Journal of Electronic Testing (2024) 40:117–135
https://doi.org/10.1007/s10836-024-06100-2

Non-Invasive Hardware Trojans Modeling and Insertion: A Formal Verification Approach

Hala Ibrahim[1] · Haytham Azmi[2] · M. Watheq El-Kharashi[1] · Mona Safar[1]

Received: 1 November 2023 / Accepted: 10 January 2024 / Published online: 20 March 2024

Abstract
In modern chip designs, shared resources are used extensively. Arbiters usage is crucial to settle conflicts when multiple requests compete for these shared resources. Making sure these arbiter circuits work correctly is vital not just for their proper functionality, but also for security reasons. The work in this paper introduces a method based on formal verification to thoroughly assess the proper functional aspects of various arbiter setups. This is achieved through SystemVerilog assertions and model checking. Additionally, we explore a non-invasive method for the modeling and insertion of different types of hardware Trojans. These Trojans, with their unique triggers and payloads, are modeled formally without the need for any alterations to the actual circuit. The results provide a detailed analysis of the cost involved in running the formal verification environment on versions of arbiters that are free from Trojans. This analysis is carried out using Questa PropCheck formal analysis tool, which offers valuable insights into the time and memory resources required. Furthermore, the results highlights how the formally modeled and inserted Trojans interfere with hold criteria of the arbiters' properties, where at least a single property fires due to the inserted Trojan. This work can be extended to be a generic approach with the potential to validate both the proper operation and security aspects of complex systems.

Keywords Hardware arbiters · Model checking · Trojan modeling · Trojan insertion · Trojan trigger

1 Introduction

Arbiter circuits are massively used in all systems that depend on resource sharing mechanisms. Additionally, arbiters represent good candidates for malicious attacks that tend to inject hardware Trojans. These hardware Trojans can cause a Denial-of-Service (DoS), a deadlock, or data alternation. Modeling hardware Trojans that have different trigger conditions or payloads in the functional verification stage is something that needs modifications to the Design Under Test (DUT). This is not always achievable, as the DUT might be an encrypted IP or a gate-level netlist. This introduction will discuss the different arbiter circuits, formal verification principles, and hardware Trojan models.

1.1 Arbiters

Due to the scale of modern System-on-Chips (SoCs), shared resources are widely used across them. So, arbiter logic is the main conflict-handling mechanism in such systems. An arbiter is a device that allocates access to shared resources [9, 11]. These shared resources include shared buses, memories, buffers, or CPUs. Clients of shared resources request access to this resource and the arbiter gives a grant to only one client according to the arbiter's algorithm. Arbiters can be classified according to multiple criteria. First, it can be classified according to its grant time. Arbiters can be single-cycle grant, multi-cycle grant, or till-release grant. Second, it can be classified according to its fairness. It can have

Responsible Editor: O. Sinanoglu

✉ M. Watheq El-Kharashi
watheq.elkharashi@eng.asu.edu.eg

Hala Ibrahim
hala.ahmed259@gmail.com

Haytham Azmi
haitham@eri.sci.eg

Mona Safar
mona.safar@eng.asu.edu.eg

[1] Computer and Systems Engineering, Ain Shams Universty, Cairo, Egypt

[2] Microelectronics Department, Electronics Research Institute, Cairo, Egypt

weak-fairness, where every request is eventually granted. It can have strong fairness, where requests are served equally often, or have a First-In-First-Out (FIFO) fairness. Third, it can be classified according to the arbitration stages, where it can be a single-cycle arbiter or a multiple-cycle arbiter. According to the variations of the above criteria, arbiter types are as follows:

1. Fixed priority arbiter: A fixed priority is assigned to each request. Accordingly, the grant is served to the request with the highest priority. Fixed priority arbiter has weak fairness, as it does not guarantee grants for lower-priority requests.
2. Variable priority arbiter: Unlike fixed priority arbiters, a variable priority arbiter maintains a one-hot priority signal to determine the priority of each request. This priority signal changes from cycle to cycle. Variable priority arbiters have multiple types according to how the priority signal is driven.
    (a) Random priority arbiter: In a random priority arbiter, the priority signal is generated in a total random behavior with no knowledge about the requests or grants history. It is usually implemented through a pseudo random number generator that uses a Linear Feedback Shift Register (LFSR) to generate random priority each clock cycle.
    (b) Round Robin arbiter: Round Robin arbiters are the most widely-used arbiter circuits, as they achieve strong fairness between requests. A Round Robin arbiter operates on the principle that a request that was just served should have the lowest priority on the next round of arbitration. This is accomplished by generating the priority vector signal from the current grant vector signal.
    (c) Grant hold arbiter: In some applications, clients required uninterrupted use of the resource for several cycles. The duration of a grant can be extended in the arbiter by using a grant-hold circuit. The hold circuit leverages a hold vector signal that specifies when the resource is ready to be released.
    (d) Weighted Round Robin arbiter: Some applications require an arbiter that is unfair to a controlled degree, so that one requester receives a larger number of grants than another requester. A weighted Round Robin arbiter exhibits such behavior. Each requester is assigned a weight that indicates the maximum fraction of grants that the requester is to receive according to its fractions. A requester with a large weight receives a large fraction of the grants, while a requester with a small weight receives a smaller fraction. A weighted Round Robin arbiter can be realized by preceding an arbiter with a circuit that disables a request from an input that has already used its quota.
3. Matrix arbiter: A matrix arbiter implements the least recently served priority scheme by maintaining a triangular array of state bits w_{ij} for all $i < j$. The bit w_{ij} in row i and column j indicates that request i takes priority over request j. Each time a request is granted, it clears all bits in its row, and sets all bits in its column to give itself the lowest priority since it was the most recently served.
4. Queuing arbiter: A queuing arbiter provides FIFO priority. It accomplishes this by assigning a time stamp to each request when it is asserted. During each time step, the earliest time stamp is selected by a tree of comparators. The cost of the arbiter is largely determined by the number of bits used to represent the time stamp, since this determines the size of the registers in the stamp blocks and the width of the comparators in the comparator tree. Queuing arbiters are particularly useful at ensuring global (or system-level) fairness through multiple stages of arbitration.

An arbiter is the core device for any resource allocation conflict resolution. Accordingly, it is a tempting target for attackers to insert Trojans that cause unfair distributions of resources, DoS attacks (starvation), eavesdropping on the granted client.

1.2 Formal Verification

Formal verification is the process of checking whether a design satisfies some requirements (properties). It is based on a mathematical method to prove or detect whether the circuit implements all the functions of the circuit design. Figure 1 shows a simplified formal verification workflow, where a model checker is used to check if the DUT is equivalent to the properties describing it [24]. This is achieved through converting both the RTL and the assertion to internal mathematical equations. These equations might be solvable and in this case a counterexample is generated to show a waveform, where the property is

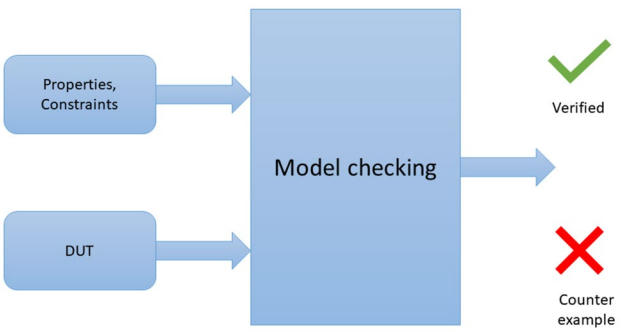

Fig. 1 Workflow of formal verification

Table 1 Properties components from a formal verification perspective

Component	Description	Example
Safety	It checks if an unexpected functionality happens	assert property (@(posedge clk) !(read & write));
Liveness	It checks if the correct behavior will eventually happen	assert property eventually (@(posedge clk) state == IDLE);
Vacuity	It checks if the property activation condition happens	assert property (@(posedge clk) req ⇒ ##[1:2] ack);
Sanity	It checks if the correct functionality can be achieved at any time	cover property (@(posedge clk) state == BUSY);
Deadlock	It checks if the observed deadlock can be escaped by some input sequence	N/A

Examples are written in SVA language

violated. Or, the mathematical equation is not solvable and this means the RTL holds against the property for all possible paths. For hardware formal verification, designs are described using Register Transfer Logic (RTL) languages, such as Verilog, or VHDL. Properties are described in an assertion language, such as SystemVerilog Assertions (SVA). The output from the model checking tool is either a pass or a fail (counterexample). A counterexample is a sequence of events that violates a property, and it is usually represented as a waveform. Any property can have single or multiple components, as illustrated in Table 1.

