System-on-Chip Information Flow Validation under Asynchronous Resets

Samit S. Miftah*, Kshitij Raj†, Xingyu Meng*, Sandip Ray†, Kanad Basu*

*ECE Department, University of Texas at Dallas, Richardson, TX, USA
†ECE Department, University of Florida, Gainesville, FL, USA

***Abstract*—Modern System-on-Chip (SoC) designs comprise hundreds of individual IP blocks, each with its custom implementation of reset signals in most cases. The asynchronous nature of these resets while crossing different reset domains makes the SoC prone to various vulnerabilities if not implemented and validated thoroughly. A key aspect in validating system functionality is to ensure the functionality under reset is verified. Traditional simulation-based validation techniques often become a bottleneck in complex SoC designs due to the large control path of these designs. We propose SoCCAR, a SoC validation framework that addresses this problem. SoCCAR leverages control flow graphs (CFG) of the design to extract the control flow associated with property violations caused by reset domain crossings due to asynchronous resets. SoCCAR efficiently tracks the chain of events leading to the payload without suffering from state space explosion, a common challenge in complex designs. We test the efficacy of SoCCAR in detecting such vulnerabilities by developing multiple SoC benchmarks, each embedded with custom vulnerability originating from reset implementations across different domains. These vulnerabilities reflect practical design complexity and correspond to security violations encountered in practice as a result of multiple asynchronous resets. SoCCAR successfully detected all violations with minimal computation overhead and runtime, making it a viable approach for detecting such violations in complex SoC designs.**

***Index Terms*—RTL Verification, Asynchronous reset, Multiple Reset Domain, Hardware Security.**

I. Introduction

The growing demand for advanced System-on-Chips (SoCs) in modern computing systems has led to larger and more intricate RTL designs, resulting in increased complexity and, as a consequence, a greater likelihood of functional errors and security vulnerabilities [1]. In order to address these challenges, SoCs undergo a rigorous functional verification process, accounting for over 70% of resources and engineering time [2]–[4]. Another crucial aspect of this verification effort is security validation to provide assurance for both reliability and security [4].

A major contributing factor to the increased design complexity stems from asynchronous events such as asynchronous resets, dynamic clock switching, software activities, and analog/mixed-signal (AMS) events [5]. Their unpredictable nature can create vulnerabilities that malicious actors may exploit, jeopardizing the system's integrity. Furthermore, the presence of multiple reset domains in an SoC poses added security challenges. These challenges include potential issues like data corruption and synchronization problems when signals traverse domains, particularly in cases where multiple IPs from untrusted vendors are integrated, each with its distinct reset implementations. Due to the wide range of potential asynchronous triggers, it becomes impractical for an SoC security architect to foresee system behavior, anticipate security risks, and proactively design safeguards. Consequently, the reliability of SoC designs heavily relies on security validation to identify and rectify problems arising from asynchronous events. In practice, detecting corner cases related to asynchronous events represents a complex challenge and constitutes a substantial portion of the validation costs in industrial SoC security validation.

There have been works on developing various methodologies for security verification of SoC designs. These methodologies include techniques like fuzzing, concolic testing, assertion checking, and information flow tracking [6]–[10]. However, these tools cannot comprehensively identify all the vulnerabilities arising from asynchronous resets. In particular, dynamic (simulation-based) techniques cannot exhaustively test all possible reset combinations. On the other hand, (commercial) formal verification tools typically necessitate the identification of all reset signals and analyzing the design under the assumption that the system remains uninterrupted by resets during execution. Although it is theoretically possible to relax the restriction on reset signals and treat them as non-deterministic free inputs, this approach leads to an overwhelming increase in the number of reachable design states, exceeding the capacity of existing formal tools. Moreover, the prevalent method employed to tackle scalability issues, namely manual abstraction, often eliminates the specific edge cases in the design that give rise to these violations. It is imperative to develop a security validation technique capable of detecting asynchronous reset-related violations while effectively managing the complexities and scalability challenges inherent in contemporary SoC designs without relying on manual abstractions.

In this paper, we introduce a novel approach, SoCCAR (SoC Security Checker under Asynchronous Resets), to address security vulnerabilities within the domain of SoC systems. SoCCAR offers a solution for the identification of security vulnerabilities originating from asynchronous resets. This solution hinges on two pivotal constituents: (1) an efficient algorithm for the extraction of the control flow graph governing reset-triggered events from the RTL design and (2) a symbolic testing framework designed for the comprehensive exploration of the design space, thereby ensuring detection of latent security vulnerabilities. Our methodology relies upon the utilization of symbolic execution techniques applied to the RTL code, thereby enabling the rigorous assessment of the behavior of reset signals on the security properties that undergo scrutiny. To tackle state space explosion, we extract a control path originating from the reset signals and terminating at the desired property. Subsequently, with the control path extracted and the number of states to be traversed significantly reduced, we engage in an exhaustive symbolic execution procedure. The complexity arising from multiple resets is addressed through the extension of the control path across multiple modules. This ensures the inclusion of any signals traversing reset signal domains due to a reset event and exerting an influence on other modules within the extracted controlled path. To test the efficacy of SoCCAR and conduct an impartial evaluation, we adopt a red team/blue team approach. In this approach, one team is entrusted with the systematic insertion of security vulnerabilities in accordance with a structured methodology, while the other team is tasked with the detection of these vulnerabilities employing SoCCAR. It should be noted that this paper proposes a methodology to detect security vulnerabilities originating from asynchronous resets. Defenses against them do not fall under the scope of this work.

The paper makes the following major contributions.

- SoCCAR, to the best of our knowledge, is the first hardware security validation framework that comprehensively accounts for asynchronous events during security validation.
- We introduce path extraction from the control flow of a hardware design. This technique enables the execution of exhaustive symbolic analysis within a reduced search space of states, effectively mitigating the issue of state space expansion.
- We develop a methodology to address challenges posed by multiple resets, particularly regarding the vulnerabilities they can introduce across different reset domains.
- We evaluate SoCCAR's effectiveness via a systematic testbed using a red team/blue team approach. One team introduced bugs into six variants of SoCs, while the other deploys SoCCAR to detect them. The SoCs used are simplified and sanitized versions of automotive (AutoSoC) and mobile (ClusterSoC) SoCs. SoCCAR successfully identified all bugs in this evaluation.

The rest of the paper is organized as follows. Section II provides essential background information on RTL design verification, and in Section III, we discuss the research challenges, primarily centered around RTL verification. Section IV reviews prior works in this domain, while Section V presents the architecture and operational principles of the SoCCAR framework. Section VI provides an evaluation of the SoCCAR's performance on benchmark SoCs, and Section VII concludes this paper.

II. Background

In this section, we provide an overview of the key concepts that are related to our framework. We will discuss the concepts of Reset Domain Crossing (RDC), Asynchronous reset, and security validation in contemporary industrial practice.

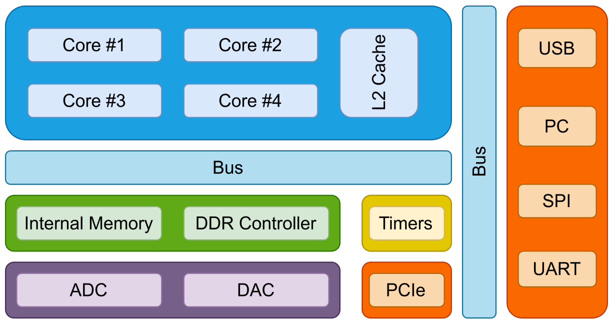


Fig. 1: Simple SoC Architecture Differentiated by reset domains.

A. *Asynchronous Resets in SoC Designs*

The complexity of SoC designs has increased due to the integration of diverse subsystems (computing, memory, cryptography, communication, etc.), each consisting of multiple IPs operating asynchronously with distinct clock domains. The practice of resetting individual modules has become a prevalent approach to address runtime issues such as crashes, violations, interrupts, etc. Modern SoC designs incorporate multiple reset domains, each governed by independent reset signals. An example of a simple SoC is shown in Figure 1. Currently, industrial-grade SoCs encompass tens to hundreds of reset domains, with each domain being managed by a combination of asynchronous reset signals [11]. Nevertheless, the application of asynchronous resets during execution can lead to unpredictable system behavior, yielding elusive defects that are arduous to detect.

B. *Reset Domain Crossing*

Reset domain crossing (RDC) refers to a sequential path in the design where the source and the destination sequential elements operate on independent resets [12]. Modern SoC designs operate in multiple domains (clock, reset, power) to accommodate various functionalities from different IPs and enable these IPs to operate in diverse configurations [11]. Nonetheless, the presence of multiple reset domains introduces several security vulnerabilities. Data transfer between reset domains must be accurately synchronized to avoid data corruption and reliability issues. Problems stemming from multiple resets also include untimely deactivation, polarity errors, activation sequencing, and insufficient reset cycles [5].

