SoCCAR: Detecting System-on-Chip Security Violations Under Asynchronous Resets

Xingyu Meng[1], Kshitij Raj[2], Atul Prasad Deb Nath[2], Kanad Basu[1] and Sandip Ray[2]

[1]ECE Department, University of Texas at Dallas, Richardson, TX, USA
[2]ECE Department, University of Florida, Gainesville, FL, USA

***Abstract*—Modern SoC designs include several reset domains that enable asynchronous partial resets while obviating complete system boot. Unfortunately, asynchronous resets can introduce security vulnerabilities that are difficult to detect through traditional validation. In this paper, we address this problem through a new security validation framework, SoCCAR, that accounts for asynchronous resets. The framework involves (1) efficient extraction of reset-controlled events while avoiding combinatorial explosion, and (2) concolic testing for systematic exploration of the extracted design space. Our experiments demonstrate that SoCCAR can achieve almost perfect detection accuracy and verification time of a few seconds on realistic SoC designs.**

I. Introduction

Most modern computing devices are typically designed via System-on-Chip (SoC) architecture, through integration and composition of pre-designed hardware intellectual property (IP) blocks procured from a variety of globally distributed supply chain. IPs in a typical industrial SoC can be diverse, including a variety of processor cores, memory modules, cryptographic blocks, communication modules (*e.g.*, wireless and LTE modules), debug and peripheral driving interfaces (*e.g.*, JTAG, HDMI, USB, etc). SoC designs promise much faster design turnaround time, robustness, and configurability than custom hardware. However, an unfortunate effect of this integration has been a steep increase in design complexity, together with a corresponding increase in security vulnerabilities. Therefore, it is imperative to develop techniques for systematic detection of security violations in modern SoC designs.

A key source of complexity in modern SoC designs is the plethora of *asynchronous events*, *i.e.*, events triggered by a condition independent of the main execution flow of the system. Such triggers include asynchronous resets, dynamic clock switching, software activities, analog/mixed-signal (AMS) events, etc. Unfortunately, the unpredictability in system behavior resulting from an asynchronous event can result in subtle corner case vulnerabilities that can be exploited by an adversary to compromise the integrity of the entire system. On the other hand, given the plethora of potential asynchronous triggers, it is impossible for a SoC security architect to anticipate the behavior of the system in response to these events, formulate their security implications a priori, and design mitigations. Consequently, trustworthy SoC designs critically depend on security validation to identify violations resulting from asynchronous events. Indeed, asynchronous event corner cases represent some of the most hard-to-detect bugs in industrial SoC security validation practice and account for the majority of validation cost.

In this paper, we develop a framework, SoCCAR ("**So**C Se**c**urity **C**hecker Under **A**synchronous **R**esets"), for detecting SoC security violations that comprehends one of the most pervasive and fundamental asynchronous events, partial resets. Partial asynchronous resets have been enabled in current industrial SoC designs with multiple reset domains, and permit partial initialization of selected IPs and design functionality in the middle of the execution while obviating the need for a full system reboot. However, they can be source of subtle security vulnerabilities resulting from inconsistencies in microarchitectural states of different IPs, unanticipated effects on finite state machines (FSMs) that govern the execution control flows, etc. SoCCAR provides a systematic mechanism for detecting security violations that comprehends asynchronous resets through (1) a new technique for efficiently extracting the Control Flow Graph of reset-controlled events from the RTL design, and (2) a concolic testing framework for effective exploration of the extracted design space.

The paper makes several important contributions. *First*, while there has been significant research in hardware and SoC security validation techniques including mature commercial tools (see Section VI), the analysis performed has implicitly assumed the absence of asynchronous events *e.g.*, formal techniques [1], [2] introduce assumptions specifying absence of resets which are critical to their scalability and precision. To the best of our knowledge, SoCCAR represents the first security validation framework that comprehends and accounts for the critical role of asynchronous events. *Second*, SoCCAR shows how to apply concolic testing efficiently on complex SoC designs while obviating manual design decomposition or abstraction or losing verification accuracy: SoCCAR provides almost perfect detection accuracy on realistic SoC designs with multiple reset domains, with a verification latency of only a few seconds. *Third*, as part of the evaluation of SoCCAR, we have developed a comprehensive experimental testbed that includes multiple SoC designs as well as a systematic methodology for inserting security bugs. The SoC designs include realistic features that reflect the complexity of modern industrial systems, *e.g.*, reset domain crossings, multiple asynchronously controlled IPs, hierarchically organized subsystems with heterogeneous communications, etc., and can act as authoritative benchmarks for future security validation research.

