Automated Generation of Security Assertions for RTL Models

HASINI WITHARANA, University of Florida, USA
ARUNA JAYASENA, University of Florida, USA
ANDREW WHIGHAM, University of Florida, USA
PRABHAT MISHRA, University of Florida, USA

System-on-Chip (SoC) security is vital in designing trustworthy systems. Detecting and fixing a vulnerability in the early stages is easier and cost-effective. Assertion-based verification is widely used for functional validation of Register-Transfer Level (RTL) designs. Assertions can improve the controllability and observability that can lead to faster error detection and localization. While assertions are widely used for functional validation of RTL models, there is limited effort in applying assertions to detect SoC security vulnerabilities. Specifically, a fundamental challenge in SoC security and trust validation is how to develop high-quality security assertions. In this paper, we perform automated vulnerability analysis of RTL models to generate security assertions for six classes of vulnerabilities. Experimental results show that the generated security assertions can detect a wide variety of vulnerabilities. Our automated framework can drastically reduce the overall security validation effort compared to the manual development of security assertions. Automated generation of security assertions will enable assertion-based verification as one of the most promising pre-silicon security sign-off solutions.

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

Additional Key Words and Phrases: Security Assertion, Vulnerability Detection, Assertion-based Verification, Hardware Security, Trust Verification

1 INTRODUCTION

Functional validation is widely acknowledged as a major bottleneck due to increasing System-on-Chip (SoC) design complexity coupled with reduced time-to-market constraints. According to the Wilson Research 2020 functional verification study [1], 51% of development time in hardware designs were spent in verification. In spite of such extensive effort, only 32% of the systems can achieve the first silicon success. As shown in Figure 1, the study also highlights that more than 50% designs utilized Assertion-based Verification (ABV) to address this fundamental bottleneck in functional validation and there is a steady increase in the adoption of ABV over the years. Functional assertions represent the important properties (behaviors) of a design that should be correct. For example, in case of a two-input adder, we can write an assertion to check whether the output is always equal to the summation of the two inputs. Assertions increase the observability of a design. Assertions can also be used to improve the controllability of internal signals.

In recent years, SoC security has become a major concern due to the increasing integration of Intellectual Property (IP) cores from potentially untrusted third-party vendors [2]. While functional assertions are successful in capturing functional behaviors, they are not suitable for capturing security vulnerabilities. Specifically, the functional assertions represent expected functional behaviors whereas security assertions are supposed to capture unexpected security vulnerabilities [3]. One of the major bottlenecks in assertion-based security validation is how to develop a set of security assertions for a given design. Manual development of security assertions require designers with in-depth security expertise. Most importantly, manual assertion development can be infeasible for industrial SoC designs due to its inherent limitations such as ad-hoc nature, cumbersome, time-consuming and error-prone. There are successful efforts for automated generation of functional assertions [4] that are not designed to detect security vulnerabilities. While there is an early effort

Authors' addresses: Hasini Witharana, University of Florida, Gainesville, FL, 32611, USA, witharana.hasini@ufl.edu; Aruna Jayasena, University of Florida, Gainesville, FL, 32611, USA; Andrew Whigham, University of Florida, Gainesville, FL, 32611, USA; Prabhat Mishra, University of Florida, Gainesville, FL, 32611, USA.

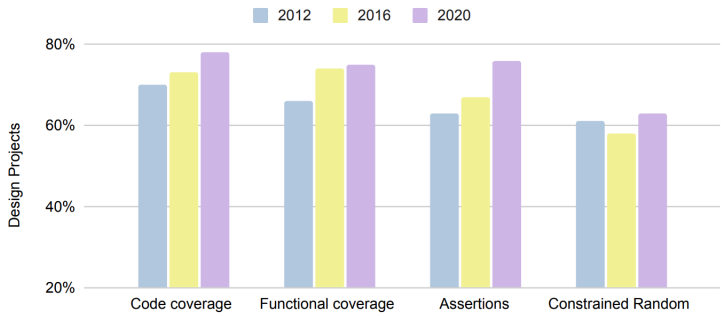


Fig. 1. ASIC functional verification trend [1].

in generating security assertions [5], it has limited applicability for only hardware Trojans. Clearly, there is a need for developing efficient mechanisms for automated generation of security assertions to cover a wide variety of security vulnerabilities. Automated generation of security assertions is the key ingredient in making assertion-based verification as one of the most promising pre-silicon security sign-off solutions.

1.1 Overview and Contributions

In this paper, we propose an automated security assertion generation framework for RTL models using both static code analysis and execution trace analysis. Figure 2 shows our proposed framework that consists of three major tasks. The first task performs vulnerability analysis of an SoC design (RTL models) to identify a wide variety of potential vulnerabilities. Specifically, we consider six types of vulnerabilities including malicious implants, information leakage, illegal states and transitions, permissions and privileges, resource management, and buffer issues. The second task enables an automated generation of security assertions to capture the potential vulnerabilities. The goal of the third task is to perform assertion-based security validation. There are two possible outcomes of the third task: (i) the generated assertion is not valid (modify the assertion generation) or (ii) the generated assertion activated a vulnerability (SoC vulnerability should be fixed).

The focus of this paper is to analyze the RTL models to generate security assertions. Unless we can formally verify the absence of all possible vulnerabilities, assertion-based security validation provides the ability to check for vulnerabilities during execution. Therefore, the assertions at pre-silicon level provides two opportunities: (i) runtime monitoring of security vulnerabilities, and (ii) fixing the vulnerabilities in the early stages of the design. The generated security assertions are primarily used for pre-silicon security verification. Once pre-silicon security sign-off is done, these security assertions can be removed from the pre-silicon models. Also, these assertions can be synthesized as runtime checkers for post-silicon validation. Due to hardware overhead (e.g., area and power) constraints, it may not be feasible to synthesize all the security assertions generated by our framework. We can utilize existing assertion synthesis methods to explore trade-off between the hardware overhead and post-silicon security coverage [6] by focusing on assertions that are most relevant for post-silicon debug [7]. Moreover, our approach can be extended to accommodate scenarios when RTL (source code) is not available. Similar to the scenario when verification engineers write test cases based on architectural specification (even before RTL is ready) to cover expected functional scenarios, we can produce security assertions based on unexpected scenarios. For example, irrespective of the implementation, "*request* $\rightarrow$ *acknowledgment*" should hold if the design has a handshaking protocol. Of course, this assumes that we have information about the expected (unexpected) behaviors from the specification. If the model is executable, we can generate assertions from the execution traces using the methodology outlined in [8]. Once we have the

assertions, we can run them concurrently with the black-box IPs provided by the third-party vendors. This may limit the types of assertion variables (e.g., only input/output variables of the IPs).

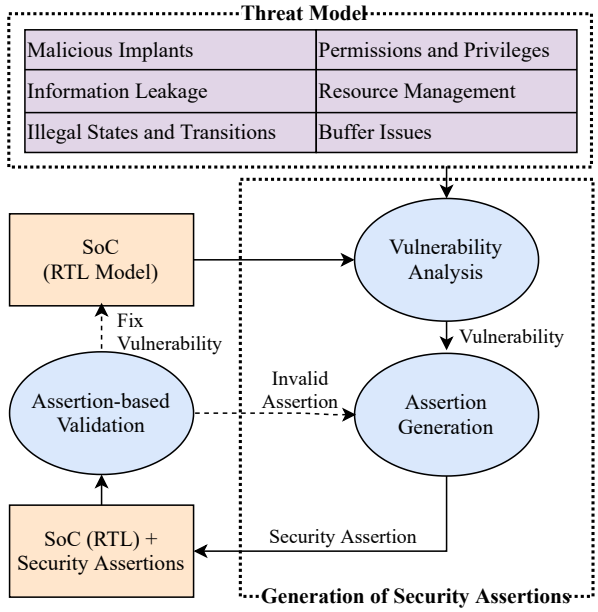


Fig. 2. Overview of the proposed assertion generation framework that consists of three major tasks: vulnerability analysis, assertion generation and validation of assertions.

To the best of our knowledge, our proposed approach is the first attempt in automated generation of security assertions for a wide variety of SoC vulnerabilities. Specifically, this paper makes the following major contributions:

(1) We propose vulnerability analysis for six classes of SoC vulnerabilities using both static code analysis and execution trace analysis.
(2) We implement efficient algorithms for automated generation of security assertions.
(3) We perform assertion-based security validation of RTL models to demonstrate that the generated security assertions can detect a wide variety of security vulnerabilities.

The remainder of the paper is organized as follows. Section 2 surveys the related efforts. Section 3 describes the threat model. Section 4 presents an overview of the proposed methodology. Section 5 – 10 describe the automated assertion generation for six classes of vulnerabilities. Section 11 presents the experimental results. Finally, Section 12 concludes the paper.

2 BACKGROUND AND RELATED WORK

This section surveys the related efforts in three broad categories to highlight the need for the proposed work.

2.1 Automated Generation of Assertions

There are many promising efforts [8–17] for automated generation of assertions that can be broadly divided into two categories. The methods in the first category rely on static analysis of RTL models to generate functional assertions [16, 17]. Schema-driven assertion generation [16] is an early attempt to automate assertions using static analysis and schema or template-based libraries. The authors in [17] statically analyze the syntax and extract the properties. The methods in the second

category perform dynamic analysis of simulation traces to generate functional assertions [4, 18, 19]. The tool IODINE [20] utilizes dynamic analysis to extract design properties such as state transitions and request signal pairs from simulation traces. GoldMine [4, 21] uses both dynamic analysis and data mining to mine complex assertions for a RTL design. GoldMine can generate both propositional and temporal assertions. A-TEAM [8] uses execution trace analysis for assertion generation and uses decision tree based techniques to mine functional assertions. The DOVE framework relies on the symbolic simulation of the firmware to search for corner cases in its computational paths to generate a compact set of assertions representing these paths [9]. The work presented in [13] uses data mining approaches to analyse the execution traces and extract assertions. Some of the approaches use both static and dynamic analysis. The authors in [14] use a system level approach that works on non-Boolean data types as well. There are also efforts for automated re-use of RTL assertions for validation of TLM models [11]. There is a recent effort for generating assertions to detect hardware Trojans [5]. The authors in [15] use execution trace provided by mutation testing to mine assertions for don't care condition exploitation of hardware Trojans. Our proposed framework is applicable for generating a wide variety of security assertions.