III. Challenges of Asynchronous Resets on Multiple Modules

To motivate the role of an asynchronous reset in the information flow violation of an SoC, we present the following examples.

Example 1: Insufficient or incorrect reset logic in cryptographic modules can lead to the failure of erasing memory [13] at its intended timestamp. Additionally, metastability problems may arise at the destination register when the reset of the source register is not the same as that of the destination register in an SoC [5].

Example 2: Furthermore, improper gating of reset signals within multiple reset domains can result in reset domain crossing (RDC) signals, leading to unexpected system behavior [14]. Such issues encompass failure to flush memory, metastability problems, and undesired state changes. For instance, a reset in the RAM of an SoC may induce meta-stability issues in the memory control unit. Another vulnerability related to RDC involves a reset in the memory control unit, altering the state of buffer register arrays, preventing data flushing, and potentially causing data leakage in the system.

During dynamic verification of SoC designs, exhaustive testing of all possible reset combinations throughout the execution is impractical. Formal verification involves the use of mathematical analysis to validate system properties; however, it depends on predetermined reset signals and is incapable of detecting violations that may arise due to asynchronous resets during execution. The overwhelming number of states that would result from considering reset signals as non-deterministic inputs would surpass the capacity of the current formal tools.

Attempts to address scalability issues through manual abstraction often eliminate critical corner cases leading to violations. Therefore, there is a pressing need for a security validation technique that detects violations resulting from asynchronous resets while managing the complexities and scalability challenges of modern SoC designs without resorting to manual abstractions.

IV. Related Works

Hardware security validation encompasses diverse research domains, one of which pertains to detecting information flow violations [15]–[17]. Investigations in this domain involve the application of formal methods, annotations, and type systems to develop designs with proven security.

Formal method-based approaches have been extensively researched and repurposed for directed test generation in RTL verification [18], [19]. Furthermore, formal techniques are employed to verify the validity of information flow policies and to authenticate the integrity of RTL designs against such policies [20], [21]. In order to make the formal verification methods memory efficient, several methodologies have been developed, such as symbolic model checking, partial order reduction, symmetry reduction, and bi-simulation minimization [22]–[27]. As designs become larger and more complex, these techniques face challenges due to "state space explosion" [4], [28]. In such cases, concolic testing is a promising approach that employs symbolic execution for generating directed tests [7], [29]. This method has shown success in both software and hardware applications. Automated concolic test generation techniques have also been explored to enhance coverage for large-scale designs [30], [31]. In recent work, a concolic testing-based approach was developed to validate the security of an IP core [3].

In light of the increasing prevalence of SoC designs featuring multiple clock and reset domains, several techniques have been proposed to verify the secure transfer of signals between said domains, referred to as clock domain crossing (CDC) and RDC, in order to guarantee trustworthy designs [32]. Several studies have looked into techniques for identifying and managing errors in CDC [33], [34]. The challenges that are specific to reset verification in hardware have been categorized [5]. In order to address these challenges, the use of assertions in conjunction with a reset distribution tree has also been explored [11]. –However, and in spite of critical need, we have not found any existing research on systematically examining the potential security implications of these features.

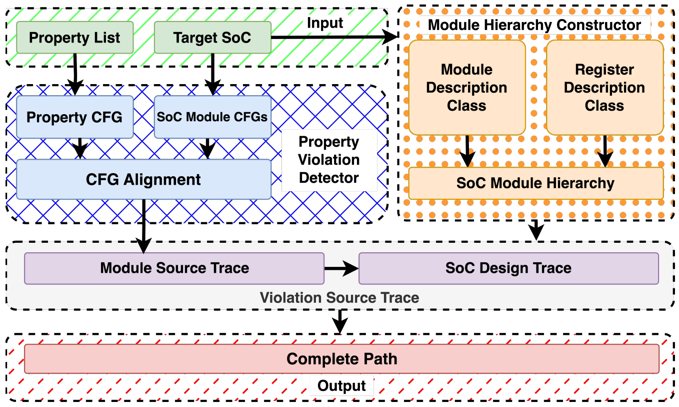


Fig. 2: SoCCAR Framework Workflow

V. SoCCAR Architecture

SoCCAR has been developed to address the security challenges of detecting vulnerabilities arising from asynchronous resets. The workflow of SoCCAR is shown in Fig. 2. SoCCAR takes a security property list and the target SoC as inputs and applies the verification steps discussed below. If any property is violated, SoCCAR extracts the complete path from the originating reset signal to the property violation as its output.

Consider an SoC design, $\mathcal{S}$, containing IP modules $\langle \mathcal{M}_1, ..., \mathcal{M}_k \rangle$, where each module $\mathcal{M}_i$ includes a list of clock signals $\mathcal{C}[\mathcal{M}_i]$ and reset signals $\mathcal{R}[\mathcal{M}_i]$, together with a (partially ordered) set of hardware events $E[\mathcal{M}_i]$. These events are composed of standard hardware operations and require specific signals to trigger them. We define the relationship between an event and its triggering signal as the signal governing the event. In other words, if a signal, v, governs event, e, we can assert that event, e, is controlled by signal, v, within the module M_i, where e is an element of the set $E[M_i]$. Each signal v_i defines a chain of events E_{vi} from the control flow of $\mathcal{M}_i$. v_i is referred to as the governing signal for E_{vi} in M_i. The list of properties, $\mathcal{P}_{list}$, defines the security specifications of the SoC. Therefore, to ensure that an SoC is secure from vulnerabilities arising from asynchronous resets, SoCCAR needs to verify the properties, $\langle \mathcal{P}_1, ..., \mathcal{P}_k \rangle \in \mathcal{P}_{list}$.

SoCCAR requirements: SoCCAR has two major functionalities: 1) traversing through the graph, and 2) running symbolic execution using an SMT solver. To traverse and select paths from the CFG, SoCCAR uses the networkX library [35] and to run symbolic execution SymbiYosys [36].

To detect vulnerabilities originating from asynchronous resets, SoCCAR includes four modules:

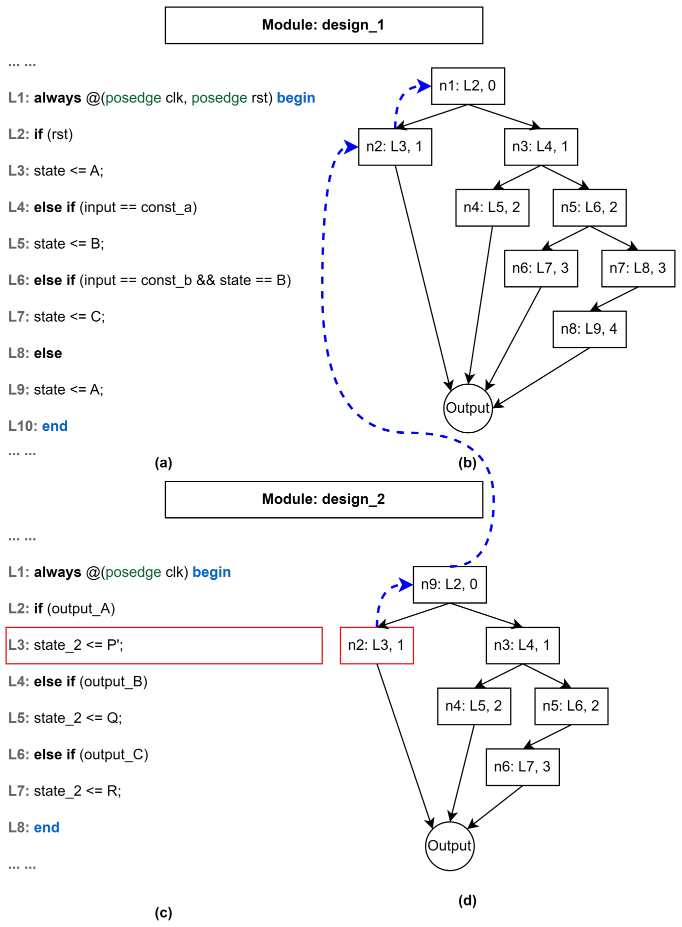
1) The *CFG generator* examines module design files and creates control flow graphs (CFGs) for each module and the properties. The "property CFG" is the control flow graph representing conditions and assignments asserted by a property. These CFGs are then utilized to identify violations and track the source of the signal causing the violation if one is found.
2) The *module hierarchy constructor* reads the RTL files. Subsequently, it indexes the modules and constructs a hierarchy of the modules within the SoC design. This assists in the subsequent stage of tracing the origin signal of the violation.
3) The *property violation detector* module first generates CFGs for the listed properties and the specified modules within the target SoC. Subsequently, it compares these property CFGs with the module CFGs to detect violations. When a violation is detected, the corresponding module is designated as the target module from which the signal's source is traced. This module of SoCCAR also limits the number of state spaces, thereby overcoming the problem of state space explosion.
4) Subsequently, the *violation source tracer* module traces the RDC signal from the security violation to the source signal (*i.e.*, asynchronous reset). It constructs the symbolic path of the violation.

