RTL-Spec: RTL Spectrum Analysis for Security Bug Localization

Samit S. Miftah*, Shamik Kundu*, Austin Mordahl†, Shiyi Wei†, Kanad Basu*

*ECE Department, University of Texas at Dallas, Richardson, TX, USA
†CS Department, University of Texas at Dallas, Richardson, TX, USA

***Abstract*— Modern System-on-Chip (SoC) designs are integrated with intellectual property (IPs) cores to achieve complex functionalities. While this integration significantly improves the computing power of SoCs, it also leads to an increase in verification complexity pertaining to the security of the SoC design. Existing SoC verification techniques do not offer localization capability to pinpoint the root causes of security vulnerabilities in the register transfer level (RTL) code. This leads to significant delay, incurred due to SoC debugging. Fault localization techniques, such as spectrum-based methodologies, are used predominantly in software validation and testing to debug and localize bugs in programs. However, due to the absence of any such techniques that correlate faulty output with individual lines of RTL code, approaches that can detect vulnerabilities in the hardware have not yet been adopted. In order to circumvent this, in this paper, we, for the first time, propose RTL-Spec, a RTL level security vulnerability localization framework that aims to pinpoint buggy lines in the RTL code to ensure the security of the SoC design in its pre-silicon phase. RTL-Spec achieves this by establishing a correlation between the effects of input patterns in simulation runs and the corresponding statements in the RTL code. The subsequent stage employs a spectrum-based localization technique to identify buggy statements in RTL that might cause vulnerabilities. The efficacy of RTL-Spec was assessed using a buggy version of PULPissimo SoC, utilized in the "Hack@DAC2018" competition. RTL-Spec accurately identified the origin of all 14 vulnerabilities in the RTL code and achieved a precision of 100% in 10 out of the 14 cases.**

***Index Terms*—Security Verification, Bug Localization, Hardware Debugging, Hardware Security.**

I. Introduction

System-on-chips (SoCs) are the brains behind modern computing devices. However, with increase in design complexity, reduction in feature size and time-to-market, a lot of functional bugs are manifested in modern SoCs, which require a thorough verification procedure to address [1]. SoC functional verification already poses a significant challenge in modern chip design. Over 70% of resources and engineering time are dedicated to verification tasks [2]–[4]. In modern SoCs, used for mission-critical environments, security verification manifests as a critical offshoot. This is because an adversary can potentially exploit the corner cases uncovered by traditional simulation-based functional verification, and introduce vulnerabilities, which may jeopardize the overall system. SoC security verification is imperative to address these critical vulnerabilities before a chip is shipped to a customer. In order to ensure security robustness, semiconductor companies practice a very rigorous security development lifecycle (SDL) process concurrently with the conventional hardware development cycle [5], [6].

Till date, existing research has developed various SoC security verification methodologies. These methods include fuzzing, concolic testing, assertion checking, and information flow tracking [7]–[11]. However, none of these approaches are able to localize or diagnose the exact location of the RTL code, which may lead to the security vulnerability. The debugging process is typically manual, which may incur significant latency, thus, extending the time-to-market for the SoC. For example, the password checking statements in lines 6 and 7 of the code example presented in Listing 1 exhibit a security vulnerability. This vulnerability prevents the password checking function from granting access even when the correct password is entered. Existing RTL security verification methodologies, mentioned before, can detect this vulnerability. However, none of them can identify which lines in the RTL code result in this vulnerability. Therefore, this would require a manual inspection of the RTL-code of around 400 lines in order to identify the security vulnerability. Although post-silicon debug techniques [12]–[15] can be used to identify the buggy RTL locations, they: (1) incur higher overhead due to addition of design-for-debug components, (2) lead to respin, which extends the time-to-market.

```
1  `STATE_run_test_idle: begin
2      if(tms_pad_i && (passchk))
3          next_TAP_state=`STATE_select_dr_scan;
4      else begin
5          next_TAP_state = `STATE_run_test_idle;
6          if(correct >= 32'h0001_FFFF)
7              passchk = 1;
8          else if(tdi_o == pass[bitindex]) begin
9              correct++;
10             bitindex++; end end end
```

Listing 1: Password checking bug.

Software validation techniques such as slicing, spectrum, program state analysis, data-mining *etc.* offer localization of bugs in program code [16]–[18]. For example, one of the most prominent approaches, spectrum, accomplishes this by tracking executed and non-executed lines of code in each program run for a given input and subjecting this data to statistical analysis. However, such software bug localization approaches cannot be directly extended to encompass RTL security bugs. This is because, unlike sequential software code execution, RTL codes can also be executed in a concurrent fashion, making it more complex to trace. To the best of our knowledge, there is no available technique to directly associate

This work is supported by NSF grants #2223046, CCF-2047682, CCF-2008905, NSF graduate research fellowship program and Eugene McDermott Graduate Fellowship 202006.

the outcomes of each simulation run with individual statements of the RTL code, making it extremely difficult to localize the security vulnerabilities, once a violation is detected.

To this end, in this paper, we, for the first time, introduce RTL-Spec, a novel framework designed for precise identification of security vulnerabilities in an SoC design, directly in its pre-silicon phase. Our proposed approach consists of two main steps: slicing and spectrum analysis. Slicing is used to reduce the search space of the hardware design by activating relevant paths or states. Spectrum analysis is used to rank the suspiciousness of statements based on the trust scores of input nets, which are derived from test cases. The approach also uses a "z-score" method to distinguish between incorrect and missing logic. The approach introduces novel features that distinguish it from software bug localization techniques, such as handling of concurrent execution of RTL-code, computing and normalizing suspiciousness scores, and detecting both incorrect and missing logic. When evaluated using a buggy RISC V-based PULPissimo SoC provided at the "Hack@DAC2018" competition [19], RTL-Spec was able to successfully detect the RTL statements responsible for 14 security vulnerabilities. Our primary contributions can be summarized as follows:

- We present RTL-Spec, a novel bug localization framework, that can identify buggy lines of RTL code that cause security vulnerabilities in an SoC. To the best of our knowledge, this is the first ever framework to diagnose the manifestation of security bugs in pre-silicon phase.
- We introduce slicing in the control flow of the RTL to simulate a portion of the entire SoC. This aids in reducing the overall time required to verify each security property of the entire SoC design.
- We develop a novel RTL-level spectrum analysis, which aids in assigning a score to individual RTL code statements, that indicates the probability of each statement harboring a potential security bug.
- RTL-Spec, when evaluated on the PULPissimo SoC used in the "Hack@DAC2018" competition, revealed that it was able to successfully locate all 14 inserted security vulnerabilities in the designs correctly [19]. We also evaluate the precision with two standard localization metric "Top-k rank" and "precision" to demonstrate RTL-Spec's localization capability.

II. Background and Related Work

This section presents the essential concepts for our proposed methodology. Furthermore, we review the related work in RTL verification and debugging to elucidiate the rationale and inspiration behind our approach.

A. SoC Security Verification

Security verification in SoC designs has emerged as a critical concern within the semiconductor manufacturing industry. Substantial efforts have been dedicated to tackling this issue through extensive research and development endeavors. Concolic testing, a semi-formal verification approach, has recently been developed for hardware security verification, by reducing the symbolic execution search space and preventing state space explosion [8], [20]–[22]. Following symbolic execution, this method runs simulation with concrete inputs to further explore the corner cases. This technique was found to surpass existing state-of-the-art commercial EDA tools like Cadence JasperGold [20]. Another viable approach is assertion-based verification, where logical propositions are employed to define the security properties of the SoC within the code. These propositions are required to remain `TRUE` during specific execution phases for the property to be verified [10], [23]. Recent studies have introduced fuzzing-based verification methods, which hold the potential to cover complex corner cases in hardware designs [7], [24], [25]. Nonetheless, these methodologies do not offer localization of the bugs in RTL code. In order to diagnose manifested security vulnerabilities, manual inspection is required to locate the buggy statements in the code. This manual process can introduce delays in meeting time-to-market goals and may introduce human errors. Therefore, it is imperative to develop a methodology capable of automating bug localization, to improve the efficiency of secure SoC development.

