SEVNOC: Security Validation of System-on-Chip Designs with NoC Fabrics

Xingyu Meng[1], Kshitij Raj[2], Sandip Ray[2] and Kanad Basu[1]

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

***Abstract*—Modern System-on-Chip (SoC) designs include a variety of Network-on-Chip (NoC) fabrics to implement coordination and communication of integrated hardware intellectual Property (IP) blocks. An important class of security vulnerabilities involves a rogue hardware IP interfering with this communication to compromise the integrity of the system. Such interference includes message mutation, misdirection, delivery prevention, or IP masquerading, among others. In this paper, we propose a scalable RTL-level SoC validation scheme, SeVNoC, for systematic detection of security violations in inter-IP communications for SoC designs with NoC fabrics. Given a target security property to be validated, SeVNoC entails extraction of the control flow graph of the relevant SoC, which is analyzed through a security property-based model comparison, without incurring state space explosion. Our experiments on full scale realistic SoC designs with multiple IPs and NoC architecture indicate that SeVNoC detects security violations in NoC communications with near perfect accuracy, within only a few minutes.**

***Index Terms*—SoC validation, Network-on-Chip, Security**

I. INTRODUCTION

Modern Internet-of-Things (IoT) systems are built on System-on-Chip (SoC) architecture, which encompass a plethora of functionalities, while providing optimized performance and overhead. SoC designs include pre-designed hardware modules, — referred to as *Intellectual Properties* or IPs, — that coordinate through on-chip communication fabrics to realize system functionality. SoC security validation has manifested as a critical bottleneck in IoT system design. Security vulnerabilities in SoCs often result from corner-case or rare scenarios, which are challenging to ascertain using existing industrial validation frameworks. Information flow violation, which involves transmitting of sensitive information to an untrusted entity, is one of the most precarious issues in a SoC. Existing research to ameliorate this issue focuses on only a single core, without scalability to a commercial-scale SoC consisting of multiple IPs.

Many modern SoC designs include network-on-chip (NoC) fabrics to implement inter-IP communications. NoCs realize on-chip communication with a collection of routers connected in a topology customized for the target SoC, and can enable integration of diverse, heterogeneous hardware Intellectual Property (IP) blocks with a variety of interfacing protocols [1], [2]. NoCs have become popular in recent years, as it is getting increasingly difficult to scale traditional crossbar and bus-based communications with the increasing number of IPs. Unfortunately, NoCs can also induce subtle security vulnerabilities that can compromise the integrity of the entire system by launching several attacks, *e.g.*denial-of-service, malfunction, leaking sensitive information, etc. For example, a Hardrware Trojan planted in an NoC can snoop transactions and leak sensitive data [3]. One of the primary challenges of detecting such a vulnerability is the fact that any number of IPs can participate in a communication utilizing the NoCs, and any one or multiple of them can collude together to inflict this malice. Moreover, as explained in Section III, even an IP which is not part of the communication can lead to a denial-of-service attack. Therefore, it is insufficient to simply analyze the IPs engaging in a communication or knowledge of the communication protocol to secure the SoC using static or dynamic validation methodologies. Hence, it is crucial to develop robust verification techniques to detect security compromises through NoC interconnects.

In this paper, we develop a framework for systematic detection of security violations in SoC designs resulting from vulnerabilities in NoC communication. In contrast to existing formal or simulation-based methodologies, we propose a semi-formal technique based on symbolic analysis. Unlike formal verification frameworks, the proposed method does not suffer from state space explosion, while providing sufficient guarantees for detecting corner-case vulnerabilities. Our framework, SEVNOC, consists of a novel algorithm for efficiently extracting a Control Flow Graph (CFG) of the design that enables efficient analysis of security properties through state exploration. The extraction procedure can avoid state explosion while obviating coarse abstractions. A unique feature of SEVNOC is the breadth of the attack scope, *e.g.*, the same framework applies to the swath of NoC adversaries, encompassing message mutation, IP masquerade, delivery prevention, etc. We evaluated SEVNOC using an experimental testbed with two different SoC designs, each instrumented systematically with a number of NoC vulnerabilities. The SoCs included a variety of realistic features and involved tens of thousands of lines of RTL code. SEVNOC could detect all but one of the vulnerabilities and completed in a couple of minutes. The paper makes the following important contributions:

- For the first time to our knowledge, we develop an infrastructure for RTL-level SoC security validation that systematically handles security vulnerabilities in untrusted inter-IP communications.
- Our CFG extraction algorithm provides a unique approach for addressing the trade-off between accuracy and scalability needs in SoC security validation in practice. The communication CFG for each module is generated and connections between these CFGs of various modules are explored.
- We performed extensive experimental evaluation of SEVNOC on multiple realistic NoC designs with several variants including multiple subtle vulnerabilities.
- We developed a comprehensive experimental testbed that includes multiple realistic SoC designs as well as a systematic methodology for inserting bugs on NoC communications. Aside of our own evaluation, the testbed can serve as a flexible experimental platform for evaluating other related research on SoC security.

The remainder of the paper is organized as follows. Section II discusses the background of the security validation and NoC designs. Section III discusses the SoC security challenges induced by NoC communications. We discuss the proposed SEVNOC design in Section IV. Section V-A discusses our experimental testbeds, SoC implementations and the experimental results. We discuss related work in Section VI and conclude the paper in Section VII.

II. Background

A. Security Validation in Research and Practice

Security validation of hardware designs entails 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 [4]–[6]. Nevertheless, security validation remains a complex and expensive process in industrial practice. In particular, many security bugs are corner-case scenarios, possibly activated through on-chip communications or asynchronous events [7], which are difficult to stimulate through either dynamic or formal tools. Consequently, industrial flows today depend critically on human expertise to identify such scenarios, perform white-box intrusion testing, or design manual abstraction to enable efficient application of analysis tools [8].

A critical area of security validation relevant to our work is information flow analysis, which entails checking whether malicious or unauthorized agents can access sensitive assets in the design. There has been significant recent progress in both static and dynamic approaches to this problem. Static and formal tools typically employ path sensitization, *i.e.*, identifying if there is any design path from a sensitive asset location to an untrusted IP or output [9]–[13], while dynamic tools target tracking the "flow" of an asset during system execution [14]–[19]. However, both static and dynamic approaches have targeted microcontrollers or IP cores, *e.g.*, typically RTL or netlist models with single clock, no inter-IP communication. To our knowledge, there is currently no automated framework for systematically detecting information flow violation among IPs in a SoC with realistic features.

B. Symbolic Analysis

Formal or symbolic validation entails developing mathematical analysis tools to prove that a system satisfies specification. When applicable, this approach can give a much stronger assurance than dynamic validation or testing, and can capture subtle design corner cases, which can be typically missed in simulation. Since security assurance solutions require exploring such corner cases, there has been significant effort to develop such solutions based on formal analysis. Formal security verification traditionally use interactive theorem proving [20]–[22], which obviously requires significant human expertise and is difficult to scale. Recent research adopted the same for state space exploration, particularly in the context of information flow, as mentioned above. This has resulted in several commercial tools for security validation [6], [23]. Formal tools in their classic form tend to suffer from state explosion [24]. Recently, there have been approaches to symbolic testing for security validation [25], which exploits the benefits of formal but re-targets it for test generation and largely avoids state space explosion at the cost of being incomplete. Recent research on deploying concolic testing directly on RTL models has been proposed [26], [27] to verify the functionality of the design. However, these approaches overlook the security vulnerabilities that might manifest in complex SoC designs, consisting of multiple IPs and interconnection fabrics.