The motive behind using formal verification is stress-testing the DUT under all input stimulus in all states, unlike simulation that depends on random or constrained-random stimulus. Along with verifying the functionality of the system, the verification of the security aspects is also required. Security verification is to guarantee the system does not have any extra functionality that might affect the original functionality or leak confidential information. Formal verification can be extended to verify the design's security through identifying hardware Trojans. Given the extensiveness of formal verification, it is most-suited to verify security-related properties in the case of the inserted-Trojans by firing properties that include the Trojan location in their active logic. Formal verification does not only help in detecting the Trojans, but also helps in debugging their root causes by providing counterexamples.

1.3 Hardware Security and Trojans

A Trojan is a malicious modification of the circuitry of an Integrated Circuit (IC) chip [1, 33]. It is done during the design or the fabrication of a chip. A Trojan consists of two components, which are a Trojan trigger that specifies when the hardware Trojan affects the circuit and a Trojan payload that specifies how the hardware Trojan affects the circuit According to the different variations for a Trojan insertion phase, trigger, and payload [13], Fig. 2 illustrates the Trojan taxonomy. A Trojan can be classified according to the insertion phase, abstraction level, activation mechanism, effect on the circuit, location in the circuit, and physical characteristics [28, 29].

For a formal verification framework to be able to identify design Trojans, we need to have a complete set of properties that evaluate all the design's intended functionalities. This set of properties should hold to make sure there are no bugs inserted into the design and that there are no hardware Trojans. Inserting hardware Trojans without modifying the RTL is a challenging task. Formal methodologies can be used to model both the trigger and the payload of the most of the Trojans types.

The contributions of the paper are as follows:

1. Modeling of four different types of arbiters, which are Round-Robin, fixed-priority, variable-priority, and matrix arbiters.

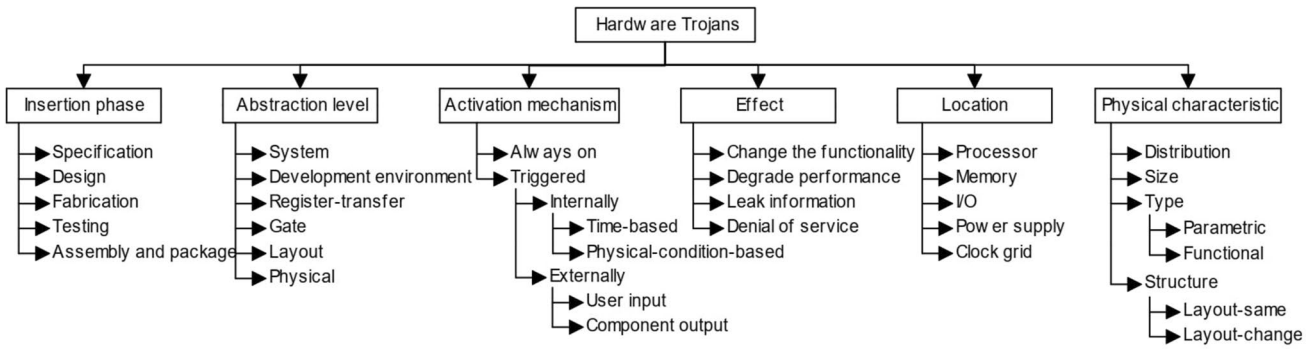

Fig. 2 Hardware Trojans classification [33]

2. Formally-verify the full requirements of each arbiter through developing SystemVerilog assertions. The required cost of running the formal verification environment is also calculated.
3. Formally-model and inject hardware Trojans against the fully-verified circuits.
4. Verify that the inserted Trojan fires at least one property to alert the verification engineer and accordingly indicate full coverage.

This is accomplished using a generic formal verification environment that can be applied on different arbiter algorithms and sizes. Additionally, the formal environment is able to insert different types of hardware Trojans and checks the effectiveness of the verification framework to detect them. In this introduction, different arbiter types are illustrated. Then, the basic concepts of formal verification and assertion-based verification are explained. At last, an illustration of different types of Hardware Trojans is also discussed.

This paper is organized as follows. Section 2 presents the work related to arbiter verification using formal verification, security verification, and hardware Trojans insertion and detection. Section 3 illustrates the proposed methodologies and detailed implementation. Section 4 discusses the obtained results. Section 5 concludes the paper.

2 Related Work

The work presented in this section covers studies related to formal modeling, verification, and validation of different types of arbiter circuit, considering it to be one of the main components used in hardware systems. It also covers work related to formal verification being used to identify hardware Trojans. Additionally, it presents the previous work related to methodologies in Trojan insertion and detection mechanisms.

2.1 Verification of Arbiters

Slimane et al. provided a high-level formal model using the BIP software for specifying arbitration protocols in an SoC architecture. The authors employed the RTD-finder model checker tool for formal analysis to assess deadlock freedom, protocol verification, and mutual exclusion of the used arbiters [39].

Ikram et al. presented an overview of building a formal verification environment for verifying a multi-stage arbiter circuit. They discussed the arbiter under test, preparation of the design for formal verification, and the set of properties tested, including exclusive grants and latency between requests and grants for same-priority agents [19].

To calculate the request-to-grant delay for random priority arbiters, the property was transformed into a bounded property. The delay value was determined using Complete Random Sequences (CRSs) generated by a pseudo-random number generator (PRNG) for random priority assignment. The CRS is the number of cycles required to cover all different random numbers a PRNG generates. The bounds of the CRS length were known, enabling the calculation of the request-to-grant delay [22]. Abstractions were made to switch between liveness properties and bounded-liveness properties, as well as between random-seed PRNG and fixed-seed PRNG, resulting in improved bug detection performance within a timely manner [4].

A generalized asynchronous arbiter was implemented and formally verified to be free from deadlocks and hazards. However, stalling possibilities in such an arbiter still depended on the correctness of the priority function [14].

A traffic-aware adaptive hybrid arbiter was developed based on a basic matrix arbiter. Simulation-based verification was used to test various aspects of the modified arbiter, including last-recently-served operation, arbiter response under light load, and blocking of input request signals [23].

The MODEVES framework, MOdel-based DEsign Verification for Embedded Systems, was proposed to support both static and dynamic assertion-based verification. The arbiter was used as a case study to validate properties related to granting requests, correct idle states, and fairness among requests [3].

Formal concepts were applied to reduce the logic inserted by High-Level Synthesis (HLS) tools. The HLS tools handle the micro-architecture and transform untimed or partially timed functional code into fully timed RTL implementations. Formal analysis and satisfiability were used to evaluate whether specific requests from certain sources to specific resources were never asserted, enabling the removal of arbitration logic [7].

As an application of formal verification to assess arbitration logic functionality, formal verification was used to check the security properties of a Network-on-Chip (NoC) router along with verifying other components of the router, such as buffers, routing algorithm, crossbar switch. The attacks were modification, duplication, packets dropping, DoS, and timing attacks. It mapped different types of security attacks to a set of properties that should hold to make sure this attack does not affect the NoC security [34].

2.2 Trojan Modeling

A Hardware Trojan Horse (HTH)-based attack was presented on the True Random Number Generator (TRNG) of an FPGA-based cryptosystem. A Trojan Horse attack is often described as Trojan that is made to look legitimate. The proposed Trojan reduced entropy and increased predictability of generated keys. The attack went undetected when analyzing the FPGA bitstream, but a detection mechanism using Wavelet Transform on the compromised bitstream was proposed [15].