The remainder of the paper is organized as follows. Section II provides the relevant background in SoC security validation, asynchronous resets, and concolic testing. Section III explains the challenges in security validation under asynchronous resets. We discuss SoCCAR architecture and implementation in Section IV. Section V presents our experimental setup and the experimental evaluation of SoCCAR. We discuss related work in Section VI and conclude in Section VII.

II. Background

A. Asynchronous Resets in SoC Designs

Reset sequences have been traditionally used in microelectronic systems to enforce a system boot. As complexity of SoC designs increased through the introduction of a variety of subsystems (*e.g.*, compute, memory, crypto, communication, etc.), each consisting of multiple IPs which execute relatively asynchronously, often using different clocks; reseting an IP or module has become a common workaround for a variety of runtime issues, including hangs,

crashes, violations, interrupts, etc. On the other hand, enforcing a full system boot has also become extremely expensive since it results in significant computation loss and ultimately reduced throughput. Furthermore, full system boot is often precluded in SoC designs for safety-critical applications except under very rare situations. To address this problem, most modern SoC designs include multiple reset domains governed by independent reset signals. Current industrial SoCs include tens to hundreds of reset domains; each reset domain may be controlled by a combination of multiple asynchronous reset signals. However, asynchronous resets in the middle of the execution can result in unpredictable system behavior, resulting in hard-to-detect bugs. While there has been some work on analysis of reset domain crossing, as mentioned in Section VI, to the best of our knowledge there is no formal tool that comprehensively comprehends the role of resets in functional, performance, or security validation.

B. SoC Security Validation Practice

Security validation of hardware designs entails thorough exploration, analysis, and evaluation of a diverse set of attack surfaces originating from malicious third-party IPs, malicious software and firmware, insecure on-chip communications, and many other potential sources, that can compromise trusted system operation. The area is extremely broad, with significant academic research as well as mature commercial tools [3], [1], [2]. Nevertheless, security validation remains a vexing problem in current industrial practice. In particular, most validation tools do not scale to the size and complexity of current industrial microelectronics. Furthermore, we are aware of no tool or framework that can handle asynchronous events, including resets, analog/mixed-signal activities, etc. Consequently, security validation critically relies on insights of human experts, to perform white-box intrusion testing, or design manual abstractions to facilitate application of analysis tools [4].

C. Concolic Testing

Concolic testing is a hybrid testing technique that integrates concrete execution with symbolic execution. The name "concolic" is a portmanteau of "concrete" and "symbolic". The key idea is to repeatedly execute a program on concrete inputs, but piggyback symbolic execution on top of concrete execution. At the end of each concrete run, a concolic testing tool heuristically selects another execution path (*e.g.*, by flipping some of the conditionals in the concluded run). This new execution path is then encoded symbolically and the resulting formula is solved by a constraint solver, to yield a new concrete input. The concrete execution and the symbolic analysis alternate until an intended level of structural coverage is reached. Concolic testing has been applied successfully on both software and hardware domains [5]. However, none of these methods indulge in the security vulnerabilities caused due to asynchronous events in hardware designs.

III. The Challenge of Asynchronous Resets

To motivate the role of asynchronous resets in SoC security, consider the following example. Albeit highly simplistic, it actually is a sanitized version of a real bug in an industrial SoC design that escaped to production.

An Example Bug: A secure firmware is left encrypted in DRAM but decrypted after being loaded to secure SRAM during execution. Access to the SRAM is gated during that execution. However, an asynchronous reset during firmware execution unlocks the gating logic but fails to clear the SRAM itself, leaving the firmware exposed to unauthorized access.

Why is detecting such a violation difficult? Note that if a security validator had knowledge of the exact scenario in which the bug manifested, then exercising it would be straightforward. However, in the absence of such knowledge, it is highly non-trivial to detect such vulnerabilities using either dynamic (simulation-based) or formal security validation methodologies. During dynamic validation, it is clearly prohibitive to comprehensively exercise all possible reset combinations at all points during the SoC execution. Correspondingly, formal verification, *i.e.*, the use of mathematical analysis to verify desired system properties, typically requires identification of all reset signals a priori. Moreover, the analysis proceeds under the constraint that the system is not reset during at any point during the execution. This obviously precludes identification of violations resulting from asynchronous resets. In principle, it is possible to relax this restriction on reset signals and treating them as non-deterministic free inputs. However, doing so results in unmanageable explosion in the set of reachable states of the design beyond the capacity of current formal tools. Furthermore, the common approach to used to address the scalability problems, *viz.*manual abstraction, typically eliminates the very corner cases of the design that lead to such violations. ***Consequently, it is imperative to develop a security validation technique that detects violations resulting from asynchronous resets while being able to handle the complexities and scalability challenges of modern SoC designs without manual abstractions.***