2.2 Assertion-based Verification

Assertions are used for both pre-silicon and post-silicon SoC validation [22]. Pre-silicon assertion-based validation is preferred since it is less expensive to capture bugs in the early stages of the design. Moreover, post-silicon validation faces inherent observability limitations. In the pre-silicon stage, there are two ways to validate the assertions. The approaches in the first category perform simulation-based validation of the design using different test vectors to activate assertions [23–26]. The approaches in the second category utilize formal methods (such as model checking) to prove if the assertions are valid [27, 28]. For post-silicon validation, the assertions are synthesised as checkers to monitor the respective functional behaviors during post-silicon debug [29–31]. All of these methods deal with functional assertions. While there are recent efforts in defining security assertions [3] as well as generating test patterns for activating them [32], there are no existing efforts in automated generation of security assertions.

2.3 Integration of Assertions in RTL Models

There are two types of assertions: immediate assertions and concurrent assertions. Immediate assertions enable checking a property when the control reaches an exact location in the code during simulation whereas concurrent assertions are checked in each clock cycle to verify the behavior. Note that the immediate assertions can be converted to concurrent assertions by modifying the antecedent. There are two types of approaches to define assertions: library-based and language-based. In library-based approach (e.g., Accellera Open Verification Library (OVL) [33]), assertion support is added to the existing languages. In contrast, the syntax of the language is used to write assertions in language-based approaches. Property Specification Language (PSL) [34] and System Verilog Assertions (SVA) [35] are examples of language-based approaches. Our framework supports both PSL and SVA assertions. Our framework also supports both immediate and concurrent assertions.

3 THREAT MODEL

There are a wide variety of SoC related security vulnerabilities. In this paper, we consider six classes of vulnerabilities: malicious implants, information leakage, illegal states and transitions, permissions and privileges, resource management, and buffer issues. These vulnerabilities can get introduced in various stages during the design cycle [36] - high-level specification, RTL implementation, synthesis (e.g., gate-level netlist), layout, fabrication, post-silicon debug, or in-field

deployment. These vulnerabilities can be intentional (malicious) or unintentional. For example, malicious implants or backdoors can be introduced by a rogue designer or untrusted third-party tools. Similarly, a designer mistake (e.g., incorrect branch condition) or buggy tools (e.g., unspecified states or illegal transitions) can create a vulnerability that can be exploited by an attacker for information leakage or unprivileged access to security assets.

3.1 Malicious Implants

Insertion of malicious code into the original code is referred as code injection in the software community. A similar concept is also applicable for hardware designs and popularly known as hardware Trojans. Untrusted third-party vendors, during the process of fabrication, or even a buggy design tool can cause insertion of hardware Trojans. These Trojans are hard to detect due to their stealthy trigger conditions, and they can be activated by specific input patterns. Unintended consequences such as information leakage can occur as a result of hardware Trojans. An attacker is likely to insert malicious implants in the hard-to-detect and rare-to-activate areas in the design such that the hardware Trojans can stay hidden during traditional validation.

3.2 Information Leakage

Modern SoCs support a variety of processes with different security levels on one physical system. A process should not leak secure information, such as a cryptographic key, into another process. Some systems, like ARM TrustZone [37], partition processes into separate trusted and untrusted environments. Another approach is to directly track the flow of information at the hardware level [38]. We need to identify and eliminate channels implicitly leaking secure information.

3.3 Illegal States and Transitions

Finite state machines are used to describe the expected behavior of an SoC in the RTL description. Functional validation covers all the expected and valid transitions in the FSM. However, there can be undefined states and transitions. Illegal states and transitions can facilitate attackers to get access to protected states and compromise the operation of the system [39]. It is important to find all possible undefined states to generate assertions to alarm invalid states and transitions.

3.4 Permissions and Privileges

Access control is used to grant permission for different privilege levels separately. As an example, the ARM7 processor comprises seven different privilege levels such as user mode, interrupt mode, and supervisor mode. Changing between these privilege levels requires permission grants by different trigger conditions which are defined by the access control system. It is essential to validate the permission grant conditions before changing the user levels. Otherwise, attackers can get access to the superuser level in an unauthorized way.

3.5 Resource Management

SoC utilizes diverse resources for various purposes. For example, a JTAG port allows a user to dump various internal structure during offline debug. Similarly, embedded trace buffers enable the recording of important signals (events) during execution. Attackers can misuse these resources to get access to the SoC and dump sensitive information using JTAG. Therefore, it is essential to verify the behavior and access policies of such resources and implement safeguards. For example, JTAG should not be enabled during normal execution (only during offline debug).

3.6 Buffer Issues

Buffers are commonly used in modern SoC designs to store data during communication between various components. Based on the complexity of the design, it may support advanced features (e.g., out-of-order and speculative execution) as well as a large number of heterogeneous buffers. Due to the complexity and unique implementation requirements of the buffers, a significant validation effort is needed to detect any hidden flaws. For example, in a single port buffer both wr_en and rd_en should not be enabled at the same time.

4 OVERVIEW

Figure 3 presents an overview of our proposed security assertion generation framework for six classes of vulnerabilities outlined in Section 3. The basic idea is to perform either static (structural) or dynamic (simulation) analysis of the SoC design (RTL models) and automatically generate security assertions for the potential vulnerabilities. For example, Algorithm 1 (Section 5) performs rareness analysis based on dynamic execution (simulation) and generates security assertions related to malicious implants. Similarly, Algorithm 2 (Section 6) performs static flow analysis of RTL models and generates information leakage related security vulnerabilities to capture potential flow violations. Section 7 (Algorithm 3) describes FSM analysis for automated generation of security assertions for illegal states and transitions. Section 8 (Algorithm 4) describes the generation of permission/privilege related security assertions based on access violation analysis of RTL models. Similarly, Algorithm 5 (Section 9) performs resource analysis to enable automated generation of resource-related security assertions. Finally, Section 10 (Algorithm 6) presents buffer analysis followed by generation of security assertions related to buffer issues in RTL models. The effectiveness of the generated security assertions is evaluated in Section 11.

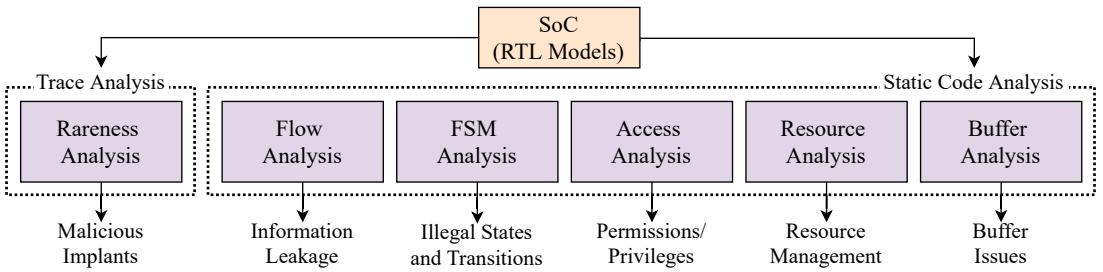


Fig. 3. Overview of automated security assertion generation for six classes of SoC security vulnerabilities.

5 ASSERTION GENERATION FOR MALICIOUS IMPLANTS

As outlined in Section 3, attackers are likely to construct hard-to-detect trigger conditions using rare events. Therefore, the first step would be to identify the rare nodes of a RTL design. These rare nodes are likely candidates for Trojan triggers. Therefore, if we monitor such rare nodes or associated triggers using security assertions, we will be able to detect malicious implants. In other words, if any of the suspicious rare triggers get activated, security assertions will fail and the designer can check the validity of the potential threat. Figure 4 shows an overview of assertion generation. This section is organized as follows. First, we define few terms used in the assertion generation for malicious implants. Next, we propose our assertion generation algorithm. Finally, we perform complexity analysis.

Definition 1: Given a finite sequence of time steps $< t^1, \ldots, t^m >$ and a set of signals $< S_1, S_2, \cdots >$, an Execution Trace (E) is a sequence of tuples $(t^k, < S_1^k, ..., S_i^k, ... >)$ where $1 \leq k \leq m$ (total execution time) and $1 \leq i \leq$ number of all signals. S_i^k is the value of signal S_i at time step t^k. □

Definition 2: Let S_i^k be a x bit signal in the design such that $S_i^k =< s_{i,1}^k, ..., s_{i,j}^k, ..., s_{i,x}^k >$, where $s_{i,j}^k$ is the value of j^{th} bit of S_i at time step k and $1 \leq j \leq x$. □

Definition 3: Let r be a real-valued rareness threshold such that $0 \leq r \leq 1$. □

Definition 4: Let $f_{ij}(t) : R^+ \rightarrow \{0,1\}$ be a function expressing the Boolean value of $s_{i,j}^k$ for any time step (t). The probability of $s_{i,j}$ being 1 can be expressed as shown in Equation 1. Then the probability of $s_{i,j}$ being 0 can be expressed as shown in Equation 2. □

$$P(s_{i,j} = 1) = \frac{1}{m} \sum_{t=1}^{m} f_{ij}(t) \tag{1}$$

$$P(s_{i,j} = 0) = 1 - P(s_{i,j} = 1) \tag{2}$$

Definition 5: A signal ($s_{i,j}$) can be termed as a rare node for value 1 (or value 0) if $P(s_{i,j} = 1) \leq r$ (or $P(s_{i,j} = 0) \leq r$).

