Journal of Hardware and Systems Security
https://doi.org/10.1007/s41635-022-00129-5

Network-on-Chip Trust Validation Using Security Assertions

Aruna Jayasena[1] · Binod Kumar[2] · Subodha Charles[3] · Hasini Witharana[1] · Prabhat Mishra[1]

Accepted: 7 October 2022

Abstract
Recent technological advancements enabled integration of a wide variety of Intellectual Property (IP) cores in a single chip, popularly known as System-on-a-Chip (SoC). Network-on-Chip (NoC) is a scalable solution that enables communication between a large number of IP cores in modern SoC designs. A typical SoC design methodology relies on third-party IPs to reduce cost and meet time-to-market constraints, leading to serious security concerns. NoC becomes an ideal target for attackers due to its distributed nature across the chip as well as its inherent ability in monitoring communications between the individual IP cores. This paper presents a comprehensive NoC trust validation framework using security assertions. It makes three important contributions. (1) We define a set of security vulnerabilities for NoC architectures, and propose security assertions to monitor these pre-silicon vulnerabilities. (2) In order to ensure that the generated assertions are valid, we utilize efficient test generation techniques to activate these security assertions. (3) We develop on-chip triggers based on synthesized security assertions as well as efficient security-aware signal selection techniques for effective post-silicon debug. Experimental results show that our proposed framework is scalable and effective in capturing security vulnerabilities as well as functional bugs with minor hardware overhead.

Keywords Network-on-chip · Security assertions · Hardware security · Security verification · Pre-silicon validation · Post-silicon debug

1 Introduction

System-on-Chip (SoC) integrates a wide variety of hardware components (e.g., processor, memory, controllers, converters) into a single integrated circuit to provide the backbone of modern computing systems. With the rapid adoption of multi-processor/multi-core based SoCs, network-on-chip (NoC) has become a crucial component for delivering high performance in a wide range of applications. Since NoC has access to various components in an SoC, it is a prime target for security attacks. NoC becomes more vulnerable due to the current trend of integrating diverse third-party intellectual property (IP) cores into the SoC design.

In order to meet the performance requirements of different IP cores, NoC design has evolved to be quite complex as different techniques are employed to accommodate high communication bandwidth. Figure 1 shows an example NoC architecture consisting of several IPs connected together via routers and electrical wires (links). IPs are connected to the routers via a network interface (NI). The combination of an IP, an NI and a router is referred to as a "node" in the NoC. NoC architectures use packets to communicate between IPs. For example, when a memory instruction (Load/Store) is executed by source IP (S), the private caches located in the same node are checked first and if it is a miss, the off-chip memory at destination IP (D) has to be accessed to retrieve the data. Therefore, a memory fetch request message is created and injected in the appropriate virtual network. The message created by the IP is first received by the NI, which converts it to network packets before sending the packets into the network via the local router[1].

The packets are routed through the routers and links according to the routing protocol stated in the *route_algorithm* until the destination node is reached [1].

✉ Aruna Jayasena
arunajayasena@ufl.edu

[1] University of Florida, Gainesville, USA

[2] Indian Institute of Technology, Jodhpur, India

[3] University of Moratuwa, Moratuwa, Sri Lanka

[1] Most NoC architectures facilitate flits, which is a further breakdown of a packet used for flow control purposes. We stick to the level of packets for the ease of explanation as our method remains the same at the flit level as well.

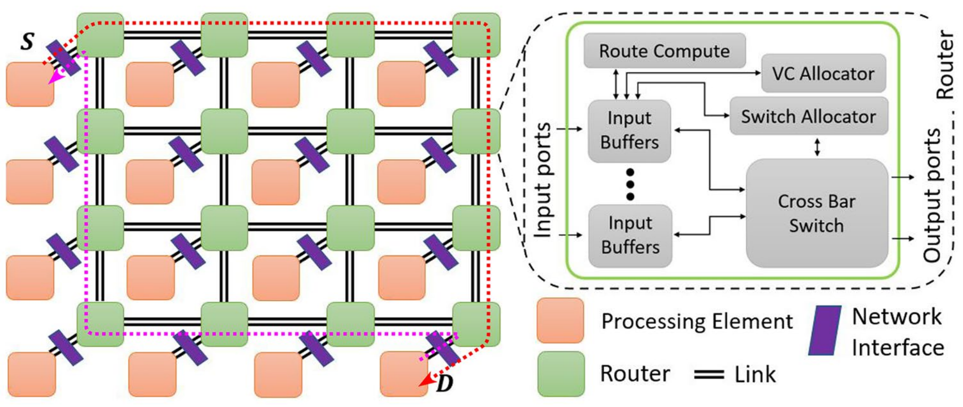


Fig. 1 Example of an NoC connecting 16 IPs. The red and pink lines show sample paths (using X-Y routing) between source (S) and destination (D) for data and response transfers, respectively

The NI connected to *D* recreates the message from the packets and passes it to *D*, which initiates the memory access. The response message from memory follows a similar process when going from *D* to *S*. For example, the red and pink lines show sample data and response transfer paths, respectively, using the X-Y routing protocol. Similarly, all IPs integrated into the SoC leverage the resources provided by the NoC to communicate with each other.

1.1 Threat Model

With all of the advanced performance features, extra logic and buffers involved in the NoC operations, it has definitely become a challenge to ensure correct functionality under all possible scenarios for the entire NoC [1–4]. Different components in the NoC design are susceptible to different types of attacks. The most commonly explored attacks can be divided into three broad categories: (i) eavesdropping attacks through packet duplication, (ii) sabotaging communication by packet corruption, and (iii) performance degradation and denial of service through packet starvation, packet dropping and packet misrouting. These attacks can even get implanted during outsourcing of different stages of NoC life cycle to third-party vendors. These attack scenarios are described in detail in Sect. 3.

1.2 State of the Art

Traditional SoC validation methodology is unlikely to detect security vulnerabilities since it is infeasible to get 100% coverage of functional scenarios for complex (billion-gate) SoC designs [5]. For example, malicious implants such as hardware Trojans can stay benign most of the time and act maliciously when a predefined trigger condition is met, which can be extremely rare [6]. Moreover, a carefully crafted Trojan has a very low performance and power footprint that can be hidden in typical process variations and environmental noise margins. While formal verification is promising at pre-silicon stage, it has two fundamental limitations. The formal methods can be applied to only small designs (e.g., individual IPs) due to state space explosion. Therefore, it will not be able to detect system-level vulnerabilities consisting of NoC communicating with multiple IPs. Most importantly, an attacker may introduce the vulnerability during the later stages in the design (e.g., during synthesis, layout or fabrication). Therefore, it is critical to monitor security vulnerabilities during runtime (pre-silicon simulation or post-silicon execution). In this paper, we show that security assertions (and associated synthesized checkers) are effective in runtime validation of security vulnerabilities.

For functional verification of SoCs, assertion-based validation (ABV) is one of the primary industrial techniques. Assertions can be thought of as certain kinds of checkers embedded in the design. Failure to adhere to the assertion condition can trigger warnings helping in runtime validation. For example, assertions can check whether the output of an adder is always equal to the sum of the two inputs. While ABV is widely used for functional validation, there is a limited effort in utilizing assertions to detect security vulnerabilities [7]. There is a fundamental difference between the objectives of functional and security assertions. While functional assertions monitor expected behaviors, security assertions are designed to monitor unexpected vulnerabilities. We utilize these assertions to carry out the comprehensive validation of NoC security at different stages of the SoC development cycle.

The primary objective of our proposed approach is to show that the security assertions can be effectively utilized for NoC trust validation. The proposed framework enables pre-silicon as well as post-silicon security validation of NoC-based SoCs. At the pre-silicon stage, the RTL model is input to the validation framework whereas the post-silicon validation is carried out on the chip model (silicon) which has highly restricted observability. Section 4 describes the three primary tasks during pre-silicon validation: assertion generation, directed test generation, and coverage analysis. We generate the security assertions in the first task. The next task enables the automated generation of directed tests to activate the security assertions. The final task performs injection of vulnerabilities and coverage analysis to demonstrate that the generated assertions can capture the security flaws.

Section 5 describes the three major tasks in post-silicon debug: signal selection, trigger generation, and post-silicon debug of security failures. The first task enables security-aware signal selection without compromising observability requirements. The second task leads to the automated generation of trigger logic based on security assertions. The final task performs trace analysis for post-silicon debug of security vulnerabilities.