IV. SoCCAR Architecture and Implementation

Our proposed technique, SoCCAR addresses the above problem by enabling systematic exploration of the impact of asynchronous resets on SoC security using concolic testing. We introduce a few notations and definitions before discussing the SoCCAR architecture. For the purpose of this description, we assume that an SoC design $\mathcal{S}$ includes a list of modules $\langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$, where each $\mathcal{M}_i$ includes a list of clock signals $C[\mathcal{M}_i]$ and reset signals $\mathcal{R}[\mathcal{M}_i]$[1], together with a (partially ordered) set of hardware events $\mathcal{E}[\mathcal{M}_i]$. We leave the description of the events unspecified, but they will be assumed to consist of standard RTL operations. For each event $e \in \mathcal{E}[\mathcal{M}_i]$ and a signal v in $\mathcal{M}_i$ we will say that *v governs e* if e is executed only when $v = 1$. Obviously, each signal v defines a projection $\mathcal{P}_v$ of the Control Flow Graph of $\mathcal{E}[\mathcal{M}_i]$ that includes only the events governed by v. We refer to $\mathcal{P}_v$ as the *governing CFG* of $\mathcal{M}_i$ with v. The notion of governing CFG naturally extends to a set of signals $V \triangleq \{v_1, \ldots, v_n\}$, by defining $\mathcal{P}_{\mathcal{V}} \triangleq \bigcup_{i=1}^{n} \mathcal{P}_{v_i}$. If the set $\mathcal{V}$ is the set $\mathcal{R}[\mathcal{M}_i]$, we refer to the governing CFG $\mathcal{P}_{\mathcal{V}}$ as *the Asynchronous Reset CFG* (AR_CFG for short) of $\mathcal{M}_i$. We use $\mathcal{AR}[\mathcal{M}_i]$ to denote the AR_CFG of $\mathcal{M}_i$. The Asynchronous Reset CFG of the SoC $\mathcal{S}$ is then naturally defined as the interactive composition $\mathcal{AR}(\mathcal{S}) \triangleq \mathcal{AR}[\mathcal{M}_1] \,||\, \mathcal{AR}[\mathcal{M}_2] \,||\, \ldots \,||\, \mathcal{AR}[\mathcal{M}_k]$.

Fig. 1 shows the overall workflow of SoCCAR. Roughly, it includes the following three components.

1) For each module $\mathcal{M}_i$ of $\mathcal{S}$ construct the AR_CFG $\mathcal{AR}[\mathcal{M}_i]$. This is done by analyzing the CFG of $\mathcal{E}[\mathcal{M}_i]$ to identify process blocks governed by asynchronous resets.
2) Assemble and connect the individual AR_CFGs to create the AR_CFG $\mathcal{AR}(\mathcal{S})$. This is done by computing and developing connection profiles of all constituent modules.
3) Given the AR_CFG $\mathcal{AR}(\mathcal{S})$, concolic testing is used to systematically explore the impact of asynchronous resets.

[1] Instead of manually specifying reset signals, it is possible to use automated clock and reset analysis available in most EDA tools to automatically identify these signals.

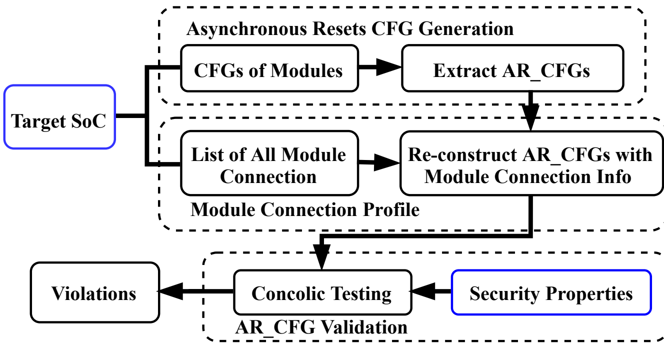

Figure 1. SoCCAR Framework Workflow

Algorithm 1 Asynchronous Resets *CFG* Generation

Input: $\mathcal{S}$, *IPs*
Output: *AR_CFG* of $\mathcal{M}_i$

1: **Initialize** $\mathcal{AR}[\mathcal{M}]$
2: **for** all *IPs* **do**
3: **Initialize** $[\mathcal{M}_i]$
4: $[\mathcal{M}_i]$ **append** $\mathcal{M}_i$
5: **for** all $\mathcal{E}[\mathcal{M}_i]$s **do**
6: $\mathcal{E}[\mathcal{M}_i] \leftarrow e$
7: $[\mathcal{M}_i]$ **append** $\mathcal{P}_{v_i}$
8: $[\mathcal{M}_i]$ **append** $\mathcal{E}[\mathcal{M}_i]$
9: **end for**
10: **Initialize** $\mathcal{AR}[\mathcal{M}_i]$
11: **for** all $\mathcal{E}[\mathcal{M}_i]$s in $[\mathcal{M}_i]$ **do**
12: **if** $\mathcal{R}[\mathcal{M}_i]$ in $\mathcal{E}[\mathcal{M}_i]$ **then**
13: $\mathcal{AR}[\mathcal{M}_i]$ **append** $\mathcal{P}_{v_i}$, $\mathcal{E}[\mathcal{M}_i]$
14: **end if**
15: **end for**
16: $\mathcal{AR}[\mathcal{M}]$ **append** $\mathcal{AR}[\mathcal{M}_i]$
17: **end for**