B. Fault localization in Software

Fault localization, in the software domain, refers to the process of identifying a set of statements in a code that may cause the program to fail. Fault localization techniques have evolved with the increasing size and complexity of software programs. This led to the development of several powerful and effective localization approaches that automate the fault localization process. Among these techniques, two of the most popular software fault localization approaches are *slicing*- and *spectrum*-based methods.

1) Slicing-based fault localization: In this localization technique, a slice of a program is taken with regards to a variable on a given statement. A forward slice from that variable includes all of the statements that the variable may/must affect. A backward slice from that variable includes all of the statements that may/must affect the variable. Whether or not we use a may- or a must-relation is dependent upon whether we take a static or dynamic slice, respectively. Slicing is particularly useful to limit the code that a developer needs to inspect [17], [26]–[28]. This approach also contributes to expediting the validation process by selectively verifying a segment of the program, rather than its entirety. Slicing techniques have been incorporated into hardware design verification methodologies primarily with the aim of test case generation and memory conservation [9], [29]. This is achieved by selecting paths pertaining to properties for verification within the control flow of the hardware design and constraining the exploration of states within these sliced paths.

2) Spectrum-based fault localization: Spectrum-based approaches are powerful techniques that have seen extensive use in software testing and debugging [16]. These techniques collect information about the execution of the program (*e.g.*, the statement execution trace), called *program spectra*. By comparing spectra from passing and failing test cases, the localizer can judge the *suspiciousness* of each statement of the program (*i.e.*, the likelihood that it contains a fault). The executable statement hit spectrum (ESHS) which is used by popular fault localization techniques like Tarantula [18] or Ochiai [30], records the executed statements for analysis [16].

For example, consider a statement s in a program, where $p(s)$ gives the number of passing test cases the statement was executed in, $f(s)$ denotes the number of failing test cases that the statement was executed in, and $total\_passed$ and $total\_failed$ denote the total number of passed and failed test cases, respectively. Tarantula [18], considers the suspiciousness, $Sus(s)$, of a statement s to be:

$$Sus(s) = \frac{\frac{f(s)}{total\_failed}}{\frac{f(s)}{total\_failed} + \frac{p(s)}{total\_passed}} \tag{1}$$

Spectrum-based approaches can assist developers in identifying specific program areas while troubleshooting faults. These methods analyze lines of code and furnish them a 'suspiciousness score', indicating how likely they are to cause a program to crash.

In this paper, we adapt and utilize these principles of slicing and spectrum-based localization techniques in software to develop our novel RTL-Spec framework, that aims to pinpoint buggy lines of code in an RTL design. By assigning suspiciousness scores to statements within an RTL design, RTL-Spec enables the localization of security vulnerabilities during the pre-silicon phase of SoC design.

III. Proposed RTL-Spec Methodology

This section introduces RTL-Spec, a novel framework that uses slicing and spectrum analysis to evaluate the security of an RTL design and detect any bugs in the RTL code due to property violations. We first explain the terms required for our approach. Next, we give a summary of RTL-Spec, followed by detailed description of its modules.

A. Definitions

a) Nets: A *net* is any variable in a design that can be assigned a value. Examples include wires, registers, logics, and bit type variables in the RTL code.

b) Guard Nets and Conditions: In order to execute a statement in RTL, a condition or a set of conditions may need to be satisfied. Those conditions will be regarded as *guard conditions* and the nets controlling these conditions are regarded as *guard nets*. For example, let us assume a conditional statement $\mathcal{C} := n\, binop\, V\ ?\ \mathcal{S}_1 : \mathcal{S}_2$, where *binop* is some binary relation like $=$ or $>$. Assume n is a net and V is either a net or a constant value. n is considered a guard net, as is V if it is a net.

c) Coverage: In a module, each net has an associated area of influence, which we refer to as its coverage. For assignment statements like $\mathcal{S}_1 := n \leftarrow V$ or $\mathcal{S}_2 := A \leftarrow n$, we consider the statement "covered" by net n. In the case of a conditional statement $\mathcal{C} := n\, binop\, V\ ?\ \mathcal{S}_1 : \mathcal{S}_2$ both S_1 and S_2 covered by the guard net n, and by V if V is a net.

d) Dependency Graph of a Net: If there exists a statement, such that $\mathrm{S} := n \leftarrow f(v_0, v_1, ..., v_m)$, where each argument $v_i | 1 \leq i \leq m$ is either a net or a constant, net n will be defined as dependent upon every v_i that is a net. Moreover, n is dependent on both guard and assignment nets.

e) Trust Score and Suspiciousness Score: In this study, two metrics are used to localize bugs in the RTL code. The "trust score" is a metric that represents the probability of a net or code statement being free from or unrelated to the violation. The "suspiciousness score" is a metric indicating the likelihood of a net or statement causing a security violation. These scores are calculated by running a statistical analysis on the "passed" and "failed" test cases acquired from the simulation results. Trust score of each input pin in an RTL design is calculated using Equation 2.

$$Trust\ Score = 1 - \frac{\frac{f(s)}{totalfailed}}{\frac{f(s)}{totalfailed} + \frac{p(s)}{totalfailed}} \tag{2}$$

These calculations are elaborated in detail in Section III-B3b.

f) Input Coverage: In spectrum-based fault localization, the suspiciousness score of a statement is determined through a combination of coverage information and test results. Unlike software, hardware statements are not executed line-by-line, thus analogous statement coverage is not directly applicable. Instead, statements can be "executed" (*i.e.*, activated) depending on inputs. Therefore, we determine the coverage of a statement based on input patterns. Assuming $\mathcal{I}_i$ is a subset to the set of all input nets $\mathcal{I}$ for the design, the statements $\mathcal{S}_i$ that are governed by $\mathcal{I}_i$ will be regarded as covered by $\mathcal{I}_i$. We discuss this in detail in Section III-B1a.

B. RTL-Spec Framework

Figure 1 provides an overview of the RTL-Spec's workflow, which is divided into five key sections. The first section encompasses RTL design and security properties (cross-hatched background). Next, the slicing design for simulation is detailed in Section III-B1 (hatched background). Section III-B2 covers simulation and test case generation. In Section III-B3, we explore spectrum analysis, which identifies suspicious lines in the code (dashed hatched background). Finally, the output of RTL-Spec is represented by the dotted background, producing a scored CFG. Each CFG statement is assigned a suspiciousness score, indicating the likelihood of containing a bug.

Furthermore, RTL-Spec is developed in such a way that verification tools like Cadence JasperGold, Synopsys verification tools, Coppelia, or RTL-Contest can be employed to generate test cases for running spectrum analysis [3], [20], [31], [32]. From Figure 1, the "Simulation Path Selection & Testbench Generation" and "Simulation & Testcase Generation" blocks can be replaced by aforementioned verification tools. If a violation is detected, RTL-Spec runs spectrum analysis using the test cases generated from the verification process. Both simulation or emulation can be used to generate test cases and run spectrum analysis to identify buggy statements in the RTL code. This study utilizes slicing to extract pathways from the CFG of the RTL design, minimizing unnecessary path exploration during simulation. RTL-Spec assumes that security properties are available, and the bugs have an impact on the property. This helps classify the simulation results as passed or failed test cases.