Definition 6: A trigger condition of size n is the output of a Boolean AND operation on n rare signals. If there are N rare signals, there are $\binom{N}{n}$ possible trigger conditions of size n. If we allow any trigger sizes up to (including n), the number of possible trigger conditions would be $\sum_{i=1}^{n} \binom{N}{n}$. □

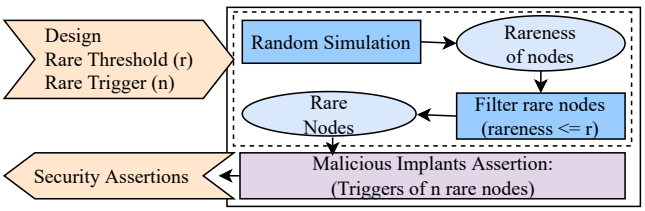


Fig. 4. Generation of security assertions for malicious implants based on rare nodes and trigger conditions.

5.1 Algorithm

Algorithm 1 outlines the security assertion generation procedure for malicious implants. It consists of three major phases: random simulation, trace analysis for identification of rare nodes, and construction of security assertions. First, the design is simulated using millions of random tests. Next, the execution trace is analyzed to identify the rare nodes based on the rareness threshold (r). Finally, the selected rare nodes are used to construct security assertions based on the number of rare triggers (n).

Algorithm 1 Assertion Generation for Malicious Implants

Input: Design D, RareThreshold r, NumberOfTriggers n.
Output: SecurityAssertions A.

```
1: function ASSERTION GENERATION(D, r, n)
2:     ExecutionTrace E ← RandomSimulation(D)
3:     A, Nodes, RareNodes ← ∅
4:     Nodes ← ExtractNodes(E)
5:     for node in Nodes do
6:         if node.rareness ≤ r then
7:             RareNodes ← RareNodes ∪ node
8:     A ← ConstructAssertions(RareNodes, n)
9:     return A
```

The rareness of a signal is calculated by dividing the number of occurrences for a specific signal value by the total number of execution cycles as shown in Equation 1. Specifically, our rareness

analysis consists of four major steps (Line 2-7 in Algorithm 1). First, we flatten the design preserving the instance names and signal names. Next, we simulate the design for the pipeline depth of the design (and repeat it for multiple times using thousands of random test patters). Then, we analyze each signal in the Value Change Dump (VCD) to compute how many times each specific signal obtained the value '1' or '0'. Finally, we select the (signal, value) pairs that have appeared in the simulation less than the "rareness threshold" for a specific value. For example, consider a one bit signal S which can have either value 1 or 0. Assume that the simulation traces exhibit the fact that S has a value 1 for 92% of the time (i.e., value 0 only for 8% of the time). The "signal S for value 0" (S,0) will be treated as a rare signal if the "rareness threshold" is 0.1 (i.e., 10%). In this case (S, 1) will be treated as a non-rare instance of the signal. While rareness analysis can be performed at different granularity (e.g., bit-level versus word-level), we have performed bit-level rareness analysis.

Listing 1. Generic template for malicious implants with N rare nodes

```
assert property (!RARE_1 & !RARE_2 & ... & !RARE_N);
```

Random simulation has the advantage of activating easy-to-detect scenarios. Therefore, we run a large number of random simulations (in the order of thousands to millions) to different rare and non-rare signals. Suppose there is a one bit Trojan trigger *trig*, which never gets activated during random simulation (i.e *trig* is always 0). With the execution trace analysis, we can identify that *trig* being 1 is an extremely rare case. A designer needs to carefully decide the rareness threshold since it presents a trade-off between the ability to detect potential Trojans and the number of rare nodes and corresponding assertions. A large threshold can lead to many rare nodes and a large number of assertions. While a small threshold can lead to a small set of rare nodes, the respective assertions may not be able to capture some potential Trojans.

Listing 2. Combination of rare triggers for n=3

```
Rare Nodes = {A, B, C}
assert property (!A);
assert property (!B);
assert property (!C);
assert property (!A & !B);
assert property (!A & !C);
assert property (!B & !C);
assert property (!A & !B & !C);
```

Once rare nodes are identified, the next step considers a set of rare trigger conditions to construct the security assertions. Listing 1 shows the generic template for specifying assertions related to malicious implants. The template uses the negation of the rare node values. Therefore, if any of the rare nodes gets activated, the assertion will fail. An important consideration is to decide the number of nodes in the trigger and respective combinations since it presents a trade-off between the ability to detect potential Trojans and the number of security assertions. A large n can lead to an exponential number of assertions. While a small n can lead to a reasonable number of assertions, the respective assertions may not be able to capture some potential Trojans. Listing 2 shows an illustrative example of using the template in Listing 1 for capturing malicious implants with combinational rare triggers. The listing considers three rare nodes and all possible combinations (three one-node trigger, three 2-node triggers and one three 3-node trigger) for $n \leq 3$. There are many alternatives to trade-off between threshold and the number of assertions. For example, we can select a lower rareness threshold to allow a larger set of rare signals, but we can reduce the trigger size to control the number of assertions. For example, we can have 5 rare nodes, but limit the trigger size of 2 (five one-node trigger, and 10 2-node triggers).

5.2 Complexity Analysis

Algorithm 1 consists of three important phases: random simulation, trace analysis and assertion generation. Therefore, the time complexity is bounded by $O(t_{RandomSimulation})$ + $O(t_{TraceAnalysis})$ + $O(t_{AssertionGeneration})$. Since test generation time is negligible compared to simulation time for larger designs, the first phase is bounded by the time taken for millions of random simulation ($t_{RandomSimulation}$). The trace analysis time, $O(t_{TraceAnalysis})$, is expected to be comparable to the simulation time since it is linearly related to $s \times O(v) \times c$, where s is the number of simulations, c is the number of simulated cycles and v is the number of variables in the design. The assertion generation time, $O(t_{AssertionGeneration})$, is the time required to construct the required number of combinations based on trigger size (n). While it can be exponential based on trigger size (n), it is negligible in practice since an attacker is likely to use a small trigger to avoid detection. Typically, n is assumed to be less than 16 for million-gate designs. Overall, the runtime of the algorithm is bounded by the simulation time, which is in the order of seconds for small RTL models. The memory requirement is bounded by the size to store the results of random simulation ($s \times O(n) \times c$). The memory complexity can be improved to $O(v)$ by considering the fact that we can maintain one data structure of all the variables and process one simulation at a time for rareness computation.

6 ASSERTION GENERATION FOR INFORMATION LEAKAGE

In this section, we consider assertion generation to cover possible scenarios for information leakage. The objective is to ensure that the design assets should not be propagated to any insecure/public outputs of the design. We utilize tagging and taint analysis to identify such leaky paths as outlined in Figure 5. This section is organized as follows. First, we propose our assertion generation algorithm. Next, we perform complexity analysis.

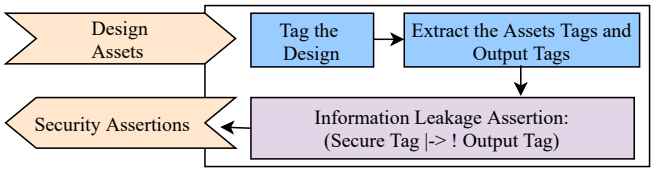


Fig. 5. Generation of security assertions for information leakage. Leaky path identification uses tagging and taint analysis.

6.1 Algorithm

Algorithm 2 outlines the major steps of our proposed assertion generation procedure. It accepts the design and secure assets as inputs and produces security assertions. The first step of our algorithm is to tag the design for taint analysis. Our tagging implementation relies on existing works on taint analysis in [40]. Tagging paradigms vary in accuracy based on two distinct attributes: explicit/implicit and precise/imprecise. Explicit tagging tracks information exclusively in the data flow, while implicit tagging further includes information flow in the control flow. At the gate-level, these two flow types are indistinguishable as all information flow can be expressed as data flow between gates. However, the distinction exists at the RTL level [41]. Precise tagging computes the taint of a variable strictly according to the definition of information flow: information is leaked only if a change in a secure input can cause a change in the value of the variable. Imprecise tagging assumes that information is leaked if any of the input to the variable is secure, regardless of their value. Listing 3 and 5 show the examples of implicit imprecise and explicit imprecise tagging, respectively. Note that the original design had only three variables (a, b and c). The other three variables (a_t, b_t and c_t) and associated two assignments are introduced during tagging.

Algorithm 2 Assertion Generation for Information Leakage

Input: Design D, Assets a.
Output: SecurityAssertions A.

function ASSERTION GENERATION(D, a)
  $A \leftarrow \emptyset$
  $P_{tree} \leftarrow$ GenerateParseTree(D)
  $D_{tag} \leftarrow$ TagDesign(P_{tree})
  $sTags \leftarrow$ ExtractSecureTags(a, P_{tree})
  $oTags \leftarrow$ ExtractOutputTags(P_{tree})
  for i in $sTags$ **do**
    for j in $oTags$ **do**
      $A \leftarrow A \cup$ assert property ($sTags_i$ |=> $!oTags_j$)
  return A

The tagging variables (e.g., a_t for signal a) are widely used to analyze control or data flow during simulation to understand if the value in a specific variable (e.g., secret asset) can flow to another variable (e.g., primary output). In Listing 3, implicit tagging is presented where the information in the control flow is considered for tagging. For example in line 5, the assignment for the signal c depends not only on signal b but also on signal a in the 'if' condition in line 4. Therefore, we can represent the equivalent information flow for reg c using c_t <= $a\_t|b\_t$ (line 6). In contrast, Listing 5 considers explicit tagging to analyze data flow information. For example in line 5, the assignment for the signal c depends on reg b according to the data flow. Therefor, we can represent the equivalent information flow for signal c using c_t <= b_t (line 6).