For hardware Trojan implantation, a method utilizing Evolvable Hardware (EHW) technology was proposed. EHW is the application of evolutionary algorithms to hardware systems during design, operation, or both. It involved decomposing the circuit, redesigning the sub-circuit, and replacing the original circuit with the evolved circuit to implant a hidden hardware Trojan [26].

In the context of network communication, a network packet redirection attack was presented. This attack involved the insertion of a hardware Trojan into reconfigurable network devices. The Trojan had probabilistic functionality with a multi-level trigger mechanism. In the MAC layer attack, the Trojan redirected a frame to an incorrect port, potentially enabling eavesdropping and denial-of-service. In the network layer attack, it forwarded all IP packets through a suboptimal router port, causing a denial-of-service attack on the receiver [30].

An implementation of hardware Trojans in a floating-point multiplier was discussed. The Trojans were implemented by adding new components (multiplexers and registers) to the RTL. Two types of Trojans were considered: an Add-Sub functionality Trojan and an Arithmetic-Shift Trojan. These Trojans were designed to activate randomly during the multiplication process [31].

An Artificial Intelligence (AI)-guided framework for automatic Trojan insertion was proposed. The framework aimed to generate a large population of valid Trojans for a given design by mimicking the properties of a small set of known Trojans. Machine learning models were used to analyze the structural and functional features of existing Trojan populations, generating "virtual Trojans" for the design. These Trojans were then bound into the design by matching their functional and structural properties with suitable internal logic structure nets [8].

2.3 Trojan Detection and Verification

Formal verification techniques are employed to detect specific types of hardware Trojans in SoCs. Each IP within an SoC can independently satisfy the required security properties, despite the possibility of malicious IPs. The overall trustworthiness of the SoC is ensured by the compliance of all IPs with the bus protocol, typically the AMBA bus. The entire SoC is formally modeled by generating a gate-level netlist and creating Finite State Machines (FSMs) using the REFSM tool. Formal analysis is conducted using the UPAAL model-checking tool, with a focus on checking information leakage and DoS attacks against the FSMs [18].

Trojans are classified based on trigger, behavior, and data feedback, and a set of formal-analysis-based approaches for Trojan detection is proposed. Techniques such as model checking using path tree logic (PTL) properties, reachability analysis, equivalence checking using Reduced Order Binary Decision Diagram (ROBDD), and structural analysis are suggested based on the formal modeling of hardware designs and Trojans. The choice of technique depends on factors such as the presence of new states in the system's FSM and the availability of a golden reference [36]. A practical hardware Trojan detection model, known as the golden layout model, was introduced. This model eliminated the need for fabricated golden chips and utilized a calibration algorithm to account for process variations and random noise. Trojan detection was formulated as a two-class classification problem [27].

To detect Trojans, a multistep flow is employed, considering their occurrence under rare conditions typically modeled using if-else, switch-case, or loop statements. The process starts with simulation tests on an IP software model, followed by parsing the C code to identify untested conditional statements. Assertions with opposite conditions are inserted to check the reachability of potential Trojan triggers. Failed assertions provide counterexamples, which are obtained by parsing the waveform files and extracting test vectors for the software model. The new test vectors are then used to detect and analyze behaviors that could potentially trigger a Trojan [38].

In the formal verification of Third-Party Intellectual Properties (3PIPs), the focus is on identifying information leakage, particularly related to confidential data like encryption keys. Trojans that leak the entire key, part of the key, or functions of the key (inverted key) can be detected. The verification also includes identifying triggers that are consistently active or span multiple clock cycles. However, the employed bounded model checking has a limitation in detecting Trojans triggered beyond a maximum number of clock cycles [35].

A survey work focused on presenting formal analysis techniques for hardware Trojan detection, including theorem proving and equivalence checking mechanisms. The survey discussed various approaches such as proof-carrying hardware (PCH), proof-carrying based netlist verification, and proof-carrying code [16]. The PCH combines a hardware module and a formal proof of safety which adheres to a previously established safety policy.

Tang et al. proposed a hardware Trojan detection method called D2ACGAN (Dual Discriminator Assisted Conditional Generation Adversarial Network) to address the limitations of side channel information-based methods. The D2ACGAN model can learn valid information from the tested chip, distinguish between side channel data with and without hardware Trojans, and classify hardware Trojans using extended data [32].

To overcome the limitations of golden chip and test pattern-based approaches, SC-COTD (Sequential/Combinational Controllability and Observability Trojan Detection) was proposed as an effective hardware Trojan detection method. It utilized a machine learning approach with an ensemble classifier based on k-means clustering and a decision-making procedure using majority voting [37].

Bitstream reverse engineering was employed for hardware Trojan detection. The approach involved inspecting the bitstream, identifying FPGA components from the layout, and detecting malicious configurations. This bottom-up approach enabled the detection of malicious circuits that are difficult to identify using conventional design analysis methods [10].

Kumar and et al. implemented a secure SoC prototype based on the RISC-V architecture. The prototype included a secure boot protocol ensuring first-instruction integrity, a key management framework based on a physical unclonable function (PUF), and a memory protection unit to ensure confidentiality and integrity properties. PUFs are innovative physical security primitives that produce unclonable and inherent instance-specific measurements of physical objects. Additionally, a proposal for a new side-channel attack countermeasure strategy was evaluated using a RISC-V out-of-order core [25].

A lightweight mutual authentication and key agreement protocol named ASSURE was proposed, based on the Rivest Cipher (RC5) and PUFs. The effectiveness of this protocol was evaluated through rigorous security analysis under various cyber-attack scenarios. A wireless sensor network using typical IoT devices was used to evaluate the overhead of memory and energy consumption [41].

A runtime system-level method was introduced to detect hardware Trojans in Third Party Behavioral Intellectual Properties (3PBIPs). The method proposed the inclusion of a small hardware Trojan detection circuit called trust filters to detect hardware Trojans at runtime. With the help of a cycle-accurate simulation model, it was possible to fine-tune these filters so that the overall system had no performance degradation [42].

Hamlet et al. developed a design-time method for automatically and systematically modifying portions of a design that exhibit characteristics of hardware Trojans. After each modification, the functionality of the design is verified against a comprehensive simulation suite to ensure that the intended circuit functionality was not changed [17].

A modeling approach was presented to capture rare nets using word-level statistics of the inputs. The approach provided the capability to locate macro-block(s) in an RTL design to estimate the rare triggering nets. Given the RTL description of a design, the authors decomposed the design into a subset of basic arithmetic modules, by which they evaluated the design for a quick estimation of hardware Trojan-vulnerable macro-block(s) [20].

A threat model for Microprocessor-System-on-Chips (MPSoCs) stemming from the use of malicious third-party NoCs was explored. The authors illustrated a rogue NoC (rNoC), which can selectively disrupt the perceived availability of on-chip resources, thereby causing large performance bottlenecks for the applications running on the MPSoC platform. Further, to counter the threat posed by rNoC, they proposed a runtime latency auditor that enabled an MPSoC integrator to monitor the trustworthiness of the deployed NoC throughout the chip lifetime [21]. Manipulating the output port of each incoming flit in a network-on-chip re-directs all flits to one port, causing disruption in tile communication. Then, the effect of this inserted hardware Trojan is detected through performance degradation across buffer utilization, virtual channel utilization, number of flits processed, and link utilization [5]. In the related work, no work presented how to verify all types of arbitration circuits with different specifications and sizes. Additionally, no work was focused on pre-silicon formal verification that models non-invasive hardware Trojans without modifying the design or that has the ability to detect several types of hardware Trojans using formal verification.

3 Proposed Framework

The work in this paper is divided into three parts. First, we discuss the modeling of the arbiter DUT. Second, we discuss the formal verification framework used to fully verify the arbiters. Third, we discuss how to leverage the formal verification framework to model and insert different types of hardware Trojans to assess the security aspect of the arbiter.

