Scalable Hardware Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution

Alif Ahmed
University of Florida
Gainesville, FL, USA

Farimah Farahmandi
University of Florida
Gainesville, FL, USA

Yousef Iskander
Cisco Systems Inc.
Knoxville, TN, USA

Prabhat Mishra
University of Florida
Gainesville, FL, USA

***Abstract*—Intellectual Property (IP) based System-on-Chip (SoC) design is a widely used practice today. The IPs gathered from third-party vendors may not be trustworthy since they may contain malicious implants (hardware Trojans). To avoid the detection of the Trojan, adversaries usually hide it under rare branches or rare assignments triggered under extremely rare input sequences. Due to exponential input space complexity, state-of-the-art constrained-random test generation methods are not suitable for activating these rare scenarios. While existing model checking based directed test generation approaches are promising, they are not capable of generating tests for large RTL designs due to the capacity restrictions of formal methods. In this paper, we propose an automated and scalable test generation approach for activation of hardware Trojans in RTL designs. This paper makes three important contributions. First, it provides a scalable test generation framework by effective utilization of symbolic execution and concrete simulation. Next, it is a fully automated approach for generating directed tests for activating rare branches and rare assignments. Finally, our experimental results demonstrate that the generated tests are able to activate hard-to-cover Trojans in large and complex RTL benchmarks.**

I. Introduction

Modern System-on-Chip (SoC) designs consist of a wide variety of computation, communication and storage related Intellectual Property (IP) blocks. Developing and verifying each of these IP blocks in-house is infeasible due to time-to-market and budget constraints. It is a common trend in industry to rely on third-party IPs to keep the cost low and to meet firm deadlines. However, using IPs gathered from third-party vendors introduces security and trust concerns. These IPs may come with hardware Trojans inserted by an adversary. These hardware Trojans can be hidden in a way such that they are triggered under extremely rare input sequences. As a result, traditional validation approaches are unable to activate them. A Trojan can leak secret information, create backdoor for attackers, alter functionality, degrade performance, halt the system, etc. [1]–[4]. Therefore, it is crucial to have effective validation techniques to detect hardware Trojans.

Trojan detection methods based on side-channel analysis monitor changes in physical characteristics such as power and delay [5]–[8], [65]. However, these approaches cannot detect functional hardware Trojans since they usually consist of a few gates which have a negligible impact on physical characteristics. The other class of methods relies on statistical parameters to distinguish Trojan-inserted circuits from Trojan-free circuits [9], [10]. Unfortunately, many times these methods provide false positives even if the circuit is Trojan-free. Logic testing based methods focus on functional comparison instead of looking at side-channel signatures. These techniques require input vectors for activating the Trojan to measure the deviation from expected behavior. Researchers have proposed model checking approaches [11] for activating hardware Trojans. However, these methods suffer from inherent capacity restrictions of formal methods while dealing with large designs. Therefore, such methods are not effective for activating hardware Trojans in complex register-transfer level (RTL) models.

In this paper, we propose a directed test generation method to activate potential hardware Trojans in RTL designs using concolic testing. Concolic testing is an effective combination of concrete simulation and symbolic execution. Unlike formal methods, concolic testing is scalable since it can avoid state space explosion by exploring one execution path at a time. While concolic testing has shown promising results in software verification domain [12], [13], it has not been explored in the context of test generation for detecting hardware Trojans. This paper makes the following four important contributions.

- To the best of our knowledge, our proposed approach is the first attempt in developing an automated and scalable technique to generate directed tests to activate hardware Trojans in RTL models.
- We develop a threat model involving rare branches and rare assignments for RTL designs. This threat model leads to a list of potential security targets for directed test generation.
- We propose an effective combination of concrete simulation and symbolic execution to generate directed tests to activate these security targets.
- We show that detection of hardware Trojans boils down to coverage of rare branches and assignments in RTL models. Our experimental results demonstrate the effectiveness of our approach by activating hard-to-detect Trojans in large and complex benchmarks.

The remainder of the paper is organized as follows. We discuss related work in Section II. We provide an overview of concolic testing in Section III. We present our threat model in Section IV. Section V describes our test generation framework for Trojan detection. Section VI presents our experimental results. Finally, Section VII concludes the paper.

978-1-5386-8382-8/18/$31.00 ©2018 IEEE

II. Related Work

Existing Trojan detection techniques can be broadly classified into the following four categories: side channel analysis, statistical methods, formal verification, and functional test generation.

A. *Side-Channel Analysis*

Existing techniques based on side channel analysis rely on the change of physical characteristics caused by the Trojan circuit - mostly in the form of current, power or delay [6], [14], [15]. When a Trojan is partially or fully activated, it would increase the switching activity compared to Trojan free circuit. Wang et al. used this property to isolate Trojan [5]. MERS utilized test generation to improve the Trojan detection sensitivity [7]. Their approach selected the nodes with low transition probability as suspicious nodes. Then test vectors are applied in such a way that switching activity of these suspicious nodes become much higher than other nodes, increasing side-channel emission. Side-channel based approaches face difficulty if the Trojan circuit is small. This is because of the difference in side channel signature due to the Trojan can be negligible compared to process variations. These methods also require Trojan free golden reference models. As side-channel analysis is carried out after fabrication, the chip may require re-spins if Trojan is detected. Thus, methods that can detect Trojan in the design stage is highly desirable.

B. *Statistical Methods*

Statistical Trojan detection methods try to differentiate the Trojan-inserted circuit from the Trojan-free version using properties of known Trojans. FANCI is one such approach [9]. FANCI marks gates that weakly influence output signals as suspicious. Their proposed algorithm uses approximate truth table for each signal to infer its effect on the outputs. However, FANCI has a high false positive rate. A similar method named VeriTrust marks redundant logic gates as suspicious [10]. Initially, all gates that are not covered during verification phase are considered as suspicious nodes and further analysis is carried out to confirm redundancy. FANCI and VeriTrust can detect only Trojans with always on or combinational type triggers (a trigger that depends only on current inputs). They cannot detect sequential Trojans, which is exploited by DeTrust benchmarks [16]. Hicks et al. proposed an approach for defeating Trojan based on unused circuit detection [17]. This method relies on the assumption that Trojan circuits will reside on unused portion of the circuit. However, their algorithm failed to detect Trojans that do not rely on unused circuits [18].