Listing 3. Example of taint logic for implicit imprecise tagging

```
1. reg a, b, c;
2. reg a_t, b_t, c_t;
3. always (@posedge clk) begin
4.       if (a)
5.           c <= b;
6.           c_t <= a_t | b_t;
7.       else
8.          c <= 0;
9.          c_t <= a_t;
10. end
```

For the tagging logic, we have utilized an implicit and imprecise paradigm wherein any input to a variable being tainted implies the variable is tainted. Specifically, for a variable, v, expressed as an original logic function, f, the taint, v_t, is the disjunction of the taints of all the inputs to the original logic function: $v = f(a_1, a_2, \ldots, a_n) \implies v_t = \sum_{i=0}^{n} a_{i_t}$, where a_{i_t} is the taint of a_i, the taint of a variable x is $x_t \in \{0, 1\}$, and + is the disjunction operation. To adapt it to the RTL level, each assignment operation in the design had an additional tagging assignment added in the same branch. The set of inputs $A = \{a_1, a_2, \ldots, a_n\}$ was assumed to consist of variables on the right-hand side of the assignment operation and all signals within the branch predicates leading to the current branch. The clock and reset signals were assumed to be untainted.

Listing 4. Generic template for information leakage

```
assert property (SECURE_TAG |=> !OUTPUT_TAG);
```

Once we get the tagged design based on the assets, we construct a list of secure tags ($sTags$) and non-secure (output) tags ($oTags$). The generation of security assertions are based on considering the possible combinations between all secure and non-secure (output) tags using the template as shown in Listing 4. Note that a structural analysis of the design will be able to show the possible paths, however, activation of such an assertion during execution guarantees that there is definite information leakage.

Listing 5. Example of taint logic for explicit imprecise tagging

```
1. reg a, b, c;
2. reg a_t, b_t, c_t;
3. always (@posedge clk) begin
4.       if (a)
5.           c <= b;
6.           c_t <= b_t;
7.       else
8.          c <= 0;
9.          c_t <= 0;
10. end
```

Listing 6 shows an illustrative example of information leakage. In this example, register s is a secure asset and register b is an insecure output node. As shown in line 4, the instrumented design will contain tags for all the nodes. In line 7, secure asset s is assigned to a a, and then in line 10, a is assigned to b. This leads to a leaky path from secure asset s to non-secure output b. To detect information leakage from the secure variable s to non-secure (output) variable b, we need to check if the corresponding tags (s_t and b_t, respectively) match as shown in Listing 7.

Listing 6. Example of information leakage

```
1. reg s  \\Secure Asset
2. reg a
3. reg b  \\Insecure Node
4. reg s_t, a_t, b_t   \\Tags
5. always (@posedge clk) begin
6.       .........
7.       a = s
8.       a_t = s_t \\Tagging statement
9.       ..........
10.      b = a
11.      b_t = a_t \\Tagging statement
12. end
```

Listing 7. An illustrative example of using the template in Listing 4 for capturing information leakage for the Listing 6

```
assert property (s_t |=> !b_t);
```

6.2 Complexity Analysis

Algorithm 2 consists of three important phases: tagging the design, extract asset/output tags using static analysis, and assertion generation. Therefore, the time complexity is bounded by $O(t_{Tagging})$ + $O(t_{ExtractTags})$ + $O(t_{AssertionGeneration})$. For the first phase, we traverse the abstract syntax tree after parsing to tag the design. The vertices and edges in the tree depend on the lines of code (l) in the design. Hence the $t_{Tagging}$ is linear to $O(l)$. When traversing through the syntax tree in the first phase, we store the variables and the associated tags in a hash table. Therefore, the $t_{ExtractTags}$

will take constant time for extracting assets/tags. Assertion generation time ($t_{AssertionGeneration}$) is effectively the time to construct the assertions for all possible combinations of the secure tags (assets) and non-secure tags (public outputs). Given that the number of assets is likely to be a small number of variables (constant complexity), $t_{AssertionGeneration}$ will be bounded by $O(v)$, where v is the number of variables in the design. Typically, the number of variables are linearly related to the design size (number of lines of code). Overall, the runtime of the algorithm is bounded by $O(l)$. Similarly, the memory requirement is bounded by the size to store the hash table for tags, which is bounded by $O(l)$.

7 ASSERTION GENERATION FOR ILLEGAL STATES AND TRANSITIONS

Figure 6 shows an overview of our assertion generation framework for illegal states and transitions. The basic idea is to extract the finite-state machine (FSM) from the design and identify illegal states and transitions to construct security assertions. Functional validation typically checks the valid transitions in an FSM. In this section, our focus is on vulnerabilities related to illegal transitions between legal states or potential (illegal) transitions from unspecified (don't care) states.

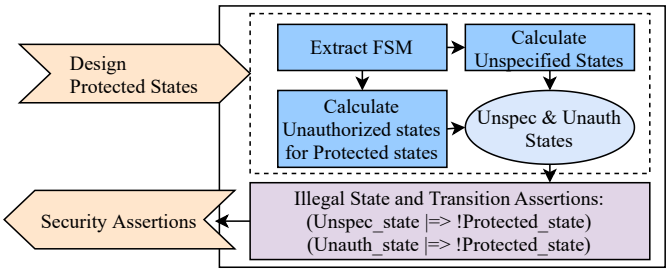


Fig. 6. Generation of security assertions for illegal state and transition related security vulnerabilities.

There can be different types of states as shown in Figure 7:

(1) *Protected State (S_{11}):* This state has valuable information (asset). For example, S_{11} has been identified as a protected state by the designer.
(2) *Authorized State (S_{10}):* A protected state can be accessed only from an authorized state (e.g., a state for checking a password).
(3) *Unspecified State (S_{00}):* These don't care states are likely created during synthesis. These states can be exploited as a backdoor to capture assets.
(4) *Unauthorized State (S_{01}) :* Even though this state is defined properly, there cannot be any transition from an unauthorized state to a protected state.

The remainder of this section is organized as follows. First, we propose our assertion generation algorithm. Next, we perform complexity analysis.

7.1 Algorithm

Algorithm 3 outlines the major steps. We first extract the FSM from the RTL models. Then identify which states are unspecified and unauthorized. In this paper, we will use the term illegal states to refer to both unspecified and unauthorized states. Next, we generate assertions using the template as shown in Listing 8. The assertion should monitor any illegal transitions between illegal states to protected states. Algorithm 3 does not create any assertions for illegal states. Rather, it analyzes the design to identify the illegal states, and constructs assertions for illegal transitions such as a transition from an unspecified state to a protected state. It is reasonable to have an unspecified state, however, it is a vulnerability for an unspecified state trying to make a transition to a protected state.

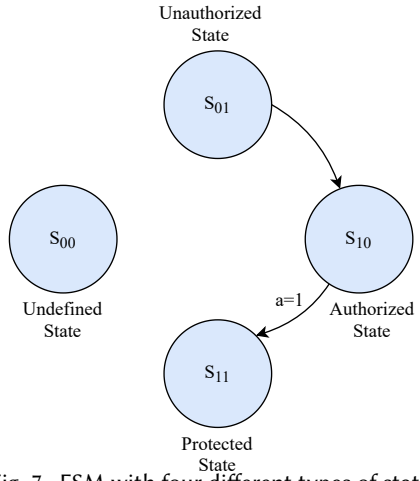


Fig. 7. FSM with four different types of states

Listing 8. Generic template for illegal states and transitions

```
assert property (ILLEGAL_SATE |=> !PROTECTED_STATE);
```

Listing 9 shows an illustrative example of using the template in Listing 8 for capturing illegal transitions for the FSM presented in Figure 7 . In the FSM, S_{11} is a protected state and both S_{00} and S_{01} are illegal states. The assertions shown in Listing 9 will fail, if any transition happens from any of the illegal states (S_{00} and S_{01}) to the protected state (S_{11}).

Listing 9. An illustrative example of using the template in Listing 8 for capturing illegal transitions for the FSM presented in Figure 7

```
assert property (S_00 |=> !S_11);
assert property (S_01 |=> !S_11);
```

Algorithm 3 Assertion Generation for Illegal Transitions

Input: Design D, Protected State pS, Authorized States aS.
Output: SecurityAssertions A.

1: **function** Assertion Generation(D, pS, aS)
2: $A \leftarrow \emptyset$
3: $fsm \leftarrow$ ExtractFSM(D)
4: $iStates \leftarrow$ GetIllegalStates(fsm, pS, aS)
5: **for** i in $iStates$ **do**
6: **for** j in pS **do**
7: $A \leftarrow A \cup$ assert property ($iStates_i$ |=> $!pS_j$)
8: **return** A

7.2 Complexity Analysis

Algorithm 3 consists of three important phases: extract FSM based on static analysis, generate the list of illegal states, and assertion generation. Therefore, the time complexity is bounded by $O(t_{FSMExtraction}) + O(t_{IllegalStates}) + O(t_{AssertionGeneration})$. The first phase is bounded by the number of lines of the code ($O(l)$). The second phase is bounded by $O(f)$, where f is the number of

states in the FSM. Assertion generation time ($t_{AssertionGeneration}$) is effectively the time to construct the assertions for all possible illegal transitions. So $t_{AssertionGeneration}$ will be bounded by $O(f^2)$. Overall, the runtime of the algorithm is bounded by the FSM complexity ($O(f^2)$). Considering the fact that the number of protected states are expected to be a small number, the time complexity would be $O(f)$, which is linearly related to the design complexity ($O(l)$). The memory requirement is bounded by the size to store the FSM ($O(f^2)$).