1.3 Contributions

This paper makes the following major contributions:

- Develops security assertions for a wide variety of NoC vulnerabilities based on a comprehensive survey to identify common vulnerabilities and attack scenarios in NoC architectures.
- Generates directed tests using bounded model checking as well as concolic testing to activate the security assertions. These tests are effective for activating pre-silicon security assertions as well as post-silicon (synthesized) checkers.
- Enables post-silicon security validation and debugging of NoC-based SoC architectures. This involves effective signal selection and automated offline analysis of selected signals.
- Demonstrates the efficacy of utilizing security assertions for capturing both pre-silicon vulnerabilities as well as post-silicon security failures in NoC-based SoCs.

1.4 Paper Organization

The remainder of the paper is organized as follows. Section 2 surveys the related approaches. Section 3 outlines the different security threat models considered in this paper. Section 4 describes the pre-silicon security validation framework. Section 5 discusses the post-silicon security debug of NoC architectures. Section 6 presents the experimental results. Finally, Sect. 8 concludes the paper.

2 Background and Related Work

In this section, we first introduce the NoC and its components. Next, we discuss the existing efforts related to NoC validation at different stages in two broad categories: pre-silicon security validation and post-silicon debug of security vulnerabilities. We provide a qualitative comparison with the state-of-the-art approaches in Sect. 6.4.

2.1 Network-on-Chip Components

As shown in Sect. 1, the router plays a major role in internal operations of the NoC. Specifically, the router is responsible for receiving packets, decoding the destination, and forwarding it to the correct output. Since NoC consists of multiple IP's that share the same communication medium, there is arbitration logic involved to avoid communication interruptions. During the arbitration, the data is stored temporally inside the *flit_buffer*. Therefore, *flit_buffer* implementation typically includes all the simple memory signals such as *rd_ptr*, *wr_ptr*, *wr_en*, *rd_en*. Furthermore, router consists of *arbiter*, *route_algorithm*, and other *miscellaneous* components. Routing algorithm determines the way to transfer the data packets through the network. Post-silicon debug is challenging due to the integration of all the complex components in the NoC. Trace buffers are widely used as a Design-for-Debug (DfD) structure in traditional post-silicon validation methodology [8, 9]. The main purpose of the trace buffer is to capture a small set of internal signals during execution that can be offloaded for offline analysis.

2.2 Pre-silicon Security Validation

Formal methods have been extensively used in performing security verification. Previous work has explored proof-based methods [10] and formal verification [11, 12]. Sepúlveda et al. [12] explored formal verification in NoC considering security vulnerabilities in routing protocols. While formal verification is promising, it has two major limitations: (i) formal methods cannot be applied on large designs due to state space explosion, and (ii) it cannot perform runtime attack detection. As discussed in Sect. 3, even if there are no Trojans detected during design time, attackers can still launch attacks during runtime which cannot be detected using design time formal verification.

While formal methods can provide guarantees, it can be infeasible to apply them on large NoC IPs. On the other hand, simulation based techniques are scalable but cannot give guarantees about the verification completeness. Assertion-based Validation (ABV) provides a middle ground by utilizing the best of both worlds. A major challenge in post-silicon debug is how to increase the controllability and observability of the hardware design. The ability to control the internal signal is referred to as controllability, whereas observability refers to the ability to view the internal signals by propagating them to observable points (such as primary outputs). A recent survey [13] shows that ABV has shown promising results in validation of functional behaviors [14–16] as well as non-functional (e.g., security) requirements [17, 18]. Assertions can capture unusual behavior and depending on where the assertion is embedded, it can give information about the internal state of the design. This

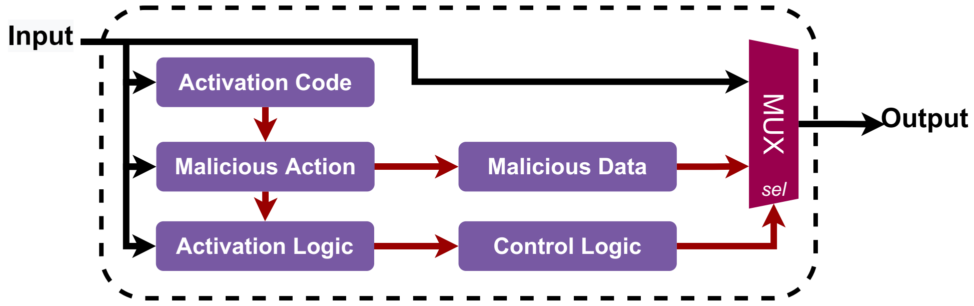


Fig. 2 An example Trojan that modifies the packet during transit

increased observability reduces overall hardware validation time significantly. While assertions do not directly improve controllability, there are efforts to generate tests that can activate the assertions. *Assertions have been extensively used for functional validation of SoC designs, however, the use of assertions for security validation of NoC-based SoCs has not been well studied in the literature.*

2.3 Post-silicon Security Validation

While pre-silicon security validation can be effective at detecting design vulnerabilities, runtime attacks are still possible unless adequate defense measures are adopted. For example, even if there are no hardware Trojans found during pre-silicon verification, attackers can still implant Trojans during fabrication and launch attacks during runtime. There are many approaches for runtime security monitoring and mitigation. Dubrova et al. [19] proposed *built-in self test* (BIST) as a solution to prevent attacks caused by hardware Trojans added into the SoC during the manufacturing stage. However, BIST cannot address diverse threats and can provide very limited coverage. Boraten et al. [20] use model checkers to alert the SoC if the buffers, VCs, and switch allocators are illegally utilized causing DoS attacks. Other techniques to mitigate attacks during runtime include traffic monitoring [21–23], partitioning [24] and cryptographic defenses [25, 26]. However, the success of these mitigation techniques is limited only to a certain set of attack scenarios. Hence, if the mitigation technique is aiming to prevent one attack type, it will fail in a different attack scenario. Moreover, these runtime mitigation techniques introduce significant hardware overhead in terms of area, power and performance. Our proposed approach effectively utilizes the existing trace buffer design and develops trigger logic to provide a lightweight solution for seamless detection of runtime security vulnerabilities related to NoC operation.

Post-silicon validation and debug of SoCs has emerged as a challenging problem [8, 9]. Recent research efforts have addressed post-silicon functional validation issues for NoC architectures [1, 27–29]. Rout et al. [27–29] explored efficient router and trace buffer design for post-silicon validation of NoC-based SoCs. These techniques mainly cater to ascertaining the functional correctness of NoC designs, and are not designed for post-silicon security validation. *To the best of our knowledge, our approach is the first attempt in exploring the effectiveness of security assertions for both design time (pre-silicon) and runtime (post-silicon) security validation of NoC architectures.*

3 Modeling Different Types of Threats

We consider various attack scenarios outlined in the recent literature [6]. For example, the attacker can be a rogue designer who is able to tamper with the NoC IP and implant Trojans in the routers during design time. Similarly, a compromised CAD tool can introduce malicious implants at various stages of the design cycle such as synthesis, scan-chain insertion, verification, and layout. An attacker can also insert malicious implants at the foundry via reverse engineering. A vulnerability can also be created unintentionally by a CAD tool. Once integrated, the Trojans remain hidden (deactivated) in order to avoid detection. Pre-programmed wake times and/or a specific activation logic can be used to fully activate the Trojans. Even when behaving maliciously, Trojans exhibit negligible power and performance overhead. For example, Sepúlveda et al. [30] explored a similar threat model. In fact, our modeling of security assertions builds on top of the modeling of security properties for formal verification of pre-silicon models [30]. We follow a functionality-directed approach for developing the assertion set for pre-silicon verification. In other words, we show how to

Table 1 Notations for properties in Tables 2 3, 4, 5, and 6

Symbol	Operator	Description
$X\phi$	Next	Property should hold in the next cycle
$G\phi$	Always	Property should always hold
$F\phi$	Eventually	Property will at some point in time (future) hold
$P\phi$	Previous state	Specifies a state at some point in time in the past
$\phi U\omega$	Until	ϕ should be true until ω becomes true

Table 2 Properties to detect Packet Duplication

P#	Description of Security Properties
d1	Always the number of packets entered the router should be equal to the sum of packets in the router and the number of packets that left the router $\sum(wr\_en \wedge \neg rd\_en \wedge \neg full) == \sum(rd\_en \wedge \neg wr\_en \wedge \neg empty)$
d2	Rd/Wr flags should reset once data has been read from /written to the buffer $(wr_{done} \rightarrow X(\neg wr\_en)) \wedge (rd_{done} \rightarrow X(\neg rd\_en))$
d3	Multiplexers should not alter the input data at the output $G((\sum_{i=0}^{N_{ports}}(select_i \wedge (data_{in}i == data_{out}))) == 1)$

generate security assertions for specific NoC vulnerabilities. A designer can create additional assertions (if needed).