A. Asynchronous Resets CFG Generation

Algorithm 1 extracts the process flows from the IPs. We initialize the procedure by searching for $\mathcal{M}_i$ and add it into a list $\langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$ to initialize its *CFG*. The algorithm then searches for all the hardware events $\mathcal{E}[\mathcal{M}_i]$s and their procedural blocks in the IP designs. Each $\mathcal{E}[\mathcal{M}_i]$ is isolated to establish its condition v and execution e in the *CFG*. A *subCFG* is defined as the connection between a v and its e. Simultaneously, we define the trigger condition $\mathcal{P}_{v_i}$ for each $\mathcal{E}[\mathcal{M}_i]$ by placing their procedural blocks (*always* @) between each *subCFG*. Once a complete *CFG* $[\mathcal{M}_i]$ of its associated $\mathcal{M}_i$ is generated, the algorithm starts extracting the asynchronous resets and their procedural blocks from the $[\mathcal{M}_i]$. We achieve this by searching the $\mathcal{E}[\mathcal{M}_i]$s that contain reset signals $\mathcal{R}[\mathcal{M}_i]$. $\mathcal{R}[\mathcal{M}_i]$ is usually defined in a universal naming format with terms such as *resetn* or *rst*. The algorithm will process all the generated $[\mathcal{M}_i]$ and reorganize all the $\mathcal{M}_i$ and their associated asynchronous resets *AR_CFG* in a new $\mathcal{AR}[\mathcal{M}_i]$ for the next task.

B. Module Connection Profile

To complete construction of $\mathcal{AR}(\mathcal{S})$ for each $\mathcal{M}_i$, we also need to generate inter-connection profiles of all the sub-modules in the SoC. Algorithm 2 explores the design flow of each $\mathcal{M}_i$ to determine the connection with the other $\mathcal{M}_x$s. An connection profile is created for each module to assemble its dedicated $\mathcal{AR}[\mathcal{M}_x]$ from its inter-connected $\mathcal{M}_x$s. The algorithm performs a static traversal of the module structure of each module $\mathcal{M}_i$, collecting all sub-module invocations as well as logistic information required to compute the connected CFG (*e.g.*, clocks, resets, etc.). The result is the creation of a list $\mathcal{CN}[\mathcal{M}_i]$ that identifies each module $\mathcal{M}_x$ invoked by $\mathcal{M}_i$. The construction of $\mathcal{AR}(\mathcal{S})$ is then done by composing for each $\mathcal{M}_x \in \mathcal{CN}[\mathcal{M}_i]$ the CFG $\mathcal{AR}[\mathcal{M}_x]$ into the top module for $\mathcal{M}_i$.

Algorithm 2 Module Connection Profile

Input: $\mathcal{S}$, $\mathcal{IP}$s
Output: $C[\mathcal{M}]$

1: **Initialize** $\mathcal{A}[\mathcal{IP}]$
2: **for** all $\mathcal{IP}$s in $\mathcal{S}$ **do**
3: $\mathcal{A}[\mathcal{IP}]$ **append** $\mathcal{IP}$
4: **end for**
5: **Initialize** $\mathcal{A}[\mathcal{M}]$
6: **for** All $\mathcal{IP}$ in $\mathcal{A}[\mathcal{IP}]$ **do**
7: **if** $\mathcal{M}_x$ in $\mathcal{IP}$ **then**
8: $\mathcal{A}[\mathcal{M}]$ **append** $\mathcal{M}_x$
9: **end if**
10: **end for**
11: **Initialize** $\mathcal{CN}[\mathcal{M}]$
12: **for** all $\mathcal{IP}$s in $\mathcal{S}$ **do**
13: **Initialize** $\mathcal{CN}[\mathcal{M}_i]$
14: **if** $\mathcal{M}_x$ in $\mathcal{M}_i$ **then**
15: $\mathcal{CN}[\mathcal{M}_i]$ **append** $\mathcal{M}_x$
16: **end if**
17: **end for**
18: $\mathcal{CN}[\mathcal{M}]$ **append** $\mathcal{CN}[\mathcal{M}_i]$