8 ASSERTION GENERATION FOR PERMISSIONS AND PRIVILEGES

In this section, we generate security assertions to ensure that assets should not be accessed without appropriate permissions and privileges. Figure 8 shows an overview of the assertion generation framework. This section is organized as follows. First, we propose our assertion generation algorithm. Next, we perform complexity analysis.

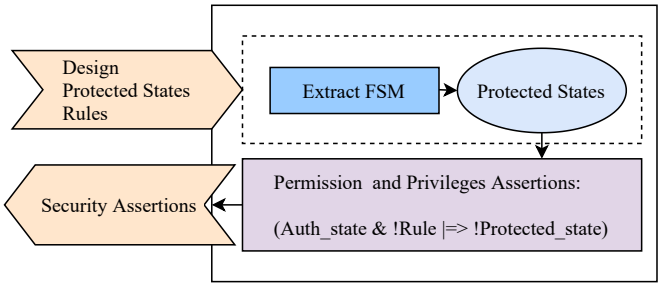


Fig. 8. Generation of security assertions for permission and privileges related security vulnerabilities

8.1 Algorithm

Algorithm 4 shows two major steps: FSM extraction and assertion generation. It assumes that the designer will specify the protected states as well as the authorized states with specific permissions (rules) to transition to protected states. It is a usual practice today to expect the designers to provide information about security assets (e.g., encryption key) for vulnerability analysis. It is possible to figure out the protected states by observing the states that manipulate assets. Once we identify the protected states, it is possible to identify the authorized states by observing the states that are making transitions to the protected states. Unfortunately, such a derivation is fundamentally flawed since it will miss the vulnerability in the design (e.g., an unauthorized state will be treated as an authorized state simply because it has a transition, which should not be there in the first place). In other words, it is expected that a designer provides this information based on the specification (or double check the information derived from the implementation).

Listing 10. Generic template for permission and privileges

```
assert property (AUTHORIZED_SATE & RULE |=> !PROTECTED_STATE);
```

We assume that there is a one-to-one mapping between the number of authorized states and the number of rules since there would be one unique transition. For each authorized state and associated transition rule, we can generate a security assertion based on a template as shown in Listing 10. By using this template, we ensure that the transition to the protected state is only possible from the authorized state when the necessary permission (rule) is true.All the invalid transitions to the protected states are covered under Section 7.

Consider the example FSM in Figure 7. The inputs for the Algorithm 4 will be protected state (S_{11}), authorized state (S_{10}) and the rule ($a == 1$). Listing 11 shows an illustrative example of using the template in Listing 10 for capturing permission and privilege for these inputs. The assertion

shown in the listing ensures that a transition between S_{10} and S_{11} is only possible when the value of a is 1.

Algorithm 4 Assertion Generation for Permissions and Privileges

Input: Design D, Protected States ps, Authorized States as, Permissions $rule$.
Output: SecurityAssertions $\{p\}$.

1: **function** ASSERTION GENERATION($D, ps, as, rule$)
2: $A \leftarrow \emptyset$
3: $fsm \leftarrow$ ExtractFSM(D)
4: **for** i in as **do**
5: **for** j in ps **do**
6: **if** as_i has a transition to ps_j using $rule_i$ **then**
7: $A \leftarrow A \cup$ assert property (as_i & ! $rule_i$ |=> ! ps_j)
8: **return** A

Listing 11. An illustrative example of using the template in Listing 10 for capturing permission and privileges for the FSM presented in Figure 7

```
assert property ((s_10 & !(a==1)) |=> !s_11);
```

8.2 Complexity Analysis

Algorithm 4 consists of two important phases: FSM extraction and assertion generation. Therefore, the time complexity is bounded by $O(t_{FSMExtraction}) + O(t_{AssertionGeneration})$. First phase is bounded by the number of lines of the code ($O(l)$). Assertion generation time will be bound by the number of authorized states as well as protected states provided by the user ($O(f^2)$), where f is the number of states in the FSM. Considering the fact that the number of valid transitions between authorized and protected states tends to be a small number, the overall run-time of the algorithm is bounded by the FSM extraction time ($O(l)$). The memory requirement is bounded by the size to store the FSM ($O(f^2)$).

9 ASSERTION GENERATION FOR RESOURCE MANAGEMENT

Figure 9 shows an overview of security assertion generation for resource management related vulnerabilities. Securing the resources that may interface with external devices/tools from attackers is the primary objective here. This section is organized as follows. First, we propose our assertion generation algorithm. Next, we perform complexity analysis.

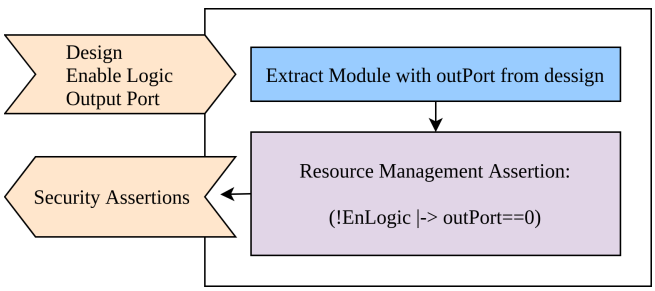


Fig. 9. Generation of security assertions for resource management related security vulnerabilities

9.1 Algorithm

Algorithm 5 shows the major steps of our assertion generation framework. It assumes that the user will provide the resource enable logic (e.g., in case of JTAG, JTAG enable logic) and the output port of the resources under consideration. The algorithm will traverse the design to find the resource output port. Next, the assertions will be generated using the template presented in Listing 12. Finally they will be added at the relevant place in the RTL description.

Listing 12. Generic template for resource management

```
assert property (!RESOURCE_ENABLE_LOGIC |=> (OUTPUT_PORT==0));
```

Algorithm 5 Assertion Generation for Resource Management

Input: Design D, EnableLogic en, OutputPort op
Output: SecurityAssertions A.

1: **function** FIND OUTPUT PORT(D,en, op)
2: $A \leftarrow \emptyset$
3: $M \leftarrow$ ExtractModules(D)
4: **for** $module \in M$ **do**
5: **if** op in $module$ **then**
6: $A \leftarrow A \cup$ assert property ($!en \mapsto op == 0$)
7: **return** A

Let us consider a simple SoC that consists of multiple IP cores. For the debug purposes of the SoC, trace buffers are used. Register data during the operation is collected in the trace buffer. When to collect data to the trace buffer is determined by the trigger logic associated with every register. The trace dump can be analyzed during post-silicon debug. To transfer the trace details from the SoC to the external analysing tool, JTAG port is used. Although JTAG port is designed for debugging purposes, attackers can get access and alter the behavior of the design. Therefore, it is important to verify that JTAG can only be activated by JTAG enable logic during debug mode and otherwise, the JTAG output would be null. Listing 13 shows an illustrative example of using the template for capturing resource for JTAG port. This assertion verifies that JTAG can only be activated by JTAG enable logic during debug mode.

Listing 13. An example of using the template in Listing 12 for capturing resource management for JTAG

```
assert property (!debug_enable |=> (JTAG==0));
```

9.2 Complexity Analysis

Algorithm 5 consists of two important phases: extract modules based on static analysis of the design and assertion generation. Therefore, the time complexity is bounded by $O(t_{ModuleExtraction})$ + $O(t_{AssertionGeneration})$. The first phase is bounded by the number of lines of the code ($O(l)$). Assertion generation time will be bound by the number of resources in the design. This time is negligible compared to $t_{ModuleExtraction}$ since there will be a small number of critical resources. Overall, the run-time of the algorithm is bounded by the design complexity ($O(l)$). The memory complexity is also bounded by the design complexity ($O(l)$) to store the design since the number of resources would be small.

10 ASSERTION GENERATION FOR BUFFER ISSUES

Buffers are a crucial part of hardware designs and also highly susceptible to vulnerabilities as outlined in Section 3. Figure 10 provides an overview of the automated assertion generation procedure for buffer related vulnerabilities. This section is organized as follows. First, we propose our assertion generation algorithm. Next, we perform complexity analysis.

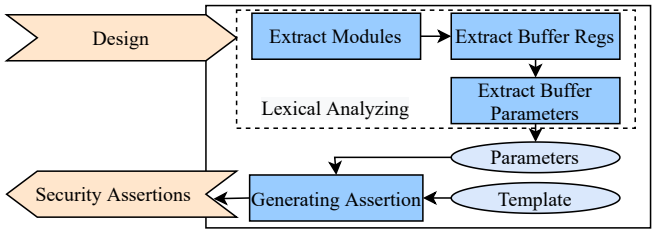


Fig. 10. Generation of security assertions for buffer issues

10.1 Algorithm

Algorithm 6 describes the major steps involved in generating buffer-related assertions. The first step is to extract all the buffer instances and associated parameters. Next, the security assertions are generated utilizing templates. Listing 14 shows an example template. Note that the usage of buffers for different use-cases can be varied due to the functional requirements. Even for the same use-case such as cache buffers, the usage of buffers can be varied for different designs with design specific requirements. For example, in one design, the writing to the buffer will only be allowed when write enable flag is on and read enable flag is off. In a different design, the writing to the buffer might rely on more parameters such as an internal ready signal even when the write enable is on and read enable is false.