In this paper, we assume that the Trojans can manifest through duplication, corruption, starvation, dropping, and misrouting of packets when packets pass through routers. Furthermore, we assume that the trace buffer logic is trustworthy. Similarly, we also consider single vulnerability model. Therefore, simultaneous activation of multiple dependent vulnerabilities is out of scope. Figure 2 shows a block diagram of a Trojan architecture that can facilitate the attacks. The capabilities of the Trojan include all possible attacks that can be caused by a Trojan-infected router including packet duplication (Sect. 3.1), packet corruption (Sect. 3.2), packet starvation (Sect. 3.3), packet loss (Sect. 3.4), and packet misrouting (Sect. 3.5). Based on the threat model, we derive a set of security properties that will be embedded in the design for monitoring runtime vulnerabilities. The notations used to denote these security properties are shown in Table 1. Tables 2, 3, 4, 5, and 6 outline the individual properties that should hold during execution for each category of threat model. The first column indicates the property number (P#). The second column provides an intuitive description of the property behavior as well as a temporal logic description that can be implemented as SystemVerilog assertions.

3.1 Packet Duplication

IPs rely on the NoC to ensure secure data communication. An attacker can eavesdrop on the packets in an attempt to leak sensitive information. A common threat model is a hardware-software coalition attack where a Trojan-infected router and an accomplice application work together to eavesdrop. When packets are received at the input buffer of the router, the Trojan copies the packets, modifies the destination address in the header so that the new destination is an IP that runs an accomplice malicious application, and places it back in the input buffer. The NoC then routes the duplicated packets to the malicious application. The same threat model has been widely used to explore eavesdropping attacks in NoC [25, 31]. Table 2 outlines the properties that should hold true to prevent packet duplication.

3.2 Packet Corruption

Integrity of data communicated through the NoC is crucial for correct execution of tasks. If an attacker corrupts data intentionally, it can cause erroneous behavior and/or system failure. Furthermore, since corrupted data can trigger re-transmissions, it can incur significant power and performance overhead leading to denial-of-service attacks. The Trojan architecture in Fig. 2 facilitates data corruption by replacing the packet content with the content in a malicious register. A similar threat model that discussed eavesdropping, denial-of-service and illegal packet forwarding, all of which utilized packet corruption at a router was presented in [32]. Table 3 outlines the properties that should hold true to prevent packet corruption.

3.3 Packet Starvation

Denial-of-Service (DoS) is one of the most common way attackers find for attacking the systems. There are multiple avenues in which the DoS manifestation affects the operation. Performance guarantees of the design are tightly coupled with operation of certain important components. For example, the response time of a memory controller that

Table 3 Properties to detect Packet Corruption

P#	Description of Security Properties
c1	Router can issue only one request at a time $G((\sum_{i=0}^{N_{ports}} req\_port_i) \leq 1)$
c2	Arbiter cannot issue multiple grants at the same time $G((\sum_{i=0}^{N_{ports}} gnt\_port_i) \leq 1)$
c3	Error checking code should match the data when data is written to and read from the buffer $G(parity_{out} == \exists P(parity_{in}))$
c4	Multiplexers should not alter the input data at the output $G((\sum_{i=0}^{N_{ports}}(select_i \wedge (data_{in}i == data_{out}))) == 1)$

Table 4 Properties to detect Packet Starvation

P#	Description of Security Properties
s1	Rd/Wr pointers should be always sequentially incremented $rd\_en \wedge (\neg wr\_en \wedge \neg empty) \leftrightarrow (X(rd\_ptr) == (rd\_ptr + 1)) \wedge$ $wr\_en \wedge (\neg rd\_en \wedge \neg full) \leftrightarrow (X(wr\_ptr) == (wr\_ptr + 1))$
s2	Rd/Wr pointers are not incremented when the buffer is empty/full $(rd\_en \wedge \neg wr\_en \wedge empty \rightarrow (rd\_ptr == X(rd\_ptr))) \wedge$ $(wr\_en \wedge \neg rd\_en \wedge full \rightarrow (wr\_ptr == X(wr\_ptr)))$
s3	Write address range should be equal to read address range $G(rd\_address_{range} == wr\_address_{range})$
s4	Arbiter should eventually grant the opportunity for every available request $(req\_port\ U\ gnt\_port) \rightarrow F(gnt\_port)$
s5	Route should issue a request whenever data is valid $G(data\_valid \leftrightarrow (\sum_{i=0}^{N_{ports}} req\_port_i) == 1)$
s6	Multiplexers should not alter the input data at the output $G((\sum_{i=0}^{N_{ports}} (select_i \wedge (data_{in}i == data_{out}))) == 1)$

provides the interface to off-chip memory can be critical in serving all the memory requests. If an attacker intentionally delays packets originating from such a critical component, the SoC performance can suffer significant degradation. Delays can lead to catastrophic consequences in real-time safety-critical applications. A Trojan can selectively delay packets originating from an IP, which is referred to as "packet starvation". Starvation can be caused by a Trojan-infected router de-prioritizing packets from a particular origin at the arbiter [33]. In other words, packets are treated unfairly such that all the input ports do not get an equal chance of accessing the output. Table 4 outlines the properties that should hold to prevent packet starvation.

3.4 Packet Loss

From the scenario of packet starvation considered in Sect. 3.3, packet loss comes out as another manifestation. In starvation, packets are intentionally delayed and can reach the destination at some point. However, when the packets are dropped, unless packets are re-transmitted, the destination will not receive the packets. Similar to the consequences of starvation, packet loss can cause severe performance degradation and malfunction [34]. Table 5 outlines the properties that should hold true to prevent packet loss.

3.5 Packet Misrouting

The NoC uses routing protocols to route packets between the senders and the receivers. A key requirement of routing protocols is to ensure packet routing without causing deadlocks and livelocks. A Trojan that corrupts packet header information and/or routing tables can force some packets to loop around and force deadlocks and livelocks. Such attacks are capable of rendering single application to full chip failures [35]. Rerouting of packets is also a critical component in eavesdropping attacks as explained above (see *Packet Duplication*). Table 6 outlines the properties that should hold true to prevent packet misrouting.

4 Pre-silicon Security Validation

Pre-silicon validation section of Fig. 3 shows an overview of our proposed NoC trust validation framework using security assertions. It consists of three major tasks: assertion generation, test generation, and coverage analysis. First, we describe how to generate security assertions for the NoC design (Sect. 4.1). Next, we present how to generate test cases to activate the security assertions (Sect. 4.2). Finally, we discuss the assertion coverage to prove the effectiveness of the security assertions (Sect. 4.3).

Table 5 Properties to detect Packet Loss

P#	Description of Security Properties
l1	Rd/Wr pointers should be sequentially incremented only when rd_en/wr_en are set $rd\_en \wedge (\neg wr\_en \wedge \neg empty) \leftrightarrow (X(rd\_ptr) == (rd\_ptr + 1)) \wedge$ $wr\_en \wedge (\neg rd\_en \wedge \neg full) \leftrightarrow (X(wr\_ptr) == (wr\_ptr + 1))$
l2	Route can issue only one request at a time $G((\sum_{i=0}^{N_{ports}} req\_port_i) \leq 1)$
l3	Route should issue a request whenever data is valid $G(data\_valid \leftrightarrow (\sum_{i=0}^{N_{ports}} req\_port_i) == 1)$
l4	Routing algorithm (XY) should be correctly implemented $G((dest_x > current_x \leftrightarrow destport_{next} == EAST) \vee (dest_x < current_x \leftrightarrow destport_{next} == WEST) \vee (dest_y > current_y \leftrightarrow destport_{next}$ $== SOUTH) \vee (dest_y < current_y \leftrightarrow destport_{next} == NORTH) \vee (destport_{next} == LOCAL))$
l5	Only one grant can be issued by the arbiter at a time $G((\sum_{i=0}^{N_{ports}} gnt\_port_i) \leq 1)$
l6	Arbiter should eventually grant the opportunity for every available request $(req\_port\ U\ gnt\_port) \rightarrow F(gnt\_port)$
l7	Multiplexers should not alter the input data at the output $G((\sum_{i=0}^{N_{ports}} (select_i \wedge (data_{in}i == data_{out}))) == 1)$