3.1 Selected Arbiters

In this paper, experiments are performed on four arbiter types, which are fixed-priority arbiter, Round Robin arbiter, matrix arbiter, and random-priority arbiter. All arbiters are discussed in Subsection 1.1. RTL code of all arbiters is written in a generic format in Verilog, where number of requests and grants can be adjusted through parameters. In the case of fixed priority arbiter, which depends on a priority decoder, a developed software utility is used to generate the RTL based on the number of requests. The Round Robin arbiter is implemented using two simple priority arbiters with a mask [40]. For the matrix arbiter, the implementation leveraged Verilog generate statements to account for the columns and rows values of the matrix [6]. In the case of random-priority arbiter, it is implemented through an LFSR followed by a fixed-priority arbiter. The LFSR is implemented using a shift register that is configured to produce the longest CRS [2]. According to the number of requests of the arbiter, certain bits from the LFSR's output are encoded to be the requests' priority.

3.2 Formal Verification of Arbiters

We have two types of properties that should be tested against the arbiters. General properties that should prove for all

Table 2 Protocol-compliant properties

Round Robin	Fixed-Priority	Matrix	Random-Priority
No two consecutive grants are given to the same request	Grant is given to a request only if all the higher-priority requests are not asserted	Grant is given to the least-recently-served request	Request-to-Grant delay through bounded liveness

arbiters. Protocol-compliant properties that depend on the type of the arbiter. General properties include deadlock freedom or starvation-free arbitration, no grant without a request, and exclusive grants. Table 2 shows protocol-compliant properties for each arbiter type.

Properties are needed to be written in SVA and run against a formal verification tool for all the types and sizes of arbiters. Table 3 shows the multiple iterations that were needed for correct interpretation of requirements. It also shows the modifications or additions required for the formal environment. A python utility is created to ease the automation of the following:

- Creating RTL files with the correct number of requests and grants, specially for the arbiters that depend on decoders.
- Creating SVA files and the SV bind module.
- Generating scripts to automate running the formal analysis tool.
- Making sure the exit criteria is met, that is, all properties are proven.

SVA files generation is generic based on the arbiter size and type. So, the set of instantiated assertions and assumptions vary according to the run's configurations.

The syntax of properties are shown in Listing 1–8. Some properties work on a single request and grant level. Some others work on a request and grant vector level. Signals "req_" and "gnt_" notate single requests and grants. Signals "req" and "gnt" notate request and grant vectors, respectively. Along with the assertions, the assumption in Listing 9 is used to assume that a request is high until granted. Additionally, for the LFSR used in the random priority arbiter, the property in Listing 10 is used to ensure having random output from the LFSR at least for two consecutive cycles, where "random_q" is the output of the LFSR.

Listing 1: Deadlock freedom property (vacuity and liveness)

```
property deadlock_free (clk, rst, req_, gnt_);
    @(posedge clk) disable iff (rst)
       $rose(req_) |=> s_eventually(gnt_);
endproperty
```

Listing 2: Mutual exclusion property (safety)

```
property multual_exclusion (clk, rst, gnt);
    @(posedge clk) disable iff (rst)
    $onehot(gnt) || gnt=='0;
endproperty
```

Listing 3: No grant without request (vacuity and safety)

```
property no_gnt_without_req (clk, rst, req_, gnt_);
    @(posedge clk) disable iff (rst)
    gnt_ |-> $past(req_);
endproperty
```

Listing 4: Round Robin protocol (vacuity and safety)

```
property rr_protocol (clk, rst, gnt_);
    @(posedge clk) disable iff (rst)
      gnt_ |=> !gnt_;
endproperty
```

Table 3 SVA equivalent properties

Original Property	Reason of modification	What is required
request implies grant	A counterexample is produced that a request is asserted and then de-asserted before it is granted	An assumption is needed to assume that a request is high till granted
request implies grant	A counterexample is produced that a request is high for multiple cycles, and it is granted one time	The property needs to be changed to indicate the rising of a request from zero to one (rising edge)
grant is onehot	A counterexample is produced, where we have zero requests and accordingly zero grants	The property needs to be changed to check for the condition, where no requests are available
request-to-grant delay	A counterexample is produced (Round Robin arbiter) as it was assumed to be # requests - 1	The property needs to be changes as the Round Robin arbiter inserts some stall cycles between consecutive requests and the request-to-gnt delay in this case is 2*# requests - 1

Listing 5: Matrix protocol (vacuity and safety)

```
property matrix_protocol(clk, rst, gnt_, req,req_);
    @(posedge clk) disable iff (rst)
    gnt_ & $onehot(req) & !req_ |=> !gnt_;
endproperty
```

Listing 6: Fixed-priority protocol (vacuity and safety)

```
property fixed_priority_protocol(clk, rst, req, gnt, i);
    @(posedge clk) disable iff (rst)
      req[i] & (~|req[i-1:0]) |=> gnt[i];
endproperty
//For the first request
property fixed_priority_protocol_0(clk, rst, req, gnt);
    @(posedge clk) disable iff (rst)
      req[0] |=> (gnt[0]);
endproperty
```

Listing 7: Maximum latency (vacuity and bounded liveness)

```
property max_latency (clk, rst, req_, gnt_);
    @(posedge clk) disable iff (rst)
    $rose(req_) |-> \#\#[1:2*WIDTH-1] $rose(gnt_);
endproperty
```

Listing 8: Maximum latency for random priority arbiter (vacuity and bounded liveness)

```
parameter LONGEST_CRS = 2**LFSR_WIDTH - 1;
property max_latency_random_priority_arbiter
    (clk, rst, req_, gnt_);
    @(posedge clk) disable iff (rst)
    $rose(req_) |-> \#\#[1: LONGEST_CRS] $rose(gnt_);
endproperty
```

Listing 9: Request is high until granted

```
property req_high_till_gnt (clk, rst, req_, gnt_);
@(posedge clk) disable iff (rst)
      $rose(req_) |-> (req_ until gnt_);
endproperty
```

Listing 10: LFSR random output

```
property random_lfsr (clk, rst, random_q);
    @(posedge clk) disable iff (rst)
      (##1 $changed(random_q);
endproperty
```

3.3 Formal Modeling and Verification of Hardware Trojans

To formally-verify the security aspects of a system, verification engineers tend to modify the RTL to inject Trojans to make sure the formal verification environment is able to catch them. This is not doable in all cases as the hardware might be an encrypted IP, or a gate-level netlist. Accordingly, leveraging the formal verification concepts to insert hardware Trojans without modifying the RTL is very beneficial. Before elaborating on the types of Trojans that can be modeled, some formal concepts need to be introduced, as follows:

- A cut signal: this is a formal way of manipulating a netlist and abstracting all the logic that drives a specific net, making this net a free-input.
- A constraint or an assumption: this is a formal way to guide the formal analysis to constraint the control points of the free input to a certain behavior to avoid getting the design into an invalid state.
- A virtual wire: this is a way to generate a wire on-the-fly, which can be used during the formal analysis to drive a certain net.

A flowchart that illustrates the Trojan modeling, insertion, and verification flow is shown in Fig. 4. A cut signal is conditionally or unconditionally applied to one of the DUT's internal states. These internal states are captured during an exploratory run to capture the DUT's structure. The conditions of applying the cut signals are determined by constraints applied to the cut signal enablers, which can depend on a primary input/output value, another internal state, or upon a certain number of cycles to be triggered. The payload of the Trojans (driving value of the cut signal) is either left for the formal analysis to decide, given as a random value set by the proposed framework, or a certain value set by the verification engineer.