Algorithm 6 Assertion Generation for Buffer Issues

Input: Design D.
Output:SecurityAssertions A.

1: **function** GENERATE ASSERTIONS(D)
2: $A \leftarrow \emptyset$
3: $Buffers \leftarrow ExtractBuffers(D)$
4: **for** b in $Buffers$ **do**
5: $A \leftarrow A \cup ConstructAssertions$(b)
6: **return** A

Listing 14. Generic template for buffer issues

```
1. //Index checking
   assert property (WRITE_POINTER<(MAX_LIMIT)&READ_POINTER<(MAX_LIMIT));
2. //Ensure no data will be loss
   assert property (WRITE_DATA_SIZE==(DATA_WIDTH) & READ_DATA_SIZE==(DATA_WIDTH));
3. //Sequential buffer access
  assert property (@(posedge clk)
     WRITE_ENABLE |-> ( WRITE_POINTER == $past(WRITE_POINTER)+1));
  assert property (@(posedge clk)
     READ_ENABLE |-> ( READ_POINTER == $past(READ_POINTER)+1));
4. //Read and write cannot happen at same time
  assert property (!(READ_ENABLE && WRITE_ENABLE));
```

Let us consider a Network-on-Chip (NoC) router buffer as an illustrative example. The buffer will hold the data until the arbitration logic lets the router to transmit data. Therefore, the simplest implementation will store and transmit data based on the first-come-first-out (FIFO) policy. As a result, the buffer reads and writes are sequential. Assume that the NoC router buffer used in this example is a single port router. Listing 15 shows an illustrative example of using the template in Listing 14 for capturing NoC-specific buffer issues.

Listing 15. An illustrative example of using Listing 14 for capturing vulnerability in a NoC router

```
assert property (wr_ptr<(bufferMaxLimit) & rd_ptr<(bufferMaxLimit));
assert property (wr_data.size==(data_width) & rd_data.size==(data_width));
assert property (@(posedge clk) wr_en |-> ( wr_ptr == $past(wr_ptr)+1));
assert property (!(rd_en && wr_en));
```

10.2 Complexity Analysis

Algorithm 6 consists of two important phases: buffer extraction and assertion generation. Therefore, the time complexity is bounded by $O(t_{BufferExtraction}) + O(t_{AssertionGeneration})$. The buffer extraction time is linearly related to the design complexity (lines of codes($O(l)$)). Since we are using a small number of templates, the assertion generation time is linearly dependent on the number of buffers in the design, which is also bounded by $O(l)$. Overall, the time complexity is bounded by $O(l)$ for large designs. The memory requirement is also bounded by the design complexity ($O(l)$) to store the design as well as buffers.

11 EXPERIMENTS

This section demonstrates the effectiveness of our proposed framework in automated generation of security assertions. First, we describe our experimental setup. Next, we present our assertion generation results for six classes of vulnerabilities. Finally, we discuss the assertion validation results.

11.1 Experimental Setup

We have explored security assertion generation for six vulnerability classes. We have utilized Pyverilog, Icarus Verilog and Yosys APIs for the implementation of the algorithms shown in Section 5 - 10. We have evaluated the assertion generation algorithms with respect to runtime, memory requirements as well as scalability. In order to ensure that the generated assertions are valid, we have performed test generation using EBMC [42] model checker and used the tests to activate the assertions.

We have used a wide variety of vulnerability injected benchmarks from TrustHub [43]. We have also modified golden benchmarks from OpenCores [44] by injecting diverse vulnerabilities to demonstrate the applicability of our assertion generation framework. We ran our experiments on Intel i7-5500U @ 3.0GHz CPU with 16GB RAM machine.

11.2 Assertion Generation Results

This section presents assertion generation results for all the six vulnerability classes. The number of assertions generated for each benchmark with respect to a specific vulnerability class is shown in Table 1. We did not consider all benchmarks for all types of vulnerabilities for practical reasons. For example, we cannot evaluate buffer issues in a design without any buffers.

11.2.1 Malicious Implants. When generating security assertions using execution trace analysis, we need to simulate the design using test vectors. As discussed in Section 5.2, the runtime of Algorithm 1 is bounded by the simulation time during rareness analysis. Table 2 shows the rareness

Table 1. Number of security assertions generated for different benchmarks with respect to Malicious Implants (MI), Information Leakage (IL), Illegal States (IS), Permissions and Privileges (PP), Resource Management (RM) and Buffer Issues (BI).

Benchmark	No of Gates	Security Assertions						Validation (Coverage)
		MI	IL	IS	PP	RM	BI	
Arbiter	5	1	2	0	0	0	0	100%
b01	45	1	2	6	1	0	0	100%
b06	50	1	4	6	1	0	0	100%
b10	121	4	3	14	1	0	0	100%
RS232-T100	136	8	4	15	1	1	0	100%
b11	401	3	2	42	3	1	0	100%
FIFO	1542	12	7	0	0	0	4	100%
PIC16F84-T400	1903	16	18	0	0	1	4	100%
b14	2157	1	4	33	5	1	0	100%
cpu8080	4441	10	6	32	2	1	48	100%
mips	13886	6	1	0	0	1	5	100%
mor1kx_cache	17020	25	9	4	1	1	10	100%
AES-T1100	40365	22	1	0	0	1	0	100%
AES-T2000	42626	19	1	0	0	1	0	100%
AES-T1300	43550	27	1	0	0	1	0	100%
AES-T500	43558	14	1	0	0	1	0	100%

threshold (r) and the number of rare triggers (n) used for each benchmark. For example, we have considered 0.002 as the rareness threshold and trigger size of 1 for the AES benchmark. We have evaluated different rareness thresholds with respect to three factors to decide the values in Table 2: Trojan coverage, false positive rate, and the number of rare signals. Our primary objective is to achieve a high Trojan coverage using a small set of assertions. These are contradictory requirements. For example, a large rareness threshold can improve Trojan coverage. However, when the rareness threshold is too large with respect to the design, it can lead to a large number of rare signals, which can lead to a prohibitively large number of assertions due to the exponential number of possible triggers based on the rare signals. As shown in Figure 20, the false positive rate also increases with the increase of rareness threshold. As a result, we need to choose a threshold that will provide reasonable Trojan coverage without leading to unacceptable assertion overhead or false positives.

Table 2. Rareness threshold (r) and number of rare trigger combinations (n) for malicious implants.

	Arbiter	b01	b06	b10	RS232	b11	FIFO	PIC16F84	b14	cpu8080	mips	Mor1kx	AES(avg)
r	0.3	0.2	0.2	0.01	0.001	0.001	0.001	0.001	0.001	0.001	0.002	0.002	0.002
n	1	1	1	1	1	2	1	1	1	1	1	1	1

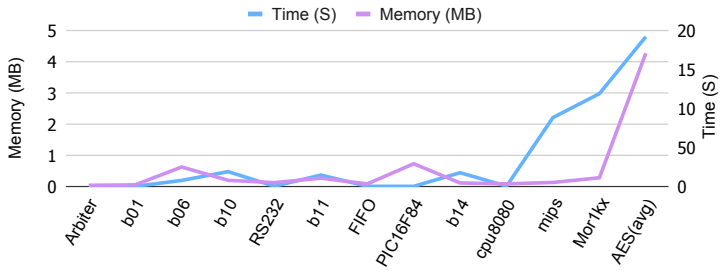


Fig. 11. Assertion generation time and memory requirements for malicious implants.

Figure 11 presents the assertion generation time (s) and memory (MB) requirements for malicious implants. The left y-axis represents the memory requirements (in MB) and the right y-axis represents

the runtime (in seconds). The x-axis indicates the benchmarks, which are sorted in the increasing order of design complexity. As expected based on the discussion in Section 5.2, larger designs like AES takes more time and memory than smaller designs like Arbiter. This is because the simulation with thousands of random inputs will take significant time and memory for complex designs like AES. As shown in Table 1, the number of generated assertions increases with the complexity of the code. This is expected since the number of rare nodes increases with the complexity of the design. The difference would be more drastic if we use the same rareness threshold across benchmarks.

11.2.2 Information Leakage. Figure 12 shows the assertion generation results based on Algorithm 2. The assertion generation time is bounded by the tagging time and the number of variables in a design. These two parameters increase with the design complexity in the benchmark. When the design complexity increases, the time taken for security assertion generation also increases.

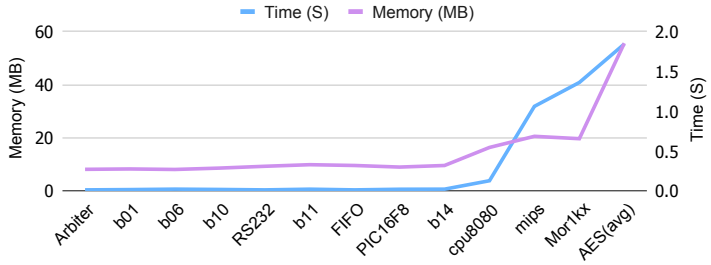


Fig. 12. Assertion generation time and memory requirements for information leakage.

As expected, the runtime is higher for a larger design (e.g., AES) compared to a simpler design (e.g, Arbiter). The memory requirement is also expected to increase with the increasing design size as shown in Figure 12. Moreover, the number of assertions generated is proportional to the number of output ports since the number of assets is typically small. For example, both Arbiter and AES benchmarks have only one asset. Since the Arbiter benchmark has 2 output ports and AES has 1 output port, the number of assertions generated are 2 and 1, respectively, as shown in Table 1.

