Directed Test Generation for Activation of Security Assertions in RTL Models

HASINI WITHARANA, YANGDI LYU, and PRABHAT MISHRA, University of Florida, USA

Assertions are widely used for functional validation as well as coverage analysis for both software and hardware designs. Assertions enable runtime error detection as well as faster localization of errors. While there is a vast literature on both software and hardware assertions for monitoring functional scenarios, there is limited effort in utilizing assertions to monitor System-on-Chip (SoC) security vulnerabilities. We have identified common SoC security vulnerabilities and defined several classes of assertions to enable runtime checking of security vulnerabilities. A major challenge in assertion-based validation is how to activate the security assertions to ensure that they are valid. While existing test generation using model checking is promising, it cannot generate directed tests for large designs due to state space explosion. We propose an automated and scalable mechanism to generate directed tests using a combination of symbolic execution and concrete simulation of RTL models. Experimental results on diverse benchmarks demonstrate that the directed tests are able to activate security assertions non-vacuously.

CCS Concepts: • **Hardware → Assertion checking**; • **Security and privacy → Security in hardware**;

Additional Key Words and Phrases: Assertion-based validation, directed test generation, SoC security validation

ACM Reference format:
Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. 2021. Directed Test Generation for Activation of Security Assertions in RTL Models. *ACM Trans. Des. Autom. Electron. Syst.* 26, 4, Article 26 (January 2021), 28 pages.
https://doi.org/10.1145/3441297

1 INTRODUCTION

System-on-Chip (SoC) security is critical as more and more personal computing needs as well as physical infrastructures are controlled by a chip with the prevalence of Internet-of-Things (IoTs). Attackers take advantage of security vulnerabilities to inject malicious software. Tools such as anti-virus are not enough to protect from these attacks if the underlying hardware (SoC) is vulnerable. A typical SoC consists of a wide variety of components including processor cores, memory, controllers, converters, and so on. Due to time-to-market constraints coupled with cost considerations, SoC design methodology involves multiple third-party vendors in a long supply chain. As a result, SoC security vulnerabilities can arise in any stage, from design to fabrication, as well as post

This is an extended version of the article that appeared in ASPDAC 2020 [1].

This work was partially supported by the National Science Foundation (Grant No. CCF-1908131) and Semiconductor Research Corporation (Grant No. 2020-CT-2934).

Authors' addresses: H. Witharana, Y. Lyu, and P. Mishra, University of Florida, Gainesville, FL, 32611, USA; email: witharana.hasini@ufl.edu.

1084-4309/2021/01-ART26 $15.00
https://doi.org/10.1145/3441297

deployment. In contrast to a software vulnerability that can be modified after deployment, fixing a hardware vulnerability becomes more and more difficult and expensive in later stages. Existing approaches try to mask some of these vulnerabilities using firmware patching or utilizing in-built reconfigurable primitives. However, these approaches may not work in all scenarios. Therefore, detecting and removing vulnerabilities in early stages is very important in SoC designs. In this work, we plan to utilize assertions for monitoring SoC security vulnerabilities.

Assertion-based Verification (ABV) is a common practice in industry today for functional validation of SoC designs [2, 3]. Assertions define the properties that should hold. For example, a functional assertion can check that the output of an adder is equal to the sum of two inputs irrespective of the implementation. In addition to checking the inputs and outputs, assertions can also increase the observability of internal states, which enables runtime error detection as well as faster localization of errors. While there is a vast literature on both software and hardware assertions for monitoring functional scenarios, there is limited effort in utilizing assertions to monitor SoC security vulnerabilities.

One major challenge in assertion-based validation is to efficiently activate all assertions. Coverage of all assertions is fundamentally different from code coverage due to the vacuity problem. Directed tests are promising in activating assertions, since a significantly smaller number of directed tests can achieve the same coverage goal compared to random or pseudo-random tests [4–13]. Simulation-based validation can handle large designs but cannot guarantee activation of assertions due to exponential input space complexity. In practice, designers need to manually write directed test patterns to cover many hard-to-activate assertions. As expected, manual test writing can be time consuming and error prone (requiring numerous trials and errors), and may not be feasible for large designs.

While formal methods are effective in automated generation of directed tests [14–39], these approaches expect formal specification and do not directly support Hardware Description Language (HDL) models. Moreover, the extra procedure of conversion from HDL to formal specification may introduce errors. Most importantly, the complexity of real world designs usually exceeds the capacity of the model checking tools, leading to state space explosion. Concolic testing is a promising direction that combines the advantages of simulation-based validation and formal methods by effective utilization of symbolic execution and concrete simulation [40].

Concolic testing has been used extensively in software domain to cover functional events [40–43] as well as assertions [44]. While early work on concolic testing of Register-transfer Level (RTL) models is promising, there are no prior efforts in activating assertions using concolic testing. In this article, we propose an automated mechanism to generate directed tests using concolic testing to activate security assertions. To the best of our knowledge, our approach is the first attempt in utilizing concolic testing for activation of security assertions.

Specifically, this article makes four major contributions:

(1) We define security assertions to monitor a wide variety of SoC security vulnerabilities.
(2) We map the problem of activating security assertions non-vacuously to the problem of activating branches in a design. We propose an efficient approach to convert security assertions to equivalent branch targets.
(3) We propose an efficient test generation method using concolic testing to cover the branch targets. The generated test vectors are guaranteed to activate the corresponding security assertions.
(4) To address the path explosion problem in concolic testing, we develop an effective heuristic of path exploration to quickly reach the target.

The remainder of the article is organized as follows. Section 2 discusses the background and the related work. Section 3 presents an overview of our test generation framework. Sections 4 and 5 describe security assertions and concolic testing for activating security assertions, respectively. Section 6 presents the experimental results. Section 7 describes the applicability and limitations of our approach. Finally, Section 8 concludes the article.

2 BACKGROUND AND RELATED WORK

In this section, we present the related efforts in assertion-based validation as well as test generation for activating assertions.

2.1 Assertion-based Validation

There are mainly two types of approaches for defining hardware assertions: language-based and library-based [45]. Language-based approaches provides syntax for formally defining assertions. Two of the most popular assertion specification languages are Property Specification Language (PSL) [46] and System Verilog Assertions (SVA) [47]. Both of these languages support temporal assertions and formally is an extension of temporal logic [48]. Some other examples are ForSpec [49], SALT [50], a SystemC extension [51], and so on. However, library-based approaches adds assertion support to existing languages. One such example is Open Verification Library (OVL) [52]. OVL has support for Verilog, VHDL, PSL, and SystemVerilog. Library-based approaches can be used to quickly write common types of assertions. Unfortunately, they are not generic enough to cover all possible scenarios. In this article, we use SVA to express our SoC security assertions.

2.2 Concolic Testing for Activating Assertions

Concolic testing is a directed test generation technique combining symbolic execution [53] and concrete simulation. It addresses the state explosion problem in formal methods, such as bounded model checking [54]. Concolic testing explores one path at a time by alternating one of the branches from the previous simulation path until reaching the target statement. Concolic testing has been extensively explored in software domain to cover functional events [40–43]. These approaches utilize different path selection heuristics and optimizations to achieve specific coverage goals. Korel and Al-Yamo [44] explored concolic testing to find an input that violates assertions by analyzing data dependency to guide test generation. However, these approaches are all designed for software (sequential) models, and are not suitable for hardware designs where multiple modules are running concurrently with different clock domains and interacting with each other. While there are initial efforts in applying concolic testing on RTL models for test generation [8, 55], none of them deals with hardware assertions.

2.3 Test Generation for Hardware Assertion Coverage

Existing test generation approaches for activating hardware assertions can be broadly classified in two categories: simulation-based and formal methods. The first category uses simulation-based methods [56, 57]. In transaction-level, Ferro et al. [56] proposed a framework for supervising SystemC TLM simulation of PSL temporal properties with combinatorial testing tools. In register-transfer level (RTL), Pal et al. [57] restricted that assertions are defined over the interface of a module (input and output) and proposed an approach to bias random test generation for assertion coverage. The second category uses model checking [58–60]. From non-deterministic finite automata (NFA), Tong et al. [59] utilized model checking to generate test for assertions with the assumption that the signals in assertions refer to the primary inputs. The above approaches have one major drawback. To enable test generation, they restrict the assertions to have variables of only specific types (e.g., primary inputs/outputs of modules). As a result, these approaches

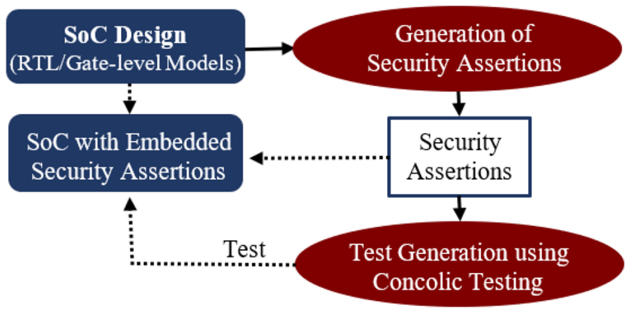

SoC Design
(RTL/Gate-level Models)
Generation of Security Assertions
SoC with Embedded Security Assertions
Security Assertions
Test
Test Generation using Concolic Testing

Fig. 1. Overview of our proposed methodology that consists of two major steps: generation of security assertions and test generation for activation of security assertions. The generated security assertions can be embedded in the design or defined as separate validation goals. Our approach converts security assertions to branch targets and activates them non-vacuously using concolic testing.

cannot be applied on assertions that may require complex interactions between any internal variables. Our test generation framework does not impose any restriction on assertion variables, and thereby enables test generation for activating a wide variety of assertions.

3 OVERVIEW

Our proposed methodology consists of two major steps as shown in Figure 1. The first step is to analyze the RTL model and identifying the security vulnerabilities. Then insert security assertions to detect those vulnerabilities. The second step converts the security assertions to branch statements and embed them into the design. Then, it utilizes concolic testing to generate a compact test set to efficiently cover (activate) the target branches (security assertions). While formal methods try to explore all possible paths at the same time (can lead to state space explosion), concolic testing has the inherent advantage of scalability, since it explores one execution path at a time. Note that the embedded branch targets are used for test generation purpose only. Once test generation is completed, these branch targets should be removed from the design (RTL model) and replaced with the original secuirty assertions. The two major steps of our proposed methodology are described in details in Sections 4 and 5, respectively.