C. NoC Interconnects

A Network-on-chip (NoC) realizes the system-level coordination of IPs in an SoC design through message-based communication. An NoC includes a collection of routers that are organized into a *routing topology*. NoC topologies in practice typically constitute tree, mesh, or cycle structures, although other more complex topologies are possible [28]. Routers in an NoC typically include configurable routing tables that map the destination of a message packet to an output port; router functionality entails transferring an input packet (which is typically delivered asynchronously to its ingress queue) to an output port (where it is deposited to an egress queue for asynchronous processing) based on the routing table map. The map can be reconfigured if necessary by the operating system under certain circumstances during execution (*e.g.*, on detection of congestion or transmission failure in certain paths). The configurability of routing tables also enables the reuse of the same router IP to realize different network topologies,

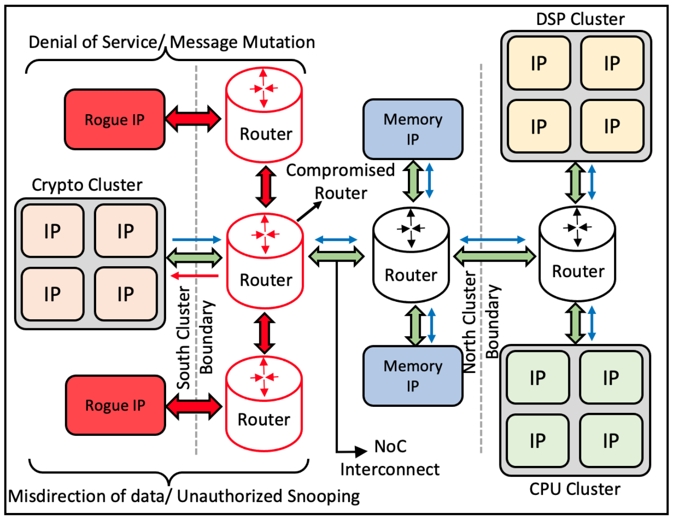


Fig. 1. Attack Surface in NoC-based System-on-Chip Designs. The compromised components can now include routers and communication links in addition to buggy or rogue IPs.

III. Security Challenges in NoC Fabrics

An unfortunate upshot of adoption of NoC fabrics is the introduction of new classes of vulnerabilities that target communication of messages during their transmission through the on-chip network [29]. As shown in Fig. 1, the vulnerable components on NoC-based SoCs include malicious and vulnerable routers and communication links, in addition to buggy or rogue IPs. This expansion of the attack surface can result in subtle security bugs that can undermine the entire system. Furthermore, many of the rogue elements can collude to undermine system integrity. Consider the following two examples. Albeit simplified, these are sanitized versions of security bugs in industrial SoCs that escaped to silicon.

Example 1: Assume that a router R is connected to a CPU which can re-configure the routing table of R (*e.g.*, to support in-field update). A bug (or malicious Trojan) implanted in the CPU microarchitecture causes this functionality to be invoked under a certain rare-to-detect corner case, enabling malicious re-configuration of the destination fields of the routing table in R. In a subsequent boot, use of this maliciously updated routing table results causes R in the transmission of a secure cryptographic key (*e.g.*, master key) through an untrusted link, compromising the integrity of the entire SoC.

Example 2: Under a certain (rare) scenario, a rogue IP I continually transmits garbled or meaningless (but not malformed) messages across a communication link to a router R after T = t_f. Fig. 2 illustrates this attack. Although each message is discarded by R, the ingress queue of R and the incoming communication link are

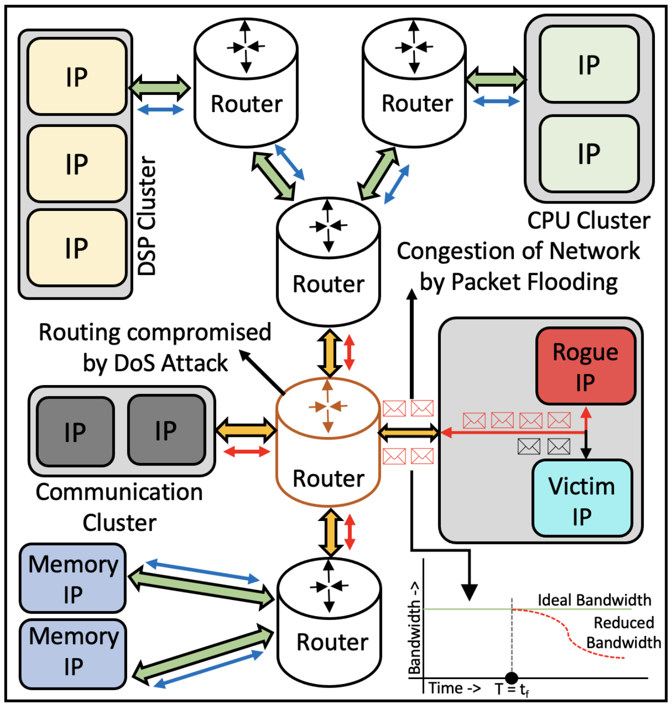


Fig. 2. Denial-of-Service Attack on NoC Caused by a Rogue IP. Here the router is benign. A rogue IP floods the link with irrelevant messages preventing transmission of communication from the victim IP. The graph shows how the available bandwidth of the link decreases for the victim IP as a result of the attack.

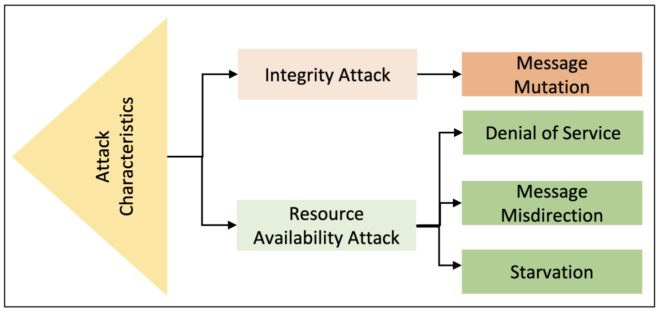


Fig. 3. Taxonomy of attacks on NoC Fabrics

overloaded by this transmission. Another IP V (*i.e.*, a victim IP) in the same subsystem is unable to transmit messages through the communication link or through R itself. This leads to a reduction in available bandwidth on the channel, as shown in the graph in Fig. 2. An upshot of this interruption is that any IP B waiting for a message from V can wait forever. A cascading sequence of such operations can eventually lead to system deadlock. Even if the deadlock is broken (*e.g.*, by a non-deterministic external event or through I ceasing transmission), it still leads to a significant degradation in performance.