The diagram shown in Fig. 3 illustrates how the proposed methodology is captured. The RTL files of the arbiter are bound to the SVA assertion in order for the formal verification to drive

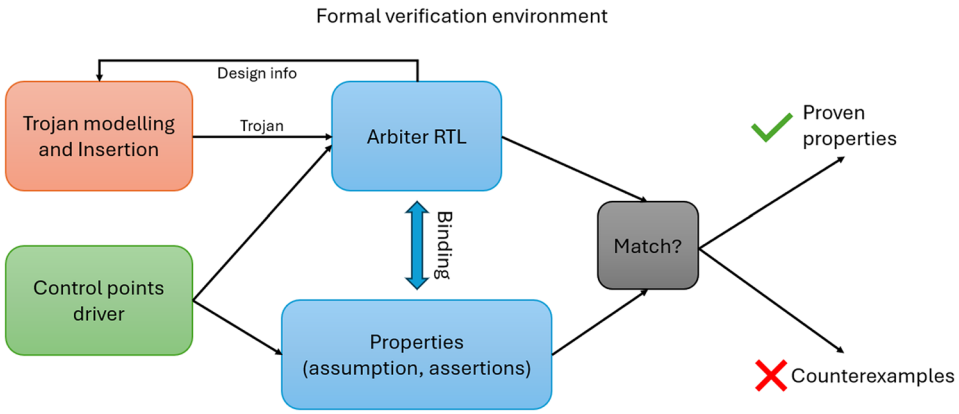

Fig. 3 Block diagram of the proposed methodology

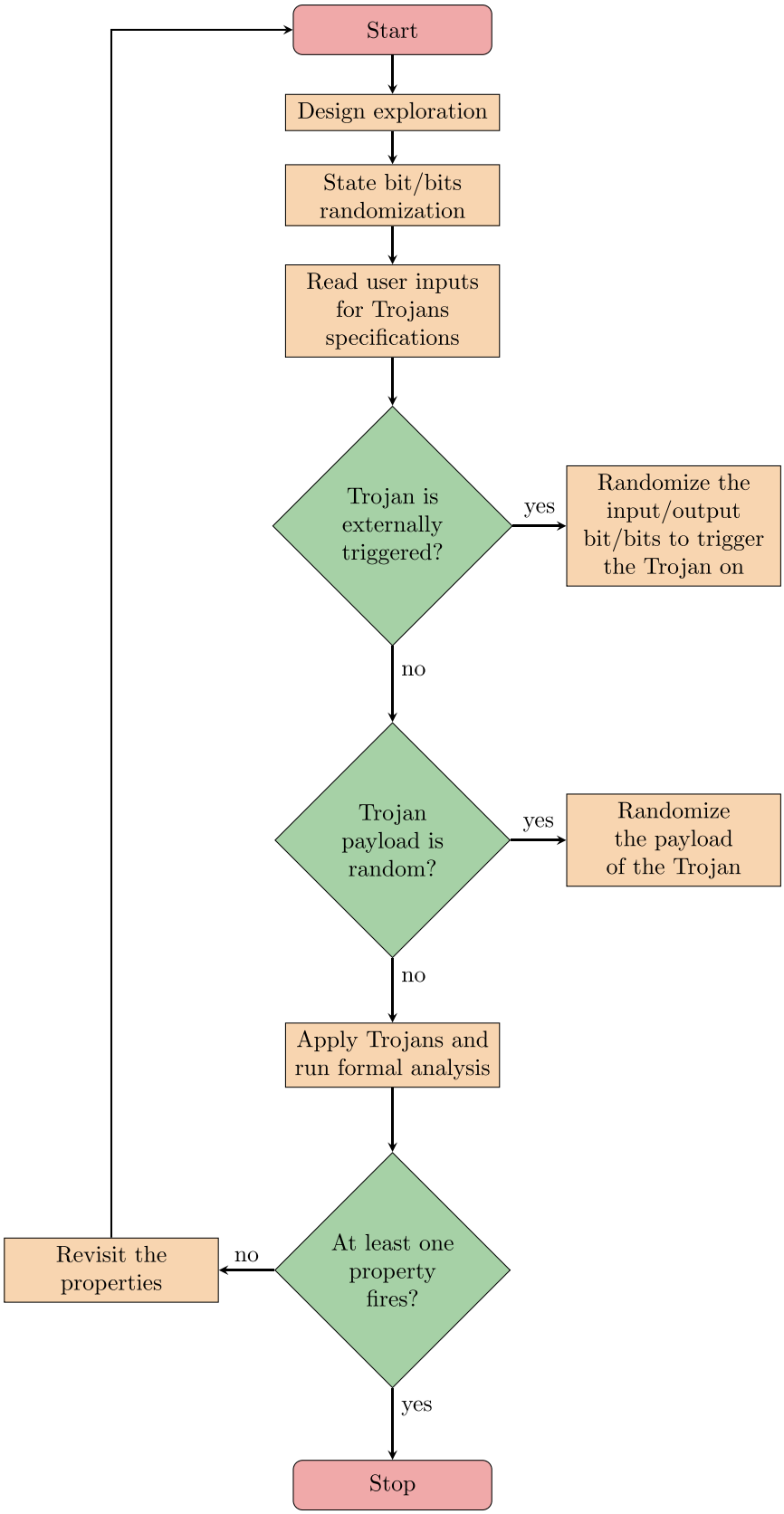

Fig. 4 Trojans modeling and insertion

Table 4 Trojan trigger mechanism

Trojan Trigger	Mechanism
ALWAYS_ON	The cut signal is unconditionally applied to the internal state
EXTERNAL_INPUT	The cut signal is conditionally applied upon a certain value of an input
EXTERNAL_OUTPUT	the cut signal is conditionally applied upon a certain value of an output
INTERNAL_TIME_BASED	The cut signal is applied only after waiting for a certain number of operating cycles

the control points (primary inputs, cut signals, blackbox outputs) in both the RTL and the assertions with the same values. The output of the design is then compared to the expected values based on the assertions. If they match, this means the properties are proven and hold for the system. If not, it means a counter-example is generated to illustrate how the property is violated. The Trojan modeling and insertion blocks reads some design information at the beginning to decide on the Trojan trigger and payload. It also applied the Trojan to the design in a non-invasive way within the formal verification environment.

According to the above definitions, multiple Trojans types are modelled according to different trigger types and payload types. A Trojan affects a bit or a group of bits in a state element in the design. Some internal tool APIs are used to extract the information about the design's state elements and the width of each of them. Then a random bit or bits are chosen to apply the Trojan on. Accordingly, the supported Trojan types (in terms of triggers) are: ALWAYS_ON, EXTERNAL_INPUT, EXTERNAL_OUTPUT, and INTERNAL_TIME_BASED.

Table 4 and 5 illustrate how the different Trojans triggers and payloads are modeled using the formal verification concepts specified above.

This Trojan modeling and insertion framework can be applied on RTL and third-party IPs. It can also be applied on encrypted IPs as long as the interface of this IP is not encrypted, which should be the common case. This framework is tied to having a set of properties that fully describe the functionality of the system. Assertion-coverage holes might lead to Trojans go undetected as there are no assertions that cover the Trojan location.

So, for the proposed methodology to work effectively, it needs the following:

- Properties that cover all logic inside the design, achieving formal coverage.
- Properties to be proven before the Trojans insertion.

The same methodology can be applied on a newly-developed arbiter as long as it's fully proven to meet the requirement before the Trojan detection.

4 Results

Results section is divided into two parts. The first part shows the results related to Trojan-free formal verification in terms of the number of properties, CPU time, and peak memory taken for each run. The other section shows the results related to Trojan formal verification in terms of the number of fired properties. In our experiments, we used Questa PropCheck formal verification tool [12]. The runs were made on an Intel(R) Xeon(R) Gold 6242R CPU @ 3.10 GHz machine to conclude all results.