4 SYSTEM-ON-CHIP SECURITY ASSERTIONS

While there is a vast literature on both software and hardware assertions for monitoring functional scenarios, there are limited efforts to define and utilize assertions to monitor SoC security vulnerabilities [61, 62]. We have reviewed a wide variety of security vulnerabilities listed in the National Vulnerability Database [63] and related research [61], and identified common SoC security vulnerabilities that are related to SoCs [64]. Note that there is a fundamental difference between exceptions and security vulnerabilities. The exceptions are defined today by SoC designers based on the point of view of functional correctness, whereas the security vulnerabilities outlined in this article are solely from security and trust perspectives. For example, an adversary may trigger (e.g., using a Trojan) an exception (e.g., divide by zero) solely to gain a higher privilege level, such that private memory or registers can be accessed.

This section tries to answer two important questions about security assertions: how to generate them and how to embedded them in RTL designs. First, we describe eight classes of SoC security vulnerabilities. Next, we outline how to generate security assertions to capture these vulnerabilities. Finally, we discuss how to embed these assertions in RTL designs.

4.1 Eight Classes of SoC Security Vulnerabilities

4.1.1 Permissions and Privileges. Permissions and privileges are the main components of the access control subsystem. Specifically, different resources are controlled by different permissions and privileges. For example, in ARM7 processor, seven different modes are defined, such as user mode, interrupt mode, and supervisor mode. It is critical to check whether the conditions for triggering privileged modes are satisfied before changing modes.

4.1.2 Resource Management. Certain resources should be protected against any illegal access, including accessing special hardware from non-privileged modes, misuse of design-for-debug infrastructures during normal usage, and so on. For example, JTAG allows engineers to trace secure memory during post-silicon validation and debug of security features. However, JTAG should never be enabled during normal usage.

4.1.3 Illegal States and Transitions. The behavior of an SoC can be modeled as a finite state machine (FSM). The valid states or transitions can be verified during functional validation. Attackers are more interested in the backdoor that allows undefined states/transitions. To verify the existence of illegal states and transitions, we could use assertions of valid states and transitions to alarm any violation, or enumerate all possible invalid states and transitions to prevent specific vulnerabilities.

4.1.4 Buffer Issues. Modern SoCs consist of advanced features (e.g., out-of-order execution and speculative execution) as well as a large number of heterogeneous buffers. Similar to software buffer errors, these buffers in deeper pipelines require significant validation efforts to detect any remaining flaws. For example, prefetched instructions in buffers should be flushed if the branch prediction is incorrect. Otherwise, these flaws can be exploited to mount an attack.

4.1.5 Information Leakage. Modern SoCs try to isolate a secure world from a non-secure world. Information from the secure world should never be leaked to non-secure world directly. ARM uses TrustZone as an approach to provide the secure world [65]. There should be safeguards present to prevent non-secure world from accessing TrustZone directly.

4.1.6 Numeric Exceptions. Numeric exceptions represent the erroneous/illegal behaviors (e.g., divide by zero) during arithmetic computations. Even if the program does not lead to illegal numeric computation, an attacker can make it happen and utilize it to create a vulnerability.

4.1.7 Malicious Implants. In software community, code injection means allowing attackers to run arbitrary code. Similarly, hardware Trojans, inserted by an untrusted third party, allow attackers to execute an arbitrary path after applying specific input patterns. This can lead to information leakage or other unintended consequences. Trojans are usually inserted in hard-to-detect and rare-to-activate areas, making it hard to detect them during validation.

4.1.8 Spectre and Meltdown. Spectre [66] and Meltdown [67] vulnerabilities were discovered in January 2018. These types of vulnerabilities allow an attacker to steal valuable information from any device that uses a vulnerable processor. In Spectre attacks, an attacker can read arbitrary memory location from an allocated memory location whereas Meltdown allows to read all memory locations of a system. These two attacks exploit speculative execution, branch prediction and caching, which are widely used to improve processor performance. Due to speculative execution, a process may be able to operate on data before it is determined whether the process has the permission to access the data. In other words, the protected data is stored in the CPU cache without checking the proper permission. A hacker can utilize this vulnerability and apply side-channel

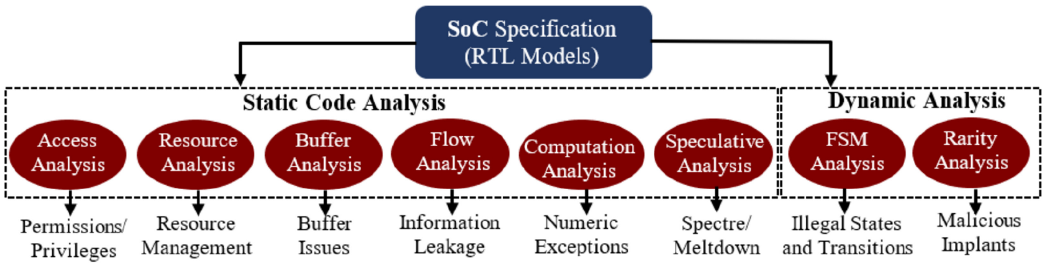


Fig. 2. Overview of assertion generation framework for the eight classes of SoC vulnerabilities.

attacks to identify the protected resource. Note that this vulnerability exploits architectural side-channel but does not rely on physical side-channel analysis [68–73].

4.2 Generation of Security Assertions

The security assertions can be generated by analyzing the design (RTL models) for the vulnerabilities described in Section 4.1. More assertions can be added based on designer inputs or application-specific considerations. Figure 2 shows an overview of assertion generation. Assertions for several types of vulnerabilities can be derived based on static code analysis, whereas we need to employ dynamic analysis to generate assertions for two cases. Please note that if the RTL models of specific components are not available, we need to utilize trace analysis to generate system-level assertions. For example, to generate security assertions related to malicious implants, we need to first determine rare nodes (signals) in the design, and then generate combinations of these rare nodes as potential triggers as security assertions. In this section, we outline the assertion generation for each vulnerability class.

4.2.1 Permissions and Privileges. By analyzing the specification, we first determine the variable that represent the privilege level, e.g., CPSR in ARMv7. For each entry to a privileged operation block, we need to generate an assertion. For the ease of illustration, we use `user` to represent current privilege, and `admin` to represent root privilege. Before privileged operation blocks, we are interested in the variable α : `privilege_level`, and define the property P : `privilege_level == admin`, as shown in Listing 1. We can use static analysis of the design to identify permissions and privileges and generate assertions.

```
always (@posedge clk) begin
    assert(privilege_level == admin);
    privileged operation block
end
```

Listing 1. Permissions and Privileges.

4.2.2 Resource Management. Concurrent assertions are powerful in protecting resources from misuse in an unexpected way. For example, to protect JTAG from getting used during normal operation mode, we need to generate the assertion as shown in Listing 2. Based on a list of resources and their usage scenarios, we can generate security assertions that capture unspecified usage of resources.

```
assert property (normal |-> !JTAG_enable);
```

Listing 2. Resource Management.

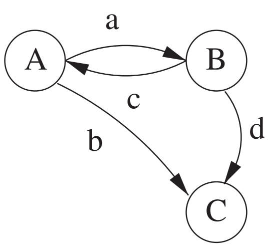


Fig. 3. FSM example.

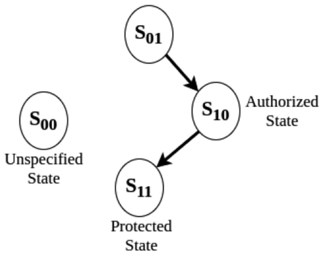


Fig. 4. FSM with protected state.

4.2.3 Illegal States and Transitions. The behaviors of modules as well as their interactions (protocols) can be expressed in the form of FSM. Therefore, we can express both valid and invalid (illegal) transitions as security assertions. As shown in Figure 3, circles represent states and edges represent transitions between states. Listing 3 gives an example of two assertions. For example, if there is a valid transition from state A to state B when variable a is true, then it can be encoded as the first assertion shown in Listing 3. Similarly, the second assertion in Listing 3 can be used to ensure that no transitions are allowed from state C to state A.

```
assert property (A && a |=>  B);
assert property (C |=> !A);
```

Listing 3. Illegal States and Transitions.

Some of the common illegal vulnerability attacks are due to unspecified (do not care) states, laser attacks and step-up time violation. To identify whether the unspecified states are handled without any vulnerability, we can add assertions to these states and check whether there is access to a protected state from an unauthorized state. To generate corresponding security assertions, we extract the finite state machines from a design, identify the unspecified states/transitions and generate the assertions to prevent accessing the protected states from unauthorized states. Moreover, we can write assertions to prevent access of protected state from any other state that does not have any transition. As shown in Figure 4 States S_{01}, S_{10} and S_{11} are defined states. State S_{11} is a protected state and state S_{00} is a do not care state (likely created during synthesis). We can generate assertions as shown in the Listing 4 to detect any illegal access to protected state S_{11}.

```
assert property (S00 |=> !S11);
assert property (S01 |=> !S11);
```

Listing 4. Protected state.

4.2.4 Buffer Issues. Assertions can be generated for all boundary cases related to buffers. To prevent access of the buffer index beyond its limits, immediate assertions should be added before each buffer access. For example, in Listing 5, before accessing `Buffer[index]`, the variable `index` needs to satisfy property P : `index >= 0 && index <= limit`.

```
reg [limit:0] Buffer;
always (@posedge clk) begin
    assert(index >= 0 && index <= limit);
    access Buffer[index];
end
```

Listing 5. Buffer Errors.

In many scenarios, we may require concurrent assertions to ensure global interactions. For example, to ensure the flush of instruction buffer (IB) after branch prediction failure, we can use the assertion shown in Listing 6.

```
assert property (Pre_fail |=> IB_Empty);
```

Listing 6. Buffer Errors Concurrent Assertion.