A score based classification method for detecting Trojan is discussed in [19]. The classification features are based on properties found from Trojans in Trust-Hub benchmarks [20]. Scores are given to nets for each of the matching features. Nets with score above a threshold are marked as Trojan nodes. Unfortunately, these features are too specific to Trust-Hub benchmarks and thus cannot be used as a generic detection method. A recent approach proposed by Salmani et al. [21] uses SCOAP[1] controllability and observability values to detect and isolate Trojan nodes. Controllability is defined as the number of primary inputs that must be manipulated to control a signal to a particular logic value. Observability is the number of primary input manipulations which is required to make a signal observable at the primary outputs. This method works using the assumption that Trojan nodes will have higher controllability/observability values to avoid detection. However, this approach will result in false positives in designs with partial scan chains. Benign signals that are not part of the scan chain will also have controllability/observability values similar to Trojans. Recently, a Trojan clustering approach based on signal correlation is proposed in [23]. However, this method is suitable for gate-level designs, and cannot be extended to RTL models for early detection.

C. *Formal Techniques*

Researchers have proposed techniques based on formal methods to prove security-related properties that would be violated in the presence of Trojans. These methods are particularly effective for detecting Trojans inside cryptographic designs. One such method - GLIFT, looks for confidentiality and integrity property violation [11]. Confidentiality property requires that secret information never leaks to an unsecured domain and integrity property requires that untrusted data never enters the secured domain. Information flow is traced by assigning a taint bit to it. In another approach [24], a base property is used to detect information leakage which may imply the existence of a Trojan. The base property checks whether any input sequence exists such that it triggers secret information leakage to an observable point. The authors have proposed another test generation technique for Trojan detection using bounded model checking (BMC) [25]. This technique looks at the critical data registers such as processor stack pointer, router address table or cryptographic keys to generate security properties. However, the strength of these approaches is dependent on the quality of the security properties. Writing meaningful security properties that can detect Trojans often requires manual effort and full insight about internal circuit operations [66].

Equivalence checking is another way of formally proving a circuit is Trojan free. Such approaches require a golden specification to verify if it is equivalent to the implementation. Trojan inserted implementation will demonstrate functionality outside of the specification. However, traditional equivalence checking techniques suffer from state explosion problem. Several approaches have been proposed using Gröbner basis theory to alleviate this problem [26]–[30], [64]. These techniques express both specification and implementation as polynomials, and reduce the specification polynomials over a subset of implementation polynomials. If both are equivalent, then the reduction procedure should result in a zero remainder. Any non-zero remainder indicates deviation from the specification. Such methods not only detect the existence of a Trojan, but

[1]SCOAP: Sandia Controllability/Observability Analysis Program [22]

also can isolate the Trojan circuit and generate test vector for activation. Unfortunately, these approaches do not work when the specification is not available.

D. Functional Test Generation

Chakraborty et al. proposed MERO [31] which excites the rare nodes multiple times in order to increase the likelihood of Trojan activation. It is extremely difficult to detect the Trojans using such statistical tests due to the stealthiness of activation condition. Besides, this technique is only applicable to gate-level designs and does not guarantee whether the generated tests can activate the Trojans. Usually complete coverage is required to detect Trojans [32]. Researchers have explored other gate-level test generation techniques, like Automated Test Pattern Generation (ATPG). Cruz et al. have proposed a test generation technique that combines the strength of model checking and ATPG for efficient test generation [33]. Their approach partitions the design based on the scan chain. Constraints are generated for non-scan elements using model checking. These constraints as well as the scan elements are then given to ATPG for test generation. This approach is suitable only for partial scan-chain inserted designs.

In summary, none of the existing techniques can effectively activate hardware Trojans in RTL designs. In this paper, we present a scalable directed test generation method for Trojan activation using an effective combination of concrete simulation and symbolic execution of RTL models.

III. Background: Concolic Testing

Concolic testing generates tests by effective combination of concrete simulation and symbolic execution. The idea was first demonstrated in software domain, and later applied on hardware designs [35]–[38]. Figure 1 presents an overview of the concolic testing methodology [34]. Depending on the objective of the test generation, concolic testing can maximize coverage by forcing execution through different branches or can guide the execution towards a specific branch. It does so by alternating between concrete simulation and symbolic execution of the design. The first step involves the simulation of the design. For initial simulation, usually random inputs are used. The execution path taken by the simulation can be decomposed into a set of constraints, referred as *path constraints* ($C = < c_1, c_2, ..., c_n >$). Next step is to force

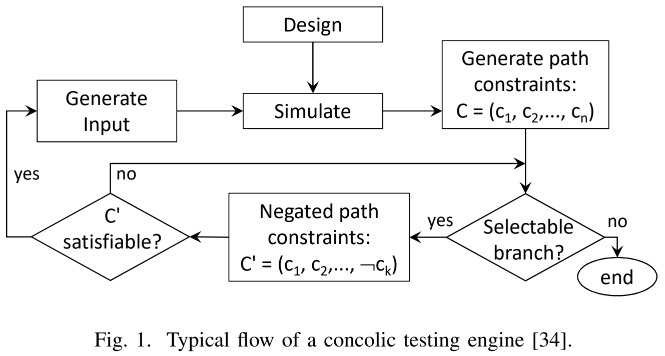
Design
Generate path constraints: C = (c_1, c_2,..., c_n)
Generate Input
Simulate
yes
no
C' satisfiable?
Negated path constraints: C' = (c_1, c_2,..., $\neg c_k$)
yes
Selectable branch?
no
end

Fig. 1. Typical flow of a concolic testing engine [34].

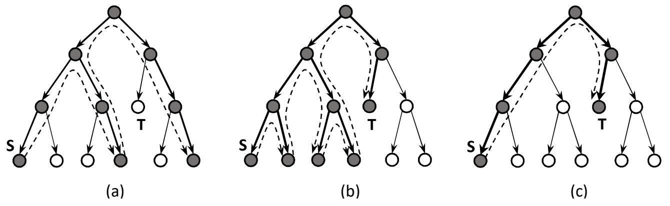

Fig. 2. CFG traversal of different concolic testing methods. S is the start (current) node and T is the target. Covered nodes are dark colored. (a) Random selection: T is not covered. (b) Uniform tests using concolic testing: T is covered after many iterations. (c) Directed tests using proposed approach: T is quickly covered.

the execution through an alternate branch. In order to do so, constraint of the selected branch (c_k) is negated and the desired path constraints for this alternate path is formed ($C' = < c_1, c_2, ..., \neg c_k >$). These new path constraints are then symbolically solved using a constraint solver. If the solver comes up with a solution input set, then for that input execution will go through this alternate branch. If no solution is found, another branch is selected for negation. These steps are repeated until required target branch is reached, or there is no solvable branch left. Other termination criteria such as timeout or coverage goal can also be used. Concolic testing avoids state explosion issue by exploring only one path at a time, instead of trying to explore all possible paths at once. This advantage makes it an attractive choice for large and complex designs.