A. CFG Generator

To facilitate the extraction of paths from the hardware control flow, SoCCAR constructs CFGs of the design using *CFG Generator*. This step entails parsing each RTL file embedded within the design and creating CFGs representing individual modules' control flow. When a security vulnerability is detected, back-tracking through these CFGs gives SoCCAR the ability to trace the vulnerability back to its origin point. Fig. 3 explains the procedure with an example. Notably, output of design_2, as shown in Fig. 3(c), depends on the output of design_1 shown in Figure 3(a). In case a vulnerability is found in design_2 at line 2, the relevant CFGs of these modules can be utilized to trace the source of this signal back to line 2 in design_1. In this scenario, the signal in question pertains to an asynchronous reset. In Fig. 3 (b) and (d), notations are employed to signify node numbers, their respective line numbers in code, and their respective flow values. Flow values refer to the level of the corresponding node in the control flow graph. The CFG generator also generates CFGs for properties, representing the conditional requirements to be met before asserting the property. Note that these property CFGs differ from module CFGs in that the conditions within them are not required to maintain a specific order.

B. Module Hierarchy Constructor

To identify the path within the SoC overall control flow, SoCCAR includes a procedure for hierarchy construction, which entails indexing both the design and header files, utilizing the *Module Hierarchy Constructor*. Here, "*module hierarchy*" refers to the arrangement of SoC modules and their organization within the overarching design. Algorithm 1



Module: design_1

```
... ...
L1: always @(posedge clk, posedge rst) begin
L2: if (rst)
L3: state <= A;
L4: else if (input == const_a)
L5: state <= B;
L6: else if (input == const_b && state == B)
L7: state <= C;
L8: else
L9: state <= A;
L10: end
... ...
```

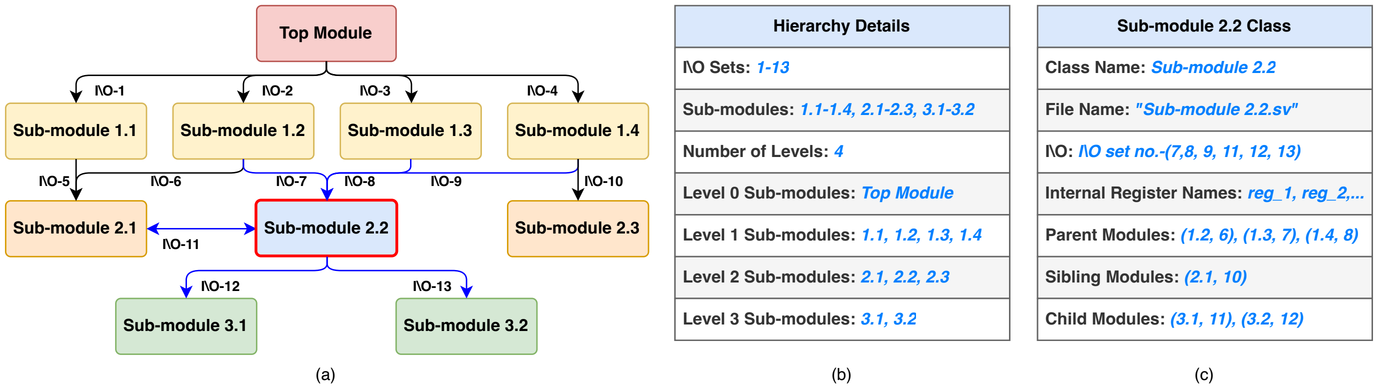
Module: design_2

```
... ...
L1: always @(posedge clk) begin
L2: if (output_A)
L3: state_2 <= P';
L4: else if (output_B)
L5: state_2 <= Q;
L6: else if (output_C)
L7: state_2 <= R;
L8: end
... ...
```

Fig. 3: Example of CFG of an RTL design vulnerable due to an RDC signal originating from an asynchronous reset. (a) verilog code of module: design_1, (b) CFG of module: design_1, (c) verilog code of module: design_2, (d) CFG of module: design_2.

describes this process. The idea is to take the complete SoC design as input and create a hierarchy, $\mathcal{H}_S$, and a list enlisting modules of interest in the SoC, $\mathcal{M}_{int}$. The design and header files are collected, as shown in line 1 in Algorithm 1. Next, all the design files are partially parsed to construct a list of module names. In this step, only the names of the modules are collected, thereby the term "partially parsing". A class, C_M, is defined to represent the modules in a structured manner (line 2). This class will be regarded as "*module class*". C_M stores the details of each module in the design, such as the module's name, design file name, input and output ports, internal register names, involved header files, and related sub-modules. This helps trace signal sources from the overall control flow of the SoC.

Suppose an SoC is based on a hierarchy as shown in Figure 4(a). To establish a relationship among the modules, we define three basic relationships. Modules that undergo instantiation are denoted as child modules, whereas those performing instantiations are termed parent modules. Modules linked directly to one another without instantiations are recognized as sibling modules. Considering sub-module 2.2 as our target module, modules that are instantiating sub-module 2.2 in their design are its parent modules, such as



Hierarchy Details
I\O Sets: *1-13*
Sub-modules: *1.1-1.4, 2.1-2.3, 3.1-3.2*
Number of Levels: *4*
Level 0 Sub-modules: *Top Module*
Level 1 Sub-modules: *1.1, 1.2, 1.3, 1.4*
Level 2 Sub-modules: *2.1, 2.2, 2.3*
Level 3 Sub-modules: *3.1, 3.2*

Sub-module 2.2 Class
Class Name: *Sub-module 2.2*
File Name: *"Sub-module 2.2.sv"*
I\O: *I\O set no.-(7,8, 9, 11, 12, 13)*
Internal Register Names: *reg_1, reg_2,...*
Parent Modules: *(1.2, 6), (1.3, 7), (1.4, 8)*
Sibling Modules: *(2.1, 10)*
Child Modules: *(3.1, 11), (3.2, 12)*

Fig. 4: SoC design hierarchy construction.

Algorithm 1 Module Hierarchy Construction

```
Input: $\mathcal{S}$, $\mathcal{P}_j$;
Output: $\mathcal{H}_S$, $\mathcal{M}_{int}$, $\mathcal{C}_{reg}$

 1: collect design_files in dir
 2: define $\mathcal{C}_M$
 3: define $\mathcal{C}_{reg}$
 4: initialize $\mathcal{M}_{list}$
 5: initialize $\mathcal{H}_S$
 6: for every reg in $\mathcal{P}_j$ append reg to $Reg_{list}$
 7: for $each\_file$ in $dir$ do
 8:     parse $each\_file$
 9:     append $module\_names$ to $\mathcal{M}_{list}$
10: end for
11: for $each\_file$ in $dir$ do
12:     parse $each\_file$
13:     for $\mathcal{M}_i$ in $each\_file$ do
14:         instantiate and update $\mathcal{C}_{Mi}$
15:         instantiate $\mathcal{C}_{reg}$ for each register in $\mathcal{M}_i$
16:         if all registers from $Reg_{list}$ in $\mathcal{M}_i$ then
17:             append $\mathcal{M}_i$ to $\mathcal{M}_{int}$
18:         end if
19:     end for
20: end for
21: construct $\mathcal{H}_S$ using $set\_of\,(C_{Mi})$
22: return $\mathcal{H}_S$, $\mathcal{M}_{int}$
```

sub-modules 1.2, 1.3, and 1.4. Sub-modules 3.1 and 3.2, which originate from sub-module 2.2, are the child modules of sub-module 2.2. sub-modules 2.1 and 2.2 are connected without instantiating each other; therefore, they are sibling modules. SoCCAR lists the modules with their relationship and the I/O ports that are used to connect the two modules. In the example of Figure 4(a), sub-module 2.2 is instantiated by its parent module sub-module 1.2 by I/O port set 7, the detail of sub-module 2.2 is shown in Figure 4(b). In Figure 4(c), *module class* for sub-module 2.2 lists the relationships and the corresponding I/O ports as tuples. The class also records the file name where the module design is located and the internal register names list. Once the class is defined and a list of module names is obtained (lines 7 to 9), SoCCAR proceeds to create instances of these classes for each module, initializing them appropriately. Alongside the module class, C_M, another class called C_{reg} is introduced to capture and store information about the internal registers within each module design (line 3). This class will be referred to as "*register class*", elaborated in detail in Section V-D. A list, Reg_{list} containing all the registers' names in the property, P_j, is also created (line 6). SoCCAR parses the design files and collects the information about the modules for constructing their respective classes, as shown in line 14. During this process, the *register class* instances are also updated (line 15). The module that contains all the registers from reg_{list} is added to the module of interest list, $\mathcal{M}_{int}$ (line 17). If no such module is found in the design files, $\mathcal{S}$ is categorized as violating the security properties, and an error flag is raised, notifying to verify the correctness of the security property. Using *module classes*, a complete graph of the SoC design is constructed, which represents the hierarchy, $\mathcal{H}_{\mathcal{S}}$, of modules in $\mathcal{S}$ (line 21).