4.2.5 Information Leakage. To protect secret information from directly leaking to non-secure world, tagging is one potential solution. We assume that the results of secure world can only be passed to non-secure world through special interface (privileged instructions). For each normal operation consisting of both secure and non-secure variables, we need to check if the result is assigned to a non-secure variable. Assume s is a secure variable in Listing 7. The operation of assigning the addition of s and a to b needs to check the tag of b. One way to generate assertions for information leakage is by using Information Flow Tracking (IFT). IFT is used to prevent information leakage. As shown in the Listing 7, we can instrument the code with tags that allow information flow tracking and then write assertions.

```
reg s; // secure variable
reg a, b;  // two registers
reg s_t, a_t, b_t;  // tags
always (@posedge clk) begin
    ...
    assert(b_t == secure_tag);
    b = s + a;
    ...
end
```

Listing 7. Information Leakage.

Another way of generating security assertions to detect information leakage is to check for paths that are likely to leak the assets (secrets). By using random simulation, we can get the probability of leaking secure assets from different paths. After getting the probabilities, we can use a high threshold to actually identify which paths are most likely to leak valuable information. Then, we can write assertions for those selected paths.

4.2.6 Numeric Exceptions. Numeric exceptions are more relevant to the implementation of SoC designs. We need to generate one assertion for each possible numeric exception during arithmetic computation. For example, in case of a divide-by-zero exception, we can generate an immediate assertion as shown in Listing 8.

```
always (@posedge clk) begin
    assert(denominator != 0);
    quotient = numerator / denominator;
end
```

Listing 8. Numeric Errors.

When generating assertions to detect numeric exceptions, we used static code analysis. We analyzed the code to check numeric exceptions such as divide by zero and incorrect usage of signed and unsigned integers.

4.2.7 Malicious Implants. Malicious modifications (e.g., hardware Trojans) can be inserted during pre-silicon or post-silicon stage. In the pre-silicon stage, hardware Trojans are usually hidden in rare-to-activate branches or rare execution of concurrent statements. For example, we can generate assertions for each rare branch by adding an assertion for each rare trigger condition as shown in Listing 9.

```
always (@posedge clk) begin
    assert(rare_trigger)
end
```

Listing 9. Malicious Implants.

First, we need to check for rare nodes (signals) and rare branches where hardware Trojans are likely to hide in a design. To identify rare nodes and branches random simulation can be used. We ran millions of random tests on a benchmark and analyzed the simulation trace to identify the branches that are not getting activated (or gets activated less than a specific threshold). These branches are identified as rare branches. Furthermore, we also analyzed the simulation trace with signal values to identify rare signals. As an example, if a signal has value 0 only for 1% of the random simulation (the signal has value 1 for the remaining 99% time), then we consider that signal having value 0 as a rare condition. When we used these mechanisms to identify the rare nodes, there can be huge number of rare nodes that need to be considered. Once we have the set of rare nodes, we can insert assertions for various trigger conditions consisting of a set of rare nodes. Clearly, there is a trade-off between rareness coverage using security assertions and potentially exponential number of triggers based on a set of rare nodes. To reduce the number of rare nodes and make the assertion generation applicable in reality, we can set a high threshold to generate only the set of extremely rare nodes.

4.2.8 Spectre and Meltdown. There are several variants of Spectre and Meltdown vulnerabilities such as Spectre-RSB (Return Stack Buffer Mispredict), Spectre-BTB (Branch Target Injection), Meltdown-US (Supervisor-only Bypass), and so on. To identify the spectre and meltdown vulnerabilities, we statically analyze the code and recognize whether the processor supports either speculative execution or out-of-order execution. Then, we can write assertions to protect resources such as return stack buffer and branch target buffer as shown in Listing 10. This assertion detects speculative exploitation of overfill of RSB.

```
assert property(btb_rd_ret_f2==1'b1 |-> rs_overpop_correct ==1'b1)
```

Listing 10. Spectre Attacks.

4.3 Embedding Security Assertions in RTL Designs

There are two orthogonal ways of embedding assertions in the RTL model of an SoC design: immediate and concurrent assertions. Please note that immediate assertions can be converted to concurrent assertions by modifying the antecedent. However, as described below, it would be natural to use a specific one depending on the type of security vulnerability.

4.3.1 Immediate Assertions. Immediate assertions are powerful in detecting vulnerabilities such as numeric errors. Immediate assertions are flexible and can vary based on the potential statements or blocks. For immediate operations, it is important to find out the exact location, the relevant variable (e.g., trigger) α, and the assertion, *assert* $(P(\alpha))$, can be inserted. Immediate assertions are inherent for checking specific operations, such as divide-by-zero checking and out-of-boundary checking.

4.3.2 Concurrent Assertions. Concurrent assertions, however, are checked each clock cycle, representing expected properties of SoCs. Concurrent assertions are useful to express any FSM-related vulnerabilities (e.g., illegal states and transitions). Each concurrent assertion can be defined using *assert property* (P). The property P should be derived from the specification of SoCs and vulnerability classes.

5 AUTOMATED GENERATION OF DIRECTED TESTS

In this article, activation of security assertions refers to finding counter-examples that fails the security assertions non-vacuously. Vacuity is defined in Reference [74] as follows: If there exist a sub-formula ψ of a formula ϕ such that ψ can be replaced with arbitrary formula and does not affect the outcome of model checking, then the formula ϕ is vacuous in model M. For example, in the formula $p \longrightarrow q$, it is vacuously valid if p is always false, since we can replace q with any sub-formula. We address the vacuity problem by converting the formulas into specific branch targets and applying concolic testing to activate them.

Listing 11 shows the branches that are converted from two types of assertions (immediate assertions and concurrent assertions) in Arbiter. Note that the conversions from assertions to branches are the same for these two types, except that an individual concurrently running block is needed to wrap the branches from concurrent assertions. In Listing 11, the first assertion is an immediate assertion and its corresponding branch is directly embedded in the same place as the assertion. However, the second assertion is converted into an always block that is running concurrently with all the other blocks. To find counter-examples that make the security assertions fail non-vacuously, we need to generate tests to activate branch targets that are converted from the security assertions.

5.1 Conversion of Security Assertions to Branches

To generate a test to activate security assertion P, we first map the assertion activation problem to branch coverage problem in concolic testing. Algorithm 1 shows our procedure to convert security assertion P to blocks containing a corresponding branch target. In this section, we introduce the details of converting security assertions to branches. In this article, we consider assertions with logic operator, implication ($|-\ >$) and delay (##). Although SVA supports many more operators, we have considered these three operators, since they can represent a wide variety of security assertions. Note that we have also utilized other SVA operators for representing security assertions in NoC architectures as well as SweRV RISC-V processor in Section 6.2. Moreover, some of the remaining SVA operators can be constructed using the above operators as discussed in Section 5.1.3. This section is organized as follows. First, we describe the major steps in Algorithm 1: abstract syntax tree (AST) generation, modification of AST with timing, and conversion of modified AST to branch targets. Next, we perform complexity analysis of Algorithm 1.

```
module arb(clk, rst, req1, req2, gnt1, gnt2);
input clk, rst, req1, req2;
output gnt1, gnt2;
reg state, gnt1, gnt2;
always @ (posedge clk or posedge rst)
    if (rst)
        state <= 0;
    else
        state <= gnt1;
always @ (*)
    if (state) begin
        gnt1 = req1 & ~req2;
        gnt2 = req2;
        // Assert 1: assert(req2 == gnt2)
        if (req2 != gnt2)
            // target 1
    end
    else begin
        gnt1 = req1;
        gnt2 = req2 & ~req1;
    end
// Assert 2: assert property(gnt1|->~gnt2)
always @ (*)
    if (gnt1)
        if (gnt2)
            // target 2
endmodule
```

Listing 11. An example of branch conversion in Arbiter.

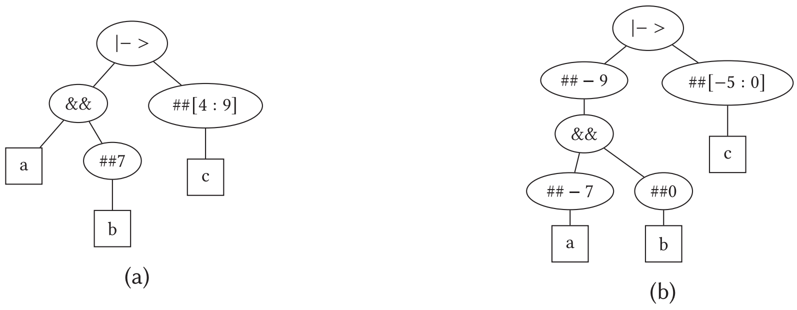


Fig. 5. (a) Simplified AST for *assert (a* ##7 *b* |− > ##[4 : 9] *c)*. Logic operator, implication and delay are non-terminal nodes (oval), and others are terminals (rectangle). (b) Readjusted AST with timing. All delays are converted to local history values.

5.1.1 Abstract Syntax Tree Generation. To understand the meaning of one assertion, we parse the assertion and build an AST for it. Three types of operators are selected as non-terminal for our simplified AST, i.e., logic operator, implication and delay. Others are treated as terminals. For example, if the original assertion is *assert (a* ##7 *b* |− > ##[4 : 9] *c)*, which means if *a* is 1 in clock 0 and *b* is 1 in clock 7, then c must be 1 in any clock between clock 11 and clock 16. The simplified AST for this assertion is shown in Figure 5(a).

5.1.2 Modification of AST with Timing. As delays represent the future events, which cannot be evaluated in the current clock cycle, we transform delays into retrieving history values. We assume that there exists a global clock counter (as shown in Listing 12), and the design remembers all the "necessary" history values. We use a[clk_cnt] to represent the history value of a in clock clk_cnt. Figure 5(b) shows the readjusted AST for Figure 5(a). There are two things to consider:

(1) Adjustment is local to its own children for each non-terminal nodes. For example, the left sub-tree in Figure 5(a) (a ##7 b) adjusts the delay of 7 to its left child. If we look at the whole expression, then the history values of a should be at least 11 cycles ahead of c. This localization property make adjustment efficient.
(2) For delay range, we adjust the longest delay to the left side and modify the range appropriately, e.g., ##[4 : 9] in Figure 5(a) rotates the ## − 9 to the left side and adjusts itself to ##[−5 : 0].

```
always @(posedge clock) begin
    clk_cnt <= clk_cnt + 1;
end
```

Listing 12. Global clock counter.