1) Slicing Design to Optimize Simulation: In order to generate test cases and construct restrictions for optimizing simulation, RTL-Spec requires a CFG. We generate CFG

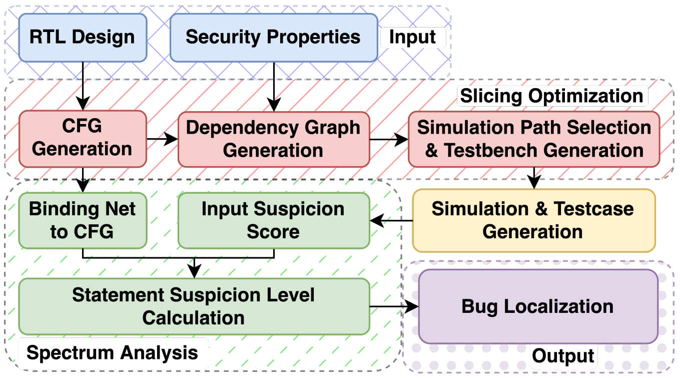


Fig. 1: Flowchart of RTL-Spec.

using our in-house CFG generator which shares similarities with *Goldmine* [33] but is customized specifically to simplify the process of slicing. The CFG is required for two distinct objectives. First, we create a dependency graph of nets that determines the effect of an input pin on the nets in the design using CFG. Second, from the CFG, RTL-Spec performs slicing in order to extract relevant simulation paths that can lead to the violation of the security property to verify. These paths are employed to formulate simulation constraints, which streamline the test case generation process. The generated test cases simulate design behavior along the selected paths. By analyzing the output, the dependency graph aids in identifying input pins with a higher likelihood of triggering security violations. This process is represented with blocks in dashed background, named "Slicing Optimization", which facilitates directed test generation to detect vulnerabilities. Setting up simulation path requires branch realignment or analysis on distributed and concurrent systems as shown in [8], [34]. Employing slicing techniques can circumvent these computations, enabling directed input pattern generation that effectively stimulate the necessary paths. We describe the modules involved in this process in the following paragraphs.

```
`timescale 1ns/1ps
module lck_reg(
    input [15:0] D_in,
    input clk, nrst, wr_q, wr_ack, lck, debug,
    wr_val,
    output wr_req,
    output reg wr_done,
    output reg [15:0] D_out);
    wire unlocked, wr_en;
    assign unlocked = !lck | debug;
    assign wr_en = wr_val & wr_ack;
    always @(posedge clk or negedge nrst) begin
        if (wr_q == 1) wr_req <= 1;
        else wr_req <= 0;
        if (!nrst) D_out <= 16'h0000;
        else if (wr_en & unlocked) begin
            D_out <= D_in;
            wr_done <= 1;
        end
        else if (!wr_en) D_out <= D_out;
    end
endmodule
```

Listing 2: RTL Example (in SystemVerilog).

a) Dependency Graph Generation: To localize the bugs in the RTL code of a design, we need to evaluate the relation of each statement to the inputs. In order to evaluate this relation, a dependency graph is generated. From the dependency graph, the impact of each input pin on the nets can be inferred. The graph is also used to generate constraints for testbenches (Section III-B2). This module of RTL-Spec is shown in Figure 1 with the block named "Dependency Graph Generation". The dependency graph is constructed using the CFG of the design. Before discussing the dependency graph generation process, an example of an RTL code for a "locked register" is illustrated in Listing 2, with its corresponding CFG illustrated in Listing 3.

In Listing 3, each line of the CFG is divided into four parts, separated by a double colon (::). The first part indicates the module name of the design. The second part shows the branch information of each statement. If the branches of CFG are independent of each other, the branch information will be enumerated, such as lines 1 to 3 in Listing 3. When the branch is dependent on the guard condition, a comma (,) is used to illustrate a new layer of branches. For example, lines 4 to 14 are dependent on the guard condition at line 3. The second number of the branch information increments when a guard condition of the same layer is introduced as seen in lines 9 and 10. Assignment-type statements under a guard condition do not contribute to the branch information. Since this CFG is primarily used to generate testbenches and does not affect the simulation, we consider the conversion of "posedge clk" to "clk == 1" and "negedge nrst" to "nrst == 0" as valid.

```
1  lck_reg :: 0 :: unlocked <== !lck | debug :: A;
2  lck_reg :: 1 :: wr_en <== wr_val & wr_ack :: A;
3  lck_reg :: 2 :: clk == 1 or nrst == 0:: Alws;
4  lck_reg :: 2, 0 :: wr_q == 1 :: C;
5  lck_reg :: 2, 0 :: wr_req <== 1 :: A;
6  lck_reg :: 2, 1 :: ++++ELSE++++ :: C;
7  lck_reg :: 2, 1 :: wr_req <== 0 :: A;
8  lck_reg :: 2, 2 :: !nrst == 1 :: C;
9  lck_reg :: 2, 2 :: D_out <== 16'h0000 :: A;
10 lck_reg :: 2, 3 :: wr_en & unlocked == 1 :: C;
11 lck_reg :: 2, 3 :: D_out <== D_in :: A;
12 lck_reg :: 2, 3 :: wr_done <== 1 :: A;
13 lck_reg :: 2, 4 :: !wr_en == 1 :: C;
14 lck_reg :: 2, 4 :: D_out <== D_out :: A;
```

Listing 3: Example CFG.

Once the CFG is developed, we can proceed to generate the dependency graph. To generate the dependency graph first we take the security property shown in Listing 4. This property defines the secure write operation of the RTL code shown in Listing 2. The property requires that when the "*lck*" flag is set to `HIGH`, the "*D_out*" flag should remain stable. The proposed bug localization assumes the presence of properties that might not directly correlate with the specific network elements of the bug. However, these properties must be influenced by the bug for the classification of test cases as passed or failed.

```
property lock_status_chck;
   @(posedge clk) disable iff(!nrst)
   lck |-> $stable(D_out);
endproperty
```

Listing 4: Security property for verification.

Algorithm 1 is employed to construct the dependency graph using the CFG and the property. This approach facilitates the construction of dependency graphs without synthesis, as opposed to relying on fan-in and fan-out. Algorithm 1 takes the RTL code $\mathcal{M}$ and a property $\mathcal{P}$ as input. From $\mathcal{M}$, the CFG is generated (line 1). The RTL code is parsed to retrieve the set of input nets $\mathcal{I}_M$ (line 2). Each relevant net n that defines the properties is collected to form the set of property nets $\mathcal{N}_p$ (line 3). In Listing 4, the relevant nets in properties are

Algorithm 1 Dependency Graph Generation.

```
Input: $\mathcal{M}$, $\mathcal{P}$;
Output: $\mathcal{G}_{depndncy}$, $\mathcal{I}_C$;

1:  $\mathcal{CFG}_{\mathcal{M}} \leftarrow Gen\_CFG(\mathcal{M})$
2:  $\mathcal{I}_M \leftarrow Construct(\mathcal{M})$
3:  $\mathcal{N}_p \xleftarrow{append}$ all $n$ from $\mathcal{P}$
4:  $\mathcal{N}_{curr} \leftarrow \mathcal{N}_p$
5:  while $\mathcal{N}_{curr} \not\subset \mathcal{I}_M$ do
6:      for each $n$ in $\mathcal{N}_{curr}$ do
7:          from $\mathcal{CFG}$:
8:          for all assignments to n do
9:              $\mathcal{N}_n.append(n_{assgnd})$
10:             $\mathcal{N}_n.append(n_{guard})$
11:         end for
12:         $Construct\ \mathcal{G}_n(\mathcal{N}_n)$
13:         $\mathcal{N}_{curr} \leftarrow \mathcal{N}_n$
14:     end for
15:     $\mathcal{G}_{depndncy} \leftarrow$ combine $and$ add$(all\ \mathcal{G}_n)$
16: end while
17: $\mathcal{I}_C \leftarrow \mathcal{N}_{curr}$
18: return $\mathcal{G}_{depndncy}$, $\mathcal{I}_C$
```

Algorithm 2 Test Case Generation and Testing.

```
Input: $\mathcal{M}$, $\mathcal{I}_C$, $\mathcal{U}_M$, $\mathcal{G}_{depndncy}$, $\mathcal{P}_{inf}$
Output: $\mathcal{W}_C$;

1:  $\mathcal{I}'_C \leftarrow (\mathcal{U}_M - \mathcal{I}_C)$
2:  $\mathcal{C}_{wI} \leftarrow$ create_class$(\mathcal{U}_M)$
3:  $\mathcal{C}_{wI} \leftarrow$ Constraints_apply$(\mathcal{I}_C, \mathcal{P}_{inf})$
4:  $\mathcal{C}_{wI} \leftarrow$ randomize$(\mathcal{I}'_C)$
5:  generate $\mathcal{TB}$
6:  $\mathcal{M} \xleftarrow{simulation} \mathcal{TB}$
7:  collect $\mathcal{S}_{data}$
8:  $\mathcal{S}_{data} \leftarrow$ purge_data$(\mathcal{I}'_C)$
9:  label $\mathcal{S}_{data}$
10: each $\mathcal{I}_C \xleftarrow{assign} w$
11: construct $\mathcal{W}_I$
12: $\mathcal{W}_C \xleftarrow{\mathcal{G}_{depndncy}} \mathcal{W}_I$
13: return $\mathcal{W}_C$
```

'*lck*', '*D_out*', and '*nrst*'. '*clk*' is excluded from the relevant nets list since clock signals have a consistent role and are independent across all RTL designs. Issues related to clock signals, including clock trees, are detected through their impact on other nets. These nodes are represented by $\mathcal{N}_{curr}$ (line 4). They are used as starting nodes, and the dependency graph of the module can be constructed with the CFG of Listing 2.