Note that the *Module Hierarchy Constructor* targets the generation of a hierarchical structure denoted as $\mathcal{H}_{\mathcal{S}}$, serving as a comprehensive representation of the overall SoC design. Furthermore, $\mathcal{H}_{\mathcal{S}}$ gives essential details regarding the I/O ports employed for the interconnection of various modules within the SoC. This feature supports the framework's "Violation Source Trace" module, making it easy to trace vulnerabilities to their point of origin. (See Section V-D).

C. Property Violation Detector

SoCCAR utilizes the *Property Violation Detector* module to identify any security property violations that may occur while reducing state space. Algorithm 2 describes the overall process of this module, how it identifies whether any module in $\mathcal{S}$ violates the security properties $\mathcal{P}_j \in \mathcal{P}_{list}$ and reduces state space. It takes three inputs: module lists, $\mathcal{M}_{list}$, list of modules of interest or victim module, $\mathcal{M}_{int}$, and the security property to verify, $\mathcal{P}_j$.

```
    property example_property;
    @(p, q)
        (cond_B & cond_D & cond_G) |->
    Assignment_X;
    endproperty //example_property
```

Listing 1: Property Example.

Algorithm 2 Property Violation Detection

Input: $\mathcal{M}_{list}$, $\mathcal{M}_{int}$, $\mathcal{P}_j$
Output: reg_{vio}, $\mathcal{M}_{curr}$

```
1:  $\mathcal{P}_{CFG} \leftarrow$ extract$_{CFG}$ $(\mathcal{P}_j)$
2:  $\mathcal{C}_l \leftarrow$ CFG2list $(all\ conditions\ in\ \mathcal{P}_{CFG})$
3:  $CFG_i \leftarrow$ extract$_{CFG}$ $(\mathcal{M}_i)$
4:  set $vio\_flag$ = FALSE
5:  initialize($symbolic\_list$)
6:  for each $\mathcal{M}_i$ in $\mathcal{M}_{int}$ do
7:      initialize($path$)
8:      find $always_{blk-ij}$ such that $Sens_{lst} == Sens_{lstP}$
9:      mark($path$)
10:     $cond_{lstM} \leftarrow always_{blk-ij}$
11:     for each $cond_M$ in $cond_{lstM}$ do
12:         find $cond_M$ such that $cond_M \equiv (cond_P \in \mathcal{C}_l)$
13:         if find == NULL then
14:             unmark($path$)
15:         end if
16:         $cond_{P1} \leftarrow cond_{P-next}$
17:         if relation $(cond_P, cond_{P1})$ = AND then
18:             $cond_{lstM} \leftarrow cond_{M-blk}$
19:         else if relation $(cond_P, cond_{P1})$ = OR then
20:             $cond_{lstM} \leftarrow cond_{lstM}$
21:         end if
22:     end for
23:     if all $cond_P \in \mathcal{C}_l$ is aligned then
24:         append($symbolic\_list$)
25:     end if
26: end for
27: $(vio\_flag, In\_pattern)$ =
            execute_symbolic_analysis($symbolic\_list$)
28: return  $In\_pattern$ if $vio\_flag$ == TRUE
            else "property verified"
```

Initially, SoCCAR analyzes the security property to extract a CFG denoted as $\mathcal{P}_{CFG}$ (line 1 in Algorithm 2). This CFG outlines the required sequence and arrangement of conditions and assignments within the SoC design. By isolating the preceding conditions of an assignment, SoCCAR constructs a list of conditions denoted as $\mathcal{C}_l$ (line 2). If the security property $\mathcal{P}$ is satisfied, the CFGs from both the module and the property should align. In order to verify the alignment, SoCCAR extracts the CFG of the victim module, which is determined using the property (line 3). The property list, derived from the security property expressed in *SystemVerilog* assertion (Listing 1), serves as an illustration. Subsequently, SoCCAR aligns the conditions from $\mathcal{P}_{CFG}$ with those from $\mathcal{M}_{CFG}$ and verifies the assignments to detect any violations. The flag *vio_flag* is initialized to `FALSE` (line 4) to indicate the absence of violations. Here, the term "align with each other" indicates that the conditions before an assignment specified in the property match with those in the module's CFG. To illustrate, consider the code snippet presented in Listing 2. The path leading to the assignment of the value *valA* to register *regA*, governed by the conditions $(A == B)$ and $(C == D)$, conforms to the corresponding property illustrated in Listing 2. We refer to such instances as being aligned with one another.

```
always @(posedge clk) begin
    if (A==B) begin
        if (C==D) begin
            regA = valA;
        end
    end
end
property regA_prop;
    @(posedge clk) disable iff(nrst)
    (A == B) && (C == D) |-> regA == valA;
endproperty
```

Listing 2: Example code(in *SystemVerilog*).

To align the conditions, the framework constructs the condition list, $\mathcal{C}_l$, under an *always* block from the property (line 2). From $\mathcal{M}_{CFG}$, *always* blocks with a sensitivity list, $Sens_{lst}$, is selected, which should be similar to the sensitivity list of the security property, $Sens_{lstP}$ (line 6). Within the selected blocks, SoCCAR identifies the conditional blocks such as a condition C, where $\mathcal{C} \in \mathcal{C}_l$. If multiple conditions exist in the security property before an assignment, SoCCAR aligns them based on their relationships. If the conditions have a relation of "`AND`", SoCCAR attempts to align using nested conditional block (lines 17-18). When the conditions have an "`OR`" relation, the aligning procedures will find both conditions on the same block level (lines 19-20). The condition alignment process does not check instances of the conditions in sequence unless it is explicitly mentioned in the security property. Upon aligning each always/conditional block, the aligned blocks are marked for symbolic execution. The block is unmarked if the next conditional block is not found in one of the marked blocks. The marking process ends once the traversal through the CFG reaches the assignment, constructing a complete path. These paths are appended to a list for running symbolic execution (line 24). This list is used by the symbolic execution engine to be restricted to selected paths.

SoCCAR runs symbolic execution on the marked paths using *SymbiYosys* (line 27). Two types of symbolic execution are executed on the module: (a) a complete symbolic execution (assuming reset as a regular signal) and (b) conventional symbolic execution (it is assumed that the reset will not be triggered). From the paths in the symbolic execution list, those related to asynchronous resets are marked for complete symbolic execution, and the rest are for conventional symbolic execution. A path is considered related to asynchronous reset if the path is sensitive to reset, *i.e.*, *always @(..., posedge/negedge reset, ...)*. By restricting the symbolic execution engine within the selected paths, SoCCAR considers all reset conditions, thereby alleviating state space explosion. If a violation is detected during the symbolic execution, the violation detector returns the input pattern needed to reach the state. Otherwise, the property is declared as verified, and the process is ended (line 28).

The clock cycles boundary allowed for symbolic execution was set to 100 cycles for each step in the path. However, this boundary can be modified by the user as required. Within this predefined limit, using symbolic analysis, SoCCAR systematically explores the subsequent steps (conditions or assignment) from the path. The property is deemed unsatisfied if these steps are not identified within the set cycle boundary. In the case of a property dictating that an assignment is accomplished after a certain number of clock cycles following a condition, n, the clock cycle boundary is set to $(100 + n)$ cycles. Furthermore, when the assignment is found within the boundary but at a different clock cycle from the property, SoCCAR also regards this as a property violation.

Current state-of-the-art formal tools such as Cadence Jaspergold and Synopsys Formal require the definition of reset signals before verification can commence. This requirement is due to the assumption that resets will not occur during runtime in order to avoid state space explosion. Consequently, these tools inadvertently overlook vulnerabilities associated with asynchronous reset signals. To circumvent this, SoCCAR adopts a strategy to prevent overlooking paths containing asynchronous resets that might be disregarded under the aforementioned assumptions. Nevertheless, if a violation is triggered by an RDC signal, the payload remains detectable through traditional methods. However, the conventional methods fail when detecting the source of the vulnerability, $i.e.$, the origin of the RDC signal. Our experimental results, shown in Table V corroborate this.

D. Violation Source Trace

The *Violation Source Trace* module traces the violation of a security property from the payload to the corresponding asynchronous reset signal using Algorithm 3. This module encompasses the process of tracing the asynchronous reset signal responsible for the violation of the property. SoCCAR takes the module violating the property, M_{curr}, as input. Next, an empty path $Pth_{\mathcal{S}}$ is initialized. To trace the violating statement to the input, a class is defined to encapsulate information about each register in M_{curr} as mentioned in Section V-B. The register represented by the class is regarded as the *host*. The class contains information such as register name, registers that are assigned to the host, and registers that guard the assignment with a condition. An example of *host* class is shown in Table I for "*reg_r1*" in module "M_1", an example register to elaborate register class which includes the module it is located, and the related registers in the module.

TABLE I: Example of Register Class.