4.1 Trojan-Free Results of Arbiters' Properties

This section illustrates the formal verification run, considering different types and sizes of arbiters. The results are reported in terms of the run's elapsed time in seconds, the peak memory reported by the tool in megabytes, the number of assertions generated for the arbiter verification, the number of assumptions used for constraining the formal run, and the results of the properties. The number of assertion for each arbiter configuration can be calculated as in Algorithm 1 according to the arbiter size and type. Properties results are shown in Table 6. For all types of arbiters, it is observed that all properties are proven, except for fixed priority arbiters. Fixed priority arbiters fire the deadlock-free properties for all the requests other than the first request (highest priority request). The counterexample for any of these firings is a waveform, where the first request is always asserted, and the grant is never given to any other request, as illustrated in Fig. 5. Questa PropCheck

Table 5 Trojan payload mechanism

Trojan Payload	Mechanism
UNKNOWN	The value driving the cut signal is left for the formal analysis to decide
RANDOM	The value driving the cut signal is constrained based on a random value set by the proposed framework
CERTAIN_VALUE	The value driving the cut signal is constrained based on an input from the verification engineer

Table 6 Trojan-free results of arbiters' properties

Arbiter type	Arbiter size	Elapsed time (s)	Peak memory (MB)	Number of assertions	Number of assumptions	Results
Fixed priority	4	2	420	13	4	3 Fired
Fixed priority	8	2	420	25	8	7 Fired
Fixed priority	16	2	420	25	8	7 Fired
Fixed priority	32	3	430	49	16	15 Fired
Round Robin	4	2	420	17	4	All proven
Round Robin	8	11	420	33	8	All proven
Round Robin	16	709	420	65	16	All proven
Round Robin	32	10141	1100	129	32	All proven
Matrix	4	2	420	17	4	All proven
Matrix	8	18	420	33	8	All proven
Matrix	16	Timeout	660	65	16	All proven, 17 bounded-proof
Matrix	32	Timeout	1130	129	32	All proven, 24 bounded-proof
Random priority	4	17	450	14	4	All proven
Random priority	8	22	490	26	8	All proven
Random priority	16	47	570	50	16	All proven
Random priority	32	232	750	98	32	All proven

formal verification tool has the ability to detect if these reported deadlocks can eventually be escaped under some input sequences. The deadlock checking depends on negating the counterexample/s of the failed deadlock-free (liveness) property to check if we can escape the deadlock loop with other values of the contributing control points. With a certain tool configuration, the deadlock properties of the fixed priority arbiter are checked, and they are found to have escapable waveforms. The escapable waveform of any deadlocked request shows all the preceding requests as low, as shown in Fig. 6. Another observation on the Trojan-free results is related to the high sizes in the case of the matrix arbiter, specifically size sixteen and size thirty-two. The formal verification run has timed out after eight hours of analysis without producing any counterexamples. Accordingly, some of the proven properties are considered as bounded-proofs. Table 7 shows a comparison with the discussed related work, elaborating how the proposed framework applied the formal verification on a wider set of arbiter types and sizes. It also shows how the framework stress-tests the arbiters using a various and generic set of properties.

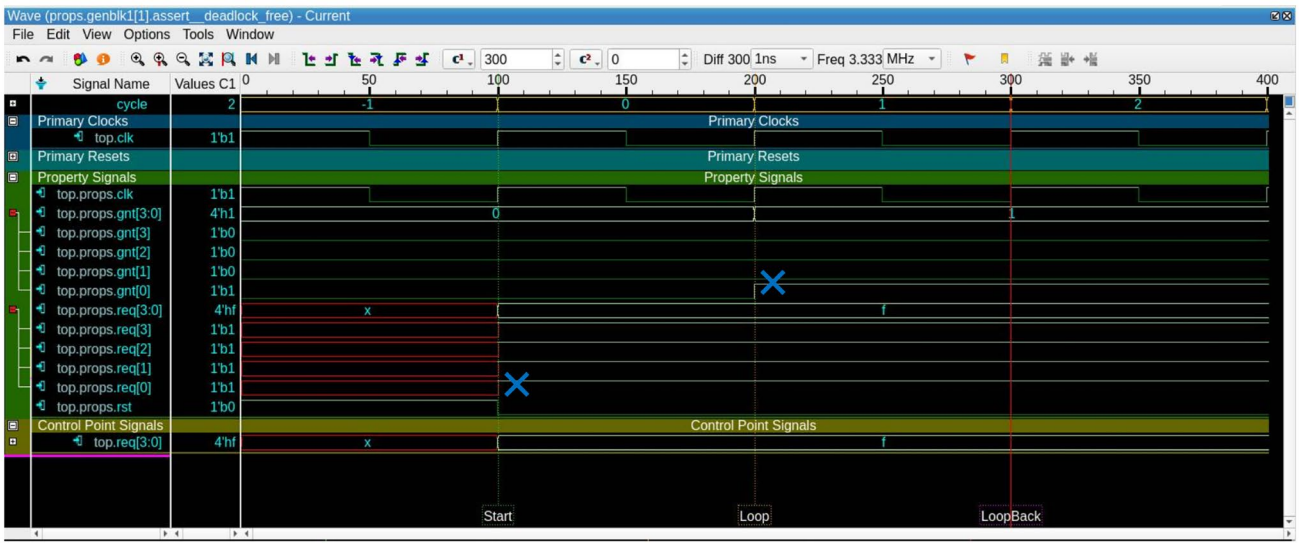

Fig. 5 Deadlock counterexample for a fixed priority arbiter

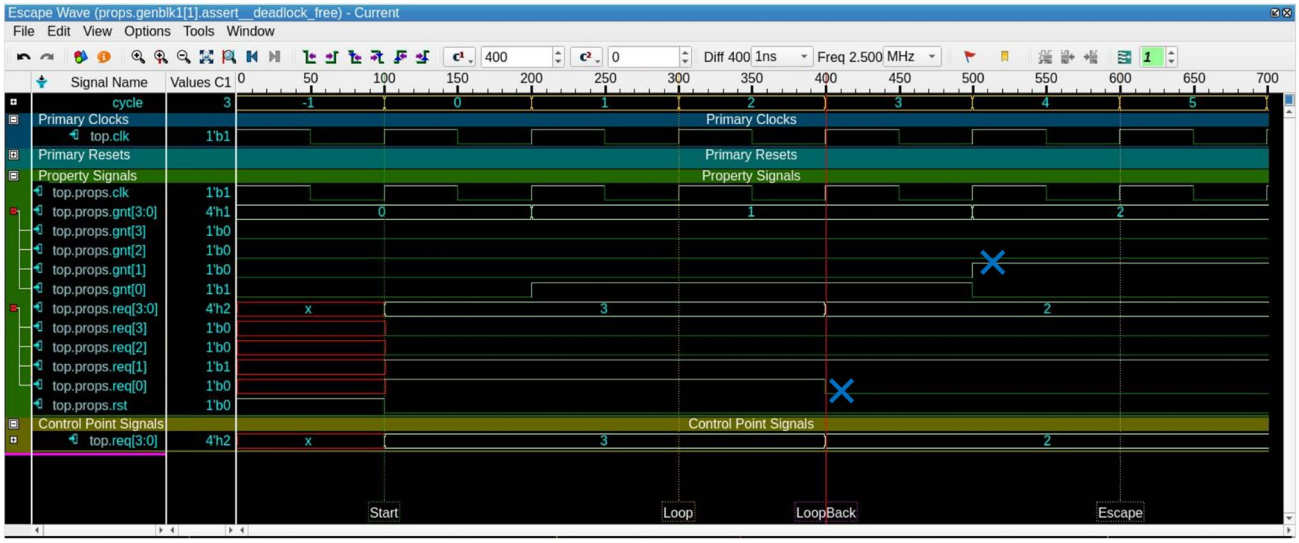

Fig. 6 Deadlock escape waveform

Algorithm 1 Calculation of the number of assertions and assumptions for an arbiter configuration

$assertions \leftarrow arbiter\_size * (deadlock\_freedom + no\_gnt\_without\_req) + multual\_exclusion$
$assumptions \leftarrow arbiter\_size * req\_high\_till\_gnt$
if $arbiter\_type$ *is* "$FIXED\_PRIORITY$" **then**
  $assertions \leftarrow assertions + (arbiter\_size - 1) * fixed\_priority\_protocol + fixed\_priority\_protocol\_0$
else if $arbiter\_type$ *is* "$ROUND\_ROBIN$" **then**
  $assertions \leftarrow assertions + arbiter\_size * max\_latency + arbiter\_size * rr\_protocol$
else if $arbiter\_type$ *is* "$MATRIX$" **then**
  $assertions \leftarrow assertions + arbiter\_size * max\_latency + arbiter\_size * matrix\_protocol$
else if $arbiter\_type$ *is* "$RANDOM\_PRIORITY$" **then**
  $assertions \leftarrow assertions + arbiter\_size * max\_latency, andom + random\_fsr$
end if

4.2 Trojan-Inserted Results of Arbiters' Properties

According to the Trojans taxonomy illustrated in Fig. 2, the Trojans used in these experiments have the following specifications:

- Insertion phase: The modeled Trojans can be inserted during the design phase.
- Abstraction level: The modeled Trojans can be applied on register-transfer or gate levels.
- Activation mechanism: The inserted Trojans can be always on, internally-triggered (time-based), and externally-triggered (user input and component output).
- Effect: The modeled Trojans can cause any of the effects according to the DUT original functionality.
- Location: The modeled Trojans can be inserted at any location in the DUT.