A dependency graph is constructed for the statements that are related to $\mathcal{N}_{curr}$. For each statement, N_n is augmented with its guard and assignment nets (lines 8-11). However, if n is an input net, it is also appended to N_n. This concludes the first level of a dependency graph (line 13). A level is defined by the current set of nodes and the nets they are directly dependent on. After constructing the first level, the next level is obtained that reflects the updated '*current set of nodes*' (line 6). This process will continue until $\mathcal{N}_{curr} \subset \mathcal{I}_{\mathcal{M}}$ (line 5), and the final dependency graph $\mathcal{G}_{depndncy}$ can be obtained (line 15). The final value of $\mathcal{N}_{curr}$ only contains the input nets for stimulating the paths related to the given security property set, $\mathcal{P}$. Therefore, the value of N_{curr} is copied into the input set governing the nets related to the security property, $\mathcal{I}_C$ (line 17). $\mathcal{I}_C$ will be used for generating restriction for simulation testbenches and spectrum analysis to verify the security property set $\mathcal{P}$. The dependency graph, $\mathcal{G}_{depndncy}$, will be used to calculate the trust score of all the intermediate nets in the design, thereby all the statements in the RTL code. Moreover, the processing of input signals in Algorithm 1 varies depending on the design and security considerations, resulting in the reduction of input pins and sliced paths during bug detection through simulations in the majority of instances. We elaborate this in Section III-B2 and Section III-B3a.

b) Slicing Simulation Path: In order to generate test cases for spectrum analysis and streamlining simulations, RTL-Spec employs a technique known as *path slicing* on the CFG. Path slicing defines a focused path for the simulation, inhibiting excessive route exploration, thus rendering the basis for creating testbenches. Additionally, specific constraints are established using $\mathcal{P}$ to guide the generation of test patterns, ensuring that the simulation concentrates solely on elements within this sliced path. To provide a clearer understanding of path slicing, an example from Section III-B1a will be used.

The property in Listing 4 specifies the characteristics of the '*D_out*' net. Therefore, all statements that assign to '*D_out*' must be verified and the property is only valid when '*nrst*' equals 1. As a result, our slicing criteria should include all lines that assign to '*D_out*' while '*nrst*' is set to `HIGH` (*i.e.*, lines 11 and 14 in Listing 2).

In order to generate the simulation path, the assignment statement on line 11 of Listing 3 is used as a starting point, which has a branch of "2,2". Here the rightmost "2" indicates that that the statement belongs to the conditional block labeled as "2". The preceding statement that belongs to the same branch of "2,2" is searched. It is either a condition statement 'C' or an always block 'Alws'. In this case, line 10 is the preceding statement. Subsequently, the previous condition statement with a branch of "2" is looked for and the next preceding statement is found on line 3. We repeat this process until we reach a statement of type 'Alws'. Since line 3 is an 'Alws' type statement, we have completed the path generation to reach line 11 (Listing 3). Therefore, the final sliced path is achieved and shown in Listing 5.

```
3  lck_reg :: 2 :: clk == 1 or nrst == 0:: Alws;
10 lck_reg :: 2, 2 :: wr_en & unlocked == 1 :: C;
11 lck_reg :: 2, 2 :: D_out <== D_in :: A;
```

Listing 5: Final sliced Path for Listing 2.

With the sliced path, we acquire the condition and net value required to reach the assignment statement line 11. Therefore, simulation constraints can be created to generate input patterns and directly verify the sliced path. The constraints have to alter the input values for '*wr_en*' and '*unlocked*', while keeping the value of '*lck*' to `HIGH` and thereby check if for any case a value can be assigned to '*D_out*'.

2) Simulating Test Cases based on the Input Coverage: RTL-Spec uses simulation to verify whether the RTL code satisfies the defined property and generate output dataset that will be used as test cases to run spectrum analysis. The test cases are categorized as "*passed*" or "*failed*". If there exists no failed cases, the RTL design is considered as secure and no further analysis is needed. However, if there are failed case(s), the following modules of RTL-Spec, discussed in Section III-B3, are employed to localize to buggy statement. In order to reduce the test cases needed, RTL-Spec generates the input patterns using the dependency graph $\mathcal{G}_{depndncy}$ and simulation test cases. RTL-Spec generates a testbench that only excites the sliced paths by setting input patterns for the statements under verification. Only the inputs that directly affect the path will be modified, while the other inputs will possess random values. Once the simulations are completed, the results are labeled showing whether the input patterns lead to a successful or failed run.

Algorithm 2 shows the process flows of optimizing the testbench and launching a controlled simulation. It utilizes the RTL design $\mathcal{M}$, inputs covering the property $\mathcal{I}_C$, and all of the inputs $\mathcal{U}_M$ to simulate the sliced paths $\mathcal{P}_{inf}$. We identify the input nets $\mathcal{I}'_C$ by creating a subset of $\mathcal{U}_M$ excluding $\mathcal{I}_C$ (line 1). An interface, $\mathcal{IF}_M$, is constructed for the testbench to connect to the design under test (DUT), and a class, $\mathcal{C}_{wI}$, is generated which assigns input patterns and restrictions for simulation (lines 2-4). For example, a class, shown in Listing 6, can be generated for the design in Listing 2. For example, in this class, input nets '*wr_q*' and '*D_in*' are not relevant to verify the property. It should be noted that "*D_in*" is not a control net, and the probability of it being 0 is very low. Even if it does occur, it has no notable impact on the localization result; therefore the assumption remains valid. Hence, their values will be assigned random bits. The test sequence contains the input pattern for '*nrst*', '*wr_ack*', '*wr_val*', '*lck*', and '*debug*' respectively from least significant bit (LSB) to most significant bit (MSB). Since the property is disabled when "$nrst == 0$", we create a constraint where "$nrst = 1$".

```
`define test_seq_len 5
class io_seq_gen;
    rand bit wr_q;
    rand bit [31:0] D_in;
    randc bit [`test_seq_len - 1 : 0] seq_in;
    constraint reset2zero {seq_in[0] == 1;}
endclass //io_seq_gen
```

Listing 6: Sequence Generator Class for Simulation.

After setting up the constraints and rules in $\mathcal{C}_{wI}$, we generate the testbench and launch the simulation with the testbench (lines 5-6). We collect the simulation results and construct a table $\mathcal{S}_{data}$ which only contains test cases of $\mathcal{I}_C$ (lines 7-8). Each of the test cases is labeled as "*Passed*" or "*Failed*" based on whether the property, $\mathcal{P}$ under verification is satisfied (line 9). Each input of $\mathcal{I}_C$ will be assigned a trust score (defined in Section III-A), w, and a vector $\mathcal{W}_I$ will be created, representing the trust score of $\mathcal{I}_C$ (lines 10-11). From $\mathcal{W}_I$, we use $\mathcal{G}_{depndncy}$ to construct another vector, $\mathcal{W}_C$, that represents trust score of all the nets in $\mathcal{M}$. The vector W_C contains trust scores for each net in the design. Initially, trust scores are set through continuous assignments, as shown in line 9 of Listing 2 of the paper (*e.g.*, "unlocked" is assigned the value "$!lck \mid debug$," resulting in an initial trust score of $\frac{1+0}{2} = 0.5$). Nets without such assignments start with a trust score of 1.