C. Concolic Testing on AR_CFGs

Algorithm 3 describes the concolic testing framework on AR_CFG. It starts with each $\mathcal{AR}[\mathcal{M}_i]$ in constructed CFGs, and the (concrete) simulation path consists of all the events e in the $\mathcal{AR}[\mathcal{M}_i]$. Note that the algorithm can randomly assign the reset and clock signals to start the first round of simulation. To address this, we assign all the registers with ones instead of zeros; consequently, we can validate the major functionalities of asynchronous resets such as register clearance and value reset. After the first round of simulation, if e is not in the execution path, the simulator computes the condition v that governs the e in the path, and solves the constraints on clock edge and reset signal. Note that clock edge can be transformed into a equivalence condition,*e.g.*, *(posedge clk)* can be translated into *(clk == 1)*. Correspondingly, reset signal can also be transformed into equivalence conditions, *e.g.*, a condition *if* (*∼reset*) is translated into *if (reset == 0)*. These equivalences are used to solve the v of the asynchronous resets.

Algorithm 3 can account for additional security constraints to enable effective path exploration. For instance, a representative constraint can be *"after a reset the data memory must be cleared"*. Such constraints are generally available as part of the security regression in industrial practice. The simulation checks each such available constraint at each round; if any of the constraints is violated, the simulation will return an invalidation message and mention the module that violates the restriction.

V. SoCCAR Evaluation

A. SoC Platforms

A critical challenge with evaluation of any research on SoC validation is the lack of open-source SoC designs of realistic complexities. For instance, SoCCAR requires SoC designs with multiple reset domains and IPs and subsystems with complex functionality governed by various asynchronous reset signals. Furthermore, a thorough

Algorithm 3 Asynchronous Events Validation

Input: $\mathcal{AR}[\mathcal{M}_1]||\mathcal{AR}[\mathcal{M}_2]||\ldots||\mathcal{AR}[\mathcal{M}_k]$, *Security_ Properties*
Output: Invalid Messages
1: *Restricts* ← *Security_ Properties*
2: **for** $\mathcal{AR}[\mathcal{M}_i]$ in $\mathcal{AR}[\mathcal{M}_1]||\mathcal{AR}[\mathcal{M}_2]||\ldots||\mathcal{AR}[\mathcal{M}_k]$ **do**
3: **Initialize Input** ← randombits()
4: **Initialize Registers** ← **ones**
5: **Path**, **Invalid** ← *Simulate*(**Input**,*Restricts*)
6: **if** $e \in$ **path then**
7: **return Input**
8: **end if**
9: **while** $e \notin$ **path do**
10: **Input** *solve v*
11: *Testcase* **append**(**Input**)
12: **Execute Path**,**Invalid**← *Simulate*(**Input**,*Restricts*)
13: **if** $e \in$ **Path then**
14: **return** *Testcase*
15: **end if**
16: **if Invaild** not empty **then**
17: **return Invalid**
18: **end if**
19: **end while**
20: **end for**

Simulate(**Input**, *Restricts*)
1: Initialize **Invalid**
2: **for** all *CFG* in $\mathcal{AR}[\mathcal{M}_i]$ **do**
3: *Variables* ← **Input**
4: *values* ← **execute** *flows*
5: **Input append** *values*
6: *check _invalid(Restricts*, **Input**)
7: **end for**

evaluation requires different variants of SoC designs with a variety of security violations. To address these challenges, we have developed a number of realistic SoC benchmarks, and a systematic methodology for inserting security violations. For evaluating SoCCAR we use two SoC benchmarks ClusterSoC and AutoSoC shown in Fig. 2, that include multiple reset domains with asynchronous resets. While simplified, these SoCs are inspired from commercial SoC designs.

ClusterSoC is a representative mobile/IoT SoC with an area efficient implementation of RISC-V cores and peripherals. It is a sanitized and simplified version of an industrial SoC used for low-power mobile systems. The RISC-V cores implement two different 32-bit ISAs, *i.e.*, the baseline integer (RV32I) and the embedded extension (RV32E) with reduced number of registers. This SoC model has a Wishbone (B3) cross-bar as the shared system bus and two SRAM modules (dual port and single port) are integrated as memory modules. Three crypto engines *i.e.*, SHA, DES3, and AES, are incorporated for various cryptographic operations. It also features three high-speed DSP cores *i.e.*, FIR, DFT, and IDFT for signal processing. ClusterSoC supports three external communication mediums including an UART module, an SPI controller, and an Ethernet controller for off-chip communication as well as connectivity to cloud services.

AutoSoC is a sanitized and simplified version of SoCs used for automotive infotainment applications. It is significantly more complex than ClusterSoC and features hierarchical and heterogeneous shared buses and application-specific subsystems. Each subsystem its own communication fabric and represents a tiled architecture. The shared buses of individual subsystems are connected to the system bus with a bus-bridge. The system bus implements a variation of AMBA bus

Table I
AREA STATISTICS OF CLUSTERSOC AND AUTOSOC. RESULTS ARE BASED ON SYNTHESIZED DESIGNS USING XILINX VIVADO.