Note that such vulnerabilities would be easy to detect if the potential trigger conditions for the malicious activity were known in advance. However, in the absence of such knowledge, it is difficult to develop a systematic methodology for detecting such violations. In particular, virtually any usage scenario entails a variety of communications across multiple fabrics, any of which can be subverted by a malicious IP or router, resulting in a compromised or non-functioning system. Furthermore, the malicious component may not even be one of the IPs taking part in any of the usage scenarios. As illustrated by both the examples above, a malicious IP can corrupt or flood a communication link or routing table *asynchronously*, with respect to the system functionality. Consequently, it is not possible to simply analyze the usages or communication protocols participating in the inter-IP coordination across the NoC in realizing a specific system-level use case; one must additionally account for any other IP with the ability to communicate through the same routers or links. This is difficult to do for both dynamic and formal validation. For dynamic validation, a rare activation scenario (or trigger of the activity) implies that it is unlikely to be hit in random simulation. For formal methods, identification of a violation scenario typically reduces to symbolic exploration of the entire SoC design, which is clearly limited by the well-known state explosion problem. Thus, it is imperative to develop alternate techniques for addressing these issues.

Threat Model and Attack Taxonomy: The focus of the paper is on system-level interaction. The threat model considered consequently targets adversaries that can corrupt inter-IP communications in SoC designs; we do not consider corruption or confidentiality breach inside an IP that does not impact its interaction with the rest of the system. Fig. 3 provides a taxonomy of NoC attacks explored in this paper. The attack taxonomy is inspired from previous work by Deb Nath *et al.* [29] on an architecture-level solution to SoC designs with NoC fabrics using security policy implementation in a centralized security-policy engine. As with that work, the adversaries here focus on *effect* of an adversarial activity on the communicated messages, rather than the *mechanism* of the activity. For example, consider e a message m destined towards an IP D at any router R. The effect of adversarial activity on m can be classified as one of three categories: (1) mutation of the payload; (2) prevention of its delivery (*e.g.*, through flooding); or (3) misdirection of m towards a different IP D'. Note that the *mechanism* through which each of these effects can be implemented can include a large and diverse set of different actions, *e.g.*, the misdirection is possible by either corrupting the destination ID of the message, or a malicious IP D' masquerading as D.

Remark 1: This paper evaluates SeVNoC on SoCs with NoC fabrics and the threat models illustrated hold true for NoC-based fabrics. While the threat models associated with bus-based systems will change, the overall methodology and the operation of SeVNoC remains the same and it is feasible to quote that this validation framework can detect vulnerabilities in bus-based systems as well. As long as the security properties are defined in a way that SeVNoC can parse it and extract the relevant CFGs, detecting such vulnerabilities is possible.

IV. SeVNoC Methodology

We assume a set $\mathcal{SP}$ of events, which contains the relevant communication events $\mathcal{E}[\mathcal{M}_s]$. Let the SoC design $\mathcal{N}$ constitute a list of modules $\langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$, where each $\mathcal{M}_i$ contains a sequence of *process blocks* $\mathcal{F}[\mathcal{M}_i]$. Informally, these process blocks consist of unspecified events and assignments. For each function $f \in \mathcal{F}[\mathcal{M}_i]$ and event trigger $s \in \mathcal{F}[\mathcal{M}_i]$, we say that s governs f if f is executed only when s is true. We define a projection $\mathcal{P}_v$ of the Control Flow Graph $[\mathcal{M}_i]$ in module $\mathcal{M}_i$ that includes the event statements containing signal v. We refer to $\mathcal{P}_v$ as the *under-testing CFG* of $\mathcal{M}_i$ with v. The notion of governing CFG naturally extends to a set of signals provided by $\mathcal{SP}$ as $V \triangleq \{v_1, \ldots, v_n\}$, by defining $\mathcal{P}_\mathcal{V} \triangleq \bigcup_{i=1}^{n} \mathcal{P}_{v_i}$. If the set $\mathcal{V}$ is in the $\mathcal{F}[\mathcal{M}_i]$, we refer to the *under-testing CFG* $\mathcal{P}_\mathcal{V}$ as *the Communication Event CFG* (CE_CFG) of $\mathcal{M}_i$. We use $\mathcal{CE}[\mathcal{M}_i]$ to denote the CE_CFG of $\mathcal{M}_i$. The communication event

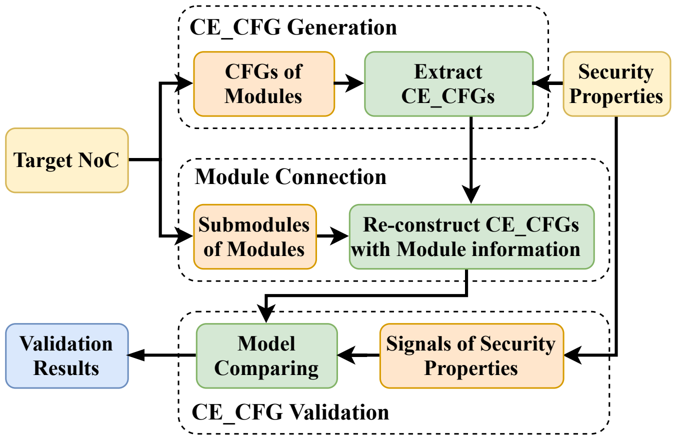


Fig. 4. SEVNOC Framework.

Algorithm 1 Communication Event *CFG* Generation

```
Input: $\mathcal{N}$, $\mathcal{SP}$
Output: $\mathcal{CE}[\mathcal{M}_i]$ of $\mathcal{M}_i$
 CFG Generation($\mathcal{N}$)
 1: for all IPs $\in \mathcal{N}$, do
 2:     Initialize $[\mathcal{M}_i]$, $[\mathcal{M}_i]$ append $\mathcal{M}_i$
 3:     for all $\mathcal{F}[\mathcal{M}_i]$s in $\mathcal{M}_i$, do
 4:         if $s$, $f$ in $\mathcal{F}[\mathcal{M}_i]$, then
 5:             $\mathcal{P}_{v_i} \leftarrow s$, $f$, then $[\mathcal{M}_i]$ append $\mathcal{P}_{v_i}$
 6:     end for
 7: end for
CE_CFG Extraction($[\mathcal{M}_i]$, $\mathcal{SP}$)
 1: Initialize $\mathcal{CE}[\mathcal{M}_i]$
 2: for all in $\mathcal{SP}$, do
 3:     for all $\mathcal{P}_{v_i}$s in $[\mathcal{M}_i]$, do
 4:         if $s, f \in \mathcal{P}_{v_i}$ in $\mathcal{E}[\mathcal{M}_s]$, then
 5:             $\mathcal{CE}[\mathcal{M}_i]$ append $\mathcal{P}_{v_i}$, $\mathcal{E}[\mathcal{M}_i]$
 6:     end for
 7: end for
 8: $\mathcal{CE}[\mathcal{M}]$ append $\mathcal{CE}[\mathcal{M}_i]$
```

CFG of $\mathcal{S}$ is then naturally defined as the interactive composition $\mathcal{CE}(\mathcal{S}) \triangleq \mathcal{CE}[\mathcal{M}_1] \,||\, \mathcal{CE}[\mathcal{M}_2] \,||\, \ldots \,||\, \mathcal{CE}[\mathcal{M}_k]$.