5.1.3 Conversion of AST to Branch Target. After we adjust AST with timing, each node is attached with non-positive delay (implicitly 0 delay). From adjusted AST, we construct branches by post-order traversal of the adjusted AST with the help of a stack S. Each part of the clause is represented by a unique variable except for the clauses that can be directly accessed. Stack S contains the visited variables that have not been combined by other clauses. Algorithm 1 shows how the target branch is generated (in italic bold text) with the help of stack S. The generated code of Figure 5(b) is shown in Listing 13. As shown in Algorithm 1, RTL code is generated based on the root type of each sub-tree. We consider the following three root types:

ALGORITHM 1: Assert2Branch

```
   /* Input: assertion P.                                   */
   /* Output: B containing generated blocks.                */
1  Construct simplified AST for assertion P
2  Readjust AST with delay information
3  Empty stack S
4  for Post-order traversal readjusted AST do
5      if current node n is an implication then
6          Convert implication to logic operator
7      end
8      if current node n is a variable then
9          Push n to S
10     end
11     if current node n is delay then
12         Pop variable a from S
13         Add delay to a
14         Push the modified variable to S
15     end
16     if current node n is a logic operator then
17         Pop all variables of its children from S
18         Combine the children with its operator
19         Push the result to S
20     end
21 end
22 Create branch to test the variable in S
```

Delay: For a single delay, we retrieve the history value of the variable, e.g., when we visit the node $\#\# - 7$ in Figure 5(b), the node a is in the top of stack S. We pop a from S, and push back a[clk_cnt - 7]. A delay range represents an OR operation on all the values, e.g., $\#\#[-5:0]c$ means $c[-5]|c[-4]|\ldots|c[0]$. Listing 13 shows the expansion of $\#\#[-5:0]c$ using for-loop and uses variable $p2$ to represent this part. When a single delay is applied, we skip generating a new variable for the clause, e.g., $\#\# - 7a$ directly utilizes the history value of a instead of generating a new variable.

Logic Operator: When the root is a logic operator, Algorithm 1 combines all its children (contains delay information) using the operator. As each child is already represented by a single variable in the stack S, we just pop all of them from S, and use a new variable to represent the combined result.

Implication: Implication, A $|-\,>$ B, contains two parts: A is called the antecedent, and B is called the consequent. There are two implication operators in SVA, i.e., overlapped implication ($|-\,>$) tests consequent sequence at the clock when its antecedent sequence is activated, while nonoverlapped implication (| =>) tests the consequent in the next clock cycle. The latter one can be converted to the previous one by adding one cycle delay to the consequent sequence. As shown in Listing 13, we convert the implication node into variable $p3$.

When we finish traversing the readjusted AST, the assertion expression is represented as a single variable in top of stack S, e.g., $p3$ in Listing 13. A branch target is created by checking the value of the final variable.

Many of the remaining operators can be constructed using the above operators. For example, repetition operators can be constructed using the delay operator [75]. There are three types of repetition operators. As described below, we can use the delay operator to simplify the operators and build the stack S.

(1) **Consecutive Repetition Operator**: Repetition can be represented using the [*n] construct, where n is a constant. A repetition range can be presented using[*m:n]. Here both m and n are constants. For example, **a[*2] ##1 b** is same as **(a ##1 a ##1 b)**.
(2) **Goto Repetition Operator**: This operator specify that there is one or more delay cycles between each repetition of the expression. For example, **a[->3]** is equivalent to **(!a[*0:$] ##1 a) [*3])**.
(3) **Nonconsecutive Repetition Operator**: This operator allows for space between the repetition of the terms. The difference between goto and nonconsecutive operator is that repeating term need not to be true for a nonconsecutive operator. For example, **a[=3]** is equivalent to **((!a[*0:$] ##1 a ##1 !a[*0:$]) [*3]**.

5.1.4 Complexity Analysis of Algorithm 1. For the ease of representation, we assume that the design remembers all "necessary" values in the previous iterations. To achieve memory efficiency, the clk_cnt can be as small as the largest delay in the whole assertion, e.g., 9 for *assert(a* $\#\#7\ b\ |-\,> \#\#[4:9]\ c$*)*, as a result of introducing new variables. If we look at the code in Listing 13, then the impact of a[clk_cnt - 16] is already stored in p1[clk_cnt - 9]. Thus, remembering older values than the longest delay is a waste of memory. After determining the largest delay, we add a modulo operation to Listing 12, i.e., clk_cnt<= clk_cnt $\mod (9+1)$, with an extra one to remember the current clock. Assume that b is the longest delay and n is the length of the assertion. The *memory requirement* complexity is $O(bn)$, since the memory usage of the tree structure, the stack S, and required new variables are linear to the length of assertion.

The running time of Algorithm 1 is dominated by post-order traversal of the AST, compared to the AST construction and adjustment. For each node, the running time is linear to the number of children. Then, each node contributes twice to the total running time. Since the number of nodes in AST is linear to the length of the assertion, the *running time* complexity is $O(n)$.

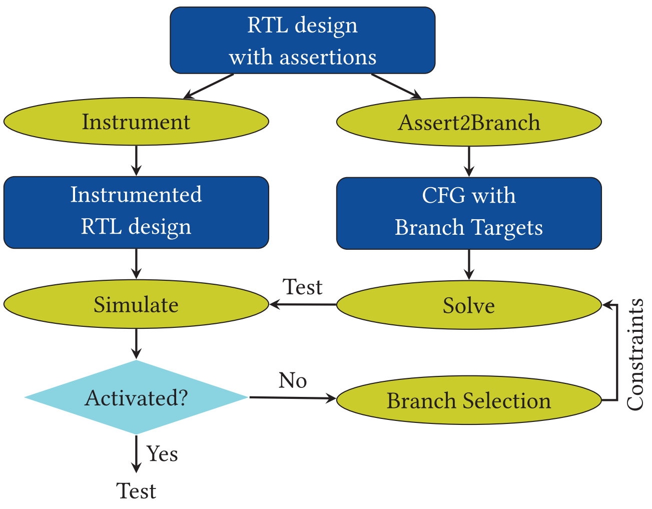


Fig. 6. Overview of our test generation framework. After converting assertions to branch targets, concolic testing is applied to generate tests to activate these branch targets (assertions).

```
always @(posedge clock)
begin
    // p1 = ##-7 a && b
    p1[clk_cnt] = a[clk_cnt - 7] && b[clk_cnt];
    // p2 = ##[-5:0]c
    p2[clk_cnt] = 0;
    for (i : [clk_cnt - 5, clk_cnt])
        p2[clk_cnt] = p2[clk_cnt] | c[i];
    // p3 = ##-9 p1 |-> p2
    p3[clk_cnt] = 1;
    if (p1[clk_cnt - 9])
        if (!p2[clk_cnt])
            p3[clk_cnt] = 0;
    // branch target
    if (!p3[clk_cnt])
        $display("Assertion fail");
end
```

Listing 13. The branch converted from Figure 5(b).

5.2 Test Generation Using Concolic Testing

Once the security assertions are converted to branches, we apply concolic testing to generate tests to cover the generated branch targets. Figure 6 shows an overview of our test generation framework. As discussed in Section 2.2, concolic testing combines concrete simulation and symbolic execution. In Figure 6, the left side shows the concrete simulation part, and the right side shows the symbolic execution part. To instruct symbolic execution, the concrete path needs to provide every branch it takes. Instead of changing simulator to execute symbolically in each branch and assignment, we use existing tools for simulation, and instrument the RTL design with *display* statement to show which branch the simulation has taken. For example, the instrumented first block of Listing 11 is shown in Listing 14.

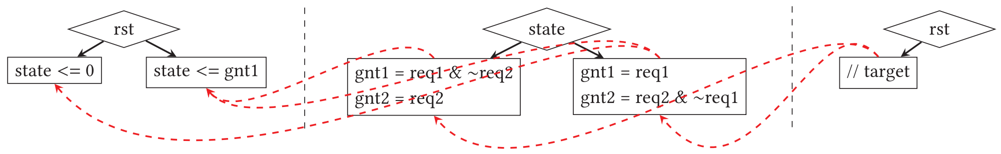


Fig. 7. Chaining of related blocks in CFGs. A block is chained to the blocks where its condition is likely to be satisfied.

```
always @ (posedge clk or posedge rst)
    if (rst) begin
        $display("arb2 branch 1 taken");
        state <= 0;
    end
    else begin
        $display("arb2 branch 2 taken");
        state <= gnt1;
    end
```

Listing 14. Instrumented first block in Arbiter.

Based on Algorithm 1, the security assertions in RTL design are converted into branch targets in control flow graph (CFG). For example, the Listing 11 has three always blocks: the first always block leads to the first CFG (CFG1), the second always block leads to the second CFG (CFG2), and the last always block represents the CFG (CFG3). For every test that is generated by symbolic execution, simulation will give the concrete execution and report every branch it takes. Based on the branch information, constraints are constructed together with all the assignments inside the corresponding blocks. The most important step in concolic testing is to find the best alternative branch to flip, which will be discussed in the next section. With the selected alternative branch, new constraints are constructed, and solved by an SMT solver to generate a new test for simulation. The general idea is to efficiently explore different paths to get closer to the branch target converted from a specific assertion.

To help alternative branch selection, we first chain the relative blocks together in the control flow graph. We use the second assertion in Listing 11 as an example. The branch target is controlled by the condition *gnt1 & gnt2*. Therefore, we need to create an edge (chain) from CFG3 (where $gnt1$ and $gnt2$ are used) to CFG2 (where $gnt1$ and $gnt2$ are assigned). Similarly, since the blocks in the second CFG are controlled by the value of $state$, we also need to link CFG2 (use of $state$ variable) to CFG1 (assignment to $state$ variable), as shown in Figure 7. This chaining process helps alternative branch selection concentrating only on related branches. When we consider the relevance of one branch with the target, we calculate the distance from the immediate block following the alternate branch to the target. In each iteration, the most relevant and reachable branch is selected as the alternative branch to construct new constraints and generate a new test.

Figure 8 shows an example of alternative branch selection. In Figure 8, the initial test resulted in the simulated path shown by dotted line. At this point, there are two choices to select alternate branch: either flip branch b1 or flip branch b2. Based on our distance calculation from the target, clearly b1 is the best choice. Once it is flipped, the target would be activated in the next iteration.