11.2.3 Illegal States and Transitions. Figure 13 shows the assertion generation results for Algorithm 3. The runtime and memory requirements for security assertion generation is bounded by the size of the FSMs. In most cases, larger designs lead to more complex FSMs with larger number of states and transitions. As shown in Figure 13, the assertion generation time and memory requirements increase with the design complexity for most of the benchmarks.

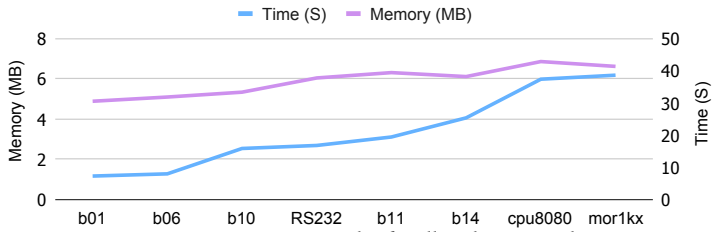


Fig. 13. Assertion generation results for illegal state and transitions.

11.2.4 Permission and Privileges. Figure 14 shows the assertion generation results for Algorithm 4. As expected based on the discussion in Section 8.2, the time for assertion generation is bounded by FSM extraction time, which is linearly related to the design complexity. The assertion generation time and memory requirements increase with the line of codes for most of the benchmarks. If we compare runtime/memory requirements of permissions/privileges (Figure 14) versus illegal

states/transitions (Figure 13), we can observe that the time and memory taken to generate assertions for permission and privileges vulnerability is always less than the time taken for illegal state and transition vulnerability. It is due to the fact that the number of valid transitions is typically less than the number of possible invalid (illegal) transitions in an FSM.

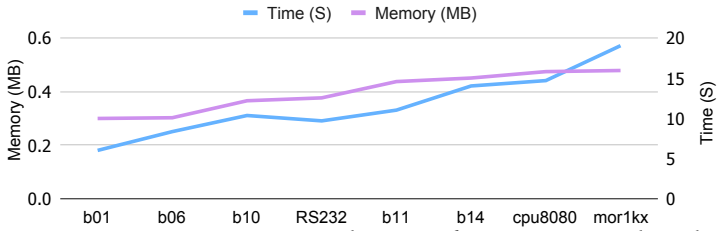


Fig. 14. Assertion generation time and memory for permissions and privileges.

11.2.5 Resource Management. Figure 15 shows the assertion generation results for Algorithm 5. The left axis shows the memory (MB) taken and the right axis shows the time taken in seconds. The complexity of the design increases from left to right in Figure 15. As expected based on the discussion in Section 9.2, the runtime and memory requirements for Algorithm 5 is bounded by the design complexity. Similar to previous ones, the time and memory footprint for larger designs are higher than smaller designs.

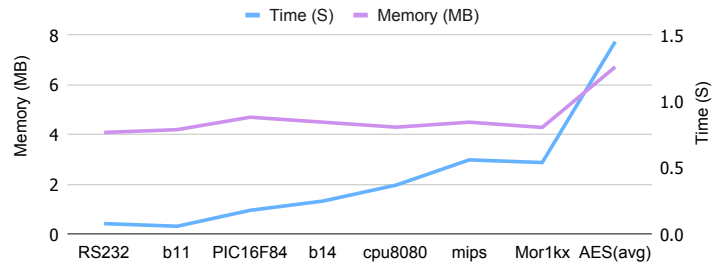


Fig. 15. Assertion generation time and memory for resource management.

11.2.6 Buffer Issues. Figure 16 shows the assertion generation results for Algorithm 6. The runtime and memory requirements for Algorithm 4 is bounded by the design complexity. Benchmark FIFO is the bare-bone first-in-first-out buffer. In case of PIC16F84, mips and mor1kx, the read from the buffer and write to the buffer is limited to at most two per single instance. Therefore, the generated number of assertions is proportional to the number of instances. However, in the case of cpu8080, there are more than 400 buffer accesses by 48 unique registers and nets. As a result, the algorithm creates assertions for every unique buffer access. Hence, cpu8080 benchmark results show higher memory usage although the design complexity is less than the mips benchmark.

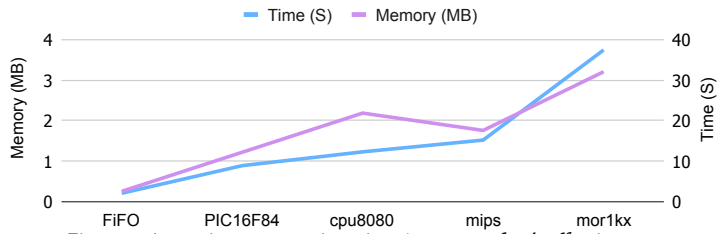


Fig. 16. Assertion generation time/memory for buffer issues.

11.3 Validation of Generated Assertions

In this section, we evaluate the quality of the generated assertions from two perspectives: (i) validity of the generated assertions, and (ii) the vulnerability coverage provided by the generated assertions. The first one checks whether an assertion is able to detect the respective vulnerabilities injected in the design. The second one determines whether the number of assertions generated is sufficient to detect all the vulnerabilities. We used EBMC [42] to generate counterexamples for assertion failures. The generated counterexamples can be used as a test case to activate the respective assertions. We are able to successfully activate all the assertions to ensure that the generated assertions are valid as shown in Table 1.

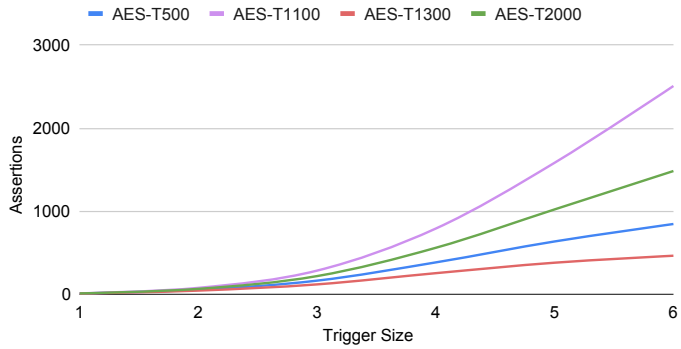


Fig. 17. Number of assertions generated for different trigger sizes (n)

We have also observed that the generated assertions are able to detect all the vulnerabilities. For example, in case of malicious implants, we have explored the number of assertions versus trigger size for different benchmarks, as shown in Figure 17. When the number of nodes in a trigger n increases, the number of security assertions generated also increases. The different trajectories in the figure are due to the fact that the number of rare nodes are different in different benchmarks. Using the rareness threshold of 0.001, the number of rare nodes in AES-T500, AES-T1100, AES-T1300, AES-T2000 are 10, 12, 9 and 11, respectively. As discussed in Section 5, the number of assertions would be higher when there are more rare nodes.

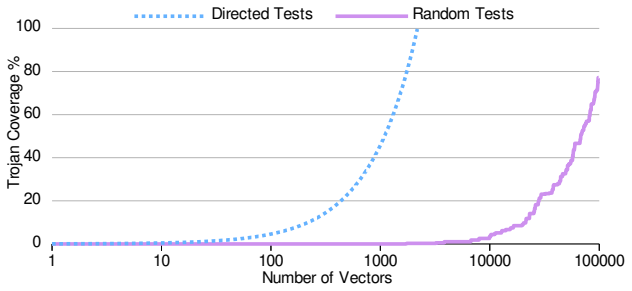
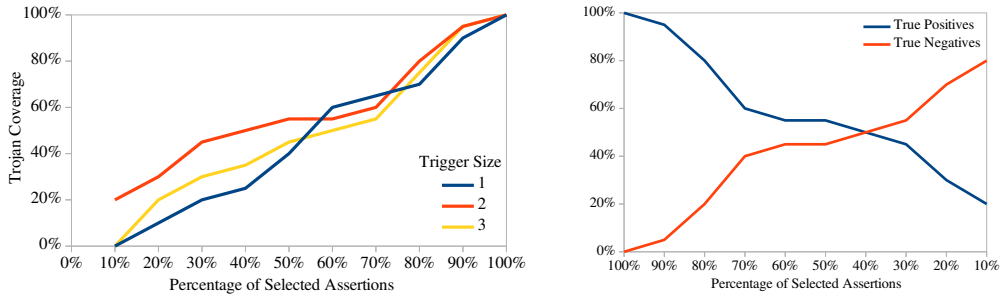


Fig. 18. Assertion validation for AES-T500 with random tests and directed tests

Let us now change our focus to evaluating the quality of the generated assertions in detecting vulnerabilities. Let us assume that the attacker is going to construct triggers with less than 4 nodes ($n < 4$). We used the Trojan injected AES-T500 and added more Trojan triggers with less than 4 nodes. Using Algorithm 1 security assertions are generated and embedded in the design. Then we used random tests and directed tests to activate the Trojans and observed whether the security assertions are capable of detecting the Trojan activation. Figure 18 presents the Trojan coverage



(a) Different trigger combinations

(b) True positives versus true negatives

Fig. 19. Assertion validation results with different percentage of assertions

with respect to number of test vectors for both directed and random tests in AES-T500. When the number of test vectors increases, Trojan coverage also increases. As shown in the figure, we are able to achieve 100% Trojan coverage with less number of directed tests whereas a large number of test vectors will be required in random testing to achieve 100% Trojan coverage. The key observation is that the generated assertions are able to detect all the injected vulnerabilities as long as the trigger can be activated by suitable test patterns.