3) Executing Spectrum Analysis: Once the slicing, simulation of the sliced path, and labeling of test cases are completed, spectrum analysis is performed on the RTL code. The execution of spectrum analysis entails the creation of vectors to bind the nets to the CFG, followed by the computation of suspiciousness scores and bug localization. RTL-Spec uses this process to rank the statements in the RTL code based on their probability to cause the violation.

a) Binding Nets to CFG: RTL-Spec binds the nets to the CFG statements using a binding vector as shown with the block named "Binding Net to CFG" in Figure 1. This binding vector represents the correlation of each net with each statements in the RTL codes. This enables RTL-Spec to establish a correlation between input patterns and RTL code statements. Furthermore, the binding vector helps in calculating statement trust scores efficiently.

The binding vectors represent a numerical value for each statement in the CFG. From the CFG, we generate these vectors and use the dependency graph, $\mathcal{G}_{depndncy}$ to update each vector during the calculation. The trust score of a net is the average of the input nets in the final level of $\mathcal{G}_{depndncy}$. For example, the input nets in the final level of the dependency graph consist of '*lck*' and '*debug*' for '*unlocked*'. The trust score of '*unlocked*' is set to 0.5 (average trust score of '*lck*'$= 1$ and '*debug*'$= 0$). The vectors consist of three elements: the previous cumulative trust scores, the datatype values, and the net roles. These elements capture the trustworthiness of each statement based on the control flow.

To illustrate this, we use the example in Listing 2. Table I shows the binding vectors for line 1 and line 10 from Listing 3 in the second and third rows, respectively. The fourth row shows the weight vector of the nets, $\mathcal{W}_C$. The vector for each statement has three parts: the first part is the cumulative trust score of the previous conditions in the CFG; the second part is the type of statement ('0' for assignment and '1' for condition); and the third part is the net roles in the statement. The net roles are encoded as '0', '1', or '2', where '0' means the net is absent, '1' means the net contains the assigned value, and '2' means the net is being assigned. This representation can facilitate the calculation of the trust score of each statement. When we create the binding vector of the CFG, we initialize all the statements with a trust score of 1. As we traverse through the CFG statements, we update the cumulative trust score of each statement based on control flow. To update the cumulative trust score, we multiply the statement's current

TABLE I: Binding Vector of Statements from the CFG and Weight vector of Net Trust Scores.

	Cumulative Probability	**Statement Type**	**clk**	**nrst**	**wr_q**	**wr_ack**	**lck**	**debug**	**wr_val**	**wr_req**	**wr_done**	**unlocked**	**wr_en**	**D_in**	**D_out**
Line 1	1	0	0	0	0	0	1	1	0	0	0	2	0	0	0
Line 10	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0
$\mathcal{W}_C$			1	1	1	1	1	0	1	1	0.75	0.5	1	0.83	1

Algorithm 3 Suspicion Score Calculation.

```
Input: $\mathcal{W}_C$, $\mathcal{CFG}_M$, $\mathcal{CFG}_V$
Output: $\mathcal{CFG}_w$;
1: for all $\mathcal{S}t_V$ in $\mathcal{CFG}_V$ do
2:     $\mathcal{S}t_{Vl} \leftarrow \mathcal{S}t_V[2::]$
3:     $\mathcal{S}t_{Vl} \leftarrow \text{dec2bin}(\mathcal{S}t_{Vl})$
4:     $\mathcal{S}t_{Vm} \leftarrow MSB(\mathcal{S}t_{Vl})$
5:     $\mathcal{S}t_{Vl} \leftarrow LSB(\mathcal{S}t_{Vl})$
6:     $\mathcal{S}t_V[0] \leftarrow \mathcal{S}t_{ch}$
7:     $\mathcal{S}t_{ts} = \mathcal{S}t_V[0] \times \frac{1}{n} \times \sum(\mathcal{S}t_{Vl} \times \mathcal{W}_C)$
8:     if($\mathcal{S}t_V[1] == 1$) $\mathcal{S}t_{ch} \leftarrow \mathcal{S}t_{ts}$
9: end for
10: return $\mathcal{CFG}_w$
```

cumulative trust score with the corresponding guard condition (defined in Section III-A). If no guard condition is present, the cumulative trust score is multiplied by one per initialization. Next, we use the binding vector of the entire CFG and the weight vector ($\mathcal{W}_C$) to calculate the trust score of each statement. We take the LSB of the net values from the binding vector to create a sub-vector and multiply it with $\mathcal{W}_C$. This yields the total trust score for the statement. Next this score is assigned to the net possessing the MSB with a value of 1 in the binding vector. Finally, the vectorized CFG, $\mathcal{CFG}_V$ is returned, incorporating the updated trust scores.

b) Calculating Suspiciousness Level in the CFG Statements: This module of RTL-Spec is responsible for calculating suspiciousness of the statements in the CFG. We compute the suspiciousness of the statements in the CFG to localize the buggy statements using Algorithm 3.

The trust score vector, $\mathcal{W}_C$, $\mathcal{CFG}_\mathcal{M}$ and $\mathcal{CFG}_V$ are taken as inputs for this algorithm. Each of the vectors in $\mathcal{CFG}$ is represented as $\mathcal{S}t_V$. For all vectors in $\mathcal{CFG}_V$, a sub-vector, $\mathcal{S}t_{Vl}$, is created by taking every element from the third element (line 2). The values of $\mathcal{S}t_{Vl}$ are converted to binary (line 3). Two vectors $\mathcal{S}t_{Vm}$ and $\mathcal{S}t_{Vl}$ are created by taking the MSB and LSB, respectively (lines 4-5). Equation 3 is used to calculate the trust score of the statements (line 7):

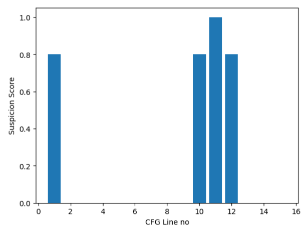
$$\mathcal{S}t_{ts} = \mathcal{S}t_V[0] \times {}^{1}\!/_{n} \times \sum(\mathcal{S}t_{Vl} \times \mathcal{W}_C) \tag{3}$$

Here, n is the number of 1's in $\mathcal{S}t_{Vl}$ and $\mathcal{S}t_V[0]$ is the cumulative trust score. We update the $\mathcal{S}t_{ch}$ with $\mathcal{S}t_{ts}$ (line 8 in Algorithm 3). We append the trust score, $\mathcal{S}t_{ts}$ to the statements in $\mathcal{CFG}_M$. For example, the suspiciousness score calculation for line 1 in Listing 3 is as follows:

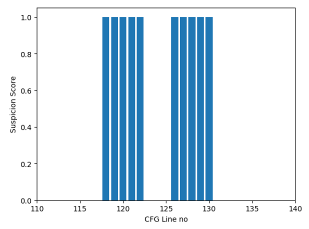
$$\begin{aligned}\mathcal{S}t_{ts} &= 1 \times \frac{1}{2} \times \sum([0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0] \\ &\qquad \times [1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1]) \\ \mathcal{S}t_{ts} &= \frac{1}{2} \times \sum([0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]) \\ \mathcal{S}t_{ts} &= \frac{1}{2} \times 1 = 0.5\end{aligned} \tag{4}$$

Here, $\mathcal{S}t_{Vl}$ is taken from the first row of Table I.

c) Bug Localization: Once calculation of the trust score for each statement in the CFG is accomplished, suspiciousness scores are assigned to the statements. These scores are analyzed in order to localize the bugs. Negative log of the trust scores are calculated using the following equation:



(a) Incorrect statement.