Depending on the objective, different strategies can be adopted for alternate branch selection as shown in Figure 2. The simplest one is random or constrained-random selection [39] (Figure 2(a)). However, it is not suitable for our goal of covering particular suspicious branches. As this strategy is random, it does not provide any guarantee of covering the target branch (shown as T in the figure). An alternate strategy is uniform selection, where the goal is to maximize branch coverage (Figure 2(b)). Most concolic testing based test generation techniques use this strategy [12], [13], [34]–[38], [40]–[43]. Uniform search is effective if we want to cover all possible branches. For our goal of Trojan activation, uniform search will eventually reach the target node but may take infeasibly long time. This is mostly because such search strategies do not prioritize a particular branch. The third strategy is directing execution towards a particular target node (Figure 2(c)). This strategy is most suitable for our objective as it will enable us to quickly cover the suspicious nodes without going through unnecessary branches. There has been extensive research on directed test generation using different methods [44]–[51]. Researchers have used concolic testing for directed test generation as well [39], [52]–[55]. However, these concolic testing methods only consider sequential execution models and are not applicable on hardware (concurrent) designs. In this paper, we propose a concolic testing based directed test generation approach for RTL designs. Our proposed method utilizes distance feedback to quickly reach the desired security targets.

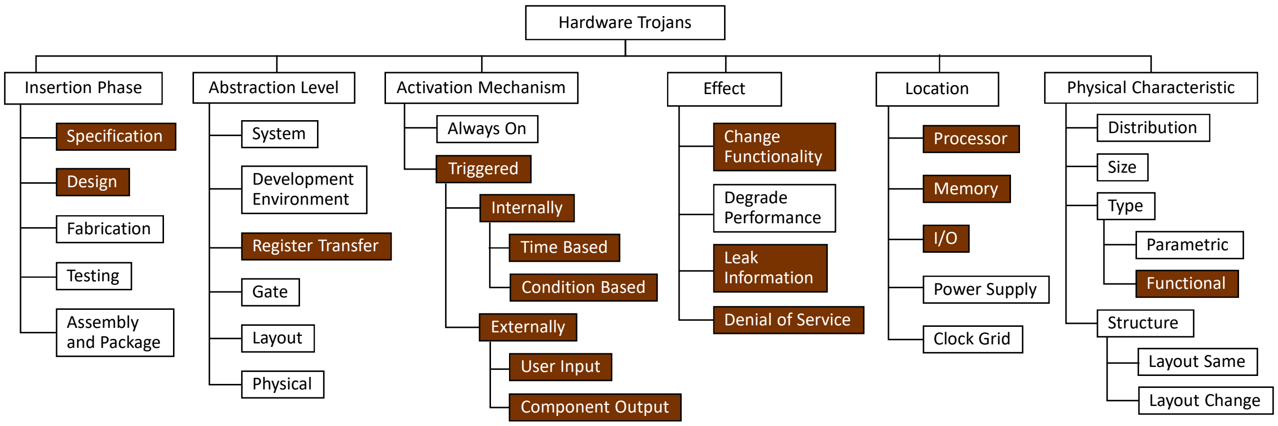

Fig. 3. Hardware Trojan taxonomy [3], [56], [57]. A wide variety of Trojans (shown in brown shaded box) can be activated using our proposed method.

IV. Threat Model

Figure 3 gives an overview of the types of Trojans that can be detected using the proposed approach. Although we have applied our method on RTL designs, it can be easily extended to higher levels of abstraction such as transaction-level models. It is possible to extend our approach on gate-level models if we have a clear mapping for branches. As our detection algorithm utilizes test generation for Trojan activation, Trojans that are always on cannot be covered by our approach. Side channel or unused circuit detection-based approaches are more suitable for such Trojans. On the other hand, our approach is effective when the Trojan must be triggered - either externally or internally.

An adversarial threat model for our proposed approach is presented in Figure 4. Our method detects Trojan during the verification of RTL design. Thus, Trojans inserted at specification or design stage can be caught. In our threat model, the untrusted-third party IPs can come with malicious implants. Untrusted EDA tools, in-house rogue employees or the SoC integrator can also insert hard-to-detect Trojans in the original RTL design. We assume that to escape detection during different steps of verification/validation procedure, Trojans are designed in such a way that only a very rare set of input sequences can trigger them. In other words, Trojans are dormant during the normal execution, and activated under unusual (rare) conditions. Therefore, a smart adversary is likely to insert Trojans in RTL designs under rare branches which may reside in the unspecified functionality of the design. Otherwise, traditional simulation-based techniques using random or constrained-random tests can detect them, and the attacker's attempt would fail. We have also considered rare continuous/concurrent assignments in our threat model. These are assignments that may not be under any branches. Therefore, there is a high chance that they are not covered by targeting rare branch coverage. We transform rare assignments to branches without changing the functionality of the design in order to generate tests to cover them. Therefore, our threat model boils down to covering only rare branches including both original and newly created ones (due to the conversion of rare assignments to branches).

Example 1: Suppose that an RTL IP contains an assignment statement which is as follows: $assign\ Tj\_Trig = count\_1\ \&\ count\_2$. Assuming Tj_Trig is the trigger signal of the hidden Trojan and it becomes true when both $count\_1$ and $count\_2$ overflow at the same time. Signal Tj_Trig becomes rare for value '1' if two of the counters are large enough. In order to consider such rare assignments in the test generation phase, we convert the assignment to a conditional statement as follows. ■

Listing 1. Converting an assign statement to a conditional statement.
```
if (count_1 & count_2)
        Tj_Trig <= 1;
else
        Tj_Trig <= 0;
```

In this paper, we aim to activate the Trojans that change the functionality (e.g., causing information leakage or denial of service) and they can be triggered internally or externally. There may be cases when the Trojan's trigger is dependent on several rare branches. However, the trigger should be used somewhere in the design to activate the malicious functionality. Since the trigger value is rare, the branches/assignments that use the trigger would be rare to be activated, consequently. Therefore, by covering all of the rare branches and assignments, we can activate hidden Trojans in RTL designs.