Table 6 Properties to detect Packet Misrouting

P#	Description of Security Properties
m1	Route can issue only one request at a time $G((\sum_{i=0}^{N_{ports}} req\_port_i) \leq 1)$
m2	Routing algorithm (XY) should be correctly implemented $G((dest_x > current_x \leftrightarrow destport_{next} == EAST) \vee (dest_x < current_x \leftrightarrow destport_{next} == WEST) \vee (dest_y > current_y \leftrightarrow destport_{next} == SOUTH) \vee (dest_y < current_y \leftrightarrow destport_{next} == NORTH) \vee (destport_{next} == LOCAL))$
m3	Only one grant can be issued by the arbiter at a time $G((\sum_{i=0}^{N_{ports}} gnt\_port_i) \leq 1)$
m4	Multiplexers should not alter the input data at the output $G((\sum_{i=0}^{N_{ports}} (select_i \wedge (data_{in}i == data_{out}))) == 1)$

4.1 Generation of Security Assertions

To launch an attack identified in Sect. 3, the hardware Trojan must change the normal behavior of the NoC. Any violation to the security properties presented in Sect. 3 can be monitored by implementing them as assertion checkers embedded in the NoC design. Table 7 shows the twelve SystemVerilog assertions that can cover the security properties defined in Sect. 3 (Tables 2, 3, 4, 5, and 6). If the security checks defined by the assertions are not violated during runtime, we can conclude that there are no ongoing attacks.

The mapping between the security properties and the security assertions is outlined in Fig. 4. The same color indicates the security properties for a specific threat model. For example, the four security properties related to packet corruption ($c1$–$c4$ in Table 3) are colored in pink. For optimization purposes, some properties are combined into a single assertion since they effectively perform the same functionality. For example, the security assertion $A7$ can cover three security properties: $c1$ (from Table 3), $l2$ (from Table 5) and $m1$ (from Table 6).

4.2 Test Generation for Activation of Security Assertions

We have explored automated test generation using two complimentary approaches. The first approach uses model checking that is suitable for small designs with simple assertions. The second approach is scalable for large designs and complex assertions due to the effective utilization of concrete simulation and symbolic execution. This section briefly describes these approaches. Section 6 demonstrates the effectiveness of these test generation techniques in activating security assertions for NoC architectures.

Given that the security assertions represent unexpected behaviors, they are not expected to be activated during the traditional validation methodology. Therefore, it is important to generate directed tests to activate the security assertions. Once an assertion is activated by a directed test, it indicates that the assertion is valid and it is able to accurately detect a specific security threat. An assertion is valid if there is at least one scenario that can violate the assertion. The invalid assertions cannot be activated under any circumstance, and therefore, those assertions should be removed. Figure 5 shows our test generation framework with two complementary approaches. We use SAT-based bounded model checking (BMC) that accepts the NoC design and assertions (negated properties) as inputs. The counterexamples generated by the EBMC model checker [36] can be used as a directed test that is guaranteed to activate the respective security assertion.

Unfortunately, EBMC may fail to handle complex properties due to state space explosion. In such cases, we use concolic testing [37] that can effectively utilize concrete

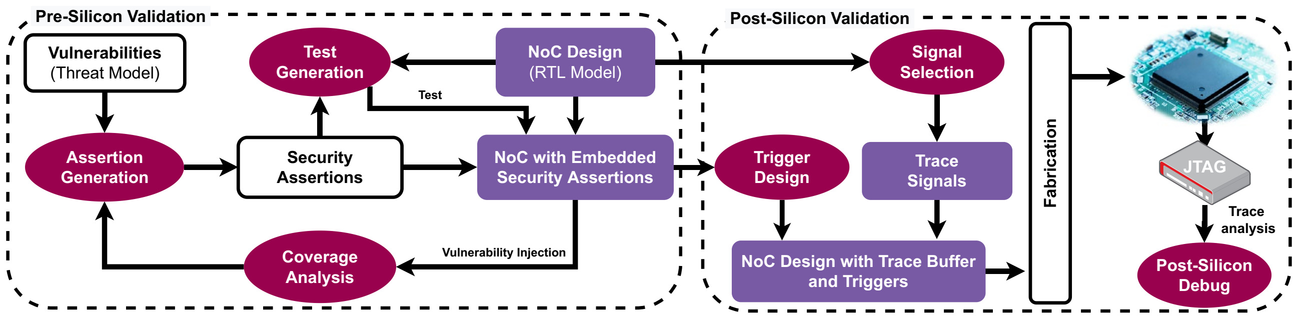


Fig. 3 Overview of our proposed NoC trust verification framework using security assertions. It consists of three major tasks for pre-silicon validation: assertion generation, test generation, and coverage analysis. For post-silicon validation, it consists of three major tasks of trigger design, signal selection and debug of post-silicon failures

Table 7 SystemVerilog Security Assertions

A#	SystemVerilog Assertions
A1	`assert property (@(posedge clk) (wr[i]&&!rd[i]) |=>(wrPtr[i] ==$past(wrPtr[i])+1));` `assert property (@(posedge clk) (rd[i] && !wr[i]) |=>(rdPtr[i] == $past(rdPtr[i])+1 ));`
A2	`assert property (@(posedge clk) (wr[i] && !rd[i] && (depth[i] == B)) |=>(rdPtr[i]==$past(rdPtr[i])));` `assert property (@(posedge clk) (rd[i] && !wr[i] && (depth[i] == {DEPTHw{1'b0}})) |=>(rdPtr[i] == $past(rdPtr[i]))) ;`
A3	`assert (maxRdPtr[i] <= maxWrPtr[i] && minRdPtr[i] >= minWrPtr[i])`
A4	`assert property (@(posedge clk) wr[i] |=> !wr[i];` `assert property (@(posedge clk) rd[i] |=> !rd[i];`
A5	`assert (parity(din)|-> s_eventually parity(dout));`
A6	`assert (count(wr_en,!rd_en)==count(!wr_en,rd_en));`
A7	`assert ($onehot0(destport));`
A8	`assert (dest_x*dest_y<=NoCSize);`
A9	`assert ((dest_x > current_x && destport_next==EAST) ||(dest_x < current_x && destport_next==WEST) ||(dest_y > current_y && destport_next==SOUTH) ||(dest_y < current_y && destport_next==NORTH) ||(destport_next==LOCAL));`
A10	`assert ($onehot0(grant));`
A11	`assert property (@(posedge clk) request[j]|->s_eventually grant[j]);`
A12	`assert (!$onehot(sel) || sel!=1'b0 || (sel[x]==1'b1 && (mux_in[OUT_WIDTH*(x)+:OUT_WIDTH]==mux_out))==1);`

simulation and symbolic execution to generate the required test patterns. Concolic testing addresses the state space explosion problem by exploring one path at a time compared to model checking that tries to explore all possible paths. To activate the security assertions non-vacuously, we first convert the security assertions into branch statements and then use concolic testing to activate the specific branches.

4.3 Assertion Coverage Analysis

One way of checking the correctness of the assertions would be to inject a wide variety of vulnerabilities into the design and check whether assertions are able to capture them. Typically for this purpose vulnerability scenarios have to be created only considering the threat models without focusing on the created assertions. After injecting the vulnerabilities, the design can be simulated with the generated tests. The process of injecting vulnerabilities and computing assertion coverage analysis need to be performed in an iterative fashion until we get 100% coverage of security assertions.

5 Post-silicon Trust Validation Using Synthesized Checkers

Due to various factors ranging from exponential validation complexity, slow simulation speed to lack of effective coverage metric, it is not feasible to capture all functional bugs as well as security vulnerabilities during pre-silicon validation stage. The focus of post-silicon validation is to capture these escaped bugs as well as security vulnerabilities. Post-silicon validation section of Fig. 3 shows the three major tasks in the post-silicon security validation framework for NoC architectures: trigger design, signal selection and debug of post-silicon failures. The remainder of this section describes these tasks in detail.