6 EXPERIMENTS

This section is organized as follows. First, we describe our experimental setup. Next, we describe the benchmarks and associated security assertions. Finally, we present our test generation results.

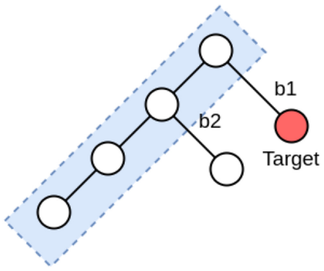


Fig. 8. A simple example for alternate branch selection.

Table 1. Types of Vulnerabilities Explored in Seven Benchmarks (More Potential Vulnerabilities are Possible)

Vulnerability	Arbiter	PCI	MEM	GNG	AES	NoC	SweRV
Permissions and Privileges			✓			✓	
Resource Management					✓	✓	
Illegal States & Transitions	✓	✓				✓	
Buffer Issues			✓			✓	
Information Leakage			✓				
Numeric Exceptions				✓			
Malicious Implants					✓		
Spectre and Meltdown							✓

6.1 Experimental Setup

To evaluate our test generation technique in activating security assertions non-vacuously, we implemented our framework in C++ using Icarus Verilog Target API [76] with Yices [77] as the constraint solver. As shown in Section 5.2, our framework first converted all security assertions to branches and inserted them into modified designs. Next, it applied concolic testing to generate test to activate the branches. Finally, we simulated the assertion-inserted instances (before converting to branches) to validate the correctness of generated test sets. Our framework is compared with EBMC [60] to show the performance improvement. We ran our experiments on two different machines to demonstrate several aspect of our framework: (i) Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM, and (ii) Intel E5-2698 v4 @ 2.20 GHz CPU with 200 GB RAM.

6.2 Benchmarks and Assertions

To demonstrate the necessity of security assertions, we analyzed five SoC benchmarks and two other large designs (Network-on-Chip benchmark and SweRV RISC-V Processor) and inserted security assertions outlined in Section 4. The types of potential vulnerabilities of seven benchmarks are shown in Table 1. Note that the number of instances are more than the number of vulnerability types, as each type may contain multiple instances. In the remainder of this section, we describe each type of vulnerability and inserted instances in detail for the seven benchmarks.

Arbiter: We first analyzed a simple design, Arbiter, as shown in Listing 11. For this simple design, we inserted the vulnerability of invalid states and transitions. Even for this small design, careless design, fault injections, or transient errors can make it behave differently. For example, if the security of the whole design relies on the *gnt1* and *gnt2* not asserted together, then *assert*(!(gnt1 & gnt2)) should be added to the design. However, the number of invalid states and transitions would be exponential when time is involved. For example, when we consider two consecutive cycles, *assert*(!(gnt1 | => (gnt2 != req2))) should hold.

Peripheral Component Interconnect (PCI): Top module pci_master32_sm from Opencores [78] contains eight modules such as pci_frame_crit and pci_irdy_out_crit. To mimic SoC design, which contains different parts from untrusted third party, we inserted invalid states and transitions to the subordinate modules (8 of 10) as well as the top modules (2 of 10). To generate vulnerable instances, we randomly changed operators in all the modules.

Memory Design (MEM): This design is created to mimic the behavior of a simplified Trusted Hardware (TH) implementation of memory, as shown in Listing 15. Trusted Hardware, e.g., Intel's Software Guard Extensions (SGX) [79], allows remote clients to upload private computation and data to a secure container of a server with a TH. One key implementation of SGX is the introduction of Process Reserved Memory and Enclave Page Cache (EPC), inhibiting invalid accesses even from the kernel. The simplified design is shown in Listing 15, which contains input signal *sc* to denote whether it is a secure access or not. The memory space is denoted by an array named *mem* with size of 1 MB.

```
module mem(clk, rst, wr, sc, address,
           in, out);
input clk, rst, wr, sc;
input [31:0]address;
input [7:0]in;
output reg [7:0]out;
reg[7:0] mem[2**20-1:0];
always @ (posedge clk)
  if (address >= 1024 || sc) begin
    if (wr) mem[address] <= in;
    else out <= mem[address];
  end
endmodule
```

Listing 15. Simplified Memory of Trusted Hardware.

(1) Permissions and privileges. Assume the lower 1 kB of memory is allocated to EPC. Since EPC should be accessed through secure container/process, each access to the lower 1 kB should be checked. Although one conditional checking is already in place, assertions may also help when implementation error, fault injection or Hardware Trojans exist, e.g., *assert*(address <= 1,024 | => sc) can be inserted before any access to memory.
(2) Information leakage. In this simplified memory implementation, it does not explicitly describe the state of *out* signal when we want to write. For a buggy CPU design that connects to this memory, a process may be able to read the previous access of another process from the out port (including secure processes) with interleaved memory access. We may add a concurrent assertion with (*assert property* (wr | => out == 0)).
(3) Buffer errors. Memory is a type of buffer, hence should be checked for buffer errors. Each access to memory should be checked with assertions to test if address is in the range of memory size. In this example, the memory size is 1 MB (2^{20}) and the length of address is 32 bits. We need *assert*(address $< 2 * *20$) to check out-of-boundary accesses.

We inserted a vulnerability related to permissions and privileges, by removing *sc* checking in the first *if* statement. For vulnerabilities related to information leakage, we assume that attackers are able to connect one specific location to *out* when it is a write operation.

Gaussian Noise Generator (GNG): We next inspected a computation-intensive design called GNG. The design is downloaded from Opencores [78]. One possible numeric error in this design is the assignments between signed and unsigned values as shown in the snippet of code in Listing 16. The first assignment assigns an unsigned variable to a signed variable. The next assignment is the computation between signed values. The final assignment assigns a signed value to an unsigned value. We are concerned with the automatic transformation between signed and unsigned values. For example, when the most significant bit of mul1 is 1, mul1_new is interpreted as a negative value using a two's complement representation. Then mul_new is added to a positive number and converted to an unsigned number again. The behavior may or may not be the original intention of this code. We want to generate assertions, e.g., *assert*(mul1[32] != 1), and guide the test plan of debug. The developer should decide if it is a numeric error or the expected behavior. Since we view this design as "buggy" by itself, we did not generate vulnerable instances for it.

```
module gng_interp (
  input clk, rstn, valid_in,
  input [63:0] data_in,
  output reg valid_out,
  output reg [15:0] data_out
);
wire [33:0] mul1;
wire signed [13:0] mul1_new;
reg [17:0] c0_r5;
reg signed [18:0] sum2;
reg [14:0] sum2_rnd;
assign mul1_new = mul1[32:19];
always @ (posedge clk)
  sum2 <= $signed({1'b0, c0_r5}) + mul1_new;
always @ (posedge clk)
  sum2_rnd <= sum2[17:3] + sum2[2];
...
endmodule
```

Listing 16. GNG_interp.

Advanced Encryption Standard (AES): AES is a very commonly used crypto core, consisting of ten rounds of block ciphers (substitution permutation networks). The substitution is shown in Listing 17. We also inserted JTAG to dump internal variables during debug. The identified vulnerabilities are:

(1) Resource management: As JTAG is for debug purpose only, it should be disabled during normal usage. As a result, the dump signal should contains nothing related to any internal signals. We inserted a concurrent assertion *assert property* `(!JTAG |-> (JTAG_out == 0))` to detect whether attackers are bypassing the JTAG checking and dump internal signals to infer the plaintext.
(2) Malicious implants: As module S contains a lot of rare branches, attackers are able to construct rare trigger conditions for hardware Trojans. For example, the probability of (out == 32'h7c7c7c7c) is 2^{-32} when (in == 8'h01) is true for all S_0, S_1, S_2 and S_3

```
module S4 (clk, JTAG, in, out, JTAG_out);
  input clk, JTAG;
  input [31:0] in;
  output [31:0] out;
  output reg [31:0] JTAG_out;
  S
    S_0 (clk, in[31:24], out[31:24]),
    S_1 (clk, in[23:16], out[23:16]),
    S_2 (clk, in[15:8],  out[15:8] ),
    S_3 (clk, in[7:0],   out[7:0]  );
  always @ (posedge clk)
    if (JTAG) JTAG_out <= in;
endmodule
module S (clk, in, out);
  always @ (posedge clk)
  case (in)
    8'h00: out <= 8'h63;
    8'h01: out <= 8'h7c;
    ...
  endcase
endmodule
```

Listing 17. AES table.

together. Assertions like *assert*(out != 32'h7c7c7c7c) in module S4 can guide the designer of a test plan to cover this specific potential trigger condition.

Network-on-Chip (NoC): We used security assertions to monitor vulnerabilities in an open source NoC-based SoC generation platform [80]. We configured a 2×2 Mesh NoC that interconnects 4 IPs. A *mor1k* processor [81] was used to configure the IPs. Then, we implemented a simple message passing application using C and compiled it using the mor1k-tool-chain. We inserted security assertions that are mentioned in the Table 2 to detect various NoC attacks. The assertions were inserted inside the router component of the NoC RTL design. These assertions can be used to identify packet duplication, packet corruption, packet starvation, packet misrouting and packet dropping as outlined in Table 3.

Spectre Attack in SweRV RISC-V Processor [82]: In this work, we are considering Spectre-RSB variant of spectre vulnerability and trying to utilize security assertions to detect this vulnerability. The processor stores the return address to RSB every time when a call instruction is executed and uses that as a return target prediction when matching return is detected. One way to attack the RSB is exploiting overfill or underfill of the RSB. We used SweRV RISC-V processor [82] as our experimental setup and wrote security assertions to detect speculative exploitation of overfill and underfill of the RSB. We inserted assertions *assert property*(btb_rd_ret_f2 == 1'b1 |− > rs_overpop_correct ==1'b1) and *assert property*(btb_rd_ret_f2 == 1'b1 |− > rs_underpop_correct ==1'b1) to detect and prevent accessing arbitrary illegal address locations.