Register Class	
Host register name:	reg_r1
Module name:	M_1
Guard register and corresponding value:	(reg_pr1, x), (reg_pr2, y), (reg_pr3, z)

The subsequent phase of the tracing process entails identifying and isolating the erroneous statement. Once such a statement is identified, the entire path, Pth_{Mcurr}, of the CFG in M_{curr} is extracted (line 2 of Algorithm 3). By utilizing the register classes associated with Pth_{Mcurr}, the framework generates a list of inputs, regarded as In_{Mcurr}, that lead to the faulty assignment (line 3). The process of extracting the path and constructing the list marks the completion of tracing

Algorithm 3 Violation Source Tracing

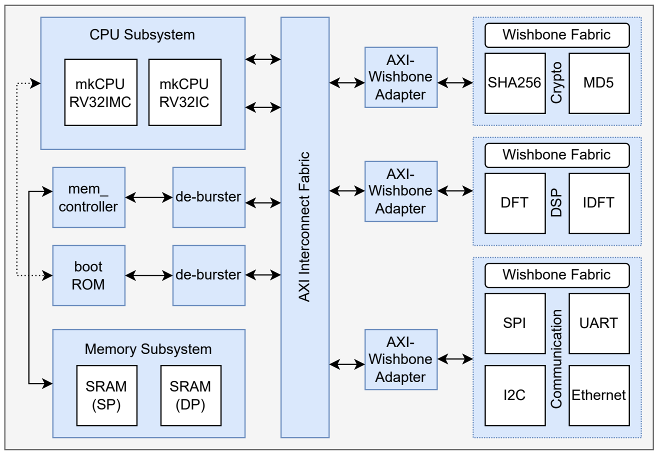
```
Input: $\mathcal{M}_{list}$, $\mathcal{H}_{\mathcal{S}}$, $\mathcal{M}_{curr}$, $IO_{\mathcal{S}}$
Output: $Pth_{\mathcal{S}}$

 1: initialize $Pth_{\mathcal{S}}$
 2: $\mathcal{M}_{curr.pth} \leftarrow$ extrct_pth_CFG $(\mathcal{M}_{curr}, \mathcal{C}_{reg.curr})$
 3: from $\mathcal{M}_{curr.pth}$ construct $In_{MCurr}$
 4: append $\mathcal{M}_{curr.pth}$ to $Pth_{\mathcal{S}}$
 5: while $In_{MCurr} \nsubseteq IO_{\mathcal{S}}$ do
 6:     traverse upper in $\mathcal{H}_{\mathcal{S}}$
 7:     $\mathcal{M}_{curr} \leftarrow$ parent $(\mathcal{M}_{curr})$
 8:     extractCFG$(\mathcal{M}_{curr})$
 9:     $\mathcal{M}_{curr.pth} \leftarrow$ extrct_pth_CFG $(\mathcal{M}_{curr}, \mathcal{C}_{reg.curr})$
10:     from $\mathcal{M}_{curr.pth}$ construct $In_{MCurr}$
11:     append $\mathcal{M}_{curr.pth}$ to $Pth_{\mathcal{S}}$
12: end while
13: return $Pth_{\mathcal{S}}$
```

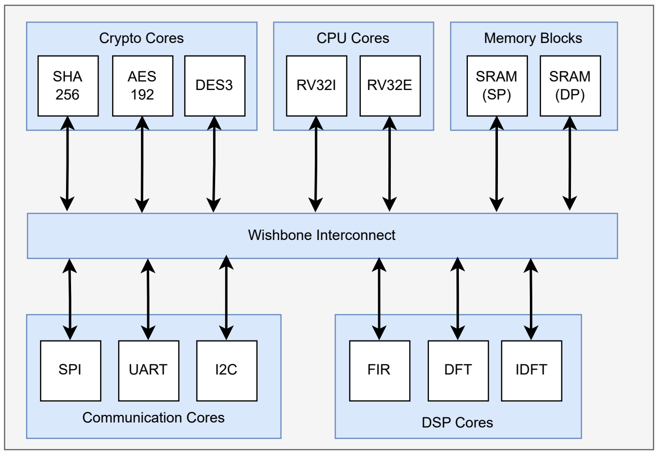
within the faulty module, and the path is appended to $Pth_{\mathcal{S}}$ (line 4). If the inputs of In_{Mcurr} form a subset of SoC, $\mathcal{S}$, the overall tracing process is completed (line 5). However, if In_{Mcurr} is not a subset of the inputs of the SoC, $In_{\mathcal{S}}$, *i.e.*, $In_{Mcurr} \in In_{\mathcal{S}}$, SoCCAR will proceed to traverse the hierarchy upwards (line 6). In the higher level of the hierarchy, SoCCAR will search the module M_i that contains the submodule M_{curr} and assigns value to In_{Mcurr}. Subsequently, this M_i becomes the new M_{curr}, and the tracing continues (line 7). The next iteration of tracing requires the construction of updated In_{Mcurr} from the new M_{curr}. SoCCAR generates and analyzes CFG of the new M_{curr} (line 8). Next, it performs symbolic execution on the updated M_{curr} to identify the path from the CFG that corresponds to the input pattern for the old M_{curr} leading to the violation (line 9). SoCCAR extracts the path and constructs In_{Mcurr} (line 10), and appends In_{Mcurr} to $Pth_{\mathcal{S}}$ (line 11). It subsequently repeats the previous phase until $In_{Mcurr} \in In_{\mathcal{S}}$ or a reset in any module. Following each tracing iteration, the extracted paths are combined to form the complete path, $Pth_{\mathcal{S}}$, as the final output of SoCCAR (line 13). This complete path, $Pth_{\mathcal{S}}$, establishes the symbolic path from the asynchronous reset of the SoC to the violation of the security properties. Through this procedure, SoCCAR tracks any RDC signal back to the specific module where the asynchronous reset was initiated. By using this process, SoCCAR extracts only CFGs from the modules in the same hierarchical path, which overcomes the challenge of extracting the complete CFG for the entire SoC while minimizing the false positive rate that comes with incomplete CFG.

VI. SoCCAR Evaluation

This section provides a detailed explanation of the evaluation approach used to assess SoCCAR's efficacy in detecting security violations caused by asynchronous resets. First, we explain how we assess its performance. Next, we discuss the process of designing representative SoCs and the bug insertion strategy into the benchmarks. In conclusion, we showcase the evaluation process, results, and findings from the evaluation of SoCCAR.



(a)



(b)

Fig. 5: SoC models used for SoCCAR evaluation. (a) AutoSoC: Hierarchical bus-based SoC topology. (b) ClusterSoC: Standard bus-based SoC topology.

A. *Evaluation Method*

The assessment of SoCCAR in our study was done using a red-team/blue-team approach. The red team studied and crafted bugs, devised a methodology for their inclusion, and developed diverse SoC benchmarks with embedded bugs. In order to ensure an impartial evaluation, no exchange of information took place between the red and blue teams with regard to bug descriptions, IP classes, the frequency of occurrence, types of bugs introduced, and their location. The blue team only received the faulty design and the designated property for verification from the red team, with no communication regarding the design and implementation of SoCCAR between the two teams. SoCCAR was evaluated based on the seven properties listed in Section VI-D. These properties represent the six violation types listed in Table IV.

B. *Representative SoC design*

We created several representative SoC benchmarks with multiple reset domains. These designs are complex enough to demonstrate SoCCAR's scalability. The benchmarks represent scaled-down versions of commercial SoCs commonly used in developing mobile and Internet of Things (IoT) devices. They have a hierarchical tiled architecture consisting of different components such as processors and peripherals, as shown in Figure 5. These components are connected through a shared bus, and there are separate subsystems for domain-specific applications. Bus bridges are used to connect the subsystems

TABLE II: Area Statistics of SoC Benchmarks. Results are based on synthesized designs using AMD-Xilinx Vivado.

SoC Architecture	SoC Variants	RTL Lines	Area Statistics			
			LUTs	LUTRAMs	FFs	BRAMs
ClusterSoC	Variant #1	≈8200	33524	2641	13214	124
	Variant #2		33862	2181	13286	124
	Variant #3		31928	2258	12682	126
AutoSoC	Variant #1	≈3200	35826	2298	15622	126
	Variant #2		38861	2971	18982	128
	Variant #3		37972	2874	17214	128

to the main system bus, which follows the AMBA specification and uses the AXI-4 protocol. However, the application-specific subsystems use different protocols, like Wishbone and APB.

There are two main types of System-on-Chip (SoC) configurations used as benchmarks for the evaluation of SoCCAR: hierarchical bus-based and standard bus-based. These configurations are depicted in Figure 5. AutoSoC is for automotive applications, and ClusterSoC is for IoT/mobile applications. Both models use open-source IP implementations from MIT-CEP benchmarks and OpenCores.