SEVNOC works by computing $\mathcal{CE}(\mathcal{S})$ which can then be efficiently explored through symbolic analysis, and no additional meta-data apart from the RTL and security properties are needed. Fig. 4 shows the overall framework of SEVNOC. It includes the following three components:

1) For each module $\mathcal{M}_i$ of $\mathcal{N}$, construct the CE_CFG $\mathcal{CE}[\mathcal{M}_i]$ by analyzing the CFG of $\mathcal{F}[\mathcal{M}_i]$ to identify whether process blocks contains signal v in security properties $\mathcal{SP}$
2) Assemble and connect the individual CE_CFGs to create the *under-testing CFG* $\mathcal{CE}(\mathcal{N})$. This is done by computing and developing connection profiles of all constituent modules.
3) Given the CE_CFG $\mathcal{CE}(\mathcal{N})$, model comparison is used to systematically verify the correctness of communication events.

A. CE_CFG Generation

Algorithm 1 generates the abstract CFG. It scans for all the process blocks $\mathcal{F}[\mathcal{M}_i]$. The trigger condition s and its execution f are extracted from each process block and added into the CFG as a *subCFG* of the module. Note that in RTL we can insert procedural blocks (*always @*) between each *subCFG* to form a complete CFG of the module $\mathcal{CE}[\mathcal{M}_i]$. Subsequently, the CFG of the module is checked to ensure whether it contains any signal v_i defined in $\mathcal{SP}$. The algorithm can then extract $\mathcal{P}_{v_i}$ from $\mathcal{CE}[\mathcal{M}_i]$ according to the extracted signal $V \triangleq \{v_1, \ldots, v_n\}$ from the security properties. The algorithm will process all the *subCFG*s and select the $\mathcal{P}_{v_i}$ to develop a new *CE_CFG* $\mathcal{P}_v$ for the module. All modules in the SoC are processed and associated with their communication event *CE_CFG* into a new $\mathcal{CE}[\mathcal{M}_i]$. Figure 5 shows the overall framework of two processes in Algorithm 1.

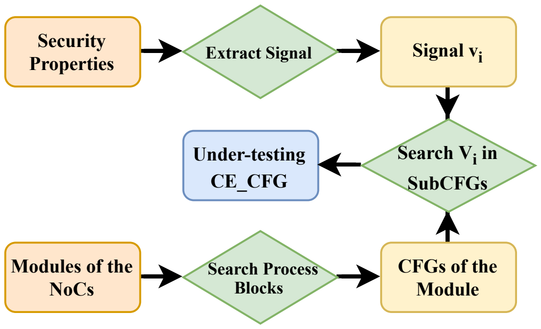


Fig. 5. Algorithm 1 Process Flow of Extraction

Algorithm 2 Module Connection Profile

```
Input: $\mathcal{N}$, $\mathcal{IP}$s
Output: $\mathcal{CP}[\mathcal{M}]$
 1: Initialize $\mathcal{A}[\mathcal{IP}]$
 2: for all $\mathcal{IP}$s in $\mathcal{N}$, do $\mathcal{A}[\mathcal{IP}]$ append $\mathcal{IP}$
 3: Initialize $\mathcal{A}[\mathcal{M}]$
 4: for all $\mathcal{IP}$ in $\mathcal{A}[\mathcal{IP}]$, do
 5:     if $\mathcal{M}_x$ in $\mathcal{IP}$, then $\mathcal{A}[\mathcal{M}]$ append $\mathcal{M}_x$
 6: end for
 7: Initialize $\mathcal{CP}[\mathcal{M}]$
 8: for all $\mathcal{M}_x$ in $\mathcal{A}[\mathcal{M}]$, do Initialize $\mathcal{CP}[\mathcal{M}_i]$
 9:     if $\mathcal{M}_x$ in $\mathcal{M}_i$, then $\mathcal{CP}[\mathcal{M}_i]$ append $\mathcal{M}_x$
10: end for
11: $\mathcal{CP}[\mathcal{M}]$ append $\mathcal{CP}[\mathcal{M}_i]$
12: for all $\mathcal{CE}[\mathcal{M}_i]$ in $\mathcal{CE}[\mathcal{M}]$, do
13:     if $\mathcal{M}_x \in \mathcal{CP}[\mathcal{M}_i]$, then $\mathcal{CE}[\mathcal{M}_i]$ append $\mathcal{CE}[\mathcal{M}_x]$
14: end for
```

B. Module Connection with CE_CFG

Algorithm 2 explores each $\mathcal{M}_i$ in NoC $\mathcal{N}$ to search for the sub-modules and complete the $\mathcal{CE}[\mathcal{M}_i]$. To achieve this, we list all the $\mathcal{M}_i$s as $\mathcal{A}[\mathcal{M}]$ and generate a connection profile for each $\mathcal{M}_i$ to determine which sub-modules are included in its design flows. The structure of each module is scanned by the algorithm to collect all the sub-modules' invocations and logistic information. A list $\mathcal{CP}[\mathcal{M}_i]$ of each module is created to identify each $\mathcal{M}_x$ that is invoked by the design flow of $\mathcal{M}_i$. The inter-connected $\mathcal{M}_x$s and its associated $\mathcal{CE}[\mathcal{M}_x]$ are then assembled to construct a complete $\mathcal{CE}[\mathcal{M}_i]$, according to the connection profile of the module. The CFG of top module $\mathcal{CE}[\mathcal{M}_i]$ will be updated with all the $\mathcal{CE}[\mathcal{M}_x]$, where its module $\mathcal{M}_x \in \mathcal{CP}[\mathcal{M}_i]$.

C. Model Comparison with Security Properties

To verify the correctness of the generated CFGs $\mathcal{CE}[\mathcal{M}_i]$, we create the model $\mathcal{M}_s$ of each specification in $\mathcal{SP}$. Algorithm 3 identifies the trigger conditions s and its executions f from each

Algorithm 3 Module Comparing

```
Input: $\mathcal{CE}(\mathcal{N})$, $\mathcal{SP}$s
Output: Invalid Message, $\mathcal{M}_{\S}$
1: Initialize $[\mathcal{M}_s]$
2: for all Specifications in $\mathcal{SP}$, do
3:     if $s$, $f$ in $\mathcal{F}[\mathcal{M}_i]$, then
4:         $\mathcal{P}_s \leftarrow s$, $f$, and $[\mathcal{M}_s]$ append $\mathcal{P}_s$
5:     else if $f$ in $\mathcal{F}[\mathcal{M}_i]$, then
6:         $\mathcal{P}_s \leftarrow f$, and $[\mathcal{M}_s]$ append $\mathcal{P}_s$
7: end for
Model Comparison($\mathcal{CE}[\mathcal{M}]$, $[\mathcal{M}_s]$)
1: for all $\mathcal{CEM}_i$ in $\mathcal{CE}[\mathcal{M}]$, do
2:     for all $\mathcal{P}_{v_i}$ in $\mathcal{CE}[\mathcal{M}_i]$, do
3:         for all $\mathcal{P}_s$s in $[\mathcal{M}_s]$, do
4:             Invalid = Comparing_model($\mathcal{P}_{v_i}$, $\mathcal{P}_s$)
5:             if Invalid == 1, then Print($\mathcal{P}_{v_i}$, $\mathcal{M}_x$)
6:         end for
7:     end for
8: end for
```