6.3 Functional Assertion versus Security Assertions

As outlined in Section 2.1, functional assertions are widely used today for assertion-based validation. It is time-consuming to insert enough assertions into an industrial SoC design. Many research efforts have been devoted for automated generation of functional assertions. Rogin et al. [83]

Table 2. Security Assertions to Detect Various NoC Attacks

A#	Description of Security Assertions
$A1$	Route can issue at most one request $G((\sum_{i=0}^{N_{ports}} req\_port_i) \leq 1)$
$A2$	Route should issue a request whenever a data is valid $G(data\_valid \leftrightarrow (\sum_{i=0}^{N_{ports}} req\_port_i) == 1)$
$A3$	Routing algorithm (XY) should be correctly implemented $G((dest_x > current_x \leftrightarrow destport_{next} == EAST) \vee (dest_x < current_x \leftrightarrow destport_{next} == WEST) \vee (dest_y > current_y \leftrightarrow destport_{next} == SOUTH) \vee (dest_y < current_y \leftrightarrow destport_{next} == NORTH) \vee (destport_{next} == LOCAL))$
$A4$	Always at most one grant issued by the arbiter $G((\sum_{i=0}^{N_{ports}} gnt\_port_i) \leq 1)$
$A5$	As long as the request is available, it will eventually be granted by the arbiter within T cycles $(req\_port\ U\ gnt\_port) \rightarrow F(gnt\_port)$
$A6$	No grant can be issued without a request $\neg req\_port \rightarrow X(\neg gnt\_port)$
$A7$	Time between two issued grants is same for all requests $G(\forall i, j \in \{north, west, south, east, local\} \Delta T_i = \Delta T_j)$
$A8$	During multiplexing, output should be equal to input data $G((\sum_{i=0}^{N_{ports}} (select_i \wedge (data_{in} i == data_{out}))) == 1)$

Table 3. NoC Security Vulnerabilities and Relevant Security Assertion Mappings

Vulnerability	Combinations
Packet Duplication	$G(A5 \wedge A8)$
Packet Corruption	$G(A1 \wedge A5 \wedge A8)$
Packet Starvation	$G(A8)$
Packet Dropping	$G(A1 \wedge A2 \wedge A3 \wedge A4 \wedge A5 \wedge A6 \wedge A8)$
Packet Misrouting	$G(A1 \wedge A3 \wedge A4 \wedge A8)$

proposed to generate properties of a design by analyzing simulation traces. Hertz et al. [84] improved the analysis process using data mining and developed a tool named Goldmine. The generated rules from simulation traces are passed through a formal verification tool to verify the correctness in the design. As the simulation data is inherently incomplete and non-deterministic, the quality of mined assertions cannot be guaranteed. Moreover, as our case studies show, these functional assertions are not suitable for detecting security vulnerabilities.

To demonstrate the necessity of security assertions and to emphasize the completeness and coverage in detecting the security vulnerabilities, we compared our method with the state-of-the-art functional assertion generation technique, GoldMine [84]. We analyzed five benchmarks and inserted security assertions using the procedure outlined in Section 4. Then, Goldmine [84] was applied on all the five SoC benchmarks to generate as many assertions as possible. To evaluate the effectiveness of our security assertions, we randomly inserted 10 vulnerabilities into each design to form 10 vulnerable instances and generated test patterns to activate these security vulnerabilities. If any security assertion generated by the two methods (ours versus Goldmine) got activated, then we claim the corresponding method detects the vulnerability. To avoid any bias, assertion generation was performed before insertion of vulnerabilities. The types of potential vulnerabilities of each benchmark and the detected instances are shown in Table 1 and Figure 9, respectively. Note

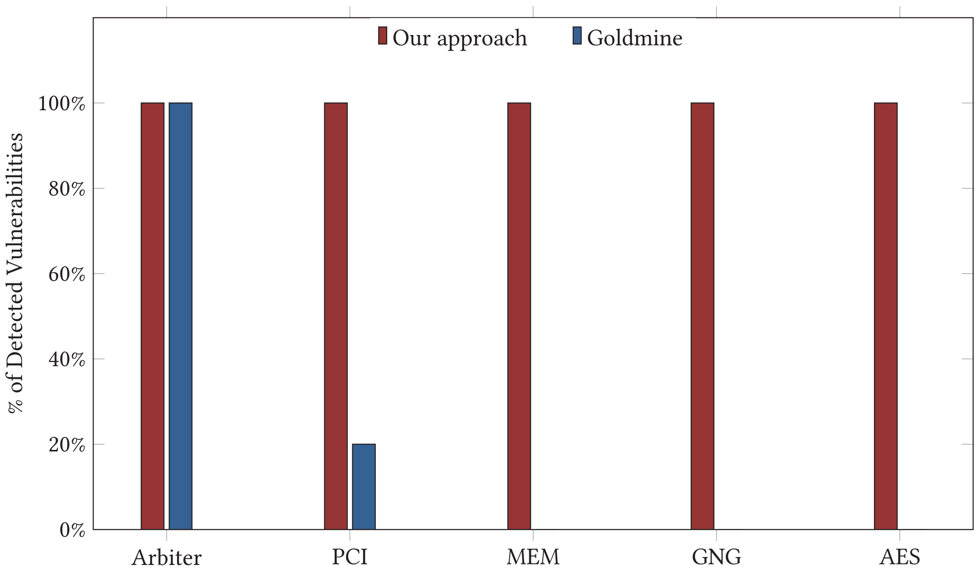


Fig. 9. Comparison between our security assertions and functional assertions generated by Goldmine [84] in detecting security vulnerabilities.

Table 4. Performance Comparison of Our Approach with EBMC [60] in Generating Tests to Activate Security Assertions for Arbiter, PCI, MEM, GNG, and AES Benchmarks Using Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM (EBMC Failed for MEM Benchmark Due to Insufficient Memory)

Benchmark	No. of Lines	EBMC [60]		Our Approach	
		Time (s)	Memory (GB)	Time (s)	Memory (GB)
Arbiter	25	0.01	0.004	0.01	0.009
MEM	40	—	—	46.85	0.89
PCI	635	0.13	0.03	0.04	0.01
GNG	696	10.04	0.1	64.04	0.03
AES	340K	118	7.6	91.05	1.07

that the number of instances are more than the number of vulnerability types, as each type may contain multiple instances.

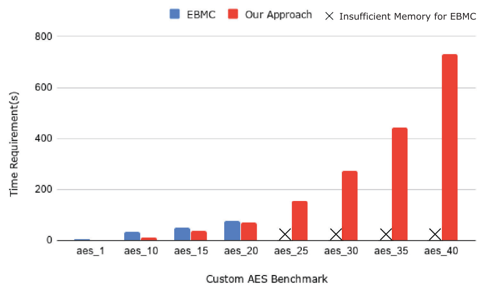
Figure 9 shows that the functional assertions generated by GoldMine [84] cannot eliminate the need for our dedicated security assertions. Specifically, our security assertions are able to detect all the implanted security vulnerabilities while the functional assertions failed to detect most of them. This is expected, because Goldmine tries to mine invariant to generate functional assertions. In other words, the functional assertions represents expected functional behaviors, which is not designed to capture unexpected behaviors of security vulnerabilities.

6.4 Test Generation Results

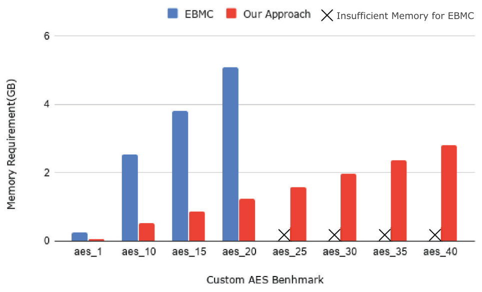
We have used our framework to generate tests for activating security assertions. The performance comparison results are shown in Table 4, Table 5, and Figure 10. Results in Table 4 and Figure 10 are generated using a machine with Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM, whereas the

Table 5. Performance Comparison of Our Approach with EBMC [60] in Generating Tests to Activate Security Assertions for Custom AES Benchmarks Using Intel E5-2698 v4 @ 2.20 GHz CPU with 200 GB RAM

Benchmark	No. of Lines	EBMC [60]		Our Approach			
		Time (s)	Memory (GB)	Time (s)	Time Improvement	Memory (GB)	Memory Improvement
cb_aes_01	33K	2.89	0.17	0.90	3.2×	0.06	2.9×
cb_aes_10	334K	58.4	3.42	12.81	4.6×	0.59	5.8×
cb_aes_15	501K	113	6.42	27.9	4.1×	0.88	7.3×
cb_aes_20	668K	178	10.3	63.7	2.8×	1.23	8.4×
cb_aes_25	836K	260	15.0	127	2.1×	1.58	9.5×
cb_aes_30	1003K	411	20.7	230	1.8×	1.97	11×
cb_aes_35	1170K	478	27.1	372	1.3×	2.36	12×
cb_aes_40	1337K	617	34.3	578	1.1×	2.81	12×
Average	—	265	14.7	177	2.6×	1.4	8.6×



(a) Time requirement with respect to the custom AES benchmarks.



(b) Memory requirement with respect to the custom AES benchmarks

Fig. 10. Performance comparison of our approach with EBMC [60] in generating tests to activate security assertions for custom AES benchmarks using Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM.

results in Table 5 is generated using a machine with Intel E5-2698 v4 @ 2.20 GHz CPU with 200 GB RAM.

Table 4 shows the performance comparison of our approach with EBMC [60] in generating tests to activate security assertions for Arbiter, PCI, MEM, GNG, and AES benchmarks using Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM machine. The first column shows the benchmarks. The second column provides the number of lines in the Verilog RTL implementation of the benchmark. The third and the fifth columns describe the time(s) taken to activate security assertions by EBMC [60] and our approach, respectively. Similarly, the fourth and sixth columns describe the memory (GB) requirement of EBMC and our approach, respectively. For smaller and less complex designs, such as Arbiter benchmark, EBMC is more efficient in time and memory as shown in Table 4. This is expected, because EBMC quickly explore all paths in these small designs, while concolic testing incurs inherent overhead associated with simulation and constraint solving. However, for complex designs our approach outperforms EBMC in memory requirements.

To highlight the importance of our approach in handling large designs, Figure 10 shows the performance comparison of our approach with EBMC [60] in generating tests to activate security assertions for custom AES benchmarks using Intel i7-5500U @ 3.0 GHz CPU with 8 GB RAM.