SoC Models	Variants	Area		
		LUT	LUTRAM	BRAM
ClusterSoC Variants	Variant #1	16906	2698	124
	Variant #2	17047	2618	126
	Variant #3	15891	2298	126
AutoSoC Variants	Variant #1	33861	2971	128
	Variant #2	32972	2874	128

protocol *i.e.*, AXI4-Lite, and the subsystems incorporate their own Wishbone (B3) bus. In comparison to ClusterSoC, each subsystem of AutoSoC incorporates additional IP cores and a variety of advanced architectural features. For instance, the CPU subsystem is scaled up from two area-optimized cores to three cores hat implement compressed/variable size (RV32IC) ISA, and multiply-and-divide ISA (RV32IM) in addition to the baseline integer ISA. The memory subsystem includes a DMA controller. The number of crypto cores and DSP blocks in AutoSoC are augmented for additional functionality such as implementation of cryptographic hash algorithms and IIR filter for signal processing.

Both ClusterSoC and AutoSoC are fully functional SoC designs. Our experiments use three variants of ClusterSoC and two variants of AutoSoC. The different variants are developed by introducing different security bugs on the base SoCs (see Section V-B). To demonstrate the realistic complexity of the SoC designs, we synthesized FPGA implementations using Xilinx Vivado. Table I shows the area statistics of these implementations. Note that different variants of ClusterSoC consume between 15891 and 17047 LUTs, while AutoSoC variants go to 32972 and 33861 LUTs. Perhaps more importantly, both SoCs can run standard RISC-V applications and Linux-based benchmarks. The use of these SoCs in our experiments help demonstrate the scalability and precision of SoCCAR for practical systems. Furthermore, we believe, the creation of such SoCs will facilitate future research on SoC validation by providing representative benchmarks that are manageable yet reflect various complex features of commercial SoC implementations.

B. Bug Insertion

Evaluating a validation technique requires not only realistic SoC designs but also a methodology for introducing realistic bugs in those SoCs. Furthermore, it is important to be able to demonstrate that the technique can identify (1) different classes of violations, and (2) rarely excitable violations. Note that these requirements are somewhat at odds. In particular, it is not realistic to evaluate a validation tool on a single SoC design with all the different classes of violations introduced: if a tool can detect violations in such an SoC, there would be no a priori reason to believe that the tool can also be used to validate an SoC which includes only a few of the violations (and consequently, the violations in the SoC are rarer). To address these two requirements, we developed a methodology for systematically developing different variants of the two SoCs, with each variant including a subset of security bugs.

Our overall bug insertion methodology works in three stages. First, we divide the IPs in the SoC into different classes as shown in Table II. The idea is to account for the fact that certain bugs are relevant to certain IP types, *e.g.*, an information flow violation that compromises a key or plaintext is relevant to a crypto core while a

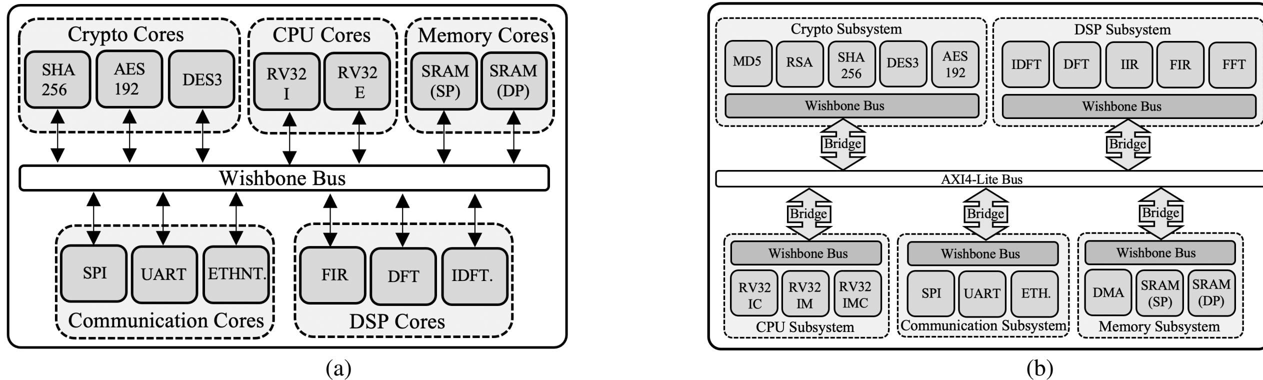

Figure 2. Two SoC Designs Used for Evaluation of SoCCAR. (a) ClusterSoC: A Simplified SoC targeted for Mobile Applications. (b) AutoSoC: A Simplified Automotive SoC.