While we use rare branches and rare assignments as our threat model in this paper, our approach can easily incorporate suspicious nodes marked by other methods such as Transition Probability Calculator (TPC) [20], FANCI [9], VeriTrust [10] as well as score-based classification methods [19] to perform our Trojan detection analysis. Moreover, while this paper uses Verilog examples, our approach is equally applicable on VHDL designs.

V. Test Generation for Trojan Activation

Figure 5 shows the overview of our proposed approach. It consists of three major steps: i) design instrumentation,

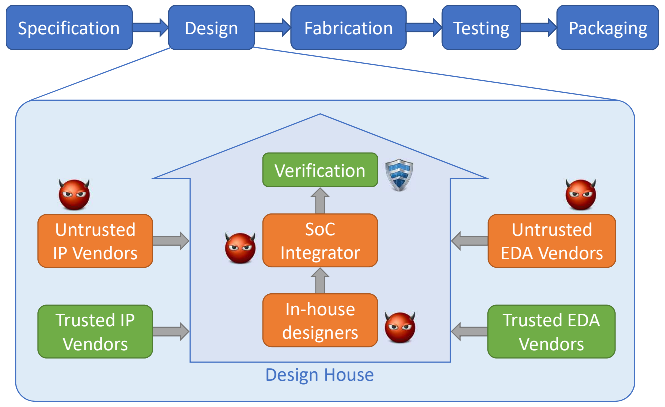

Fig. 4. Adversarial threat model for our proposed approach. Attacker can be an untrusted third-party IP vendor, untrusted EDA tool vendor, rogue in-house designer or SoC integrator itself. In our threat model, verification team is responsible for defending against attacks.

ii) obtaining security targets of the design based on the identification of the rare branches and assignments, and iii) directed test generation to activate the security targets. The remainder of this section describes these steps in detail.

A. Design Instrumentation

Design instrumentation is needed to trace the execution paths during simulation of the design. Instrumentation is done by inserting a *$display* statement for each functional statement. This insertion is automated and done during the abstract syntax tree (AST) generation phase of the simulator. Note that the instrumentation will not change the functionality of the design, since it only traces the executed statements. This trace is later used to identify rare branches as well as to generate path constraints for symbolic execution. The design needs to be instrumented only once.

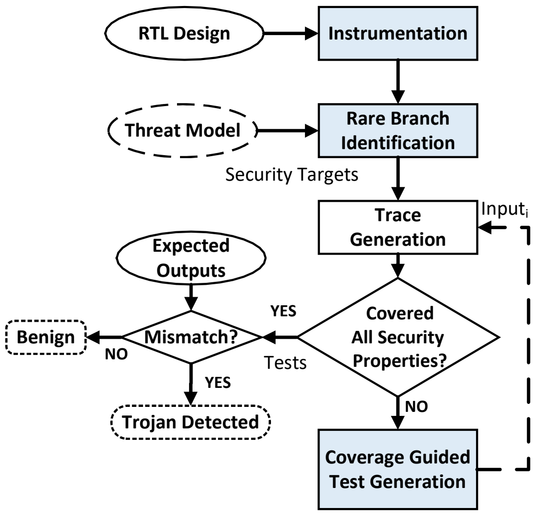

Fig. 5. Proposed hardware Trojan activation framework consists of three main tasks: i) design instrumentation, ii) finding suspicious branches and assignments, and iii) test generation for Trojan detection.

Example 2: Listing 2 shows the instrumented version of Trojan circuit in AES-T1000 benchmark [20] where "c" shows the current clock cycle (display statements are added). The Trojan is triggered when *state* variable becomes a particular value from 2^{128} possible values ($state = specific\_value$)[2]. ■

Listing 2. Instrumented version of a part of the code in AES-T1000 benchmark [20]

```
always @(rst , state)
  begin
  if (rst==1) begin
    $display("rst==1,c");
    Tj_Trig <= 0;
    $display("Tj_Trig==0,c+1");
  end
  else if (state == specific_value)
  begin
    $display("state==specific_value ,c");
    Tj_Trig <= 1;
    $display("Tj_Trig==1,c+1");
  end
end
```

B. Identification of Suspicious Branches

Random simulation is utilized to find rare branches which can potentially host hardware Trojans. We simulate the instrumented design using random inputs. Next, the number of times each branch is covered is counted. The branches that are covered less than a threshold number of times are marked as suspicious branches. For example, having a threshold of zero implies only uncovered branches as suspicious. In our experiments, we have used a threshold of zero. It gives the lowest probability of false positive. All the branches that fall within the threshold are considered as security targets for the proposed Trojan activation framework. Using random testing is not only useful to mark rare branches, but it is also beneficial to reveal existing bugs in the design. Moreover, if an attacker does not insert a Trojan smartly, there is a high chance that the Trojan may be activated during random testing. As mentioned before, other methods of detecting suspicious branches are equally applicable.

Example 3: Consider the Trojan circuit shown in Listing 2, the *Trj_Trig* signal remains zero most of the time. However, when the *state* input gets the rare value shown in line 9, the *Trj_Trig* signal is activated. The chance of the branch shown in line 9 being covered during random test simulation is extremely low (probability of $1/2^{128}$) and most likely it will not be covered. Therefore, our method marks this branch as a rare branch (security target). ■

After identifying rare branches, we model conditions of each rare branch as a security target such that the branch will be taken if the conditions are evaluated true. The security targets are used by our test generation framework to produce the input

[2] $specific\_value = 128'h00112233\_44556677\_8899aabb\_ccddeeff$

conditions (directed tests) to activate the respective rare branch in order to make sure that no Trojan or malfunction resides inside those rare branches.