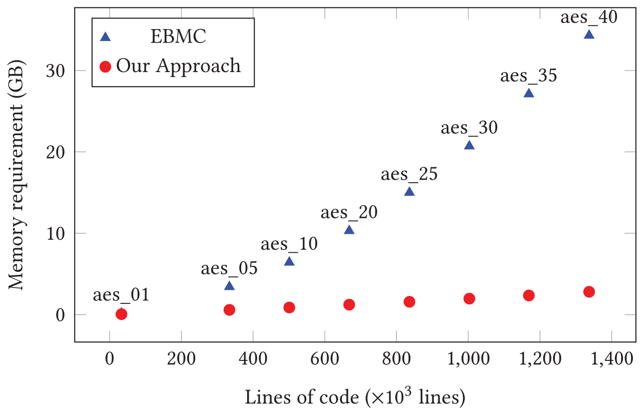


Fig. 11. Memory requirement with respect to the total lines of code in custom benchmarks of AES.

We have used custom benchmarks of AES named as cb_aes_*n* (shown as aes_*n* for the ease of illustration), where *n* is the number of rounds in AES. We varied the number of rounds to easily control the size of our benchmarks. The number of unrolled cycles is just enough to activate the assertions. For example, the number of unrolled cycles is $n + 5$ for each custom AES benchmark cb_aes_*n* as it requires n cycles to get results from output. Figure 10(a) shows the time(s) taken by these approaches to activate security assertions. Similarly, Figure 10(b) shows the memory requirement (GB) by these approaches. As shown in Figure 10, EBMC failed for cb_aes_25 (836K lines) or larger versions due to insufficient memory while our approach is able to run all the custom benchmarks using a machine with 8 GB RAM.

To further identify the performance improvement between EBMC and our approach when increasing the number of lines, we used the same custom AES benchmarks using a machine with 200 GB RAM. The results of this experiments are shown in Table 5. The first column represents the custom benchmarks. The second column indicates the number of lines in the RTL implementation of the custom benchmark. The third and the fifth columns describe the time(s) taken to activate security assertions by EBMC [60] and our approach, respectively. Similarly, the fourth and seventh columns describe the memory (GB) requirement of EBMC and our approach, respectively. The sixth column provides the time improvement of our approach compared to EBMC. The last column provides the memory improvement of our approach compared to EBMC.

Our approach is more efficient in memory usage. As shown in Table 5, our approach is up to 12× (8.6× on average) more efficient in memory usage compared to EBMC. To better visualize the relationship between the memory requirement with respect to the size of the design, we plot the memory requirement of two approaches for our custom benchmarks in Figure 11. Note that the number of lines for each custom AES benchmark is the total lines after hierarchy flattening. As we can see, the memory requirement of EBMC grows exponentially with the lines of code. It is due to the state space explosion problem of model checking. However, the memory requirement of our approach grows linearly with the lines of code, since it explores one path at a time, which is linear to the code size. For the benchmark cb_aes_40 (around 1.3 million lines of code), EBMC requires over 34 GB memory, while our approach only needs 2.8 GB. Due to exponential memory requirement, EBMC is expected to fail for larger and more complex designs, while our approach is expected to be scalable, since memory requirement increases linearly.

We ran both the EBMC and our approach to activate the security assertions mentioned in the Table 2 for the NoC design. Table 6 presents the time(s) and space (MB) taken by EBMC [85] and our approach to activate the security assertions. As shown in Table 6, our approach was able to activate

Table 6. Comparison of Test Generation Performance with EBMC [60] for Activating NoC Security Assertions

Assertion	EBMC [60]		Our Approach	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)
A1	0.02	11.7	0.01	9.6
A2	0.02	12.0	0.01	9.3
A3	0.02	11.8	0.01	9.5
A4	0.01	11.2	0.01	9.8
A5	0.01	5.1	0.01	9.8
A6	0.01	4.8	0.01	9.9
A7	—	—	8.78	37.3
A8	0.01	11.6	0.01	9.4

all the security assertions, whereas EBMC failed to activate the assertion A7. The arbiter property A7 checks for the time intervals between grants. EBMC led to state space explosion (insufficient memory) due to the liveliness nature of A7. Our approach activates the assertion A7, because concolic testing considers only one path at a time, which prevents the state space explosion problem.

7 APPLICABILITY AND LIMITATIONS

This section provides a brief discussion on applicability and limitations of our test generation framework for activating security assertions. The core part of the work can be applied to any assertions that can be translated to a branch, which is not limited by the language or the domain of the assertions. We implemented the most widely used operators that can capture a wide variety of security assertions. Moreover, many of the remaining operators can be constructed using the operators as mentioned in Section 5.1.3. In general, our approach would be able to handle any security vulnerabilities that can be represented by commonly-used SVA operators. Section 6.2 shows NoC case study using complex security assertions. Based on our experiments, we did not encounter any security vulnerability that cannot be captured using the SVA language.

There are several inherent limitations in concolic testing such as path explosion and expensive constraint solving. To mitigate the path explosion problem, we have considered the distance metric from the simulated path to the target. Our future research will explore other mitigation techniques including reinforcement learning [86] as well as path merging methods [87].

8 CONCLUSION

SoCs are widely used today in both embedded systems and IoT devices. While SoC security is paramount, there are limited prior efforts in defining and detecting a wide variety of SoC security vulnerabilities in RTL models. In this article, we explore the suitability of utilizing security assertions to monitor SoC security vulnerabilities. A major challenge in assertion-based security validation is how to activate all the security assertions to ensure that they are valid. While existing model-checking-based directed test generation is promising, it cannot generate tests for large designs due to state space explosion. We presented an automated and scalable mechanism to generate directed tests using concolic testing to activate security assertions non-vacuously. Using a diverse set of benchmarks, our experimental results demonstrated that our test generation approach is significantly faster (up to 4.6×, 2.6× on average) compared to state-of-the-art test generation methods. Most importantly, our approach is scalable, since it has linear memory requirement, while state-of-the-art directed test generation method has exponential memory requirement. Our future research will explore how to enable seamless integration of existing assertion-based functional validation with the proposed SoC trust validation using security assertions.

REFERENCES

[1] Yangdi Lyu and Prabhat Mishra. 2020. Automated test generation for activation of assertions in RTL models. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'20)*.

[2] Harry D. Foster, Adam C. Krolnik, and David J. Lacey. 2004. *Assertion-based Design*. Springer Science & Business Media, Berlin.

[3] Mingsong Chen and Prabhat Mishra. 2010. Functional test generation using efficient property clustering and learning techniques. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 29, 3 (2010), 396–404.

[4] Mingsong Chen, Xiaoke Qin, and Prabhat Mishra. 2010. Efficient decision ordering techniques for SAT-based test generation. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'10)*. IEEE, 490–495.

[5] Yangdi Lyu, Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. 2018. Directed test generation for validation of cache coherence protocols. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 38, 1 (2018), 163–176.

[6] Farimah Farahmandi and Prabhat Mishra. 2018. Automated test generation for debugging multiple bugs in arithmetic circuits. *IEEE Trans. Comput.* 68, 2 (2018), 182–197.

[7] Yangdi Lyu and Prabhat Mishra. 2020. Automated test generation for activation of assertions in RTL models. In *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASPDAC'20)*. IEEE, 223–228.

[8] Yangdi Lyu, Alif Ahmed, and Prabhat Mishra. 2019. Automated activation of multiple targets in RTL models using concolic testing. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE'19)*. IEEE, Florence, Italy, 354–359.

[9] Prabhat Mishra and Farimah Farahmandi. 2019. *Post-Silicon Validation and Debug*. Springer, Berlin.

[10] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. 2018. Directed test generation using concolic testing on RTL models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. 1538–1543.

[11] Yangdi Lyu and Prabhat Mishra. 2020. Scalable concolic testing of RTL models. *IEEE Trans. Comput.* (2020). Early access. https://ieeexplore.ieee.org/document/9099620/.

[12] Farimah Farahmandi and Prabhat Mishra. 2017. Automated debugging of arithmetic circuits using incremental gröbner basis reduction. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'17)*. IEEE, 193–200.

[13] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. 2012. *System-level Validation: High-level Modeling and Directed Test Generation Techniques*. Springer, Berlin.

[14] Yangdi Lyu and Prabhat Mishra. 2020. Automated trigger activation by repeated maximal clique sampling. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'20)*. 482–487.

[15] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. 2012. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, Berlin.

[16] Edmund M. Clarke Jr. , Orna Grumberg, and Doron A. Peled. 1999. Model Checking. In *The MIT Press*. Springer, Berlin.

[17] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. 2012. Automatic RTL test generation from SystemC TLM specifications. *ACM Trans. Embed. Comput. Syst.* 11, 2 (2012), 1–25.

[18] Xiaoke Qin and Prabhat Mishra. 2012. Directed test generation for validation of multicore architectures. *ACM Trans. Design Autom. Electron. Syst.* 17, 3 (2012), 1–21.

[19] Mingsong Chen and Prabhat Mishra. 2011. Property learning techniques for efficient generation of directed tests. *IEEE Trans. Comput.* 60, 6 (2011), 852–864.

[20] Mingsong Chen, Xiaoke Qin, and Prabhat Mishra. 2014. Learning-oriented property decomposition for automated generation of directed tests. *J. Electron. Test.* 30, 3 (2014), 287–306.

[21] Heon-Mo Koo and Prabhat Mishra. 2009. Functional test generation using design and property decomposition techniques. *ACM Trans. Embed. Comput. Syst.* 8, 4 (2009), 1–33.

[22] Prabhat Mishra and Nikil Dutt. 2008. Specification-driven directed test generation for validation of pipelined processors. *ACM Trans. Design Autom. Electron. Syst.* 13, 3 (2008), 1–36.

[23] Farimah Farahmandi and Prabhat Mishra. 2016. Automated test generation for debugging arithmetic circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'16)*. IEEE, Dresden, Germany, 1351–1356.

[24] Xiaoke Qin and Prabhat Mishra. 2012. Automated generation of directed tests for transition coverage in cache coherence protocols. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'12)*. IEEE, 3–8.

[25] Mingsong Chen and Prabhat Mishra. 2011. Decision ordering-based property decomposition for functional test generation. In *Proceedings of the Design, Automation and Test in Europe*. IEEE, 1–6.

[26] Sudhi Proch and Prabhat Mishra. 2016. Test generation for hybrid systems using clustering and learning techniques. In *Proceedings of the 29th International Conference on VLSI Design and 15th International Conference on Embedded Systems (VLSID'16)*. IEEE, 589–590.

