (cb_aes) are used to showcase the limitations of model checking and ATPG. Both custom AES benchmarks are a subset of the AES module and only include 15 and 20 key rounds, respectively. The Trojan (a comparator and FF) checks the final output of the last round against a predetermined output. If they match, the secret key is leaked through the primary output.

A machine with Intel Core i5-3470 CPU @ 3.20GHz and 8 GB of RAM is used for testing. For each benchmark, the design is simulated for millions of cycles to identify the rare branches. The benchmarks are then synthesized using design compiler with DFT scan insertion with no area or power optimization. In mapping rare branches to rare nodes for our experiments, identifying boundary signal naming from primary inouts, and registers was sufficient.

A subset of the FFs are randomly selected for scan in an iterative process to maintain a high test coverage for partial-scan insertion. To make it harder to detect and showcase the efficiency of our approach, Trojan activation and payload FFs are excluded from the scan-chain effectively simulating a scenario in which an adversary would insert a Trojan in hard to detect areas after scan-chain insertion. With the rare nodes identified and scan chain inserted, the scan FFs are then replaced with primary inputs and given to the SMV tool [22] for model checking. Properties for each rare node are written in LTL and given to the model checker along with the design, which is converted from Verilog to .smv format. The resulting counter-example traces are then given to Synopsys TetraMAX [23] for ATPG N-detect with N = 10. Stuck-at circuits are constructed from the traces using ATPG AND primitives and corresponding stuck-at 0 faults are added to the tool's fault list. For a Trojan to be detected, its effect must be propagated to an observable output. Therefore, the final phase in our approach is to translate the test vectors into testbenches. By targeting rare nodes, the test vectors generated from N-detect ATPG have an increased likelihood of activating a Trojan. The suspicious design is simulated using ModelSim with the resulting testbench and its output is compared (XOR) with either a golden model or functional specification from the IP vendor to detect the presence of a hard-to-detect functional Trojan. Some constraints were imposed on the design due to tool limitations. For example, in SMV results are not accurate in designs with multiple clock domains and in TetraMAX sequential elements with multiplexed clocking are not allowed. In order to maximize the benefits of using ATPG and model checking approaches, our framework also includes parallel execution of them. The test pattern is taken from whichever execution generates results first.

B. Results

The experimental results from designs with partial-scan insertion are described in Table 1. The benchmarks are listed in the first column. Columns 2 and 3 describe the percent of FFs that are included in the scan-chain and the corresponding test coverage. Column 4 shows the number of rare branches identified from our initial analysis simulation. Columns 6, 8, 10, and 12 report the CPU time in generating test vectors for each approach. In columns 5, 7, 9, and 11 we show whether the test vectors detected the Trojan. Finally the last three columns show the improvement of our framework over ATPG, model checking, and MERO.

For AES and RS232 circuits, sequential ATPG alone outperforms model checking and the combined approach. Note that the combined approach still generates a test vector in comparable time, yet we take the test pattern resulting from ATPG as it finishes first. The execution time difference between AES and RS232 can be attributed to the location of the Trojans. Both AES benchmarks have Trojans that compare the state (primary input) to a predetermined value. On the other hand, in RS232, the Trojans are activated as a result of internal sequential signal combination.

The custom benchmarks are used to illustrate the weaknesses in both sequential ATPG and model checking. cb_aes_15 has a total of 5889 FFs, 883 non-scan FFs, and a max non-scan sequential depth of 3. cb_aes_20 has a total

TABLE I: Comparison of our approach against ATPG, Model Checking, and MERO for Trojan Detection with Partial-Scan. MO indicates memory overflow with 8GB of RAM. TO indicates timeout after 57600s

Benchmarks	Scan FFs (Scan/Total)	Test Cov.	#Rare Bran.	ATPG		Model Chk. (MC)		MERO		Our Approach		Improvement over		
				Detect	Time	Detect	Time	Detect	Time	Detect	Time	ATPG	MC	MERO
AES-T1000	6448/6933	99%	2	✓	0.02s	✓	85.6s	X	TO	✓	8.80s	1x	4280x	–
AES-T2000	6468/7108	99%	5	✓	0.90s	✓	216.5s	X	TO	✓	22.0s	1x	241x	–
RS232-T400	30/59	97%	2	✓	0.24s	✓	3600s	✓	2810s	✓	0.52s	1x	15000x	11708x
RS232-T800	26/58	97%	1	✓	0.06s	✓	7.23s	✓	3157s	✓	0.12s	1x	120x	52617x
cb_aes_15	5006/5889	99%	1	✓	28800s	X	MO	X	15720s	✓	7.85s	3669x	–	2003x
cb_aes_20	7262/7809	99%	1	✓	28800s	X	MO	X	16740s	✓	38.3s	752x	–	437x

"✓" indicates Trojan detected, "X" indicates Trojan not detected, "–" indicates not applicable

of 7809 FFs, 547 non-scan FFs, and a max non-scan sequential depth of 3. We can see the ATPG tool took a significant amount of time in generating test patterns to trigger the rare branch. Similarly model checking fails to generate a pattern due to state explosion and the tool experiencing a memory overflow (MO). Our proposed hybrid technique is able to generate test vectors even when the circuit structure has sufficient non-scan FF depth and structure.