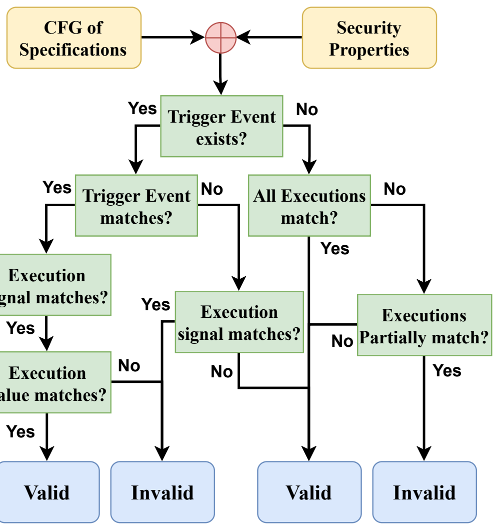
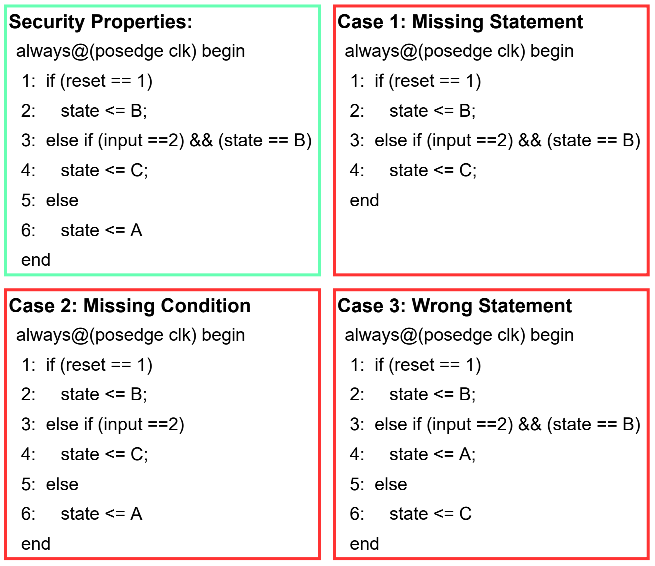


Fig. 6. Comparing Process Flow of Algorithm 3

specification, and transforms into a CFG model $\mathcal{M}_s$. When s is available in the specification, the process of generating the CFG is identical to Algorithm 1. When s is not available in the specification, f will be isolated to establish which executions should be computed simultaneously and which should not happen. $[\mathcal{M}_s]$ can be used to validate the model of CFG generated for each module $\mathcal{CE}[\mathcal{M}_x]$ by comparing whether the CFG model matches the security properties. The details of the comparison are shown in Fig. 6. Once all CFGs in $\mathcal{CE}[\mathcal{M}]$ are checked through the model comparison process, we can evaluate the invalid message returned from Algorithm 3; the message will contain the *subCFG*s and $\mathcal{M}_x$ of the $\mathcal{CE}[\mathcal{M}_x]$s that violate the $\mathcal{M}_s$. If no invalidation message is obtained, then no property from $\mathcal{SP}$ is violated.

To demonstrate the process of Algorithm 3, we include an example model process block and three cases that will violate the pre-defined security property model, as shown in Fig. 7. The original process block consists of three if-else statements and their assignments. The register *state* is assigned as *B* when *reset == 1*, as *C* when *input == 1 and state == B*, and as *A* default. Now, we will discuss the three possible scenarios.



Security Properties:

```
always@(posedge clk) begin
1: if (reset == 1)
2:    state <= B;
3: else if (input ==2) && (state == B)
4:    state <= C;
5: else
6:    state <= A
end
```

Case 1: Missing Statement

```
always@(posedge clk) begin
1: if (reset == 1)
2:    state <= B;
3: else if (input ==2) && (state == B)
4:    state <= C;
end
```

Case 2: Missing Condition

```
always@(posedge clk) begin
1: if (reset == 1)
2:    state <= B;
3: else if (input ==2)
4:    state <= C;
5: else
6:    state <= A
end
```

Case 3: Wrong Statement

```
always@(posedge clk) begin
1: if (reset == 1)
2:    state <= B;
3: else if (input ==2) && (state == B)
4:    state <= A;
5: else
6:    state <= C
end
```

Fig. 7. Examples of correct and buggy process blocks. The first one is the correct flow and the next three are manifested with certain bugs.

1) ***Case I-Missing Statement:*** In this case, the default condition is missing, which will result in undesired functionalities. When the buggy model is checked through Algorithm 3, it will match the code in line 1, 2, 3, and 4. However, when checking line 5 and 6, Algorithm 3 will follow the process: Trigger Event $\xrightarrow{\text{No}}$ All Execution Match $\xrightarrow{\text{No}}$ Executions Partially Match $\xrightarrow{\text{Yes}}$ Invalid. Hence, the algorithm detects the missing statement in the model.
2) ***Case II-Missing Condition:*** In this case, the condition (state == B) in line 3 is missing, which might result in a bypass of the privilege level. The under-testing model will match line 1 and 2. However, when checking line 3, Algorithm 3 will conduct the process: Trigger Event $\xrightarrow{\text{Yes}}$ Trigger Event Matches $\xrightarrow{\text{No}}$ Execution Signal Matches $\xrightarrow{\text{Yes}}$ Invalid. Hence, the algorithm will detect the incorrect condition in this model.
3) ***Case III-Wrong Statement:*** In this case, the assignments of line 4 and line 6 are swapped, which will cause an incorrect functionality. When the model is checked by Algorithm 3, it will match the first statement. However, when checking the second statement, Algorithm 3 will execute the process: Trigger Event $\xrightarrow{\text{Yes}}$ Trigger Event Matches $\xrightarrow{\text{Yes}}$ Execution Signal Matches $\xrightarrow{\text{Yes}}$ Execution Value Matches $\xrightarrow{\text{No}}$ Invalid. Hence, the algorithm will detect the incorrect assignment in the model.

V. Experimental Evaluation

A. SoC Benchmarks

A key challenge in research on SoC security validation is the lack of an appropriate testbed for evaluation. Evaluation of SeVNoC requires SoCs with multiple communication fabrics, disparate IP cores, asynchronous communication, etc. Unfortunately, there are no realistic open source SoCs with these features. Platforms like Openpiton and BOOM provide microprocessors, but not a framework for building SoCs with enough configurabiltiy to carry out experimental evaluations for such methodologies. To address this problem, we developed two SoC benchmark classes, NSNoC and TransNoC, that implement crucial features reflecting the complexities involved in realistic SoC designs. These SoC benchmarks have been derived using our in-house framework, SoCCom [30]. SoCCom enables SoC integration using open-source IPs and industry standard

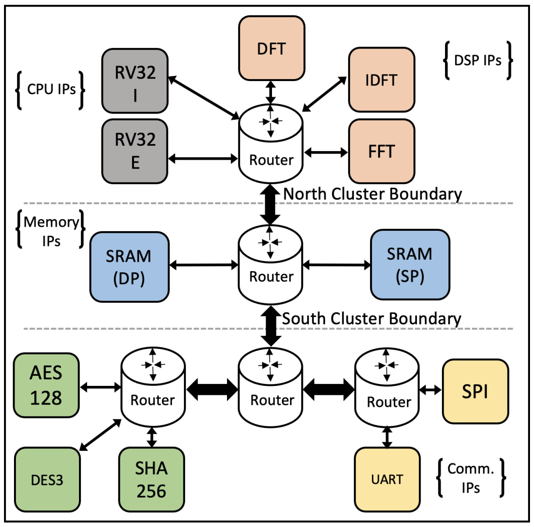


Fig. 8. NSNoC: Representative IoT SoC model. It includes one NoC realized by a tree-based topology.