Algorithm 1 Security Targets Identification

```
 1: Input: Design under test DUT, Threshold $\pi$
 2: Output: Set of security targets
 3: Security target set, $\mathbb{P} = \{\}$
 4: Instrumented design, $DUT'$ = instrument(DUT)
 5: Input vector $I$ = random()
 6: Path trace $\Phi$ = simulate($DUT', I$)
 7: $B$ = identifyRareBranches($DUT'$, $\Phi$, $\pi$)
 8: $A$ = identifyRareAssignments($DUT'$, $\Phi$, $\pi$)
 9: for each $a \in A$ do
10:     $B = B \cup$ createEquivalentBranch(a)
11: end for
12: for each $b \in B$ do
13:     $P$ = createSecurityTarget($b$)
14:     $\mathbb{P} = \mathbb{P} \cup P$
15: end for
16: return $\mathbb{P}$
```

Algorithm 1 shows the procedure to mark security targets. The algorithm takes the design under test (DUT) as well as threshold π and it produces a set of security targets $\mathbb{P}$ as output. To trace the execution path, the design is instrumented (line 4). Then, the design is simulated using random input vectors and simulation traces are stored in Φ (lines 5-6). After collecting the simulation trace, branches that are not covered more than threshold times are marked as rare branch (line 7). Subsequently, these branches are added to set B. Note that for each branch b, two conditions are considered: i) b is taken, and ii) b is not-taken. For example, consider the branch $if(var == 1)$. If the value of var was always 1 during the random simulation, then only *taken* condition for this branch is covered. We have to add the *not-taken* condition to set B as not covered. The same procedure is applied for identifying the rare assignments (line 8). Each rare assignment a is then converted to an equivalent branch and added to set B (lines 9-11). Finally, each element in B is converted to an assertion, and it is added to the output set $\mathbb{P}$ (lines 12-15).

Example 4: Consider the Trojan circuit shown in Listing 2. The branch shown in line 7, $if\ (state == specific\_value)$ has not been covered during random simulation, and thus it is marked as a rare branch. Therefore property $assert\ eventually\ state = specific\_value$ is added to the design for security validation. Similarly in Listing 1, since the condition ($count\_1 == 1$ & $count\_2 == 1$) is a rare event, we create a security target as:

$assert\ eventually\ count\_1 == 1$ & $count\_2 == 1$. ■

C. Coverage Guided Test Generation for Trojan Activation

Algorithm 2 takes an RTL design as well as security targets as inputs and generates directed tests to cover the security targets. First, we perform a preprocessing step to reduce the total number of security targets. The number of security targets

Algorithm 2 Test Generation for Trojan Activation

```
 1: Input: Instrumented design under test $DUT'$, Security
    targets $\mathbb{P}$
 2: Output: Set of test vectors $\mathbb{T}$
 3: $\mathbb{T} = \{\}$
 4: $\mathbb{P}'$ = pruneOverlappingTargets($\mathbb{P}$)
 5: Input vector $I$ = random()
 6: while $I$ is not null do
 7:     $\mathbb{T} = \mathbb{T} \cup I$
 8:     Path Trace $\phi$ = simulate($DUT', I$)
 9:     for each $P \in \mathbb{P}'$ and isCovered($P, \phi$) do
10:         $\mathbb{P}'$.remove($P$)
11:     end for
12:     if $\mathbb{P}$ is empty then
13:         Return $\mathbb{T}$          ▷ All security targets are covered
14:     end if
15:     $C$ = findConstraints($DUT', \phi$)
16:     for all uncovered branches $C$ do
17:         $b$ = branchWithLeastDistanceFromTarget($C$, $p$)
18:         $b_n = \neg b$
19:         $I$ = satisfy($C + b_n$)
20:         if $I != null$ then
21:             break                    ▷ To execute new input
22:         end if
23:     end for
24: end while
25: return $\mathbb{T}$
```

has a direct impact on the performance of the test generation approach. The number of targets can be reduced based on the dependency between them due to the fact that all branches within a rare branch are also rare. Covering the inside branch will also cover the parent branch, and thus it can be removed from the target list. Such dependency can be resolved by looking at the control flow graph (CFG) of the design. If a target is dominator of any other target, it can be pruned. An example is shown in Figure 6. Here (a) shows the initial targets as B, D, and E. However, B is a dominator of target D, hence can be removed. This is done statically, without unrolling the design for multiple cycles. The static analysis only prunes part of the dependent branches. Dynamic pruning with actual unrolling of design would result in more pruned targets, but we do not use it in this work since it is susceptible to state explosion.

After pruning step (line 4), one of the targets is selected for test generation. Distance from the target is then evaluated by running breadth-first search (BFS) starting from the target branch, and following predecessor edges in the CFG. An example is shown in Figure 6(d). Here, D is selected to be covered first. Initially, target D is assigned distance 0 and all other branches are assigned infinity. Next, we run BFS starting from D, and follow predecessor edge. After distance evaluation is finished, the distance would be: $B = 1$, $A = 2$ and others infinity. This procedure is also done statically without actually unrolling the design. Next, we apply concrete simulation

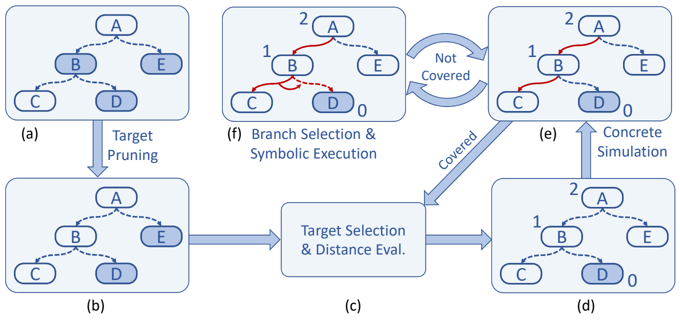
A
B
E
C
D
(a)
Target Pruning
2
1
0
(f) Branch Selection & Symbolic Execution
Not Covered
Covered
(e)
Concrete Simulation
(b)
Target Selection & Distance Eval.
(c)
(d)

Fig. 6. Overview of test generation procedure. Targets are shaded. (a) Initial targets. (b) Targets after pruning. (c), (d) Selects one target, and evaluates distance for that target. (e) Runs concrete simulation. Execution path is marked as red (solid line). (f) Selects an alternate branch and symbolically solves for input.

followed by symbolic execution for several iterations in order to generate tests to activate the potential hardware Trojan. In each iteration, the instrumented design is simulated for a specific number of clock cycles (i.e., unroll cycles) and a trace file is produced (Figure 6(e)). The information of the trace file is then converted into path constraints (line 15). These constraints model the execution path taken by the concrete simulation. In the next step, one of the alternate branches is selected to be explored. We have selected the branch which has lowest assigned distance value (line 17). In other words, we have given priority to the branches that are closer to our security target. Path constraints that lead to that branch is then symbolically solved by a constraint solver (line 19). If a solution exists, then we again do concrete simulation with that solution, this time forcing execution through that alternate branch. This concrete and symbolic execution steps are repeated until the target is covered (Figure 6(e)-(f)), or some terminating conditions are met (e.g., timeout). If all of the branches are exhausted and no new input vector I can be generated, algorithm returns generated tests (line 25).