Table 7 Comparison with the related work for the functional verification of arbiters

	Model	Arbiter type	Evaluated requirements	Verification environment
[39]	BIP	Round Robin, TDMA	Deadlock freedom, protocol verification, and mutual exclusion	RTD-finder model checker on a BIP model
[19]	RTL	Two-stage (priority and Round Robin)	Exclusive grants, latency between request, and grant fairness	VC-formal tool
[4, 22]	RTL	Random priority	Request-to-grant delay	IBM's RuleBase PE and SixthSense formal tool
[14]	Petri net	Two-stage priority arbiter	Deadlock freedom, stall-free is not proven	WORKCRAFT software
[23]	RTL	Matrix	Grant fairness, light load response	Simulation environment
Proposed work	RTL	Fixed priority, random priority, matrix, Round-Robin	Deadlock freedom, mutual exclusion, no grant without request, protocol-compliant requirement, maximum latency	Questa PropCheck Formal tool

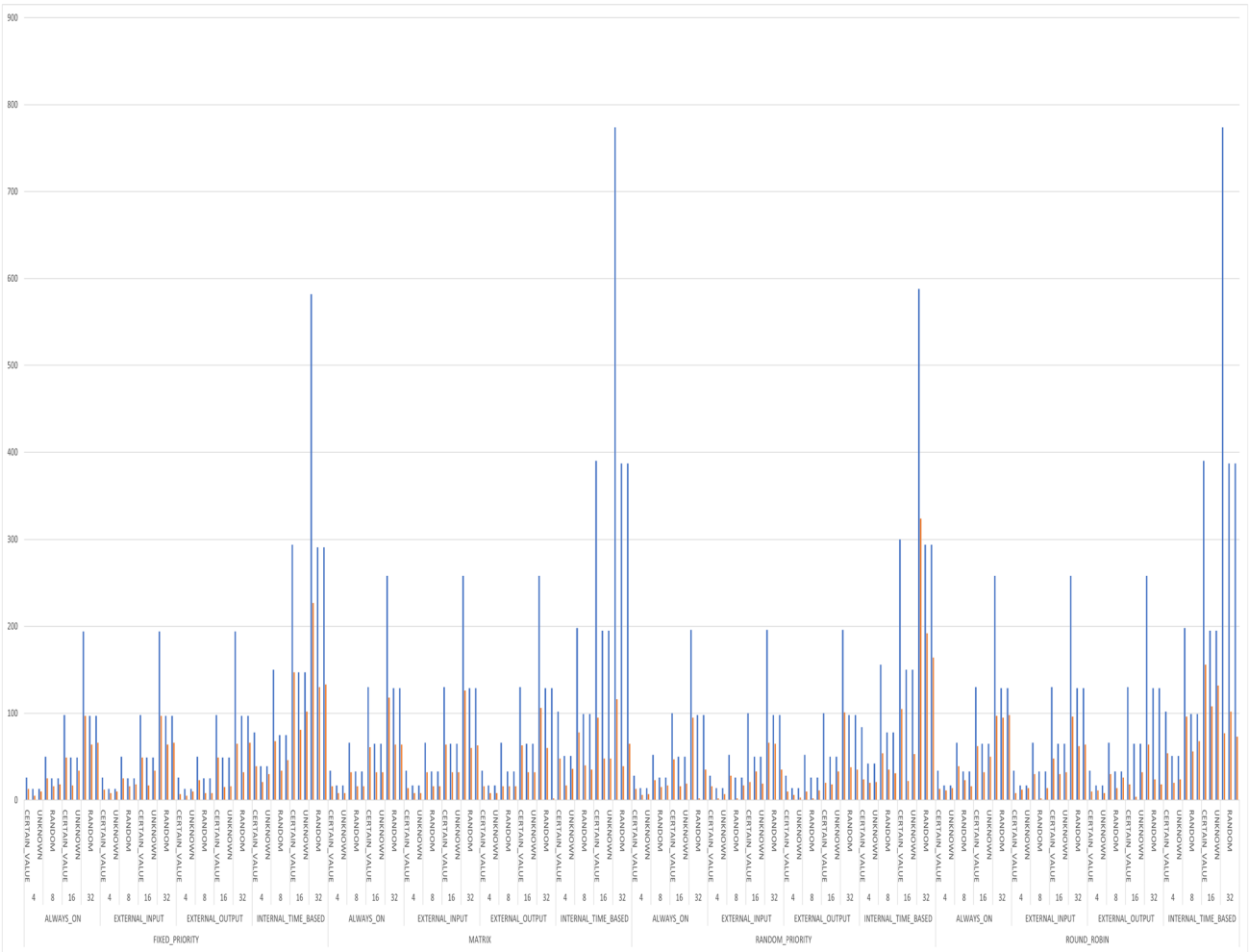

Fig. 7 Distribution of the number of assertions versus firings with respect to arbiter type, arbiter size, Trojan trigger, and Trigger payload

- Physical characteristic: The DUT with the inserted Trojans will have the same layout as the Trojans are modelled in the verification environment.

In the experiments of this section, we need to make sure that the formal environment is able to detect any inserted Trojan. Detection of Trojans is through firing at least one property to alert the verification engineer that there is something wrong with the DUT. Accordingly, the formal run in these experiments is set with five minutes timeout and checks if -at least- one property fires before this timeout. The use of formal verification abstracts detecting the effect of the inserted Trojans. It only works on proving/disproving a set of properties that describe the expected correct functionality. The verification environment has a one-hundred percent assertion coverage. So, any type of inserted Trojans will be detected as a property firing along with a counterexample that describes how the property does not hold against the DUT. A special handling is done for the fixed-priority arbiter. For the experiments of the fixed priority arbiters, the deadlock checks are enabled to make sure the fired deadlock-freedom properties are escapable (proven under some input conditions, where the preceding requests are not asserted). Through this setup, the inserted Trojans are not masked by firings.

Through varying different aspects for each run, we have results for three-hundred-eighty-four different combinations. These combinations represent different arbiter types and sizes, along with different Trojans triggers and payloads. The runs cover four types of arbiters: fixed-priority, random-priority, Round Robin, and matrix. They also cover four sizes of each arbiter, which are: four, eight, sixteen, and thirty-two. Additionally, they cover four trigger types: ALWAYS_ON, EXTERNAL_INPUT, EXTERNAL_OUTPUT, INTERNAL_TIME_BASED (covering multiple cycle thresholds). Moreover, the runs cover three Trojan payloads: UNKNOWN, RANDOM, CERTAIN_VALUE (different values are examined)). The different types of Trojan triggers and payloads are mutually exclusive. So, a Trojan trigger