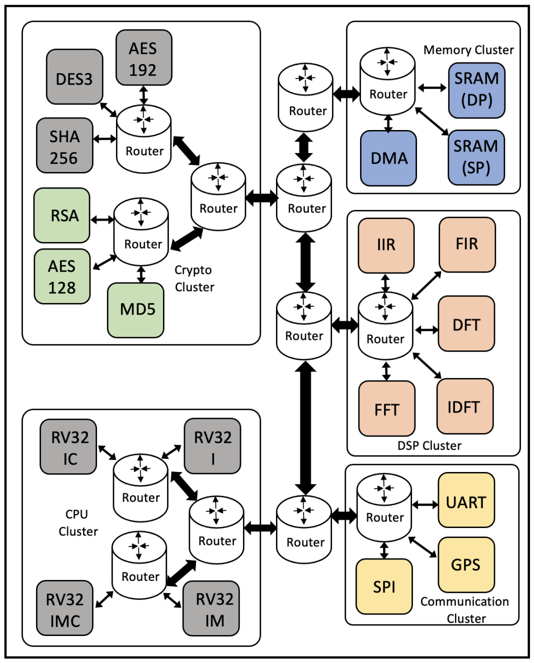


Fig. 9. TransNoC: An Automotive SoC with application specific subsystems

interfaces, supporting different interconnect fabrics and topologies. SoCs generated using SoCCom have been functionally validated against SoC integration and inter-operability of all components. Figs. 8 and 9 show high-level block diagrams for these SoCs. Our designs are inspired by our previous work [7], which introduced realistic SoC benchmarks for evaluation of information flow validation due to asynchronous resets. However, the SoCs developed in that work were bus-based. We extended them with NoC implementations and hierarchical subsystems with bus-based and NoC-based fabrics. We also introduced a mix of high-speed and low-speed IPs, different cryptographic modules, and multi-core computing systems. All IPs and routers used in designing the benchmark classes have been derived from open-source IP implementations. Note that these SoCs are not toy designs, *e.g.*, NSNoC implementation constitutes 43000 lines of (behavioral) Verilog code and TransNoC constitutes 57000 lines. Both NoCs can run realistic applications.

TABLE I
AREA METRICS OF NSNOC AND TRANSNOC

NoC Class	Variants	Area		
		LUT	LUTRAM	BRAM
NSNoC	Variant #1	19706	2951	144
	Variant #2	18137	2876	132
	Variant #3	18762	2774	142
	Variant #4	18361	2968	148
	Variant #5	19026	2815	136
TransNoC	Variant #1	41276	3192	156
	Variant #2	43162	3217	164
	Variant #3	43578	3285	168
	Variant #4	44284	3348	174
	Variant #5	40812	3098	152

NSNoC is targeted for low-power and area-efficient IoT applications. The IPs are integrated with a network adapter to form a compute-tile. It includes a RISC-V core that implements RV32I (Baseline Integer) and RV32E (Embedded Extension) ISAs. The memory subsystem comprises two RISC-V optimized variants of a single port and dual port SRAM modules that form the memory compute-tile. The North Cluster IPs are high-performance IPs comprising of the Inverse Discrete Fourier Transform (IDFT), Discrete Fourier Transform (DFT), and Fast Fourier Transforms (FFT) IPs to form the DSP subsystem. The South Cluster IPs consists of two subsystems, the crypto subsystem and the peripheral communication subsystem. The crypto subsystem consists of AES128, DES3, and SHA256 IPs for carrying out cryptographic operations. The peripheral subsystem consists of peripheral IPs like UART and SPI to enable communication with the outside world.

TransNoC has been inspired by SoC designs used in automotive applications. It features a heterogeneous hierarchical system with bus-based and NoC-based subsystems for implementing different functionalities such as DSP, crypto, computing, memory, and communication operations. The individual subsystems are interconnected using routers with different topologies. Some subsystems are internally connected via a Wishbone protocol compliant bus. TransNoC incorporates significantly more IPs than NSNoC. For instance, the computing core has been scaled up from 2 cores to 4 cores, the crypto subsystem has RSA, MD5, and DES3 IPs on top of the pre-existing IPs in NSNoC. The DSP subsystem has an additional Infinite Impulse Response (IIR) IP in its compute-tile. A GPS IP has also been integrated into the communication subsystem.

NSNoC and TransNoC benchmarks furnish a set of realistic and fully functional SoC designs. We use two variants of NSNoC and TransNoC for evaluating our security validation methodology. Each of these variants are categorised based on the class of security violation present in them. In order to provide an understanding of the design complexity of these SoC benchmarks, we performed FPGA synthesis in Xilinx Vivado. Table I shows the area statistics of these 2 classes of SoC benchmarks.

TABLE II
PLACEMENT OF SECURITY VIOLATIONS IN NSNOC AND TRANSNOC

NoC Class	**Security Violation Placement**			
NSNoC	**Message Misdirection**	**Message Mutation**	**Delivery Prevention**	**Network Congestion**
Variant #1	Crypto Cluster Router	-	-	GPS IP
Variant #2	-	Comm. Cluster Router	CPU Cluster Router	-
Variant #3	-	-	-	UART IP
Variant #4	Comm. Cluster Router	Memory Cluster Router	-	-
Variant #5	-	-	Crypto Cluster Router	-
TransNoC	**Message Misdirection**	**Message Mutation**	**Delivery Prevention**	**Network Congestion**
Variant #1	Crypto Cluster Router	-		SHA256 IP
Variant #2	-	-	Memory Cluster Router	DFT IP
Variant #3	Comm. Cluster Router	-	-	-
Variant #4	-	Memory Cluster Router	-	-
Variant #5	-	-	-	Comm. Cluster Router

B. Inserting Security Violations

To perform a thorough and sound evaluation of our proposed security validation approach, we developed a framework for systematically inserting security bugs in NSNoC and TransNoC, mimicking the realistic nature of such bugs in terms of frequency of appearance, excitation condition, and payload. Note that inserting all (or several) bugs together in the same SoC would not provide a realistic target for evaluation. The greater the number of bugs in a design, the higher the likelihood for a validation tool to hit some of them in random exploration. Consequently, if a validation tool can detect bugs in such an SoC, one cannot conclude that the tool could also successfully explore an SoC which only includes a small subset of the bugs (with the consequence that each bug would be harder to excite).

Our approach addresses this problem by developing multiple variants of each SoC, each with different classes of bugs. The key observation is that certain bugs are relevant to certain IP types, *e.g.*, a message misdirection attack by an IP requires controlling (and sometimes modifying) operations of the router attached to the IP, and typically requires a high privilege, while a flooding attack can be performed by an IP with lower privilege. For evaluating SEVNOC, we used five buggy variants of both NSNoC and TransNoC (in addition to the original bug-free one). Table II shows the IPs in which violations were inserted, and Table III provides a detailed description of each vulnerability and its system-level impact. The bugs are culled from real security vulnerabilities observed in industrial SoCs but sanitized and simplified to be applied to 10 variants of the two SoC classes as shown in Table II.