[27] Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. 2010. Synchronized generation of directed tests using satisfiability solving. In *Proceedings of the 23rd International Conference on VLSI Design*. IEEE, 351–356.

[28] Thanh Nga Dang, Abhik Roychoudhury, Tulika Mitra, and Prabhat Mishra. 2009. Generating test programs to cover pipeline interactions. In *Proceedings of the 46th ACM/IEEE Design Automation Conference*. IEEE, 142–147.

[29] Prabhat Mishra and Mingsong Chen. 2009. Efficient techniques for directed test generation using incremental satisfiability. In *Proceedings of the 22nd International Conference on VLSI Design*. IEEE, 65–70.

[30] Heon-Mo Koo and Prabhat Mishra. 2006. Functional test generation using property decompositions for validation of pipelined processors. In *Proceedings of the Design Automation and Test in Europe Conference*, Vol. 1. IEEE, 1–6.

[31] Prabhat Mishra and Nikil Dutt. 2005. Munich, Germany. In *Proceedings of the Design, Automation and Test in Europe*. IEEE, 678–683.

[32] Prabhat Mishra and Nikil Dutt. 2004. Graph-based functional test program generation for pipelined processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1. IEEE, 182–187.

[33] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. 2019. *System-on-Chip Security: Validation and Verification*. Springer Nature.

[34] Prabhat Mishra, Swarup Bhunia, and Mark Tehranipoor. 2017. *Hardware IP Security and Trust*. Springer.

[35] Yangdi Lyu and Prabhat Mishra. 2020. Scalable activation of rare triggers in hardware Trojans by repeated maximal clique sampling. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* (2020). Early access. https://ieeexplore.ieee.org/document/9179783.

[36] Zhixin Pan and Prabhat Mishra. 2021. Automated test generation for hardware Trojan detection using reinforcement learning. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'21)*.

[37] Alif Ahmed, Farimah Farahmandi, Yousef Iskander, and Prabhat Mishra. 2018. Scalable hardware Trojan activation by interleaving concrete simulation and symbolic execution. In *Proceedings of the IEEE International Test Conference (ITC'18)*. IEEE, 1–10.

[38] Farimah Farahmandi and Prabhat Mishra. 2017. FSM anomaly detection using formal analysis. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'17)*. IEEE, 313–320.

[39] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. 2017. Trojan localization using symbolic algebra. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASPDAC'17)*. IEEE, 591–597.

[40] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer, Berlin, 419–423.

[41] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 213–223. DOI: http://dx.doi.org/10.1145/1065010.1065036

[42] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224. Retrieved from http://dl.acm.org/citation.cfm?id=1855741.1855756.

[43] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. DOI: http://dx.doi.org/10.1145/2110356.2110358

[44] B. Korel and A. M. Al-Yami. 1996. Assertion-oriented automated test data generation. In *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE, Berlin, 71–80. DOI: http://dx.doi.org/10.1109/ICSE.1996.493403

[45] Giuseppe Di Guglielmo, Luigi Di Guglielmo, Andreas Foltinek, Masahiro Fujita, Franco Fummi, Cristina Marconcini, and Graziano Pravadelli. 2013. On the integration of model-driven design and dynamic assertion-based verification for embedded software. *J. Syst. Softw.* 86, 8 (2013), 2013–2033.

[46] IEEE. 2010. IEEE standard for property specification language (PSL). *IEEE Std 1850–2010 (Revision of IEEE Std 1850–2005)* (2010), 1–182.

[47] IEEE. 2013. IEEE standard for SystemVerilog–Unified hardware design, specification, and verification language. *IEEE Std 1800–2012 (Revision of IEEE Std 1800–2009)* (2013), 1–1315.

[48] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. 1983. The temporal logic of branching time. *ACTA Informatica* 20, 3 (1983), 207–226.

[49] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer et al. 2002. The ForSpec temporal logic: A new temporal property-specification

language. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, 296–311.

[50] Andreas Bauer and Martin Leucker. 2011. The theory and practice of SALT. In *Proceedings of the NASA Formal Methods Symposium*. Springer, Berlin, 13–40.

[51] Deian Tabakov, Gila Kamhi, Moshe Y. Vardi, and Eli Singerman. 2008. A temporal language for SystemC. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*. IEEE, 1–9.

[52] Harry Foster, Kenneth Larsen, and Mike Turpin. 2006. Introduction to the new accellera open verification library. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon'06)*. Citeseer.

[53] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.* 10, 6 (Apr. 1975), 234–245. DOI: http://dx.doi.org/10.1145/390016.808445

[54] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, Berlin, 359–364.

[55] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. 2018. Directed test generation using concolic testing on RTL models. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'18)*. IEEE, 1538–1543.

[56] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet. 2008. Generation of test programs for the assertion-based verification of TLM models. In *Proceedings of the 3rd International Design and Test Workshop*. IEEE, 237–242. DOI: http://dx.doi.org/10.1109/IDT.2008.4802505

[57] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta. 2008. Accelerating assertion coverage with adaptive testbenches. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 27, 5 (May 2008), 967–972. DOI: http://dx.doi.org/10.1109/TCAD.2008.917975

[58] Yann Oddos, Katell Morin-Allory, Dominique Borrione, Marc Boulé, and Zeljko Zilic. 2009. MYGEN: Automata-based on-line test generator for assertion-based verification. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI (GLSVLSI'09)*. ACM, New York, NY, 75–80. DOI: http://dx.doi.org/10.1145/1531542.1531563

[59] Jason G. Tong, Marc Boulé, and Zeljko Zilic. 2013. Test compaction techniques for assertion-based test generation. *ACM Trans. Des. Autom. Electron. Syst.* 19, 1, Article 9 (Dec. 2013), 29 pages. DOI: http://dx.doi.org/10.1145/2534397

[60] R. Mukherjee, D. Kroening, and T. Melham. 2015. Hardware verification using software analyzers. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. IEEE, 7–12. DOI: http://dx.doi.org/10.1109/ISVLSI.2015.107

[61] S. Ray, E. Peeters, M. M. Tehranipoor, and S. Bhunia. 2018. System-on-chip platform security assurance: Architecture and validation. *Proc. IEEE* 106, 1 (Jan. 2018), 21–37. DOI: http://dx.doi.org/10.1109/JPROC.2017.2714641

[62] C. Wang, Y. Cai, Q. Zhou, and H. Wang. 2018. ASAX: Automatic security assertion extraction for detecting Hardware Trojans. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference (ASP-DAC'18)*. IEEE, 84–89. DOI: http://dx.doi.org/10.1109/ASPDAC.2018.8297287

[63] National Institute of Standards and Technology. 2020. National Vulnerability Database. Retrieved from https://nvd.nist.gov.

[64] Yangdi Lyu and Prabhat Mishra. 2020. System-on-Chip Security Assertions. Retrieved from arxiv:eess.SY/2001.06719.

[65] Arm Ltd. 2020. ARM TrustZone. Retrieved from https://developer.arm.com/technologies/trustzone.

[66] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 1–19.

[67] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIXSecurity'18)*. 973–990.

[68] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. 2018. Scalable test generation for Trojan detection using side channel analysis. *IEEE Trans. Info. Forensics Secur.* 13, 11 (2018), 2746–2760.

[69] Zhixin Pan, Jennifer Sheldon, and Prabhat Mishra. 2020. Test generation using reinforcement learning for delay-based side channel analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'20)*.

[70] Yangdi Lyu and Prabhat Mishra. 2020. Automated test generation for Trojan detection using delay-based side channel analysis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'20)*. 1031–1036.

[71] Yangdi Lyu and Prabhat Mishra. 2019. Efficient test generation for Trojan detection using side channel analysis. In *Proceeding sof the Conference on Design Automation and Test in Europe Conference (DATE'19)*. 408–413.

[72] Yangdi Lyu and Prabhat Mishra. 2018. A survey of side-channel attacks on caches and countermeasures. *J. Hardware Syst. Secur.* 2, 1 (2018), 33–50.

[73] Yuanwen Huang, Swarup Bhunia, and Prabhat Mishra. 2016. MERS: Statistical test generation for side-channel analysis-based Trojan detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. 130–141.

[74] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. 2001. Efficient detection of vacuity in temporal model checking. *Formal Methods Syst. Design* 18, 2 (Mar. 2001), 141–163. DOI: http://dx.doi.org/10.1023/A:1008779610539
[75] Harry D. Foster and Adam C. Krolnik. 2007. *Creating Assertion-based IP*. Springer Science & Business Media.
[76] S. Williams. 2020. Icarus verilog. Retrieved from http://iverilog.icarus.com.
[77] Bruno Dutertre. 2014. Yices 2.2. In *Proceedings of the Conference on Computer-Aided Verification (CAV'14) (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, Berlin, 737–744.
[78] OpenCores. 2020. Retrieved from https://www.opencores.org/.
[79] Intel. 2019. Intel Software Guard Extensions. Retrieved from https://software.intel.com/en-us/sgx.
[80] Alireza Monemi et al. 2017. ProNoC: A low latency network-on-chip-based many-core system-on-chip prototyping platform. *Microprocess. Microsyst.* 54 (2017), 60–74.
[81] OpenRISC. 2020. mor1kx. Retrieved from https://github.com/openrisc/mor1kx.
[82] CHIPS Alliance. 2020. EH1 SweRV RISC-V Core 1.7 from Western Digital. Retrieved from https://github.com/chipsalliance/Cores-SweRV.
[83] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke. 2008. Automatic generation of complex properties for hardware designs. In *Proceedings of the Design, Automation and Test in Europe.* 545–548. DOI: http://dx.doi.org/10.1109/DATE.2008.4484908
[84] S. Hertz, D. Sheridan, and S. Vasudevan. 2013. Mining hardware assertions with guidance from static analysis. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 32, 6 (June 2013), 952–965. DOI: http://dx.doi.org/10.1109/TCAD.2013.2241176
[85] D. Kroening and M. Purandare. [n.d.] EBMC. Retrieved from http://www.cprover.org/ebmc.
[86] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: A survey. *J. Artific. Intell. Res.* 4 (1996), 237–285.
[87] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *ACM Sigplan Notices* 47, 6 (2012), 193–204.

Received May 2020; revised November 2020; accepted December 2020