Intuitively, our iterative procedure effectively guides the execution path towards the target. Our approach avoids the state space explosion by examining one path at a time in contrast to traditional formal methods that consider all of the paths simultaneously. Therefore, it is capable of activating hard-to-detect Trojans in large designs, as will be demonstrated in the experiments section.

Example 5: Consider the instrumented code in Listing 2. The concolic testing generates a test vector to activate the rare branch (line 7) where $rst = 0$ and $state = specific\_value$. This input vector makes Tj_Trig true, and therefore, activates the Trojan. ■

VI. Experiments

A. Experimental Setup

Experiments are performed using a 64-bit Red Hat Enterprise Linux server machine with Core-i5 3427U CPU and 16GB of RAM. The CPU has two cores running at 1.80GHz. Our hardware Trojan detection framework is implemented using Icarus Verilog Target API [58]. Icarus Verilog is also used for parsing and simulating the RTL design. Yices is used as the constraint solver [59]. Our approach is independent of the constraint solver, so other popular solvers such as Z3 [60], Boolector [61], etc. can also be used. EBMC model checker is used to perform property-based test generation [62], [63]. Our Trojan detection framework is tested with Trojan inserted designs from Trust-Hub benchmark suite [20]. Table I shows the benchmark characteristics. The selected benchmarks cover a wide range of hardware Trojans from the taxonomy given in Figure 3. Some additional custom designed benchmarks are used to demonstrate the scalability of our method. Security targets here are the rare branches. Rare branches are selected by simulating the design with random inputs for one million cycles. A branch is considered rare if it is not covered during the simulation.

TABLE I
Characteristics of Trojan inserted benchmarks used for experiments. These functional Trojans were inserted in RTL models during the design phase.

Benchmark	Activation Mechanism	Effect	Location
wb_conmax-T200	Condition Based	Change Functionality	I/O
wb_conmax-T300	Condition Based	Change Functionality	I/O
AES-T500	Time Based	Denial of Service	Processor
AES-T1000	Time Based	Leak Information	Processor
AES-T1100	Time Based	Leak Information	Processor
AES-T1300	Time Based	Leak Information	Processor
AES-T2000	Time Based	Leak Information	Processor
RS232-T100	Condition Based	Denial of Service	N/A
RS232-T200	Condition Based	Denial of Service	N/A
RS232-T400	Condition Based	Denial of Service	N/A
RS232-T800	Condition Based	Denial of Service	N/A
memctrl-T100	User Input	Degrade Performance Denial of Service	Memory
cb_aes_xx	User Input	Leak Information	N/A

B. Results

In this section, we show the efficiency of our framework to generate test vectors to activate the hidden Trojans. We compare our approach with bounded model checking tool EBMC [62], [63]. The model checker takes the design and the negation of the security targets and converts these to a set of conjunctive normal form (CNF) clauses. Then it uses a SAT solver to generate counterexamples. However, when the number of CNF clauses exceeds a threshold (e.g., about 30 million clauses), SAT solvers as well as BMC fail. To overcome the state explosion problem of BMC, we use concolic testing to generate tests to activate hard-to-detect hardware Trojans. Note that there are no existing approaches for rare branch activation using model checking for Trojan detection in RTL models. We provided these results to demonstrate the limited applicability of state-of-the-art formal methods on large designs. This also highlights the fact that our threat model and security targets can be utilized by existing approaches.

TABLE II
THE REQUIRED TIME AND MEMORY TO GENERATE DIRECTED TESTS THAT ACTIVATE THE TROJAN USING EBMC AS WELL AS OUR APPROACH IN RTL TRUST-HUB BENCHMARKS. MO = MEMORY OUT OF 16 GB.

Benchmark	Cycles Unrolled	Lines of Code[1]	#Rare Branches	Rare Branches Coverage	EBMC [63]		Our Approach		Time Improvement	Memory Improvement
					Time (sec)	Memory (MB)	Time (sec)	Mem (MB)		
wb_conmax-T200	10	63 k	1	100.00%	8.71	659.5	13.36	124.7	-1.53x	5.29x
wb_conmax-T300	10	63 k	1	100.00%	11.77	1198.9	11.06	118.8	1.06x	10.09x
AES-T500	10	455 k	5	100.00%	67.07	7436	11.67	599	5.74x	12.41x
AES-T1000	10	456 k	2	100.00%	68.37	7441	3.88	525	17.62x	14.17x
AES-T1100	10	544 k	5	100.00%	71.03	7449	11.8	601	6.01x	12.39x
AES-T1300	10	456 k	9	100.00%	68.57	7449	2.65	524	25.87x	14.21x
AES-T2000	10	456 k	6	83.33%	69.27	7554	6.75	600	10.26x	12.59x
cb_aes_01	5	33 k	1	100.00%	1.27	179.4	0.51	55.3	2.49x	3.24x
cb_aes_05	10	167 k	1	100.00%	11.47	1450.3	4.03	244.3	2.84x	5.93x
cb_aes_10	15	334 k	1	100.00%	33.17	4130.6	14.47	502.4	2.29x	8.22x
cb_aes_15	20	501 k	1	100.00%	70.78	8041.2	32.14	778.2	2.20x	10.33x
cb_aes_20	25	668 k	1	100.00%	110.13	13202.8	86.03	1085.5	1.28x	12.16x
cb_aes_25	30	886 k	1	100.00%	-	MO	150.54	1405.3	-	-
cb_aes_30	35	1003 k	1	100.00%	-	MO	243.02	1780.3	-	-
cb_aes_35	40	1169 k	1	100.00%	-	MO	371.23	2112.7	-	-
cb_aes_40	45	1693 k	1	100.00%	-	MO	851.25	2532	-	-

[1]After hierarchy flattening.