C. Evaluation and Results

We evaluated SEVNOC through a red-team/blue-team approach. The red team determined the impacts of bugs and developed the bug insertion method, while using different SoC variants. The blue team designed the validation framework and infrastructure. No information was communicated to the blue team regarding the description of bugs, IP classes, types of NoC, or the types of bugs inserted at different IPs. Correspondingly, no information was communicated to the red team regarding the implementation of SEVNOC. The number of bugs inserted at the different IPs for each variant was known only to the red team. The experiments were performed on an AMD Ryzen3 2.6GHz Dual-Core processor and 4GB RAM.

The process of CFG extraction takes less than 5 seconds to complete for both NoCs. It significantly reduces the amount of code before verification process by extracting 480 of 43000 lines in NSNoC (eight properties), and 350 of 57000 lines in TransNoC (seven properties). SEVNOC could detect the bugs in all variants of NSNoC; for TransSoC, it detected all bugs other than one. The entire validation for each variant took between 120 and 180 seconds of wall clock time to complete. Validation of NSNoC variants typically took about 120 seconds, and validation of TransNoC variants took about 180 seconds.

It is illustrative to explain the violation that could not be detected. This bug is a message misdirection bug in the communication cluster router in TransNoC #1. The bug was triggered by a specific assignment in one of the blocks, different from the event procedure block, which misdirected packets at specific intervals to a different location. This led to an RTL construct where the trigger and the event could not be mapped to each other and the extracted CFG for analysis did not contain the event and trigger together to be examined for violation. However, when the trigger and the event were included within a single procedure block (without modifying functionality of the system), the bug was easily detected by SEVNOC. The lesson from the experience is that the CFG extraction *algorithm* is capable of handling logically distant connections between events and triggers; however, generalizing the notion of module connections to account for structurally separated modules can facilitate extensibility and scope of the framework. We will explore such extended formalization in future work. When applying assertion-based verification, given the same number of security properties (15 security properties), 40 additional lines of code need to be inserted into the design. Moreover, the number of assertions will be enhanced when more security properties are involved. It will be impractical to use assertion-based runtime verification when hundreds of security properties are introduced for NoCs, since it might create substantial space overhead on the circuit designs and subvert the performance. On the other hand, simulation-based verification requires well-tuned testbenches to explore all the potential security vulnerabilities. Therefore, the

TABLE III
SECURITY VIOLATIONS

Type of Violation	**Mechanism of Attack**	**Impact**
Message Misdirection	Rogue IP with higher privilege level modifies the routing table and diverts traffic to an illegal destination	Leads to sensitive data leakage and unprivileged information access to malicious IPs .
Message Mutation	Rogue IP alters the contents of the inbound and outbound traffic on the network	Compromised data integrity and can trigger undesired response/operation from an unauthorised source. It may also trigger a streak of challenge/response to a specific destination.
Delivery Prevention	Rogue IP blocks all incoming and outgoing transmission from a particular end-point	This may lead to starvation as specific IPs may always have their packets dropped by the router and also enter a stall as acknowledgements may never reach the intended destination.
Network Congestion	Rogue IP floods the network with undesirable packets and congest the network, stopping all communication on the chip	This will cause Denial-of-Service in the system and halt normal operation. It forces the system to go into an unbounded stall. It may also lead to starvation of request access to specific IPs.

process of input generation and simulation will create a significant overhead in terms of latency. Commercial verification tools such as JasperGold took hours to completely generate all the test cases, while our framework takes a few seconds for verifying each security property.

VI. Related Work

A significant component of SoC security validation research has been on information flow analysis [9], [10]. In addition, formal methods have also been used to verify security policies and validate the correctness of the RTL for such security policies [31], [32]. Several design flows have been proposed to facilitate correct-by-construction security assurance [33].
These approaches are hard to retrofit on existing legacy IPs. Existing research has developed architectures for systematic security policy enforcement, which includes untrusted NoC communications [34]. On somewhat similar grounds, Deb Nath *et al.* [29] proposed an architecture-level solution for run-time detection of security-critical events in SoC designs with NoC fabrics using security policies implemented using a centralized security-policy engine. This, however, is an architectural feature to be invoked at run-time and not a validation technique, incurring additional integration overheads and in-field deployment. A critical component of this work involves writing fine-grained security policies for each type of NoC-induced vulnerability in the SoC, and these policies need to be revised even if one entity of the whole attack surface changes. These policies are useful in thwarting specific known IP vulnerabilities, but take a hit when it comes to the overall validation of the SoC. There has also been work on security assertions in the original RTL for detecting manifested Trojans [27].

Dynamic verification strategy to detect bugs inspired by the types of security-critical errata in the classification phase was also proposed [35]. However, this technique is limited in general-purpose functionality due to the increasing number and complexity of invariants. More recently, symbolic testing for IP security verification has been proposed [36], [37]. Finally, there has been recent work on SoC information flow validation that accounts for asynchronous events [7]. This work was applied to realistic SoC designs with multiple IP cores. However, this approach only accounted for asynchronous resets, and the SoCs considered implemented bus-based crossbar connection; security of communication fabrics was not considered.

With the proliferation of NoC fabrics in System-on-Chip designs, there has been significant research on NoC architecture, exploration, and analysis. However, we are not aware of any validation framework that accounts for security impacts of NoCs, such as message corruption, fabrication, or misdirection, as discussed in this paper. Research on NoC security validation and resilience has addressed specific attack mechanisms [3], [38]. In particular, Kumar *et al.* [39] developed a technique for runtime detection of Denial-of-Service violations in NoC by integrating traffic monitors in routers. Charles *et al.* [38] provide another solution for DoS mitigation, by flagging unintended and malicious traffic to the node and not allowing it to propagate to the entire system. There has also been work on detecting compromised communication infrastructure using special functionality in the firmware [3]. Network-based monitoring using unique hardware design probes integrated into the SoC computing tiles provides a mechanism for run-time observability of system-level security threats [40]. One pitfall to using this technique is scalability to support the increasing complexity of modern-day SoC designs with hundreds of integrated IPs, as it can increase the area and power overheads in the overall system functionality. There has also been recent work on detecting NoC-based Trojans [41], [42]. Unfortunately, none of these works provide a comprehensive RTL-level validation analysis on full-scale SoC, consisting of NoC architecture.

VII. Conclusion

We have developed an automated infrastructure, SEVNOC for security validation of SoC designs that account for corruptions and vulnerabilities arising out of NoC communications. Existing research has shown that these fabrics can be exploited to introduce vulnerabilities, resulting in malicious denouements like leaking sensitive data and denial of service. To our knowledge, SEVNOC is the first automated security validation framework that can address the spectrum of communication attacks on modern SoC designs. Our experimental results show that SEVNOC can provide near complete detection accuracy without suffering from the state explosion bottlenecks, prevalent in tools implementing symbolic analysis. In particular, for realistic SoCs involving $57,000$ lines of RTL implementation, SEVNOC could complete the validation in only a few minutes. An attractive feature of the framework is that it obviates needs for manual abstractions to enable application of the formal infrastructure: note that, SEVNOC works directly on RTL designs as is, without requiring tweaks for scalability. The automation and scalability results suggest that SEVNOC has the potential to apply to industrial SoC designs and can be smoothly integrated with industrial flows.