In case of large designs, an iterative refinement may be required. While a designer can keep on injecting various types of vulnerabilities, any coverage hole can be analyzed to generate additional assertions. For example, if a coverage hole indicates that certain malicious implants are not covered by the generated assertions, a designer can make the rareness threshold larger (creates more rare nodes) or trigger size larger that leads to a larger set of security assertions to improve the coverage. This process continues until the coverage goal is met. We have filtered the assertions based on a percentage of all possible combinations of triggers against the Trojan coverage. As expected, Figure 19a shows that the Trojan coverage increases by increasing the number of assertions. Figure 19b presents the results for varying percentage of selected assertions against the coverage of Trojans correctly (*True Positives*) as well as inability to detect Trojans (*True Negatives*). Clearly, if we have the budget to include all the assertions, it will provide 100% Trojan coverage. If we have less number of assertions, the accuracy (*True Positives*) goes down. As expected, the summation of the True Positives and True Negatives will be all the inserted vulnerabilities (Trojans).

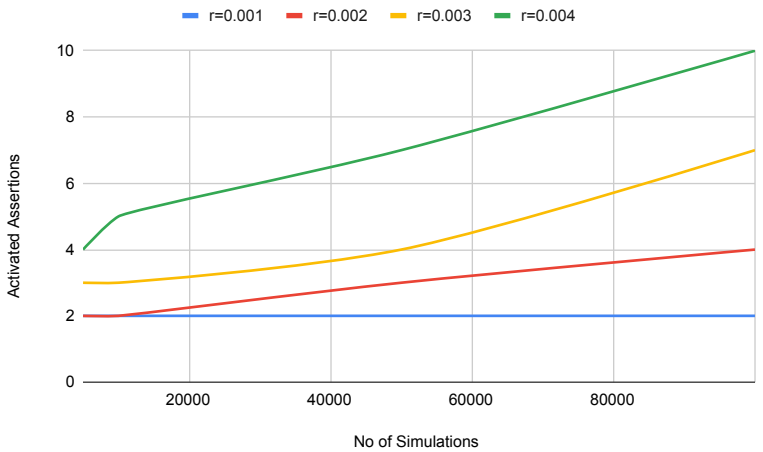


Fig. 20. Assertion activation with respect to the rareness threshold for AES-T500 Trojan free design

Finally, the generated assertions can lead to both true positive and false positive results. As discussed above, a true positive result implies that an activation of an assertion implies that the activation (presence) of a Trojan. A false positive result scenario means that an assertion is activated even in the Trojan free golden design. For example, it is possible to have a rare node activated during normal simulation. We have evaluated our assertion generation for malicious implants to identify how false positive results vary with respect to the rareness threshold. Figure 20 shows the number of activated security assertions in the y-axis whereas the x-axis represents the number of simulations (test vectors). We increase the rareness threshold (r) form 0.001 to 0.004 while maintaining the trigger size (n) as 1. When the rareness threshold is increased, the number of rare nodes increases, including some of the not-so-rare nodes that may get activated during normal simulation. For r value with 0.001, two assertions get activated in the Trojan free design. So, these two assertions can be considered as false positives. As expected, the number of false positives increases when the rareness threshold is increased.

12 CONCLUSION

SoC security is critical to design trustworthy systems. Existing research efforts cover either specific classes of vulnerabilities (e.g., hardware Trojans) or require significant changes to the specification languages and associated design automation tools. In this paper, we proposed an automated SoC security validation framework using security assertions. We have developed efficient algorithms for automated generation of security assertions for six classes of security vulnerabilities. Our experimental results demonstrated that our framework is scalable in terms of both runtime and memory requirements for large designs. Our evaluation also highlights both the validity and usefulness of the generated security assertions.

13 ACKNOWLEDGMENTS

This work was partially supported by grants from National Science Foundation (CCF-1908131) and Semiconductor Research Corporation (2020-CT-2934).

REFERENCES

[1] Harry Foster. Wilson research group functional verification study 2020.
[2] Sandip Ray, Eric Peeters, Mark M Tehranipoor, and Swarup Bhunia. System-on-chip platform security assurance: Architecture and validation. *Proceedings of the IEEE*, 106(1):21–37, 2017.
[3] Yangdi Lyu and Prabhat Mishra. System-on-chip security assertions. *arXiv preprint arXiv:2001.06719*, 2020.
[4] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *DATE*, pages 626–629. IEEE, 2010.
[5] Chenguang Wang et al. Asax: Automatic security assertion extraction for detecting hardware trojans. In *ASP-DAC*, pages 84–89. IEEE, 2018.
[6] Farimah Farahmandi, Ronny Morad, Avi Ziv, Ziv Nevo, and Prabhat Mishra. Cost-effective analysis of post-silicon functional coverage events. In *DATE*, pages 392–397. IEEE, 2017.
[7] Prabhat Mishra and Farimah Farahmandi. *Post-Silicon Validation and Debug*. Springer, 2019.
[8] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. A-team: Automatic template-based assertion miner. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.
[9] Alessandro Danese, Valeria Bertacco, and Graziano Pravadelli. Symbolic assertion mining for security validation. In *DATE*, pages 1550–1555, 2018.
[10] Nicola aand others Bombieri. Mangrove: An inference-based dynamic invariant mining for gpu architectures. *IEEE Transactions on Computers*, 2019.
[11] Tara Ghasempouri, Alessandro Danese, Graziano Pravadelli, Nicola Bombieri, and Jaan Raik. RTL assertion mining with automated RTL-to-TLM abstraction. In *Forum for Specification and Design Languages (FDL)*, pages 1–8, 2019.
[12] Tara Ghasempouri, Jan Malburg, Alessandro Danese, Graziano Pravadelli, Goerschwin Fey, and Jaan Raik. Engineering of an effective automatic dynamic assertion mining platform. In *VLSI-SoC*, pages 111–116. IEEE, 2019.

[13] Po-Hsien Chang and Li-C Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 607–612. IEEE, 2010.

[14] Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *DATE*, pages 67–72. IEEE, 2015.

[15] Nicole Fern and Kwang-Ting Cheng. Mining mutation testing simulation traces for security and testbench debugging. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 714–721. IEEE, 2017.

[16] Amir Hekmatpour and Azadeh Salehi. Block-based schema-driven assertion generation for functional verification. In *Test Symposium, 2005. Proceedings. 14th Asian*, pages 34–39. IEEE, 2005.

[17] Takamitsu Yamada. Assertion generating system, program thereof, circuit verifying system, and assertion generating method, October 13 2009. US Patent 7,603,636.

[18] Frank Rogin, Thomas Klotz, Gorschwin Fey, Rolf Drechsler, and Steffen Rulke. Automatic generation of complex properties for hardware designs. In *2008 Design, Automation and Test in Europe*, pages 545–548. IEEE, 2008.

[19] Lingyi Liu et al. Automatic generation of assertions from system level design using data mining. In *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 2011.

[20] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *DAC*, pages 775–778. IEEE, 2005.

[21] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[22] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)*, 2022.

[23] Babu Turumella and Mukesh Sharma. Assertion-based verification of a 32 thread sparc™ cmt microprocessor. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 256–261. IEEE, 2008.

[24] Vivek N Kallankara, MH Neishaburi, Katarzyna Radecka, and Zeljko Zilic. Using assertions for wireless system monitoring and debugging. In *NEWCAS Conference (NEWCAS), 2010 8th IEEE International*, pages 401–404. IEEE, 2010.

[25] Nicola Bombieri et al. On the reuse of rtl assertions in systemc tlm verification. In *Test Workshop-LATW, 2014 15th Latin American*. IEEE, 2014.

[26] Nicola Bombieri, Riccardo Filippozzi, Graziano Pravadelli, and Francesco Stefanni. Rtl property abstraction for tlm assertion-based verification. In *DATE*, pages 85–90. EDA Consortium, 2015.

[27] Dominique Borrione et al. On-line assertion-based verification with proven correct monitors. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society*. IEEE, 2005.

[28] Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Andrea Fedeli. Hybrid, incremental assertion-based verification for tlm design flows. *IEEE Design & Test of Computers*, 24(2), 2007.

[29] Jean-Samuel Chenard et al. Hardware assertion checkers in on-line detection of faults in a hierarchical-ring network-on-chip. In *Workshop on Diagnostic Services in Network-on-Chips*, 2007.

[30] Uthman Alsaiari, Fayez Gebali, and Mostafa Abd-El-Barr. Programmable assertion checkers for hardware trojan detection. In *PRIME-LA*. IEEE, 2017.

[31] Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *HOST*, pages 49–54. IEEE, 2012.

[32] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(4):1–28, 2021.

[33] Open verification language. https://www.accellera.org/downloads/standards/ovl. Accessed: 2021-05-01.

[34] 1850-2010 - IEEE Standard for Property Specification Language (PSL), 2010.

[35] 1800-2012 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.

[36] Prabhat Mishra, Swarup Bhunia, and Mark Tehranipoor. *Hardware IP security and trust*. Springer, 2017.

[37] Xia Yang, Peng Shi, Bo Tian, Bing Zeng, and Wei Xiao. Trust-e: A trusted embedded operating system based on the arm trustzone. In *IEEE International Conference on Ubiquitous Intelligence and Computing*, pages 495–501, 2014.

[38] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. *SIGARCH Comput. Archit. News*, 37(1):109–120, March 2009.

[39] Farimah Farahmandi and Prabhat Mishra. Fsm anomaly detection using formal analysis. In *ICCD*. IEEE, 2017.

[40] Wei Hu et al. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[41] Armaiti Ardeshiricham et al. Register transfer level information flow tracking for provably secure hardware design. In *DATE*, pages 1691–1696, 2017.

[42] R. Mukherjee, D. Kroening, and T. Melham. Hardware verification using software analyzers. In *VLSI*, pages 7–12, Montpellier, France, July 2015. IEEE.

[43] Trusthub. https://www.trust-hub.org/. Accessed: 2018-10-10.

[44] OpenCores. https://www.opencores.org/, 2021.