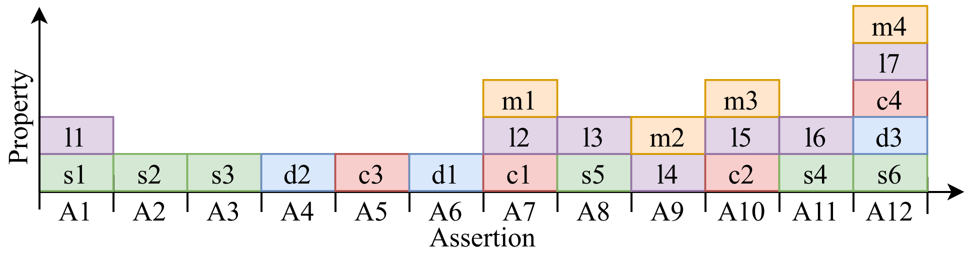
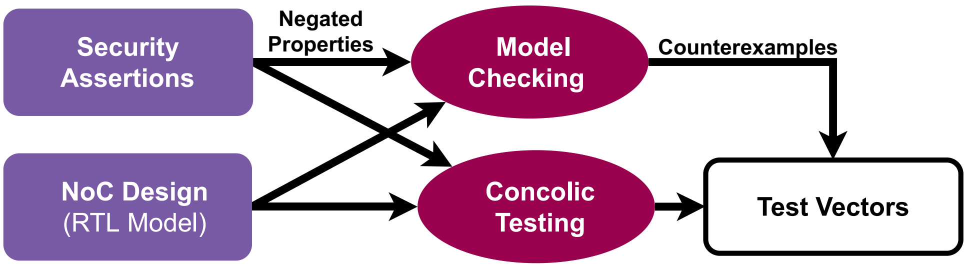


Fig. 4 Mapping between security assertions shown in Table 7 and the security properties outlined in Sect. 3 (Tables 2-6). Each color represents security properties for the same vulnerability. For example, the properties shown in pink (c1–c4) represent packet corruption in Table 3. Note that each security assertion can cover one or more security properties. For example, security assertion A7 represents three security properties (c1, l2 and m1)

Fig. 5 Directed test generation using SAT-based bounded model checking as well as concolic testing



5.1 On-chip Trigger Design Using Security Assertions

Implementation of assertions can assist in checking design correctness [4]. Similarly, security assertions [7] can also be implemented on-chip. Typically, the generation of such assertions can be done automatically [16]. However, as a large number of assertions can be obtained through mining techniques, the implementation of the assertions for on-chip triggers becomes a difficult problem owing to the associated overhead. An approach to obtain the on-chip implementation of assertions (specified in property specification language) is presented in [38] based on the concepts of automata theory. However, these techniques do not specifically cater to the objective of security assertions/properties Specifically, one of the key objectives of our approach is to create a lightweight trigger mechanism. Since the enumeration of design behaviors based on specification tend to be typically large, we adopt a threat model-centric approach for obtaining security assertions.

The assertions described in Fig. 4 were modeled based on the behavior of the threat models. The approach that was used to convert the assertions into relevant triggers is as follows. Initially, we categorized the assertions based on their types: implication assertions, immediate assertions, and concurrent assertions. Implication assertions follow the format of $(a \rightarrow b)$ where a and b can be sequence of expressions. Here, a is considered as the antecedent while b is considered as the consequent. Implication assertions simply monitor sequences based on satisfying specific criteria. For example, A2 in Fig. 4 ($s2$ in Table 4) is an implication-type security assertion. Immediate assertions are in the format of $assert(a == b)$, that check a property when the control reaches an exact location in the code. For example, A7 in Fig. 4 ($c1$ in Table 3) is an immediate type security assertion. Concurrent assertions are in the format of *assert property*$(\neg(a\&b))$ that are checked in each clock cycle to verify the behavior. Based on safety and liveness properties, assertions outlined in Table 7 can be divided into two categories. The safety properties (e.g., A1 in Table 7) try to ensure that nothing bad will happen during execution. The liveness properties (e.g., A5 in Table 7) try to ensure that something good will eventually happen.

5.2 Security-Aware Trace Signal Selection

Since NoC design contains numerous signals, selecting appropriate signals becomes a challenging problem during the design stage. Algorithm 1 outlines the proposed method for selecting trace signals to be stored in on-chip trace buffers to maximize the coverage of security assertions. With the help of dependency graph analysis, the trace signals are selected for aiding in post-silicon validation. In Algorithm 1, sv represents design variables (i.e., the variables in the RTL design). Similarly, Ω represents the modules of the design (whereas M denotes the total number of modules in the RTL design). During execution, the on-chip trace buffers contain the traced signals (denoted by $Tr_{signals}$ in Algorithm 1) that need to be dumped for error/vulnerability localization using fine-grained analysis.

Algorithm 1: *TraceSignalSelection*

```
Input: $Design, M, SA$
Output: $Tr_{signals}$
1  $sv \leftarrow \emptyset$ ;
2  $Design$ = RTL description of the $Design$;
3  $M$ = number of modules of $Design$;
4  $\Omega = \{\Omega_1, \Omega_2 ..... \Omega_M\}$;
5  $SA$ = security assertions;
6  for each module $\Omega_i$ in $\Omega$ do
7      $\mathbf{A} \leftarrow$ assertion(s) related to $\Omega_i$ from $SA$;
8      $sv_i \leftarrow$ variables from $\Omega_i$ present in $\mathbf{A}$;
9      $sv \leftarrow sv \cup sv_i$;
10     $DG_i \leftarrow$ construct dependency graph for $sv$;
11     $Edge_i \leftarrow$ calculate number of edges connected to
         each $sv_i$ from constructed dependency graph;
12 end
13 Rank variables of $sv$ across $\Omega$ by $Edge_i$ values;
14 $Tr_{signals} \leftarrow$ variables from $sv$ as per given
     trace-buffer width limit;
```

To enable effective on-chip debug and security validation, trace signals must be selected carefully. It is a major challenge to identify efficient trace signals due to the exponential nature of possible trace signal combinations as well as conflicting requirements such as error detection and internal visibility enhancement [39]. We explain the proposed algorithm using an illustrative example. Consider the following

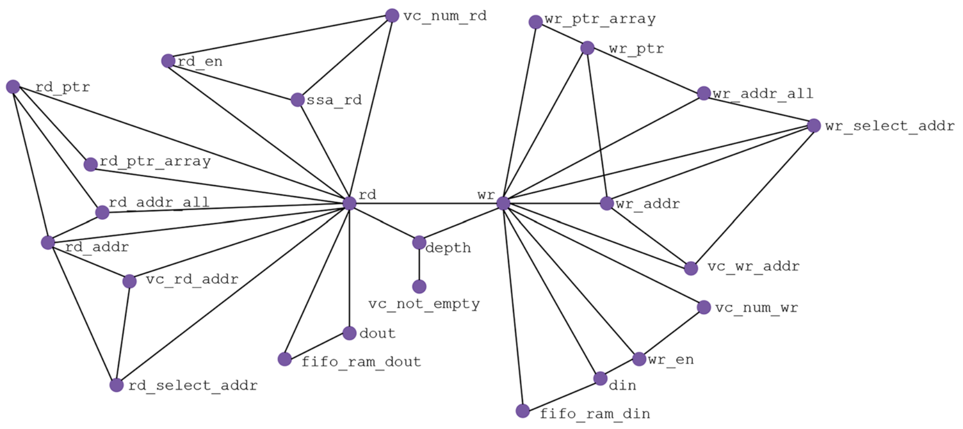


Fig. 6 An example signal dependency graph for flit_buffer

SystemVerilog assertion: $(a == 1)\&\&(b == 0) \mapsto (c == 0)$. Here, c is the destination signal, and a and b can be register-variables (flip-flops), primary inputs or wires (internal nets). The above assertion basically means that signal c is false when signal a is true and signal b is false. The condition present in the left-hand side (referred to as antecedent) can be translated into triggers and the same applies to the signal condition in the right-hand side (referred to as consequent). The goal of the signal selection is to trace variables that will be able to infer the values of signals a, b and c.

One illustration of the dependency graph is shown in Fig. 6 where the nodes represent different signals in the design and the edges depict the dependencies between them (inferred from the assignments in the RTL design description). This illustration corresponds to flit_buffer module of the NoC design. Based on these graphs, we select the signals that are maximally connected with other signals. The underlying reasoning is that maximum number of signals needed for functional behavior checking can be obtained for selection. To select variables related to security, we analyzed different types of security assertions developed in Sect. 4.1. Thereafter, the variables that were involved in the security assertions are chosen as probable candidates of trace signals. We perform a commonality search between the variables chosen from the security assertions and those chosen from dependency variable analysis.