The benchmarks utilized in this study are based on the RISC-V ISA, with a specific focus on the RV32IM and RV32IMC variants. The application-specific subsystems consist of functional units called "islands", such as cryptographic, memory, communication, and DSP subsystems. For example, the cryptographic subsystem includes modules for SHA256 and MD5, while the DSP subsystem has DFT, IDFT, and FIR modules.

In order to showcase different security vulnerabilities associated with multiple reset domains, three distinct variants were employed in creating the benchmarks. Table II presents the area metrics for all three versions, encompassing LUTs ranging from 35,800 to 38,900. These SoCs, despite their complexity, have demonstrated the successful execution of standard Linux benchmarks and RISC-V applications. Our assessment of SoCCAR confirms its scalability and accuracy when used with these SoC designs. Moreover, these SoC benchmarks serve as a useful testing ground for diverse SoC validation frameworks. They provide a manageable yet feature-rich platform that reflects the complexity of commercial SoC designs. It should be noted that other open-source designs, including CVA-6 [37] and Rocket-chip-based [38] SoC designs, were also considered as evaluation benchmarks. However, we discovered that these SoCs lack the necessary complexity to address our research goals. Specifically, our research focuses on the security implications of multiple asynchronous reset domains that interact and lead to subtle security compromises. To explore these scenarios, we need

SoCs with multiple IPs, particularly the so-called “south cluster IPs” that communicate asynchronously across various reset domains. Unfortunately, neither of the mentioned SoCs exhibit such interactions; they primarily implement processor-memory cores rather than full-blown SoC functionality (despite these interactions being common in industrial designs). Due to the scarcity of sufficiently complex SoCs, we made the decision to design and implement our own SoCs. These custom SoCs serve as platforms for us and the research community, allowing us to validate different research solutions using realistic features found in industrial SoCs.

C. Bug Insertion Strategy

It is essential to identify the different types of bugs that may occur in order to demonstrate the efficacy of SoCCAR. Moreover, it is important that the bugs are rarely activated and dispersed across various designs rather than restricted to a single module. It is crucial to note that detecting various bug types in one evaluation testbed through the verification technique does not necessarily guarantee the identification of bugs in other SoCs. To assess the thoroughness of the evaluation, a systematic approach was developed to insert suitable bugs into different variations of SoC benchmarks. The frequency of each bug class was also varied within each SoC.

The methodology for introducing violations/bugs can be broken down into three phases. First, we determine the type of violation based on the category of IP. The rationale behind this is that certain security violations are better suited for specific categories of IPs. For example, a Denial-of-Service bug is relevant to a router on the chip that connects different islands, while an information leakage bug is relevant to a cryptographic island. In the second phase, we insert these bugs into various versions of the benchmark. Based on the categorization, we have identified distinct types of bugs, which have been presented in Table III. These bugs are designed to evaluate the security of the system in a comprehensive manner. During asynchronous resets, these bugs trigger the delivery of particular payloads, which can compromise the core security aspects of the SoC designs, including integrity, confidentiality, and availability. We now describe the representative instances of violations/bugs implanted onto the benchmarks:

a) Information Leakage: During an asynchronous reset, the module fails to clear the registers storing plaintext and keys used in cryptographic calculations, allowing unauthorized processes to access these values.

b) Data Integrity: Failure of the read/write request address range check following an asynchronous reset occurs due to this bug. Unauthorized access to protected memory regions can occur if memory address registers are not cleared properly. The outcome of this could involve unauthorized read/write operations.

c) Privilege Mode: This bug prevents the state machine that manages the privilege settings of the RISC-V cores from transitioning between the default modes *i.e.*, user, supervisor, and machine. To achieve this, we intentionally introduce incorrect privilege switches during asynchronous resets in the RISC-V cores. Consequently, the privilege level register is assigned an undefined value, leading to a critical functionality error due to the absence of an available privilege level.

d) Delay of Reset Signals: To avoid metastability or sampling of asynchronous data, the clock cycle delay of reset assertion must be taken into account during the design process. If enforced incorrectly, an attacker may exploit such vulnerabilities and gain access to sensitive data assets in protected memory regions, perform unauthorized R/W operations, etc.

e) Unsynchronized RDC: When transitions in reset domains lack synchronization mechanisms, changing the asynchronous reset of a source register during operation can cause the destination register to become “*metastable*”. This condition can lead to security vulnerabilities, including information leakage or data integrity violations.

f) Unsynchronized Reset Assertion: While considering multiple reset domains, the order of reset assertion between coupled blocks with RDC should be in sequence, else the

TABLE III: Summary of Security Violations/Bugs.

Violation Type	**Trigger Condition**	**Impact**
Information Leakage	Asynchronous reset in Crypto Engine	Inadvertent disclosure of confidential assets can happen if an attacker gains access to unencrypted plain text through the cipher text port. This type of bug compromises the confidentiality of secure assets in the SoC design.
Loss of Data Integrity	Asynchronous reset at memory module	Unauthorized access of read and write operations to secure memory regions breaches the integrity and confidentiality of on-chip assets.
Unavailability of Privilege modes	Asynchronous reset at processor core	Failure to transition between privilege modes undermines the availability of essential system functions.
Delay of Reset Signals	Unaccounted clock cycle delay of reset assertion	Delays in reset signals can cause timing violations, race conditions, incorrect initialization of the circuit or memory elements, and also lead to data integrity and confidentiality violations.
Unsynchronized RDC	No synchronization mechanism between transitions in different reset domains.	May induce metastability in registers controlled by different reset domains, incorrect initialization of registers, and functional errors during run-time.
Unsynchronized Reset Assertion	Out-of-order reset assertion between coupled blocks with RDC	This may lead to incorrect/asynchronous sampling of data from registers. It may also cause data integrity and confidentiality violations in cryptographic IPs, memory blocks, etc. by allowing unauthorized read access to protected memory regions before proper reset assertion.

TABLE IV: Bugs/Violations Injected in SoC Benchmarks for SoCCAR Evaluation.

Bug No.	Violation Type	**Bug/Violation Placement**					
		ClusterSoC			**AutoSoC**		
		Variant #1	Variant #2	Variant #3	Variant #1	Variant #2	Variant #3
AR#1	Information Leakage	MD5, AES192	–	AES192, SHA256	MD5, SHA256	AES192	–
AR#2	Data Integrity	SRAM	SRAM	Wishbone Bus	SRAM	–	–
AR#3	Privilege Mode	–	RV32I	RV32E	RV32-IC, RV32-IM	RV-32IM	–
AR#4	Delay in Reset Signal	SHA-256	–	SRAM(DP)	–	–	–
RDC#1	Unsynchronized RDC	–	–	–	RV32-IC	SHA-256	–
RDC#2	Unsynchronized Reset Assertion	–	–	–	–	RV32-IMC	SRAM(DP)

possibility of sampling asynchronous data may happen. This is particularly crucial in IPs handling confidential data assets, *i.e.*, crypto IPs, memory blocks, etc., and may lead to information leakage and compromise data integrity.

In this study, we investigated bugs inspired by real security vulnerabilities found in commercial SoC designs. These bugs were then modified to match our evaluation criteria. We created multiple instances of each bug type and integrated them into our evaluation benchmarks using a random bug injection process. The bugs, their variation type, and placement in the SoC are shown in Table IV.

D. Evaluation Results

SoCCAR was successfully able to detect all three bugs, as mentioned in Table IV. In the upcoming subsections, we will discuss each bug from Table IV individually, providing more information about them as reported by SoCCAR.

In Table V, we provide a detailed comparison of our improved version of SoCCAR, the previous work [39], and the widely-used state-of-the-art formal verification tool, Cadence JasperGold [40]. Table V enlists the bug IDs used in this paper and their violation type in the first two columns respectively. The following six columns are organized in pairs of violation and source detection ability by SoCCAR v1.0, SoCCAR v2.0, and Cadence JasperGold. The tabulated findings underscore that SoCCAR v2.0 excels in both the detection of previously unidentified vulnerabilities (Bug AR#2) and the vulnerabilities stemming from RDCs (Bug RDC#1 and RDC#2). Furthermore, In our evaluation, as shown in Table V, JasperGold could not identify violations of the asynchronous reset violations (AR#1–AR#4). It was able to identify the states responsible for two of the reset domain crossing violations. Nevertheless, it lacks the capability to identify the source signal leading to the occurrence of those violations. Furthermore, we have also included the duration for detecting each bug in Table VI. With the current benchmarks, there were no false positives or negatives.

TABLE VI: SoCCAR bug detection time.

Bug No.	Violation Type	Time Duration
AR#1	Information Leakage	278 ms
AR#2	Data Integrity	312 ms
AR#3	Privilege Mode	284 ms
AR#4	Delay in Reset Signal	292ms
RDC#1	Unsynchronized RDC	748 ms
RDC#2	Unsynchronized Reset Assertion	806 ms

In the following paragraphs, we describe each of the bugs with the property they violate and the report generated by SoCCAR.