Table II
Classification of IP class and violation types

IP Class	Example IPs	Violation Type
Memory IP	SRAM(SP), SRAM(DP), DMA Engine	Loss of Data Integrity.
Processor Core	RV32I, RV32E, RV32IC, RV32IM, RV32IMC	Unavailability of Privilege Modes.
Cryptographic IP	AES192, SHA256, RSA, MD5, DES3	Information Leakage.

DoS attack making some privilege modes unavailable would make sense in a processor IP. Second, given these classifications, we define a number of bug types for SoCCAR evaluation, as shown in Table III. These bugs get triggered at asynchronous reset events and deliver specific payloads leading to eventual violation of the basic security properties of the SoC designs in terms of integrity, confidentiality, and availability. Following are representative instances of these bugs:

- **Information Leakage:** When asynchronous reset occurs, the module fails to erase the value of registers containing plaintext and keys during cryptographic computation, making these values accessible to unauthorized processes.
- **Data Integrity:** This bug leads to failure of correct address range check for read/write requests after an asynchronous reset. As memory address registers are not cleared properly, the attacker can exploit such events to get unauthorized access to the protected region of the memory and perform illegal read and write operations.
- **Privilege Mode:** The goal is for the state machine controlling the privilege specifications of the RISC-V cores to fail to switch between the default modes *i.e.*, User, Supervisor, and Machine modes. We insert incorrect privilege switches during asynchronous resets in the RISC-V cores resulting in the privilege level register to be assigned with an undefined value, which leads to a fatal functionality error caused by no available privilege level.

All the inserted bug instances are culled from real security vulnerabilities in commercial SoCs, but sanitized and reconfigured to be applicable to ClusterSoC and AutoSoC. Finally, we create different SoC variants by randomly injecting a subset of bugs. The result is three variants of ClusterSoC and two variants of AutoSoC as shown in Table IV.

C. Evaluation Methodology and Results

We evaluated SoCCAR following a red-team/blue-team approach. The red team was responsible for determining the bugs, developing the bug insertion methodology, and creating the different SoC variants. The blue team was responsible for designing the SoCCAR infrastructure. To ensure fairness in evaluation, no communication was made between the red to blue team regarding the description of bugs, IP classes, or the number and types of bugs inserted at different IPs. Correspondingly, no communication was made from blue to red team on the architecture and implementation of SoCCAR.

SoCCAR was run on the three variants of ClusterSoC and two variants of AutoSoC. Note that each variant has multiple bugs inserted at different IPs. SoCCAR detected the bugs in all variants of ClusterSoC; for AutoSoC, SoCCAR detected all bugs other than one, an information leakage bug in the SHA256 crypto core in Variant#2. Verification time for all instances lasted only a few seconds.

It is illustrative to explain the reason for the failure of SoCCAR on the SHA256 bug in AutoSoC Variant#2. Roughly, the bug is caused by an incorrect procedure block declaration: instead of the cipher assignment operations being executed during regular operation, the buggy IP would cause them to be executed only under an asynchronous reset that was composed with a specific clock edge. This resulted in an RTL construct where the asynchronous reset could not be detected as an explicit governor for the operations in the block during RTL analysis for construction of the AR_CFG. The lesson from the experience is that SoCCAR CFG extraction implementation needs to be extended to provide more refined comprehension of the RTL constructs and in particular the interplay of clock and asynchronous resets to create implicit governors. However, we view this as a weakness of the current tool implementation rather than a conceptual weakness of SoCCAR. Interestingly, given that the procedure block becomes inactive under normal (non-reset) execution, this violation can be detected by standard functional validation.

VI. Related Work

Hardware security validation is a broad area of research with several different branches [3]. One line of research closely related to our work is detection of information flow violations. Research in this area has included the use of formalisms, annotations, and type systems to develop provably secure designs [6], [7], the use of formal methods to verify correctness of information flow policies as well as certify integrity of RTL designs against such policies [8], [9].

There has also been significant research re-purposing techniques from formal methods for directed test generation for RTL verification [10], [11]. However, with increase in design size and complexity, these techniques suffer from state space explosion [12]. Recently, concolic testing has provided a promising path for generating directed

Table III
Summary of Security Bugs

Violation Type	Trigger Condition	Payload	Impact
Information Leakage	Async. reset at crypto engine	Uncleared values of plain text and crypto keys at internal registers.	Leakage of secret asset *i.e.*, unencrypted plain text can be retrieved by an attacker via cipher text port. This violates the confidentiality property of secure assets of the SoC design.
Loss of Data Integrity	Async. reset at memory module	Failure of address range check for subsequent read/write requests.	Unauthorized access of read and write operations to secure memory regions. This violates the integrity as well as confidentiality property of on-chip assets.
Unavailability of Privilege Modes	Async. reset at processor core	Processor privilege mode stuck at current state of operation.	Failure to switch between privilege modes. This violates the availability property of critical system functionality.