As discussed in Sect. 6.1, the trace buffer width is limited and a portion of it is used by the trace header data. Therefore, we cannot select all the signals to the trace buffer. We have to find the most beneficial signals that can be used to regenerate other signals during the offline analysis. For this task, we have generated the variable dependency graph for each component of the NoC design. Then we order (sort) all the signals in different modules based on their connectivity (number of edges) with the other signals. This method arranges all the variables in each module in descending order of their restoration capability. Then we have selected the most relevant signals for a particular trigger giving the priority based on the ordered variables until we reach the trace width limit. For triggers implemented at route_mesh and arbiter, the trace width was enough to fit all the variables. However, for several flit_buffer triggers that had more signals, we applied the above technique to select the most profitable ones in terms of restorability. For example, the selected trace signals for flit_buffer related triggers are listed in Table 8.

Note that our objective is to maximize assertion coverage with minor impact on overall observability. Therefore, we cannot take signals only from assertions. If we have a lot of assertions in the design, it is possible that selecting signals from only assertions can maximize both overall observability and assertion coverage.

5.3 Post-silicon Debug of Security Vulnerabilities

After the activation of on-chip triggers, the fixed number of trace buffers can store certain important information related to the design execution. The contents of these buffers need

Table 8 Selected trace signals for flit_buffer triggers

Trigger ID	Signals
T1/T2 (wr)	depth,wr_ptr,wr_ptr_next,wr_addr,vc_wr_addr,wr_en
T1/T2 (rd)	depth,rd_ptr,rd_ptr_next,rd_addr,vc_num_rd,vc_rd_addr,rd_en
T3	depth,rd_ptr,rd_ptr_next,rd_addr,vc_num_rd,vc_rd_addr,rd_en
T4	depth,wr_ptr,rd_ptr,wr_addr,rd_addr,wr_en,rd_en
T5	flit_source(dout),flit_destination(dout),parity_data
T6	depth,rd_counter,wr_counter

to be off-loaded and analyzed for several purposes. The contents of these buffers provide the understanding of internal signals after the activation of triggers leading to analysis of the bug (or, the security threat) in a fine-grained manner.

Algorithm 2: *PostSiliconTraceAnalysis*

```
Input: $D_{tr}$, $SA$
Output: $ValidationResult$
1  $D_{tr}$= data from traced flip-flops/signals;
2  $ImpSA$ ← Implication-type assertions from $SA$;
3  for each $ImpSA_i$ in $ImpSA$ do
4      $p_a$ ← antecedent signals of $ImpSA_i$;
5      $p_c$ ← consequent signals of $ImpSA_i$;
6      Check signals values of $p_a$ in $D_{tr}$;
7      if $p_a$ signal values as per $ImpSA_i$ then
8          Check signals values of $p_c$ in $D_{tr}$;
9          if $p_c$ different from $ImpSA_i$ then
10             $ImpSA_i$ fails;
11         end
12         if $p_c$ signal values as per $ImpSA_i$ then
13             $ImpSA_i$ passes;
14         end
15     end
16     if $p_a$ different as per $ImpSA_i$ then
17         $ImpSA_i$ passes vacuously ;
18     end
19 end
20 $ValidationResult$ ← failed/passed $SA_i$;
```

The methodology for offline trace analysis for implication-type assertions is presented in Algorithm 2. The validation algorithm relies on checking the security assertions (*SA*) on the off-loaded data (D_{tr} in line 1) from the trace buffer. This checking of off-loaded data in line 8 is based on the signal value comparisons with the antecedents and consequents of the respective security assertions (*SA*). Note that for non-implication type of assertions, the validation/checking is relatively simpler and based on the comparison of values of signals in the respective assertions. The observed violation (*ValidationResult* in line 20) of the security property can hint towards a possible attack scenario. Note that because of the on-chip trigger framework, it becomes compulsory that the respective trigger must have been activated. Therefore, with the help of this framework, we can ensure that the detection of the security attacks is achieved in a quick manner with minimal detection latency. Signal selection and post-silicon validation results are presented in Tables 9 and 13, respectively.

6 Experiments

This section demonstrates the effectiveness of the proposed NoC trust validation framework. First, we describe the experimental setup. Next, we present the vulnerability injection mechanism and how effective the proposed framework is for pre-silicon and post-silicon security validation. All data generated or analyzed during this study are included in this section.

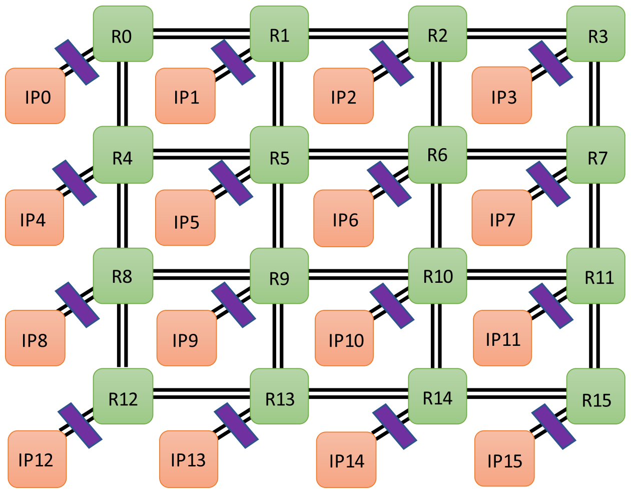


Fig. 7 4×4 Mesh NoC-based SoC used for the experimental setup

6.1 NoC Benchmark with Re-configurable Mesh

We created a re-configurable mesh NoC setup (Fig. 7 present the 4×4 configuration) consisting of “mor1k” processors in each core using the open-source ProNoC tool [40]. This Verilog RTL design has the configuration parameters of the NoC as shown in Table 9. A simple message passing scenario was designed to send three packets of data from each IP core to the IP core numbered 10. The designed scenario was implemented in C programming language. Using mor1k tool-chain, the required binaries were created for each IP core. The binary files were then subsequently placed in the RAM modules of relevant IP cores.

The simulation was done using ModelSim to verify the behavior. Once the functional accuracy of the experimental setup was verified, the selected security properties were implemented using SystemVerilog assertions. The assertions were implemented at the corresponding router components in the NoC RTL model. Table 10 provides information about the security assertion implementation. The first column provides the module (file) name in the ProNoC benchmark suite. The second column indicates the assertions implemented in that module.

We considered both safety and liveness related assertions introduced in Sect. 4.1. Implementation of liveness assertions is more complex compared to safety assertions since liveness behaviors include the “eventual” operator. Finding an exact upper bound for “eventual” operator is not possible using only the RTL design. Therefore, we derived an upper bound for the consequent to happen using simulations and used in the assertions.

In order to ensure that the generated assertions are valid, we have used directed test generation method discussed in Sect. 4. Once the pre-silicon assertions are validated, they can be mapped as post-silicon checkers. Note that SystemVerilog assertions are not synthesizable as post-silicon checkers. Previous work has proposed several alternatives to address this. Omar et al. [41] proposed a method that generates RTL netlists from assertions. We propose a different approach by creating equivalent trigger logic corresponding to each assertion. For example, Listing 1 shows the SystemVerilog description of assertion A9 as well as its equivalent trigger logic representation.

Listing 1: Synthesizable Verilog implementation of the trigger and trace logic corresponding to the A9 assertion shown in Table 7.

```
//Trigger
assign trigger_9 =
    !((dest_x > current_x && destport_next==EAST)
    || (dest_x < current_x && destport_next==WEST)
    || (dest_y > current_y && destport_next==SOUTH)
    || (dest_y < current_y && destport_next==NORTH)
    || (destport_next==LOCAL));
//Trace
assign trace= trigger_9?
    {{5{1'bX}},4'd11,current_x,current_y,dest_x,
    dest_y,destport,destport_next,{13{1'bX}}}:48'd0;
```

For post-silicon validation, we injected vulnerabilities and performed different debug experiments using Modelsim simulator. A centralized trace buffer was created, with a buffer length of fixed number of bits. (In the case of 4×4 configuration, trace width is 48 bits). Individual trigger circuits were designed and implemented to convert each of the security assertion (discussed in Sect. 4.1) to a synthesizable trigger logic and all the SystemVerilog assertions were removed from the design. The Design-for-Debug (DfD) circuit was created with a dedicated packet structure for the trace. Trace buffer width used for different NoC configurations is presented in Table 14. Figure 8 illustrates the bit structure for the trace packet. The first bit (R/IP) is used to identify the source of the trace, whether it was originated from the NoC or an IP. The next set of bits represent the routerID or IP-ID (4 bits for representing 16 routers/IPs in 4×4 instance). The next four bits represent the trace ID. The remaining bits store the selected trace signals.