1) Bug AR#1: SoCCAR detected a bug causing information to be potentially leaked due to asynchronous reset. This bug violates the property displayed in Listing 3. The property asserts that the keys and data bits be cleared upon receipt of an asynchronous reset.

```
property inf_leakage;
@(posedge wb_clk_i)
   wb_rst_i |=> (key[0]!) && (wbm_dat_o!)
endproperty //inf_leakage

```

Listing 3: Security property for Bug AR#1.

As seen from the CFG shown in Listing 4, if a reset occurs during the operation of the AES module, the key value is set

TABLE V: SoCCAR comparison with previous version and Cadence JasperGold.

Bug No.	Violation Type	SoCCAR v1.0		SoCCAR v2.0		Cadence JasperGold	
		Violation Detection	Source Detection	Violation Detection	Source Detection	Violation Detection	Source Detection
AR#1	Information Leakage	✓	✓	✓	✓	✗	✗
AR#2	Data Integrity	✗	✗	✓	✓	✗	✗
AR#3	Privilege Mode	✓	✓	✓	✓	✗	✗
AR#4	Delay in Reset Signal	✓	✓	✓	✓	✗	✗
RDC#1	Unsynchronized RDC	✗	✗	✓	✓	✓	✗
RDC#2	Unsynchronized Reset Assertion	✗	✗	✓	✓	✓	✗

to LOW. However, the data value is not being cleared thereby, causing a potential information leakage. This bug is therefore an information leakage type of bug as displayed in Table IV

```
    Module: aes_top :: 3 :: wb_clk_i == 1 :: Alws;
    Module: aes_top :: 3, 0 :: wb_rst_i == 1 :: C;
    Module: aes_top :: 3, 0 :: key[0] <== 0 :: A;
```

Listing 4: CFG of Bug AR#1.

2) Bug AR#2: SoCCAR detected a data integrity failure in the second benchmark SoC design. In this design, the property listed in Listing 5, which asserts clearing of the acknowledge flag and memory block information, is violated.

```
    property dat_intgrty;
    @(posedge wb_clk_i)
        wb_rst_i |=> (mem[waddr2][32:24] == 0) &&
    (wb_ack_o!)
    endproperty //dat_intgrty
```

Listing 5: Security property for Bug AR#2.

However, in this benchmark SoC, no such logic was implemented; thereby, upon resetting, the memory block and acknowledgment flags are not cleared. This violates the data integrity of an SoC that causes unauthorized write to memory. This bug represents Bug type 'B' from Table IV.

3) Bug AR#3: A privilege transition error is observed by SoCCAR in the third SoC design benchmark. This bug violates the set of properties listed in Listing 6, which asserts that after a reset the bus should go back to the initialization state.

```
    property dat_intgrty;
    @(posedge wb_clk_i)
        wb_rst_i |=> state == WBSTART;
    endproperty //dat_intgrty
```

Listing 6: Security property for Bug AR#3.

However, the CFG shown in Listing 7 demonstrates that the property after a reset, one of the RISC-V cores (picorv32) goes to an idle state, which does not clear the privilege flags. This poses a critical flaw in maintaining security regarding privilege level permissions of the SoC.

```
    Module: picorv32_top :: 6 :: wb_clk_i == 1 ::
    Alws;
    Module: picorv32_top :: 6, 0 :: wb_rst_i == 1
    :: C;
    Module: picorv32_top :: 6, 0 :: state <== IDLE
     :: A;
```

Listing 7: CFG of Bug AR#3.

4) Bug AR#4: An initialization error was detected by SoCCAR in the third benchmark. The property demonstrated in Listing 8 outlines the initialization protocol for the SRAM(DP) module of the benchmark SoC. It specifies that the "*data_last_buf*" and "*rp*" registers should be cleared or set to LOW one cycle after receiving a reset signal. SoCCAR detected a violation of this property in the module "*noc_buffer*".

```
    property rp_reset;
    @(posedge clk)
    rst |-> data_last_buf = 0 && rp = 0
    endproperty //rp_reset
```

Listing 8: Security property for Bug AR#4.

As evidenced by Listing 9, upon receiving reset signal in the *AXI_Adapter* module in the NoC buffer, the register *rp* is incremented by one after one clock cycle. The source signal and the payload both belong to the same module. The security of the SoC is violated by faulty logic.

```
    noc_buffer :: 4 :: clk == 1 :: Alws;
    noc_buffer :: 4, 0 :: rst == 1 :: C;
    noc_buffer :: 4, 0 :: rp <== @(posedge clk) rp
     + 1 :: A;
```

Listing 9: CFG of Bug AR#4.

Due to this bug, the AXI adapter does not clear the register '*rp*' upon a reset, causing incorrect initialization of the NoC buffers. As illustrated by Table IV, Bug C is placed in the SRAM (DP) module of the NoC in the SoC.

5) Bug RDC#1: SoCCAR detected a reset synchronization error in the first SoC benchmark in the RV32IC module. This bug violates properties shown in Listing 10, which asserts proper synchronization of instruction fetch protocol. Listing 10 shows two properties. According to the first property, upon reset, output instruction and next output of the programmable counter must be LOW. The second property mandates that if the *ifid* module is enabled, the instruction output of *ifid* must be the same as the input instruction, and the next programmable counter output must be the same as its respective input.

```
1   property reset_value_set_0;
2       @(posedge clk and negedge nRST) !nRST |=>
    ifid.instr_out == 0 && ifid.pc_next_out == 0;
3   endproperty //reset_value_set_0
4   property reset_value_set_1;
5       @(posedge clk and negedge nRST) ifid.
    enable == 1 |=> ifid.instr_out  == ifid.
    instr_in && ifid.pc_next_out == ifid.
    pc_next_in;
6   endproperty //reset_value_set_1
7
```

Listing 10: Security property for Bug RDC#1.

Listing 11 illustrates the control flow of the reported violation. As seen in lines 1 to 4, the source of the RDC signal causing the violation originates in the module *exmem*. The RDC signal propagates through the modules *datapath* and *hazard unit* (lines 4 to 11). Subsequently, it causes the *ifid* module to have a delay in assigning the next program counter value (lines 12 and 13).

The bug results in a synchronization issue within the system, as the instruction fetch module experiences a delay in assigning the next instruction. This bug was placed in the RV32IC module of the SoC by the red team as displayed in Table IV.

```
exmem :: 0 :: CLK == 1, nRST == 0 :: Alws;
exmem :: 0, 0 :: nRST == 0 :: C;
exmem :: 0, 0 :: exif.dWEN_out <== '0 :: A;
exmem :: 0, 0 :: exif.dREN_out <== '0 :: A;
datapath :: 52 :: Comb/* :: Alws;
datapath :: 52 :: huif.WEN <== exif.dWEN_out :: A;
datapath :: 52 :: huif.REN <== exif.dREN_out :: A;
hazard_unit :: 0, 4 :: ++++ELSE++++ :: C;
hazard_unit :: 0, 4, 0 :: huif.ihit == 1 and huif.REN == 0 and huif.WEN == 0 or huif.dhit == 1 :: C;
hazard_unit :: 0, 4, 0 :: huif.ifid_enable <== 1 :: A;
ifid :: 0, 2, 0 :: ifid.enable == 1 :: C;
ifid :: 0, 2, 0 :: ifid.instr_out <== #1 ifid.instr_in :: A;
ifid :: 0, 2, 0 :: ifid.pc_next_out <== #1 ifid.pc_next_in :: A;
```

Listing 11: CFG of RDC#2.

6) Bug RDC#2: SoCCAR detected an unsynchronized reset assertion type of violation that violates the property listed in Listing 12 in the first benchmark. This property ensures that the *memory control* module maintains the correct initialization of registers. The property description in Listing 12 dictates that if the *memory_control* is in '*WBCACHE1*' state and the *cache_control*'s ram state is '*ACCESS*', the value of *dwait[!dsource]* and *dwait[dsource]* must be 0 and 1 respectively in the same clock cycle.

```
property wbcache;
@(posedge CLK, negedge nRST)
    (curr_state ==WBCACHE1 & ccif.ramstate== ACCESS) |=> (ccif.dwait[!dsource] = 0 & ccif.dwait[dsource] = 1);
endproperty //wbcache
```

Listing 12: Security property for Bug RDC#2.

```
1  ram :: 5, 0 :: nRST == 0 or addr == ramif.ramaddr and ramif.ramREN == 1 or ramif.ramWEN == 1 and count >= LAT :: C;
2  ram :: 5, 0 :: rstate <== ACCESS :: A;
3  system :: 0 :: multicore.prif <== ram.prif :: A;
4  multicore :: 4 :: ccif.ramload <== scif.ramload :: A;
5  multicore :: 5 :: ccif.ramstate <== scif.ramstate :: A;
6  memory_control :: 1 :: Comb/* :: Alws;
7  memory_control :: 1, 3 :: curr_state == WBCACHE1 :: C;
8  memory_control :: 1, 3, 0 :: ccif.ramstate == ACCESS :: C;
9  memory_control :: 1, 3, 0 :: ccif.dwait[dsource] <== 0 :: A;
10 memory_control :: 1, 3, 0 :: ccif.dwait[!dsource] <== 0 :: A;
11
```

Listing 13: CFG of Bug RDC#1.