A key outcome of the work is the testbed developed for evaluation of SEVNOC. Both NSNoC and TransNoC include several realistic features, including complete processor (RISC-V) core, memories, hierarchical subsystems, etc. Aside from providing realistic targets for our own evaluations, they can address the dearth of realistic SoC benchmarks that have plagued academic research in SoC architecture and validation.

In future work, we will improve the scalability and effectiveness of SEVNOC, particularly for detecting violations triggered by a combination of multiple inputs and hardware-software interactions. We will also plan on evaluating SEVNOC on industrial applications. In particular, many industrial SoC designs for mobile and IoT applications implement NoCs that entail communication of components with varying trust levels [43], [44]. We will explore the use of SEVNOC to detect information flow violations on such systems.

REFERENCES

[1] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the tile processor," *IEEE micro*, pp. 15–31, 2007.

[2] N. E. Jerger *et al.*, "On-chip networks," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 3, pp. 1–210, 2017.

[3] D. M. Ancajas *et al.*, "Fort-nocs: Mitigating the threat of a compromised noc," in *ACM/IEEE DAC*, 2014, pp. 1–6.

[4] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-Chip Security Verification and Validation*. Springer, 2019.

[5] Cadence, "JasperGold Formal Security App," www.cadence.com.

[6] Synopsys Inc., "VCFormal Formal Security App," www.synopsys.com.

[7] X. Meng *et al.*, "Soccar: Detecting system-on-chip security violations under asynchronous resets," Cryptology ePrint Archive, Report 2021/309, 2021, https://eprint.iacr.org/2021/309.

[8] 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.

[9] X. Guo *et al.*, "Pre-silicon security verification and validation: A formal perspective," in *ACM/IEEE DAC*, 2015, pp. 1–6.

[10] A. Nahiyan *et al.*, "Hardware trojan detection through information flow security verification," in *ITC*. IEEE, 2017, pp. 1–10.

[11] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

[12] C. Wang, Y. Cai, and Q. Zhou, "Hlift: A high-level information flow tracking method for detecting hardware trojans," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 727–732.

[13] W. Hu *et al.*, "Property specific information flow analysis for hardware security verification," in *Proceedings of ICCAD*, 2018, pp. 1–8.

[14] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," *Sigplan Notices*, vol. 39, no. 11, pp. 85–96, 2004.

[15] M. Tiwari *et al.*, "Complete information flow tracking from the gates up," in *Proceedings of ASPLOS*, 2009, pp. 109–120.

[16] J. Oberg *et al.*, "Theoretical analysis of gate level information flow tracking," in *DAC*. IEEE, 2010, pp. 244–247.

[17] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*. IEEE, 2017, pp. 1691–1696.

[18] M. Qin *et al.*, "A formal model for proving hardware timing properties and identifying timing channels," *Integration*, vol. 72, p. 123, 2020.

[19] A. Ardeshiricham *et al.*, "Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties," in *Proceedings of ACM SIGSAC CCS*, 2019, pp. 1623–1638.

[20] John Rushby, "Noninterference, Transitivity, and Channel-Control Security Policies," SRI, Tech. Rep., 1992.

[21] D. A. Greve, M. Wilding, and W. Vanfleet, "A Separation Kernel Security Policy," in *ACL2 Workshop*, 2003.

[22] W. A. Hunt, Jr., R. B. Krug, S. Ray, and W. D. Young, "Mechanized Information Flow Analysis through Inductive Assertions," in *FMCAD*, 2008, pp. 227–230.

[23] Cadence, "Jaspergold formal verification platform (apps)," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, (Accessed on 05/15/2020).

[24] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model-Checking*. Cambridge, MA: The MIT Press, Jan. 2000.

[25] R. Zhang and C. Sturton, "A recursive strategy for symbolic execution to find exploits in hardware designs," in *Proceedings of ACM SIGPLAN FSM*, 2018, pp. 1–9.

[26] A. Ahmed *et al.*, "Directed test generation using concolic testing on rtl models," in *DATE*, 2018, pp. 1538–1543.

[27] Y. Lyu and P. Mishra, "Automated test generation for activation of assertions in rtl models," in *ASP-DAC*, 2020.

[28] "Intel baytrail products," https://ark.intel.com/content/www/us/en/ark/products/codename/55844/bay-trail.html.

[29] A. P. D. Nath, S. Boddupalli, S. Bhunia, and S. Ray, "Security assurance of system-on-chip designs with noc fabrics," *IEEE Transactions on Information Forensics and Security*, vol. 15, no. 1, pp. 2808–2823, 2020.

[30] A. P. D. Nath, K. Raj, S. Bhunia, and S. Ray, "Soccom: Automated synthesis of system-on-chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2022.

[31] Y. Jin *et al.*, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *IEEE VTS*, 2012, pp. 252–257.

[32] M.-M. Bidmeshki *et al.*, "Information flow tracking in analog/mixed-signal designs through proof-carrying hardware ip," in *DATE*, 2017, pp. 1707–1712.

[33] X. Li *et al.*, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of ASPLOS*, 2014, pp. 97–112.

[34] A. Basak *et al.*, "Exploiting design-for-debug for flexible soc security architecture," in *ACM/IEEE DAC*, 2016, p. 167.

[35] M. Hicks *et al.*, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proceedings of ASPLOS*, 2015, pp. 517–529.

[36] L. Shen *et al.*, "Symbolic execution based test-patterns generation algorithm for hardware trojan detection," *computers & security*, vol. 78, pp. 267–280, 2018.

[37] R. Zhang *et al.*, "End-to-end automated exploit generation for validating the security of processor designs," in *IEEE MICRO*, 2018, pp. 815–827.

[38] S. Charles, Y. Lyu, and P. Mishra, "Real-time detection and localization of dos attacks in noc based socs," in *IEEE DATE*, 2019, pp. 1160–1165.

[39] A. Kumar, P. Kuchhal, and S. Singhal, "Secured network on chip (noc) architecture and routing with modified tacit cryptographic technique," *Procedia Computer Science*, vol. 48, pp. 158–165, 2015.

[40] C. Ciordas, T. Basten, A. Rădulescu, K. Goossens, and J. V. Meerbergen, "An event-based monitoring service for networks on chip," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 10, no. 4, pp. 702–723, 2005.

[41] T. Boraten *et al.*, "Mitigation of denial of service attack with hardware trojans in noc architectures," in *IEEE IPDPS*, 2016, pp. 1091–1100.

[42] V. Y. Raparti *et al.*, "Lightweight mitigation of hardware trojan attacks in noc-based manycore computing," in *ACM/IEEE DAC*, 2019, pp. 1–6.

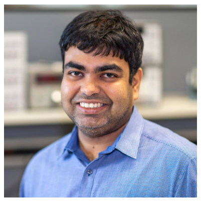
[43] J. W. O'leary, "Verification in the Age of Integration," in *ACL2 Workshop*, 2015.

[44] A. Basak *et al.*, "Security assurance for system-on-chip designs with untrusted ips," *IEEE TIFS*, vol. 12, no. 7, pp. 1515–1528, 2017.

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 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.

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 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.

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 the Internet-of-Things applications. His current research targets correct, dependable, secure, and trustworthy computing through 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 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.