Table 10 Mapping of implemented assertions to different NoC modules

Module	Implemented Assertions
flit_buffer.sv	$A1, A2, A3, A4, A5, A6$
route_mesh.sv	$A7, A8, A9$
arbiter.sv	$A10, A11$
main_comp.sv	$A12$

6.2 Pre-silicon Security Validation

In this section, we primarily discuss the test generation approaches for activating security assertions. Specifically, we explored the following two approaches for automated generation of directed tests.

6.2.1 Assertion Validation Using EBMC

The previous sections presented the approach for creating assertions to monitor vulnerabilities. In this section, we discuss the assertion validation results using the test generation framework outlined in Sect. 4.2. We used the bounded model checker EBMC [36] to generate directed tests. The generated tests were used to verify whether the added assertions are valid by activating these assertions. Table 11 presents the time (s) and space (MB) taken by EBMC [36] to activate the assertions. All the experiments were performed on a machine with an Intel i7-10510U CPU @ 1.80GHz CPU with 16GB RAM.

6.2.2 Assertion Validation Using Concolic Testing

As shown in Table 11, we are able to activate all but two assertions (A5 and A6) using EBMC [36]. These two assertions exceeded the capability of the model checker due to state space explosion (insufficient memory). We used concolic testing [37] to activate A5 and A6. Results in Table 11 show the time and memory requirement for test generation using

Table 9 Parameters used in the experimental setup

Parameter	Value
Router Type	Virtual channel (VC) based router
VCs per port	2
Payload width	32
NoC arbitration type	Round robin
Buffer flits per VC	4
Routing algorithm	X-Y routing
VC/SW combination type	Comb-Nonspec: VC allocator combined with non-speculative SW allocator where the validity of speculative requests are checked at the beginning of SW allocation

Table 11 Test generation time and memory requirement using EBMC and concolic testing to activate assertions

Framework	EBMC [36]										Concolic [37]	
Assertion	A1	A2	A3	A4	A7	A8	A9	A10	A11	A12	A5	A6
Time (s)	0.08	0.08	0.09	0.08	0.02	0.02	0.02	0.01	0.02	0.02	1.15	1.89
Memory (MB)	11.7	12.0	11.8	11.2	4.5	3.9	4.6	4.8	5.2	5.1	64.2	65.8

concolic testing. Overall, we generated tests for activating all the assertions mentioned in Fig. 4 using either EBMC or concolic testing. Therefore, we can confirm that the security assertions are valid and satisfy the design requirements.

6.3 Post-silicon Security Debug

This section has two objectives. First, we show that the synthesized checkers are capable of detecting any injected vulnerabilities. Next, we perform the overhead analysis of synthesizing security assertions on different NoC configurations.

6.3.1 Detection of Functional and Security Bugs

Several attack scenarios were created by closely following the threat models without considering the assertions or triggers. These bugs and attacks were inserted in different routers (selected randomly) to be activated at random clock cycles. Separate Modelsim simulations were carried out for each of these attack scenarios (one randomly inserted vulnerability at a time). Table 13 presents different cases of evaluation with the proposed debug framework. The first column provides different types of vulnerabilities. The second column indicates whether it is a functional bug or a security vulnerability. The third column indicates the activated trigger. The last column shows the average latency for each detecting each vulnerability. From the off-loaded trace contents, we checked for possible violation of the security assertions. As expected, triggers T5 and T6 require different numbers of cycles since they capture liveness behavior whereas the activation of the remaining triggers can be detected in one clock cycle.

6.3.2 Scalability Analysis via Overhead Calculation

As discussed in Sect. 4.1, we consider both safety (e.g., A1) and liveness (e.g., A5) related assertions. The corresponding triggers also inherit the inherent issues. As a result, the implementation is simple for safety related triggers (e.g., T1). However, implementation of liveness checking triggers (e.g., T5) involves complex logic than other types of triggers. In order to monitor the overhead of individual trigger, we have selected 4×4 multi-processor system-on-chip (MpSoC) configuration.

We have synthesized the design with the DfD circuit using Yosys [42] open synthesis tool. Table 12 shows the hardware overhead for individual security trigger. The first row provides the ID of each trigger. The second row shows the respective assertion. For example, if assertion A2 fails, trigger T2 will be activated. The last row shows overhead of each trigger.

In order to demonstrate the scalability of the approach, we have used several MPSoC configurations. Table 14 presents the hardware overhead for the proposed methodology for 2×2, 3×3, 4×4, 6×6 and 8×8 configurations. For each configuration, DfD circuit along with the trace width was accordingly adjusted. The total hardware overhead of the approach is negligible (less than 1% of the NoC area).

6.4 Comparison with Prior Efforts

The proposed approach is an attempt towards exploring the effectiveness of security assertions for validation. In existing literature, other methods have been proposed for the detection of design bugs or security threats. We present a qualitative comparison with other verification/validation methodologies in Table 15.

We analyze the merits of different methodologies on certain parameters: stage at which verification is being done (post-silicon or pre-silicon), types of bugs studied (functional or security or both), overhead (low or high over percentage of the chip area), scalability and whether runtime configurability is supported or not. *Our proposed framework outperforms existing methods in all of these categories.*

7 Applicability and Limitations

The primary objective of this paper is to demonstrate that assertion-based validation can be effectively utilized for designing security and trustworthy NoC architectures. Specifically, we have shown that we can utilize security assertions for pre-silicon trust validation as well as post-silicon security debug. Figure 3 shows different tasks of the validation methodology, including assertion generation, test generation for activation of assertions,

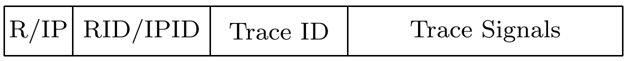


Fig. 8 Abstract trace packet structure that facilitates bug/threat localization and offline debug

Table 12 Overhead analysis for individual security triggers in 4×4 NoC configuration

Trigger ID	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	**T11**	**T12**	**Total**
Relevant Assertion	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**A11**	**A12**	
Trigger Overhead	0.01%	0.01%	0.02%	0.02%	0.19%	0.04%	0.01%	0.01%	0.01%	0.01%	0.02%	0.01%	0.36%

Table 13 Bug injections results with average cycles to activate the trigger from the activation of the bug

Vulnerability	**Type**	**Trigger**	**Latency**
Eavesdropping attack	Security	T5/T6	446/445
Packet corruption (Fifo input)	Security	T5	7
Packet missing (Fifo input)	Security	T1/T5/T6	1/330/329
Packet missing (Fifo output)	Security	T1/T5/T6	1/336/335
Packet dropping (Arbiter)	Security	T11	1
Wr/rd pointer fails when buffer is not full/empty	Functional	T1	1
Wr/rd pointer increments when buffer is full/empty	Functional	T2	1
Packet destination changing (Flit buffer)	Security	T5	450
Packet misrouting (Algorithm bug)	Functional and Security	T9	1
Invalid destination ports from route	Functional	T8	1
Multiple destination port selection (Route)	Functional	T7	1
Faulty multiplexer module	Functional and Security	T12	1
Multiple grants (Arbiter)	Functional	T10	1
Packet starvation (Flit buffer)	Security	T3	256
Packet duplication (Flit buffer)	Security	T4	1

and assertion coverage analysis, trigger generation, trace signal selection, post-silicon debug. Since each task is independent, we can replace it with an improved solution without affecting the other components. For example, we outline how to manually create NoC-specific security assertions in Sect. 4.1. This component can be improved by an automatic assertion generation framework, or using a combination of manually written assertions with automatically generated assertions. Similarly, we propose a connectivity based signal selection algorithm. Signal selection can be improved by utilizing existing trace signal selection algorithms [8, 9] considering diverse factors such as congestion (layout) constraints, restoration ratio, or dynamic signal selection based on the debug context. In other words, any signal selection algorithm is suitable in our framework as long as the objective is to maximize the assertion coverage with minimal impact on overall observability (restorability).