Listing 13 illustrates the control flow of the bug. The source signal of the bug is indicated by the first line in the reported CFG by SoCCAR. In Listing 13, it can be seen that under the condition shown in line 1, the "*RAM*" module initiates an RDC. The RDC propagates through *system* and *multicore* to the module *memory control* (lines 4 to 6). This RDC causes the module to violate the property (lines 6 to 10) by assigning both *ccif.dwait[!dsource]* and *ccif.dwait[dsource]* registers to `LOW` (line 9 and 10).

Therefore, due to Bug B, the implementation of an asynchronous reset in the SoC *RAM* leads to the occurrence of metastability issues in the *Memory Controller*. As illustrated in Table IV, the bug is placed in the RV32IMC of the SoC.

VII. Conclusion

To ensure the security of complex SoC designs that consist of multiple reset domains, it is crucial to consider violations that may arise due to erratic system behavior caused by asynchronous resets. These violations can be extremely rare and are among the most challenging bugs to detect in current industrial practices. To the best of our knowledge, our framework, SoCCAR, presents the first systematic approach that aims to detect such violations. The systematic exploration of violations caused by asynchronous resets is achieved by SoCCAR through CFG extraction and static analysis of reset logic and their impact on its propagation throughout the design. Although SoCCAR can significantly mitigate the effects of state space explosion, it still has to perform symbolic execution for 100 cycles after each step in the path. If the path is excessively long, this could lead to high memory usage and time consumption. In order to evaluate the scalability and efficacy of SoCCAR in real-world SoCs, we developed an elaborate experimental testbed. Based on our evaluation of SoCCAR, we found that it attains near-perfect detection accuracy within a few seconds of verification time. It is noteworthy that SoCCAR operates directly on the RTL implementation of complex SoCs without necessitating manual abstraction.

VIII. Acknowledgment

This research is supported by NSF grant #2223046.

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] K. L. Maidhili, F. Noorbasha, A. Vamsi, and K. H. Kishore, "Reset logic verification of an iod at system on chip level using gatesim," *International Journal*, vol. 8, no. 7, 2020.

[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] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "{HardFails}: Insights into {Software-Exploitable} hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[5] C. Kwok, P. Viswanathan, and P. Yeung, "Addressing the challenges of reset verification in soc designs," in *Design and Verification Conference (DVCon)*, 2015.

[6] 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.

[7] 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.

[8] 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.

[9] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–33, 2022.

[10] 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.

[11] I. Ahmed, K. Nouh, and A. Abbas, "Multiple reset domains verification using assertion based verification," in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2017, pp. 1–6.

[12] M. Fawzy, A. Elgohary, and H. Ibrahim, "Noise reduction in reset domain crossings verification using formal verification," in *2020 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 2020, pp. 1–5.

[13] 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

[14] P. Yeung, E. Marschner, and K. Liu, "Multi-domain verification: When clock, power and reset domains collide," in *Design and Verification Conferecen, DVCon*, 2015.

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

[16] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[17] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[18] M. Chen and P. Mishra, "Property learning techniques for efficient generation of directed tests," *IEEE Transactions on Computers*, vol. 60, no. 6, pp. 852–864, 2011.

[19] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 146–162, 1999.

[20] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *2012 IEEE 30th VLSI Test Symposium (VTS)*. IEEE, 2012, pp. 252–257.

[21] M.-M. Bidmeshki, A. Antonopoulos, and Y. Makris, "Information flow tracking in analog/mixed-signal designs through proof-carrying hardware ip," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1703–1708.

[22] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, "Symbolic model checking for probabilistic processes," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1997, pp. 430–440.

[23] D. A. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, 2003.

[24] E. M. Hahn, T. Han, and L. Zhang, "Synthesis for pctl in parametric markov decision processes," in *Nasa formal methods symposium*. Springer, 2011, pp. 146–161.

[25] A. F. Donaldson and A. Miller, "Symmetry reduction for probabilistic model checking using generic representatives," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2006, pp. 9–23.

[26] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 234–248.

[27] J.-P. Katoen, T. Kemna, I. Zapreev, and D. N. Jansen, "Bisimulation minimisation mostly speeds up probabilistic model checking," in *International Conference on tools and algorithms for the construction and analysis of systems*. Springer, 2007, pp. 87–101.

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

[29] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[30] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: concolic verification and test generation for gpus," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 215–224.

[31] K. Cong, F. Xie, and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 303–310.

[32] P. Ashar, "Static verification based signoff-a key enabler for managing verification complexity in the modern soc," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 15–15.

[33] N. Karimi and K. Chakrabarty, "Detection, diagnosis, and recovery from clock-domain crossing failures in multiclock socs," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 32, no. 9, pp. 1395–1408, 2013.

[34] K. Takara, C. Kwok, N. Jain, and A. Hari, "Next-generation power aware cdc verification–what have we learned," in *Design and Verification Conference and Exhibition United States*, 2015.

[35] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[36] YosysHQ, "Symbiyosys: A front-end driver for yosys-based formal verification," https://github.com/YosysHQ/sby, 2024.

[37] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[38] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[39] X. Meng, K. Raj, A. P. D. Nath, K. Basu, and S. Ray, "Soccar: Detecting system-on-chip security violations under asynchronous resets," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 625–630.

[40] "Jasper rtl apps | cadence - cadence design systems." [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

Samit Miftah (S'23) is a doctoral student in the Department of Electrical and Computer Engineering at the University of Texas at Dallas, Richardson, Texas as part of the TIES lab. He received his B.Sc in Engineering degree in Electrical & Electronic Engineering from Bangladesh University of Engineering and Technology (BUET) in 2019. Samit is pursuing his Ph.D. in the domain of Hardware Security. His research interests include hardware and system security, Trojan detection and hardware verification. His research has been published in Hardware Oriented Security and Trust (HOST).

Kshitij Raj (S'19) is a doctoral student in the Department of Electrical and Computer Engineering at the University of Florida, Gainesville, Florida as part of the Rising lab. He received his B.Tech degree in Electronics & Telecommunication Engineering from KIIT University, India, in 2017 and his Masters degree in Electrical and Computer Engineering from the University of Florida in 2020. Kshitij is pursuing his Ph.D. in the domain of Secure Silicon Design and Validation. His research interests lie in the field of Silicon Architecture, Design, Validation and Micro-architecture Verification. His research has been published in Design Automation Conference (DAC), Design, Automation and Test in Europe Conference (DATE), AsianHOST Conference, etc.

Xingyu Meng (S'20) is a doctoral student in the department of Electrical and Computer Engineering at the University of Texas at Dallas as part of the Trustworthy and Intelligent Embedded System (TIES) lab. He received his BE degree in Electronics Science and Technology from Nankai University in 2015, and he received his MS degree in System Engineering from the University of Texas, Dallas, in 2019. His research interests include hardware and system security, Trojan detection and hardware verification. His research has been published in Design Automation Conference (DAC), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), etc.

Sandip Ray (SM'13) is a Professor with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA, where he holds an Endowed IoT Term Professorship at the Warren B. Nelms Institute for Connected World. Before joining University of Florida, he was a Senior Principal Engineer at NXP Semiconductors, and prior to that, he was a Research Scientist with Intel Strategic CAD Laboratories. During his industry tenure, he led industrial research and R& D in pre-silicon and post-silicon validation of security and functional correctness of SoC designs, design-for-security and design-for-debug architectures, and security validation for automotive and Internet-of-Things applications. His current research targets correct, dependable, secure, and trustworthy computing through the cooperation of specification, synthesis, architecture, and validation technologies. He is the author of three books and over 100 publications in international journals and conferences. He has also served as a Technical Program Committee Member of over 50 international conferences, as the Program Chair of ACL2 2009, FMCAD 2013, and IFIP IoT 2019, as a Guest Editor for IEEE DESIGN & TEST, IEEE TMSCS, and ACM TODAES, and as an Associate Editor of Springer HaSS and IEEE TMSCS. He has a Ph.D. from the University of Texas at Austin.

Kanad Basu (S'07-M'12-SM'20) received his Ph.D. from the department of Computer and Information Science and Engineering, University of Florida. His thesis was focused on improving signal observability for post-silicon validation. Post-PhD, Kanad worked in various semiconductor companies like IBM and Synopsys. During his PhD days, Kanad interned at Intel. Currently, Kanad is an Assistant Professor at the Electrical and Computer Engineering Department of the University of Texas at Dallas. Prior to this, Kanad was an Assistant Research Professor at the Electrical and Computer Engineering Department of NYU. He has authored 2 US patents, 2 book chapters, and several peer-reviewed journal and conference articles. Kanad was awarded the "Best Paper Award" at the International Conference on VLSI Design 2011. Kanad's current research interests are hardware and systems security.