(b) Missing statement.

Fig. 2: Bug localization.

$$\mathcal{S}t_l = -log(\mathcal{S}t_{ts}) \tag{5}$$

Logarithmic scales are employed to expand the measurement range of lower trust scores. With the assignment of the suspiciousness score to each statement, the location of the bug can be ascertained through analysis of the score distribution across all statements. Bugs are classified into two categories, whether they are caused by an *incorrect* or a *missing* statement.

```
1  lck_reg :: 0 :: unlocked <== !lck | debug :: A;  0.8
2  lck_reg :: 1 :: wr_en <== wr_val & wr_ack :: A;  0
3  lck_reg :: 2 :: clk == 1 or nrst == 0:: Alws;  0
4  lck_reg :: 2, 0 :: wr_q == 1 :: C;  0
5  lck_reg :: 2, 0 :: wr_req <== 1 :: A;  0
6  lck_reg :: 2, 1 :: ++++ELSE++++ :: C;  0
7  lck_reg :: 2, 1 :: wr_req <== 0 :: A;  0
8  lck_reg :: 2, 1 :: !nrst == 1 :: C;  0
9  lck_reg :: 2, 1 :: D_out <== 16'h0000 :: A;  0
10 lck_reg :: 2, 2 :: wr_en & unlocked == 1 :: C;  0.8
11 lck_reg :: 2, 2 :: D_out <== D_in :: A;  1
12 lck_reg :: 2, 2 :: wr_done <== 1 :: A;  0.8
13 lck_reg :: 2, 3 :: !wr_en == 1 :: C;  0
14 lck_reg :: 2, 3 :: D_out <== D_out :: A;  0
```

Listing 7: Scored CFG.

Once $\mathcal{S}t_l$ for all statements are obtained, the cause of the bug can be determined by examining the score distribution for all statements, as depicted in Figure 2. Figure 2a illustrates the suspiciousness score distribution of the CFG in Listing 7. Listing 7 represents the scored version of Listing 3. The process of determining whether a security breach occurred due to a missing or incorrect statement involves calculating the "z-score" of the suspiciousness score for each statement [35]. In statistical analysis, z-score is calculated using Equation 6.

$$z = \frac{x - \mu}{\sigma} \tag{6}$$

Here, 'x' is the suspiciousness score of a statement. μ denotes the average suspiciousness score and σ is the standard deviation for the suspiciousness score of the corresponding block the statement is located in. In case of a missing statement, for every statement $x = \mu$, thereby making the z-score for every statement would be 0. However, if there exists a buggy statement, every other statement's z-score would be a negative value and only the buggy statements would have a positive z-score value. For example, in Listing 7, z-score for line 11 is 14.94 and for the other statements the z-score is -7.53. Therefore, the presence of a positive z-score makes the bug an "Incorrect statement" type bug. In case of Bug # IV from Table II, all the statements in the suspected block had an z-score of 0, thereby making it a "Missing statement" type bug and the suspiciousness score distribution is shown in Figure 2b. Henceforth, in this paper, the "Incorrect statement" and the "Missing statement" type bugs will be referred to as "I" type and "M" type, respectively.

IV. Evaluation and Results

This section presents the experimental evaluation for RTL-Spec. We outline the experimental setup and evaluation metric used to assess RTL-Spec followed by concise bug explanations and analysis of RTL-Spec's performance.

A. *Experimental Setup and Evaluation Metrics*

RTL-Spec (developed in Python 3.9, and uses NumPy and Pandas libraries) computes the suspiciousness score of RTL code statements using statistical analysis. A flawed version of PULPissimo SoC for the Hack@DAC2018 competition was used to evaluate RTL-Spec 's bug detection and localization capabilities. This benchmark represents a simplified industrial SoC configuration with common security flaws and can be generalized to other SoC benchmarks.

In order to evaluate the effectiveness of the localization technique in RTL-Spec framework, we introduce a commonly used metric for assessing software bug localization techniques, the top-k hit metric. It measures the percentage of true faulty statements that are within the top-k results ranked by suspiciousness score [36]. To assess the performance of RTL-Spec, we also measure precision, which indicates the percentage of ranked statements that are true positives (i.e., responsible for the fault). Any statement not responsible for the fault but that is ranked by the fault localizer is a false positive. Precision is formally defined as follows:

$$\text{Precision} = \frac{\#TP}{\#TP + \#FP} \tag{7}$$

In Equation 7, true positives (TP) and false positives (FP) refer to the statements that are correctly and incorrectly detected by RTL-Spec in the RTL code respectively. In scenarios where bugs entail missing statements, the calculations of TP and FP are conducted at the block level. TPs are evaluated with assertion mutation model, adopted from [37]. A TP is an occurrence that directly causes a property violation, or cause subsequent operations to violate property, or lack critical statements leading to a property violation. As discussed in Section III-B3c, the blocks are employed as the units to rank suspiciousness, such that a true positive block represents the block where the missing statement was supposed to be located.

B. *Analysis of Security Properties*

This section presents a detailed analysis of the bugs used to evaluate RTL-Spec's localization potential. Traditional tools were adept at identifying the flawed IPs, and the precision assessment was contingent upon the codebase unique to each IP with a bug. We examine the characteristics of each bug, including its effects on system functionality and security, and elucidate our methods for identifying them in the source code.

1) Bug I: In this bug, shown in Listing 2, the locked register can be written if it is in debug mode. RTL-Spec was able to reduce the number of simulation test cases from 2^6 to 2^4 by only controlling four out of six relevant input pins. By running spectrum analysis on these test cases, RTL-Spec was able to narrow down the probable location of the bug to just four lines (lines 9, 15, 16 and 17) out of 17 lines of RTL code from Listing 2 (excluding the module declarations), with lines 9, 15, and 16 being *TP* and line 17 being *FP*. RTL-Spec classified it as a 'I' type bug.

2) Bug II: This bug is shown in Listing 1. Here, the password-checking function must verify the entire password, allowing access to the advance debug unit (ADU) of the SoC via the JTAG module. The access is granted only if the condition $correct \geq 32'h0001\_FFFF$ is true (line 6). As '$correct$' cannot reach the value of $32'h0001\_FFFF$, due to the password length limitation, the ADU becomes completely inaccessible. RTL-Spec correctly identifies line 6 in Listing 1 as the most suspicious, reducing potential lines to inspect from approximately 300 to one indicating it as an 'I' type bug.

3) Bug III: This bug is also represented in Listing 1. The SoC security property necessitates a password-checking sequence to restart upon a mismatch. However, in Listing 1, the ADU fails to reset its progress for incorrect inputs. This vulnerability enables adversaries to supply incorrect inputs out of order, posing a significant security flaw. RTL-Spec identified the problematic function on lines 4 to 10 in Listing 1, where the expected protocol inconsistency occurred. No peak was detected, indicating a 'M' type bug in the design.

```
1  always @(posedge tck_pad_i or negedge trstn_pad_i)
2  begin if(trstn_pad_i == 0) begin
3    TAP_state = `STATE_test_logic_reset;
4    pass = 32'hDEADBEEF; end
5   else TAP_state = next_TAP_state;
6  ...
7  case(TAP_state)
8  `STATE_test_logic_reset: begin
9      passchk = 0;
10     if(tms_pad_i)
11     next_TAP_state = `STATE_test_logic_reset;
12     else next_TAP_state=`STATE_run_test_idle; end
```

Listing 8: Code snippet for Bugs #IV, #V.

4) Bug IV: Listing 8 shows Bug IV. The SoC security protocol mandates a full reset of all function information registers upon receiving a *reset* signal to restore the module to its default state. In the ADU, despite resetting the FSM state and password, certain registers like '*bitindex*' and '*correct*' remain unchanged. This oversight allows unauthorized access after a reset, as previous password attempts are not cleared. RTL-Spec has identified two blocks of code with 18 suspicious statements and categorized Bug IV as 'M' bugs.