Table 14 Overhead for different NoC configurations

	NOC Size				
	2x2	3x3	4x4	6x6	8x8
Number of Triggers	252	567	1008	2268	4032
Trace Width (bits)	32	48	48	64	64
DfD Overhead	0.22%	0.17%	0.19%	0.57%	0.59%
Trigger Overhead	0.41%	0.35%	0.36%	0.81%	0.91%

Table 15 Qualitative comparison with other methods

Technique	Stage	Bug Type	Overhead	Scalable	Runtime
[39]	post-silicon	functional	low	no	no
[12]	pre-silicon	security	low	no	no
[1]	post-silicon	functional	high	yes	yes
[2]	post-silicon	functional	high	no	yes
[27]	post-silicon	functional	low	no	no
Prop.	**both**	**both**	**low**	**yes**	**yes**

8 Conclusion

The increasing utilization of Network-on-Chip (NoC) in designing modern System-on-Chip (SoC) architectures manifests into NoC being a prime target for attackers. This paper presented an approach of NoC trust validation using security assertions at both the pre-silicon and post-silicon stages. The proposed approach is complementary to existing NoC verification techniques that provide only design time guarantees. Our proposed approach is applicable for both design time (pre-silicon) analysis and runtime monitoring of post-silicon vulnerabilities. We defined a set of vulnerabilities for NoC architectures, and proposed security assertions to monitor these vulnerabilities. On-chip triggers derived from these security assertions provide an opportunity to enable the debugging features with minimal latency and area overhead. With the help of on-chip trace buffers, we presented a methodology for effective offline analysis of post-silicon traces. Experimental results using an NoC benchmark demonstrated that our approach is effective in NoC vulnerability analysis using security assertions with negligible hardware overhead.

Statements and Declarations

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

Competing Interests The authors have no relevant financial or non-financial interests to disclose.

Author Contributions All authors contributed to the study conception and design. Manuscript preparation, experimental setup, and analysis were performed by Aruna Jayasena and Binod Kumar. The assertion validation part was performed by Hasini Witharana. The first draft of the manuscript was written by Subodha Charles and Prabhat Mishra and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Data Availability All the data and the analysis results that were generated during the experiments are available in the manuscript.

References

1. Parikh R, Bertacco V (2014) Forever: A complementary formal and runtime verification approach to correct noc functionality. ACM Trans Embed Comput Syst 13(3s):104:1–104:30. https://doi.org/10.1145/2514871
2. Abdel-Khalek R, Parikh R, DeOrio A, Bertacco V (2011) Functional correctness for cmp interconnects. In: ICCD, pp 352–359. http://doi.org/10.1109/ICCD.2011.6081423
3. Arteris (2009) Flexnoc resilience package. http://arteris.com/flexnoc-resilience-package-functional-safety, [Online]
4. Foster H, Lacey D, Krolnik A (2003) Assertion-Based Design, 2nd edn. Kluwer Academic Publishers, USA
5. Tehranipoor M, Koushanfar F (2010) A survey of hardware trojan taxonomy and detection. IEEE Des Test Comput 27(1):10–25
6. Bhunia S, Tehranipoor M (2018) The Hardware Trojan War. Springer'18
7. Lyu Y, Mishra P (2020b) System-on-chip security assertions. https://arxiv.org/pdf/2001.06719.pdf
8. Mishra P, Farahmandi F (2019) Post-Silicon Validation and Debug. Springer
9. Mishra P, Morad R, Ziv A, Ray S (2017) Post-silicon validation in the soc era: A tutorial introduction. IEEE Design & Test 34(3):68–92
10. Love E, Jin Y, Makris Y (2011) Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. IEEE TIFS 7(1):25–40
11. Guo X, Dutta RG, Mishra P, Jin Y (2016) Scalable soc trust verification using integrated theorem proving and model checking. In: HOST
12. Sepulveda J, Aboul-Hassan D, Sigl G, Becker B, Sauer M (2018) Towards the formal verification of security properties of a network-on-chip router. In: ETS
13. Witharana H, Lyu Y, Charles S, Mishra P (2022) A survey on assertion-based hardware verification. ACM Computing Surveys (CSUR)
14. Boule M, Zilic Z (2008) Automata-based assertion-checker synthesis of psl properties. TODAES 13(1):1–21
15. Gupta A (2002) Assertion-based verification turns the corner. IEEE Des Test Comput 19(4):131–132
16. Vasudevan S, Sheridan D, Patel S, Tcheng D, Tuohy B, Johnson D (2010) Goldmine: Automatic assertion generation using data mining and static analysis. In: DATE, pp 626–629
17. Bombieri N, Busato F, Danese A, Piccolboni L, Pravadelli G (2019) Mangrove: An inference-based dynamic invariant mining for gpu architectures. IEEE Trans on Comp 69(4):606–620
18. Danese A, Bertacco V, Pravadelli G (2018) Symbolic assertion mining for security validation. In: DATE, pp 1550–1555
19. Dubrova E, Näslund M, Carlsson G, Smeets B (2014) Keyed logic bist for trojan detection in soc. In: SoC
20. Boraten T, DiTomaso D, Kodi AK (2016) Secure model checkers for network-on-chip (noc) architectures. In: GLSVLSI
21. Charles S, Mishra P (2020) Lightweight and trust-aware routing in noc-based socs. In: 2020 ISVLSI, IEEE, pp 160–167
22. Charles S, Lyu Y, Mishra P (2019) Real-time detection and localization of dos attacks in noc based socs. In: DATE
23. Prodromou A, Panteli A, Nicopoulos C, Sazeides Y (2012) Nocalert: An on-line and real-time fault detection mechanism for network-on-chip architectures. In: MICRO
24. Wassel H, Gao Y, Jason K, Huffmire T, Kastner R, Chong F, Sherwood T (2013) Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In: ISCA
25. Charles S, Logan M, Mishra P (2020) Lightweight Anonymous Routing in NoC based SoCs. In: DATE
26. Intel (2016) Using tinycrypt library, intel developer zone. http://software.intel.com/en-us/node/734330
27. Rout S, Basu K, Deb S (2019a) Efficient post-silicon validation of network-on-chip using wireless links. In: VLSID, pp 371–376
28. Rout S, Patil SB, Chaudhari VI, Deb S (2019b) Efficient router architecture for trace reduction during noc post-silicon validation. In: SOCC, pp 230–235
29. Rout S, Badri M, Deb S (2020) Reutilization of trace buffers for performance enhancement of noc based mpsocs. In: ASP-DAC, pp 97–102
30. Sepúlveda J, Zankl A, Flórez D, Sigl G (2017) Towards protected mpsoc communication for information protection against a malicious noc. Procedia Computer Science 108:1103–1112
31. Ancajas DM, Chakraborty K, Roy S (2014) Fort-nocs: Mitigating the threat of a compromised noc. In: DAC
32. Hussain M, Malekpour A, Guo H, Parameswaran S (2018) Eetd: An energy efficient design for runtime hardware trojan detection in untrusted network-on-chip. In: ISVLSI
33. Pasricha S, Dutt N (2010) On-chip communication architectures: system on chip interconnect. Morgan Kaufmann
34. JYV MK, Swain AK, Kumar S, Sahoo SR, Mahapatra K (2018) Run time mitigation of performance degradation hardware trojan attacks in network on chip. In: ISVLSI
35. Biswas AK, Nandy S, Narayan R (2015) Router attack toward noc-enabled mpsoc and monitoring countermeasures against such threat. Circuits Systems Signal Process 34(10):3241–3290
36. Mukherjee R, Kroening D, Melham T (2015) Hardware verification using software analyzers. In: ISVLSI
37. Lyu Y, Mishra P (2020a) Automated test generation for activation of assertions in rtl models. In: ASP-DAC
38. Boule M, Zilic Z (2005) Incorporating efficient assertion checkers into hardware emulation. In: ICCD, pp 221–228. http://doi.org/10.1109/ICCD.2005.66
39. Kumar B, Basu K, Fujita M, Singh V (2020) Post-silicon gate-level error localization with effective and combined trace signal selection. IEEE Trans Comput-Aided Des Integr Circuits Syst 39(1):248–261. https://doi.org/10.1109/TCAD.2018.2883899
40. Monemi A, Tang JW, Palesi M, Marsono MN (2017) Pronoc: A low latency network-on-chip based many-core system-on-chip prototyping platform. MICPRO 54:60–74

41. Amin O, Ramzy Y, Ibrahem O, Fouad A, Mohamed K, Abdelsalam M (2016) System verilog assertions synthesis based compiler. In: MTV
42. Clifford W (2013) Yosys open synthesis suite. http://www.clifford.at/yosys/