Table II presents the results of Trojan detection using our approach on Trust-hub as well as specific custom benchmarks. The first and second columns show the type of the benchmark and the number of unrolled clock cycles, respectively. For each design, the unrolled cycle is incremented in steps until the targets are reachable. The third column shows the size of the code after hierarchy flattening. This gives an estimation of the complexity (size) of the benchmark. The fourth column shows the number of rare branches (i.e., security targets) that are used for Trojan detection. The fifth column indicates the coverage of rare branches using our approach. The sixth and eighth columns present the required time for Trojan activation using EBMC model checker and our approach, respectively. The seventh and ninth columns show the required memory to generate the test to activate the Trojan using EBMC and our approach, respectively. The last two columns demonstrate the improvement over EBMC on the required time and memory.

For large Trust-Hub benchmarks, our approach provides significantly (an order-of-magnitude) better runtime as well as memory results compared to EBMC. In addition to Trust-Hub benchmarks, few custom benchmarks are used to demonstrate the scalability of our approach. The benchmarks are named as cb_aes_xx. These are modified versions of AES benchmarks built by cascading xx number of rounds. The Trojan trigger depends on the output of the last round, so by controlling the number of rounds, xx, we can effectively control the complexity of these benchmarks. As the output depends on last round, these custom benchmarks need to be unrolled for at least xx number of cycles. As shown in Table II, the bounded model checking tool EBMC fails to generate a test case due to state space explosion (out-of-memory error). EBMC failed after the cb_aes_25 (approximately 0.9 million lines of code) while our approach worked even with cb_aes_40 (more than 1.6 million lines of code). This shows that the proposed approach can scale with design size, especially in terms of memory consumption.

Figure 7 and 8 show the effect of increasing unroll cycles keeping the design complexity constant. We have used cb_aes_10 benchmark for this comparison. The unroll cycle is increased from 15 to 40 while keeping the round number fixed to 10. In case of our approach, the time and memory requirement show very little increase with unroll cycles. On the other hand, it increases rapidly for EBMC, reaching over 10GB with unroll cycle of 40. This experiment demonstrates that even for the same design size, our approach scales better with unroll cycles. Overall, our approach is scalable both in terms of design complexity and unroll cycles. This is expected because model checkers try to explore the whole search space at the same time, whereas our approach tries to explore only one execution path at a time.

Our results demonstrate that effective interleaving of concrete and symbolic execution leads to a scalable approach for the automated generation of directed tests to activate hard-to-detect Trojans in large RTL designs. The results show that the run time of our approach is comparable to model checking for small designs, but an order-of-magnitude faster for large

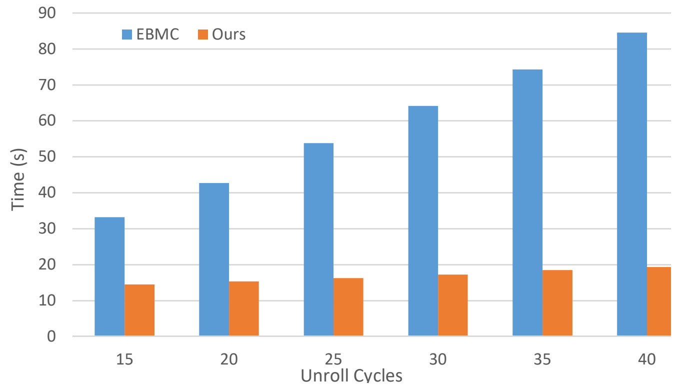

Fig. 7. Test generation time to activate the Trojan in *cb_aes*_10 benchmark with different unroll cycles.

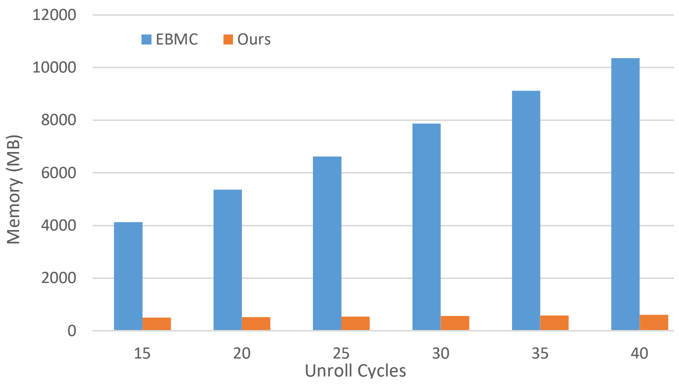

Fig. 8. Memory requirement to activate the Trojan in *cb_aes*_10 benchmark with different unroll cycles.

designs. Moreover, the memory usage of our approach is an order-of-magnitude better than model checkers. As a result, our approach can generate efficient tests to detect hidden Trojans when state-of-the-art approaches fail.

VII. Conclusion

In this paper, we presented an automated and scalable approach to activate hard-to-detect hardware Trojans in RTL designs. The first step in our detection methodology involves marking branches and assignments which are likely to contain Trojans. We proposed a threat model involving rare branches and rare assignments. Security targets are then automatically generated based on the threat model. Our approach can effectively utilize interleaved concrete simulation and symbolic execution to generate directed tests by covering security targets to detect potential hardware Trojan in the design. Our experimental results demonstrated that our test generation technique is scalable and effective for detecting Trojans in large RTL IPs when state-of-the-art methods fail. Our approach can be easily integrated into existing design flows and it can be applied on any RTL designs with single or multiple clock domains. Moreover, the proposed method can detect a wide variety of combinational and sequential Trojans in modern IP cores.

Acknowledgment

This work was partially supported by grants from National Science Foundation (CNS-1441667) and Cisco.