We also compared our approach to MERO [2]. From Table 1, it is evident that our framework outperforms this approach in partial-scan designs. Because MERO uses ATPG to justify trigger conditions, it experiences significant execution overhead. This fact is illustrated in the AES benchmarks which experience a timeout (TO). In the case of the custom AES benchmarks, MERO did not detect the Trojans because the justification portion could not activate the trigger condition given the amount of non-scan sequential elements in the design.

Our experimental results highlight two important aspects of our test vector generation framework. First, we achieve acceptable CPU times for all the test benches covered. Second, our approach is able to generate test vectors that ATPG and model checking were unable to finish in a reasonable time. Our results show that we can achieve up to four orders of magnitude faster execution times than state-of-the-art methods. This speed up is achieved by leveraging the strengths of each tool. Specifically, reducing the state space of model checking and removing the non-scan sequential complexity encountered using ATPG.

VI. Conclusion

Trust in SoC design is an ever increasing concern as more companies are including third party IPs gathered from untrusted vendors. Current Trojan detection methods that utilize ATPG and model checking tools cannot effectively handle partial-scan designs. This limitation can cause a significant execution time penalty due to the complexity of non-scan sequential ATPG or complete failure in model checking from state explosion. We proposed a framework that combines ATPG and model checking for hardware Trojan detection in partial-scan designs. Experimental results demonstrated the merits and weaknesses in both approaches and the effectiveness of combining them in case of partial-scan designs for generating test vectors targeted at hardware Trojan detection. We plan to extend our approach using Groebner basis based formal verification to help with larger circuits that model checking cannot handle [16]. Future work will also include investigating more partial-scan benchmarks with differing sequential structure and depth to further explore the effectiveness of the proposed framework.

References

[1] M. Tehranipoor and C. Wang, *Introduction to Hardware Security and Trust*, Springer, 2011.
[2] R. Chakraborty et al., "MERO: A statistical approach for hardware trojan detection", CHES, 2009.
[3] X. Zhang and M. Tehranipoor, "Case Study: Detecting Hardware Trojans in Third-Party Digital IP Cores", HOST, 2011.
[4] S. Bhunia et al., "Hardware Trojan Attacks: Threat Analysis and Countermeasures", IEEE Special Issue on Trustworthy Hardware, 2014.
[5] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection", IEEE Design & Test, 2010.
[6] J. Rajendran et al."Formal Security Verification of Third Party Intellectual Property Cores for Information Leakage" , VLSI Design, 2016.
[7] V. Chickermane and J. H. Patel, "An Optimization Based Approach to the Partial Scan Design Problem", Test Conference, 1990.
[8] T. E. Marchok et al. "Complexity of Sequential ATPG" , European Design and Test Conference, 1995.
[9] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-signal VLSI Circuits*, Springer, 2004.
[10] E. M. Clarke et al. *Model Checking*, MIT press, 1999.
[11] H.-M. Koo and P. Mishra, "Test Generation using SAT-based Bounded Model Checking for Validation of Pipelined Processor", GLSVLSI, 2006.
[12] I. Pomeranz and S. M. Reddy, "On Undetectable Faults in Partial Scan Circuits" , ICCAD, 2002.
[13] F. Wolff et al. "Towards Trojan-free Trusted ICs: Problem Analysis and Detection Scheme", DATE, 2008.
[14] M. Banga and M. S. Hsiao, "Trusted RTL: Trojan Detection Methodology in Pre-silicon Designs", HOST, 2010.
[15] P. Mishra et al., *Hardware IP Security and Trust*, Springer, 2016.
[16] F. Farahmandi et al., "Trojan Localization using Symbolic Algebra", ASP-DAC, 2017.
[17] Y.-C. Hsu et al., "Visibility Enhancement for Silicon Debug", DAC, 2006.
[18] A. Waksman et al., "FANCI: Identification of Stealthy Malicious Logic using Boolean Functional Analysis,", CCS, 2013.
[19] Y. Huang et al., "MERS: Statistical Test Generation for Side-channel Analysis based Trojan Detection", CCS, 2016.
[20] M. Tehranipoor et al., "Trust-hub.org.",http://www.trust-hub.org/
[21] H. Salmani et al., "On design vulnerability analysis and trust benchmarks development", ICCD, 2013.
[22] K. McMillan, "Symbolic Model Checking", Kluwer, 1993.
[23] Synopsys, "TetraMAX User Guide," *Version H-2013.03-SP4*, 2013.
[24] S. Saha et al., "Improved Test Pattern Generation for Hardware Trojan Detection using Genetic Algorithm and Boolean Satisfiability", CHES, 2015.
[25] F. Farahmandi et al., "Cost-Effective Analysis of Post-Silicon Functional Coverage Events", DATE, 2017.