Table IV
Security bugs inserted in various variants of both SoCs

Violation Type	ClusterSoC Bug IP Location			AutoSoC Bug IP Location	
	Variant #1	Variant #2	Variant #3	Variant #1	Variant #2
Bug #1 Information Leakage	MD5, AES192	-	AES192, SHA256	MD5, SHA256	AES192
Bug #2 Data Integrity	SRAM	SRAM	Wishbone Bus	SRAM	-
Bug #3 Privilege Mode	-	RV32I	RV32E	RV32IC, RV32IM	RV32IM

tests through symbolic exploration. Concolic testing has been successfully employed in both software and hardware domains [5], [13]. Automatic concolic test generation approaches have also been studied to improve the coverage on large-scale designs [14], [15]. Recently, a concolic testing-based approach was developed for security validation of an IP core [16].

With increasing adoption of SoC designs including multiple clock and reset domains, there has been effort on verification of clock-domain crossing (CDC) and reset-domain crossing (RDC) primarily targeted towards design sign-off [17]. There has also been work on fault detection and mitigation approaches of CDC [18], [19]. However, to our knowledge there has been no previous research on systematic analysis of security impacts of these features.

VII. Conclusion

With increasing use of complex SoCs having multiple reset domains, security validation must account for violations arising from unpredictable system behavior caused by asynchronous resets. These violations can occur on very rare corner cases and represent some of the most hard-to-detect bugs in current industrial practice. To the best of our knowledge, our framework SoCCAR represents the first systematic approaches for detecting such violations; it uses CFG extraction together with concolic testing to systematically explore violations caused by asynchronous resets. We have designed a comprehensive experimental testbed to evaluate the scalability and effectiveness of SoCCAR on realistic SoCs. Our results indicate that SoCCAR provides almost perfect detection accuracy with verification time of a few seconds. Furthermore, SoCCAR works directly on the RTL implementation of complex SoCs without requiring any manual abstraction. Finally, the testbed developed as part of the evaluation can serve as representative testbed for future research on SoC validation.

In future work, we will extend SoCCAR on other asynchronous events, including violations due to analog/mixed-signal inputs, malformed sensory or cyber-physical data, etc. We will also evaluate SoCCAR on commercial SoC designs.

References

[1] Cadence, "JasperGold Formal Security App," www.cadence.com.
[2] Synopsys Inc., "VCFormal Formal Security App," www.synopsys.com.
[3] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-Chip Security Verification and Validation*. Springer, 2019.
[4] S. Ray *et al.*, "System-on-Chip Platform Security Assurance: Architecture and Validation," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 21–37, 2018.
[5] K. Sen *et al.*, "Cute: a concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.
[6] D. Zhang, "A hardware design language for timing-sensitive information-flow security," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 503–516, 2015.
[7] X. Li *et al.*, "Caisson: a hardware description language for secure information flow," in *ACM Sigplan Notices*, vol. 46, no. 6, 2011, pp. 109–120.
[8] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *VLSI Test Symposium (VTS)*, 2012, pp. 252–257.
[9] M.-M. Bidmeshki *et al.*, "Information flow tracking in analog/mixed-signal designs through proof-carrying hardware ip," in *DATE*, 2017, pp. 1707–1712.
[10] 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.
[11] 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.
[12] M. Chen *et al.*, *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.
[13] A. Ahmed *et al.*, "Directed test generation using concolic testing on rtl models," in *DATE*, 2018, pp. 1538–1543.
[14] G. Li *et al.*, "Gklee: concolic verification and test generation for gpus," in *ACM SIGPLAN Notices*, vol. 47, no. 8, 2012, pp. 215–224.
[15] K. Cong *et al.*, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *IEEE ICCAD*, 2013, pp. 303–310.
[16] R. Zhang *et al.*, "End-to-end automated exploit generation for validating the security of processor designs," in *IEEE MICRO*, 2018, pp. 815–827.
[17] P. Ashar, "Static Verification Based Signoff: A Key Enabler for Managing Verification Complexity in the Modern SoC," in *FMCAD*, 2013.
[18] N. Karimi and K. Chakrabarty, "Detection, diagnosis, and recovery from clock-domain crossing failures in multiclock socs," *IEEE TCAD*, vol. 32, no. 9, pp. 1395–1408, 2013.
[19] P. Yeung *et al.*, "Multi-domain verification: When clock, power and reset domains collide," in *DVCon*, 2015.