References

[1] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.
[2] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, “Hardware trojan attacks: threat analysis and countermeasures,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.
[3] M. Tehranipoor and F. Koushanfar, “A survey of hardware trojan taxonomy and detection,” *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.
[4] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, “Trustworthy hardware: Identifying and classifying hardware trojans,” *Computer*, vol. 43, no. 10, pp. 39–46, 2010.
[5] X. Wang, H. Salmani, M. Tehranipoor, and J. Plusquellic, “Hardware trojan detection and isolation using current integration and localized current analysis,” in *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*. IEEE, 2008, pp. 87–95.
[6] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. G. Wolff, C. A. Papachristou, K. Roy, and S. Bhunia, “Hardware trojan detection by multiple-parameter side-channel analysis,” *IEEE Transactions on computers*, vol. 62, no. 11, pp. 2183–2195, 2013.
[7] Y. Huang, S. Bhunia, and P. Mishra, “Mers: statistical test generation for side-channel analysis based trojan detection,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 130–141.
[8] H. Salmani, M. Tehranipoor, and J. Plusquellic, “A novel technique for improving hardware trojan detection and reducing trojan activation time,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 112–125, 2012.
[9] A. Waksman, M. Suozzo, and S. Sethumadhavan, “Fanci: identification of stealthy malicious logic using boolean functional analysis,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 697–708.
[10] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, “Veritrust: verification for hardware trust,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, 2015.
[11] W. Hu, B. Mao, J. Oberg, and R. Kastner, “Detecting hardware trojans with gate-level information-flow tracking,” *Computer*, vol. 49, no. 8, pp. 44–52, 2016.
[12] K. Sen, D. Marinov, and G. Agha, “Cute: a concolic unit testing engine for c,” in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.
[13] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.
[14] D. Ismari, J. Plusquellic, C. Lamech, S. Bhunia, and F. Saqib, “On detecting delay anomalies introduced by hardware trojans,” in *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 1–7.
[15] Y. Liu, K. Huang, and Y. Makris, “Hardware trojan detection through golden chip-free statistical side-channel fingerprinting,” in *Proceedings of Design Automation Conference*. 2014, pp. 1–6.
[16] J. Zhang, F. Yuan, and Q. Xu, “Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 153–166.
[17] M. Hicks, M. Finnicum, S. T. King, M. M. Martin, and J. M. Smith, “Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically,” in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 159–172.
[18] C. Sturton, M. Hicks, D. Wagner, and S. T. King, “Defeating uci: Building stealthy and malicious hardware,” in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 64–77.
[19] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa, “A score-based classification method for identifying hardware-trojans at gate-level netlists,” in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 465–470.
[20] M. Tehranipoor, D. Forte, R. Karri, F. Koushanfar, and M. Potkonjak, *Trust-HUB*, https://www.trust-hub.org/.

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

[22] L. H. Goldstein and E. L. Thigpen, "Scoap: Sandia controllability/observability analysis program," in *Proceedings of the 17th Design Automation Conference*. ACM, 1980, pp. 190–196.

[23] B. Çakir and S. Malik, "Hardware trojan detection for gate-level ics using signal correlation based clustering," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 471–476.

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

[25] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *Proceedings of the 52nd Design Automation Conference*. ACM, 2015, p. 112.

[26] F. Farahmandi and P. Mishra, "Automated test generation for debugging arithmetic circuits," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1351–1356.

[27] F. Farahmandi, Y. Huang, and P. Mishra, "Trojan localization using symbolic algebra," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 591–597.

[28] F. Farahmandi and P. Mishra, "Automated debugging of arithmetic circuits using incremental gröbner basis reduction," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 193–200.

[29] ——, "Fsm anomaly detection using formal analysis," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 313–320.

[30] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra, "Pre-silicon security verification and validation: A formal perspective," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 145.

[31] R. S. Chakraborty, F. G. Wolff, S. Paul, C. A. Papachristou, and S. Bhunia, "Mero: A statistical approach for hardware trojan detection." in *CHES*, vol. 5747. Springer, 2009, pp. 396–410.

[32] 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.

[33] J. Cruz, F. Farahmandi, A. Ahmed, and P. Mishra, "Hardware trojan detection using atpg and model checking," in *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*. IEEE, 2018, pp. 91–96.

[34] A. Ahmed and P. Mishra, "Quebs: Qualifying event based search in concolic testing for validation of rtl models," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 185–192.

[35] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 419–423.

[36] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*. IEEE, 2009, pp. 32–37.

[37] ——, "Efficient validation input generation in rtl by hybridized source code analysis," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[38] X. Qin and P. Mishra, "Scalable test generation by interleaving concrete and symbolic execution," in *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 2014, pp. 104–109.

[39] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008, pp. 443–446.

[40] L. Liu and S. Vasudevan, "Scaling input stimulus generation through hybrid static and dynamic analysis of rtl," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, p. 4, 2014.

[41] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 35.

[42] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *ACM SigPlan Notices*, vol. 48, no. 10. ACM, 2013, pp. 19–32.

[43] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 413–424.

[44] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, 2010.

[45] ——, "Property learning techniques for efficient generation of directed tests," *IEEE Transactions on Computers*, 60(6), pp. 852–864, 2011.

[46] M. Chen, X. Qin, H.-M. Koo, and P. Mishra, *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.

[47] M. Chen, P. Mishra, and D. Kalita, "Automatic rtl test generation from systemc tlm specifications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 2, p. 38, 2012.

[48] H.-M. Koo and P. Mishra, "Functional test generation using design and property decomposition techniques," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 4, p. 32, 2009.

[49] Y. Lyu, X. Qin, M. Chen, and P. Mishra, "Directed test generation for validation of cache coherence protocols," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[50] P. Mishra and N. Dutt, "Specification-driven directed test generation for validation of pipelined processors," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 3, p. 42, 2008.

[51] X. Qin and P. Mishra, "Directed test generation for validation of multicore architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 3, p. 24, 2012.

[52] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 363–374, 2009.

[53] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 131–140.

[54] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 31–36.

[55] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 321–334.

[56] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *IEEE International Conference on Computer Design*. IEEE, 2013, pp. 471–474.

[57] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, no. 1, pp. 85–102, 2017.

[58] S. Williams, *Icarus verilog*, On-line: http://iverilog.icarus.com.

[59] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 737–744.

[60] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[61] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.

[62] D. Kroening and M. Purandare, *EBMC*, http://www.cprover.org/ebmc.

[63] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12.

[64] F. Farahmandi and P. Mishra, "Automated Test Generation for Debugging Multiple Bugs in Arithmetic Circuits," *IEEE Transactions on Computers*, 2018.

[65] Y. Huang, S. Bhunia and P. Mishra, "Scalable Test Generation for Trojan Detection using Side Channel Analysis," *IEEE Transactions on Information Forensics & Security*, 13(11), pages 2746-2760, 2018.

[66] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte and M. Tehranipoor, "Security-aware FSM Design Flow for Identifying and Mitigating Vulnerabilities to Fault Attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.