5) Bug V: The ADU's access password is hard-coded in Listing 8 at line 4. RTL-Spec identified a vulnerability between lines 1-5, along with three 'always' blocks sensitive to *clock* and *reset* signals. The '*pass*' net is influenced by only two pins, '*tck_pad_i*' and '*trstn_pad_i*'. The absence of input pins associated with '*pass*' suggests a missing statement. The password, hardcoded in one of four 'always' blocks, was considered equally suspicious by RTL-Spec due to concurrent execution and similar sensitivities. While placing reset logic in any of these blocks yields the same effect, three other detected blocks were wrongly placed in different code locations. Consequently, RTL-Spec classified this issue as an 'M' type bug.

```
assign update_sec = sec_counter == 15'h7FFF;
assign update_min = update_sec & (r_sec == 8'h59);
assign update_h   = update_min & (r_min == 8'h59);
assign update_day   = update_h & (r_h == 6'h23);
...
always @(posedge clk_i or negedge rstn_i) begin
if(!rstn_i) sec_counter <= 'h0;
else begin
    if (clock_update_i)
        sec_counter<={init_sec_cnt_i,5'h0};
```

```
11      else sec_counter <= sec_counter + 1; end end
```

Listing 9: Code snippet for Bug #VI.

6) Bug VI: The time controller module's "rtc_clock" unit performs time calculation using faulty logic, as outlined in Listing 9. The registers for minutes, hours, and days are updated using the values 8'h59, 8'h59, and 6'h23, respectively. However, these values are provided in hexadecimal format instead of decimal, resulting in a flawed clock operation. RTL-Spec highlighted suspicious statements in lines 1, 2, 3, and 4 of Listing 9, leading to the classification of these statements as incorrect. The inputs update the '*r_sec_counter*', which updates the minutes, hours, and days. Therefore, the buggy statements had a lower suspiciousness score than lines 10 and 11. Here, lines 1-4 are *TP* and lines 10, 11 are *FP*. RTL-Spec labels the bug as a 'I' type bug.

```
1 else if (write & !Write_once_status) begin
2     Data_out <= Data_in & 16'hFFFE;
3     Write_once_status <= Data_in[0]; end
```

Listing 10: Code snippet for Bug #VII.

7) Bug VII: The log-in record register is designed as a write-once system, as described in Listing 10. The 'Write_once_status' flag remains inactive until the LSB of 'Data_in' becomes `TRUE` (line 3). Therefore, the tested module was found to write multiple times without activating the 'Write_once_status' flag, indicating an 'I' type bug.

```
1 case (CS)
2     3'h0: state = 2'h5;
3     ...
4     3'h5: state = 2'h1;
5 endcase
```

Listing 11: Code snippet for Bug #VIII.

8) Bug VIII: Listing 11 demonstrates a bug in which incorrect inputs can lead to the FSM entering an illegal state, resulting in unexpected system behavior. The absence of a '*default*' case in the case block causes the FSM to reach an undefined state. The bug is detected and localized in the case block by RTL-Spec. Running inputs that assign values to '*CS*' reveals two cases where the system enters an undefined state and becomes unresponsive. It is classified as a 'M' type bug.

```
1 if(outstanding_trans_i) NS = OPERATIVE;
2 else if (error_gnt_i) NS = OPERATIVE;
3 else NS = ERROR;
```

Listing 12: Code snippet for Bug #IX.

9) Bug IX: The AXI decoder halts in an error state, as illustrated in Listing 12. It transitions '*NS*' to "operative" when there is an outstanding signal, exiting the error state (line 1). RTL-Spec detected the bug using targeted stimulation of '*CS*' and '*NS*'. Only 11 out of 2064 input pins were selected for simulation. By slicing paths to the error state, RTL-Spec reduced the number of test cases to 32. RTL-Spec's spectrum analysis identified line 1 as the most suspicious statement, thus classifying the bug as a 'I' type bug.

```
1 always_ff @(posedge HCLK, negedge HRESETn) begin
2     if(!HRESETn) begin
3         r_gpio_inten    <=  '0;
4         ...
5         r_gpio_lock     <=  '0;
6         for (int i=0;i<32;i++)
7             gpio_padcfg[i]  <=  6'b000010; end
```

Listing 13: Code snippet for Bug #X.

10) Bug X: In this case, the SoC security property dictates that lock controls must not be cleared after a reset. From Listing 13, it can observed that the property is being violated in the GPIO module as the lock control is set to "$1b'0$" following a reset (line 5). The bug is identified and located by RTL-Spec at line 292 of the design (line 5 of Listing 13). It is classified as an 'I' type bug.

```
1 case (State_SP)
2       IDLE: begin
3       ...
4       default : /* default */ ;
5 endcase
```

Listing 14: Code snippet for Bug #XI.

11) Bug XI: In the absence of a default state and incomplete conditions in the FSM cycle, the ALU enters an undefined state, as shown in Listing 14. This introduces a security vulnerability causing the instruction handling to malfunction. RTL-Spec identifies this vulnerability and designates the case statement block in the RTL design of the ALU as highly suspicious marking it as a 'M' type bug.

```
1 always @(posedge clk) begin
2     if(d[0] == 1'b1) c = aes_out;
3     ...
4     if(d[3] == 1'b1) c = temperature_out;
5     else c = 0; end
```

Listing 15: Code snippet for Bugs #XII, #XIII.

12) Bug XII: In Listing 15, it can be observed that the reset functionality is lacking in the cryptographic MUX module, which is responsible for handling the MAC output. Consequently, the module's output remains unchanged during a reset. There is no statement accounting for reset operation, resulting in compromising the integrity of cryptographic modules in the SoC. Owing to the lack of a reset statement, the entirety of the "always" block is designated as a faulty block, creating a security vulnerability. This bug is labeled as a type 'M' bug.

13) Bug XIII: As demonstrated in line 4 of Listing 15, the cryptographic MUX module obtains output data from a temperature sensor instead of the cryptographic blocks. As a result, in this state data obtained from the cryptographic MUX is unencrypted and does not contain any information other than the sensor value. RTL-Spec identified line 4 along with lines 2-3 as faulty statements in the design. In this case, Line 5 had a lower suspiciousness score, therby this bug is categorized this bug as a 'I' type bug.

```
1 always @(posedge tck_i) begin
2   if(idcode_sel & shift_dr)
3     idcode_reg <=  {td_i, idcode_reg[31:1]};
4   else  idcode_reg <=  `IDCODE_VALUE; end
```

Listing 16: Code snippet for Bug #XIV.

14) Bug XIV: The JTAG decoder disregards the LSB when examining the input for the ID code, as evident in line 3 of Listing 16. Due to the faulty logic in line 3, RTL-Spec recognizes this as an 'I' type bug.

C. Performance Evaluation

Table II summarizes the evaluation results of RTL-Spec's localization capabilities using precision and top-k rank metric. It also describes the bugs used in the evaluation, including their numbers, descriptions, CWE categories [38], and types.

TABLE II: Description of Bugs Used for Evaluation.

Bug No.	Bug Description	CWE Category	Bug Type	Rank in Spectrum	True Positives	False Positives	Precision (%)
I	Locked Register can be written in locked state if debug mode is ON.	CWE-1199	I	2nd	3	1	75
II	Password checking can never be passed logic in advanced debug unit.	CWE-1207	I	1st	2	0	100
III	Wrong password input does not clear password check progress.	CWE-1203	M	1st	1	0	100
IV	Reset signal does not clear the number of correct inputs.	CWE-1206	M	1st	2	0	100
V	Password is hard-coded and set on reset.	CWE-1207	M	1st	1	3	25
VI	Faulty logic in the RTC causing inaccurate time calculation.	CWE-1203	I	2nd	5	3	62.5
VII	Registers can be written more than once.	CWE-1199	I	1st	1	0	100
VIII	FSM goes to an undefined state.	CWE-1199	M	1st	1	0	100
IX	AXI Decoder ignores error flag.	CWE-1199	I	1st	1	0	100
X	Reset clears all lock controls in GPIO module.	CWE-1206	I	1st	1	0	100
XI	Incomplete case statements in ALU can cause unpredictable behavior.	CWE-1199	M	1st	1	0	100
XII	Output of MAC is not erased on reset.	CWE-1206	M	1st	1	0	100
XIII	Temperature sensor is muxed with the cryptography modules.	CWE-1199	I	1st	1	3	25
XIV	Instruction ID is not updated correctly.	CWE-1199	I	1st	1	0	100

The bug types are either "I" for incorrect statements or "M" for missing statements. The table shows the rank of the faulty element, the number of TPs, FPs, and the precision of RTL-Spec 's localization for each bug. For instance, Bug #I's faulty statement has the 2nd rank based on its suspiciousness score.

Analysis of the fifth column reveals that 12 out of 14 bugs are situated within the highest-scoring statement or block (the exceptions were Bugs #I and #V). Further examination of the eighth column shows that RTL-Spec achieved 100% precision on 10 out of 14 bugs. However, the results for Bugs #VI and #XIII exhibited low precision scores. In the case of Bug #VI, all "always" blocks obtain identical rankings, since they are solely sensitive to the clock and reset signals. Despite being declared at different locations in the RTL code, these blocks are executed simultaneously under identical sensitivities. This implies that the insertion of the missing statement in Bug #VI within any of these four "always" blocks would yield identical results. Nevertheless, these "always" blocks are situated disparately within the RTL code. In assessing the precision of RTL-Spec in this context, it is considered that three of the other "always" blocks are identified erroneously as false positives. Consequently, the precision of RTL-Spec in pinpointing the exact "always" block is reduced. Regarding Bug #XIII, the nets responsible for selecting the temperature sensor as output are guard nets. Therefore, all four assignments under the guard condition received an equal suspiciousness score, thereby detecting all four statements as the origin of the bug. In this case, as one of the statements in the block has a lower suspiciousness score, the z-score for the other four statements is a non-zero value. Therefore, RTL-Spec identifies this as an 'I' type bug.

During evaluation, the CFG generation time varies between 80-200 ms, depending on the code size (183ms on a codebase of around 1500 lines including comments and declarations). The spectrum analysis, which assigns suspiciousness scores to individual statements, typically takes around 100-150 ms. The overall process took 300-500ms. RTL-Spec used no more than 4GB of memory, and the octa-core processor clock speed remained 3.4GHz utilizing four cores during its run-time. RTL-Spec can handle designs with large number nets by computing suspiciousness scores with vector operations.

The findings suggest that RTL-Spec has the ability to identify erroneous statements or blocks by analyzing the impact of inputs on property violations. Two key factors reduced the precision in the evaluation process. First, we observed similar preceding conditions in the case of Bug #V and #XIII. Second, there were shared input net dependencies across multiple nets, as exemplified by Bug #I and #VI. While the second factor rarely results in severe false positives, the first type is common in RTL designs and significantly impacts precision.

V. Conclusion

This paper introduces RTL-Spec, which for the first time, enables localization of security vulnerabilities within the RTL code of SoC designs. A distinctive feature of RTL-Spec is its ability to correlate inputs and outputs from simulations with individual RTL statements, enabling precise identification of manifested security vulnerabilities. When a security breach occurs, RTL-Spec efficiently identifies the faulty statement(s) in the RTL design. RTL-Spec was evaluated with the Pulpissimo SoC used in "Hack@DAC2018" competition [19]. The RTL-Spec framework accurately identifies all 14 vulnerabilities illustrating how each input pin impacts other components in the design. It achieves 100% precision in 10 out of the 14 cases. RTL-Spec enhances the security verification process by eliminating unnecessary simulation paths and localizing faulty statements which significantly reduces the time and cost of debugging in pre-silicon security verification. In the future, we plan to refine the framework to provide a more precise localization for the missing statements that invoke security vulnerabilities. In particular, we intend to investigate parameters that distinguish between blocks declared in different locations of the RTL code, albeit having similar dependencies. Our future work also involves classifying testcases without property-based verification, which would allow us to detect and locate bugs without requiring properties.

References

[1] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[2] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. K. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs." in *USENIX Security Symposium*, 2019, pp. 213–230.

[3] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 815–827.

[4] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-chip security*. Springer, 2020.

[5] S. L. He, N. H. Roe, E. Wood, N. M. Nachtigal, and J. Helms, "Model of the product development lifecycle." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[6] *Overview: cisco public 1, cisco secure development lifecycle securing cisco technology*. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/cisco-secure-development-lifecycle.pdf

[7] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "{TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3219–3236.

[8] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1538–1543.

[9] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. S. Tupuri, "A hierarchical test generation approach using program slicing techniques on hardware description languages," *Journal of Electronic Testing*, vol. 19, pp. 149–160, 2003.

[10] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Comput. Surv.*, vol. 54, no. 11s, sep 2022. [Online]. Available: https://doi.org/10.1145/3510578

[11] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.

[12] P. Mishra and F. Farahmandi, *Post-Silicon Validation and Debug*. Springer, 2019.

[13] S. Ma, D. Pal, R. Jiang, S. Ray, and S. Vasudevan, "Can't see the forest for the trees: State restoration's limitations in post-silicon trace signal selection," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 1–8.

[14] K. Basu and P. Mishra, "Rats: Restoration-aware trace signal selection for post-silicon validation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 4, pp. 605–613, 2012.

[15] K. Rahmani, S. Ray, and P. Mishra, "Postsilicon trace signal selection using machine learning techniques," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 570–580, 2016.

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[17] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[18] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 273–282. [Online]. Available: https://doi.org/10.1145/1101908.1101949

[19] Hackdac, "Hackdac/hackdac_2018_beta: The soc used for the beta phase of hack@dac 2018." [Online]. Available: https://github.com/hackdac/hackdac_2018_beta/tree/master

[20] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, March 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2021.3066560

[21] Y. Lyu, A. Ahmed, and P. Mishra, "Automated activation of multiple targets in rtl models using concolic testing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 354–359.

[22] Y. Lyu and P. Mishra, "Scalable concolic testing of rtl models," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2020.

[23] S. Gupta, A. John, and M. Kalra, "Assertion based verification using yosys: A case study from nuclear domain," in *Proceedings of the 16th Innovations in Software Engineering Conference*, 2023, pp. 1–5.

[24] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[25] C. Chen, R. Kande, N. Nyugen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "Hypfuzz: Formal-assisted processor fuzzing," *arXiv e-prints*, pp. arXiv–2304, 2023.

[26] M. D. WEISER, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, 1979, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23. [Online]. Available: http://libproxy.utdallas.edu/login?url=https://www.proquest.com/dissertations-theses/program-slices-formal-psychological-practical/docview/302949655/se-2

[27] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 112–122. [Online]. Available: https://doi.org/10.1145/1250734.1250748

[28] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *Journal of Systems and Software*, vol. 79, no. 7, pp. 891–903, 2006, selected papers from the 11th Asia Pacific Software Engineering Conference (APSEC2004). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121205001780

[29] S. Vasudevan, E. A. Emerson, and J. A. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *International Journal on Software Tools for Technology Transfer*, vol. 9, pp. 89–101, 2007.

[30] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[31] [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[32] [Online]. Available: https://www.synopsys.com/verification.html

[33] D. Pal, V. Dodeja, A. S. Kumar, and S. Vasudevan, "Goldmine: A tool for enhancing verification productivity."

[34] M. Chen, P. Mishra, and D. Kalita, "Towards rtl test generation from systemc tlm specifications," in *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE, 2007, pp. 91–96.

[35] Jul 2023. [Online]. Available: https://en.wikipedia.org/wiki/Standard_score

[36] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1089–1113, 2019.

[37] B. Keng, S. Safarpour, and A. Veneris, "Automated debugging of systemverilog assertions," in *2011 Design, Automation & Test in Europe*, 2011, pp. 1–6.

[38] [Online]. Available: https://cwe.mitre.org/index.html