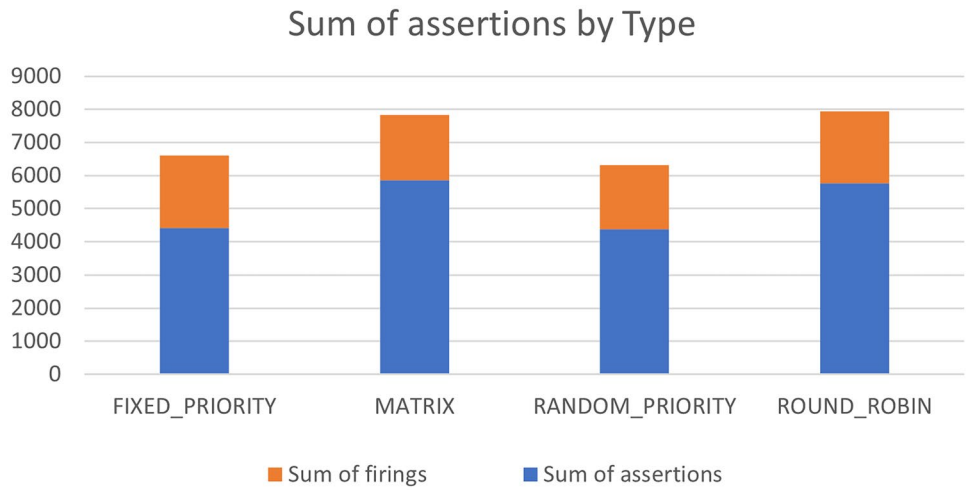

Fig. 8 The total number of assertions versus the total number of firings for each arbiter type

either depends on an external trigger (input/output value condition) or an internal trigger based on the number of cycles. Figure 7 shows the number of assertions versus the number of firings for the different combinations of arbiter sizes and types, along with different configurations of Trojan models. In this figure, a blue line shows the number of assertions associated with a configuration, where an orange line shows the number of firings in the same configuration. It is observed that we have at least one fired property for each available combination of the experiments variants, which alerts the verification engineer of a possible functional or security problem with the DUT. Figure 8 shows the commutative number of assertions along with the commutative number of firings summed for all configurations related to the same arbiter type. Also, the blue portion of each bar resembles the assertions, where the orange portion resembles the firings. Figure 9 shows a counterexample for the mutual exclusion property for a Round Robin arbiter, where the grant signal has multiple high bits. The injected Trojan is EXTERNAL_INPUT trigger and UNKNOWN value payload. The counterexample is shown, when two bits of the four-bit grant are asserted. Figure 10 shows a counterexample for the maximum latency property for a matrix arbiter. It is observed that for the second request for "req[7]", the grant signal "gnt[7]" is not asserted within the maximum latency range. The maximum latency for a matrix arbiter in the case of size-eight should be fifteen cycles maximum. The counterexample shows more cycles passed without a grant, which violates the requirement. The injected Trojan is ALWAYS_ON trigger and RANDOM value payload. Figure 11 shows a counterexample for the deadlock freedom property for a random priority arbiter. The injected Trojan

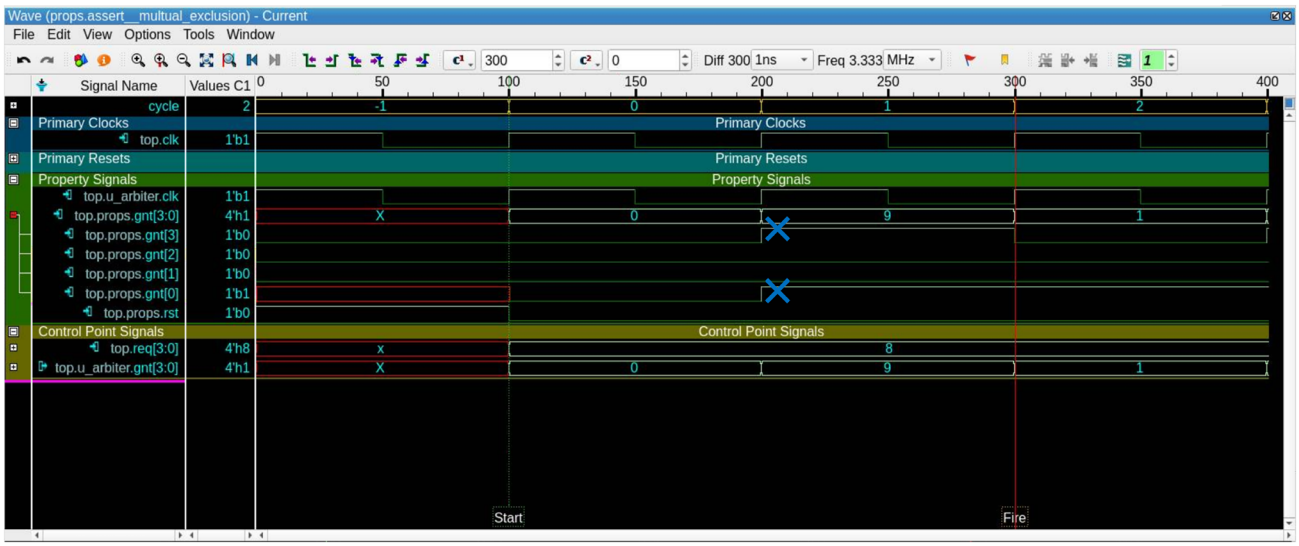

Fig. 9 A failing mutual exclusion property in a Round Robin arbiter due to the insertion of externally input-triggered unknown-value Trojan

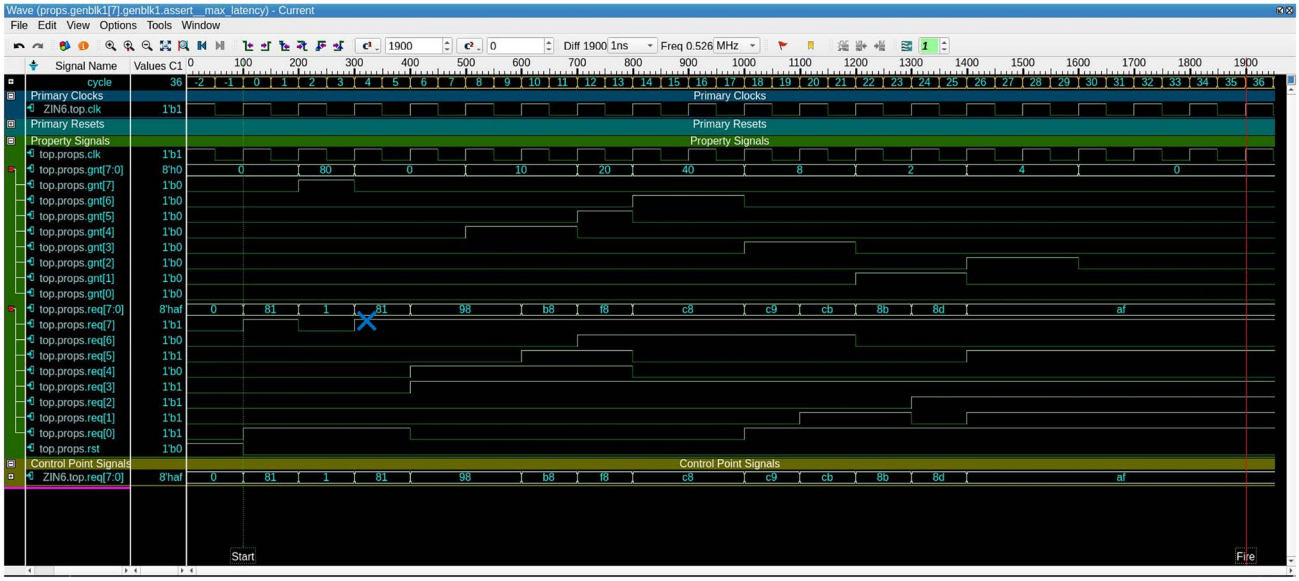

Fig. 10 A failing maximum latency property in a matrix arbiter due to the insertion of an always on random-value Trojan

is an INTERNAL_TIME_BASED trigger, which is triggered starting from the tenth clock cycle. The waveform shows that the grant signal "gnt[0]" is never asserted for the request "req[0]". It can be noted that in the case of liveness properties counterexample, the firing is shown in the form of a stuck at loop, where the needed state will never be reached. In the same formal run, some other properties hold for the inserted Trojan, such as mutual_exclusion with the sanity waveform specified in Fig. 12. In this case, the inserted Trojan does not affect the active logic contributing to this property and accordingly it holds. In other words, the randomly-selected location of the Trojan might have affected a state that is not in the cone of influence of this property.

In some experiments, a very small percentage of assertions fire. An example is the thirty-two fixed-priority arbiter with external-input trigger and certain-value payload. In this case, only one property fires out of ninety-seven assertions. On the contrary, some cases have a large percentage of firing

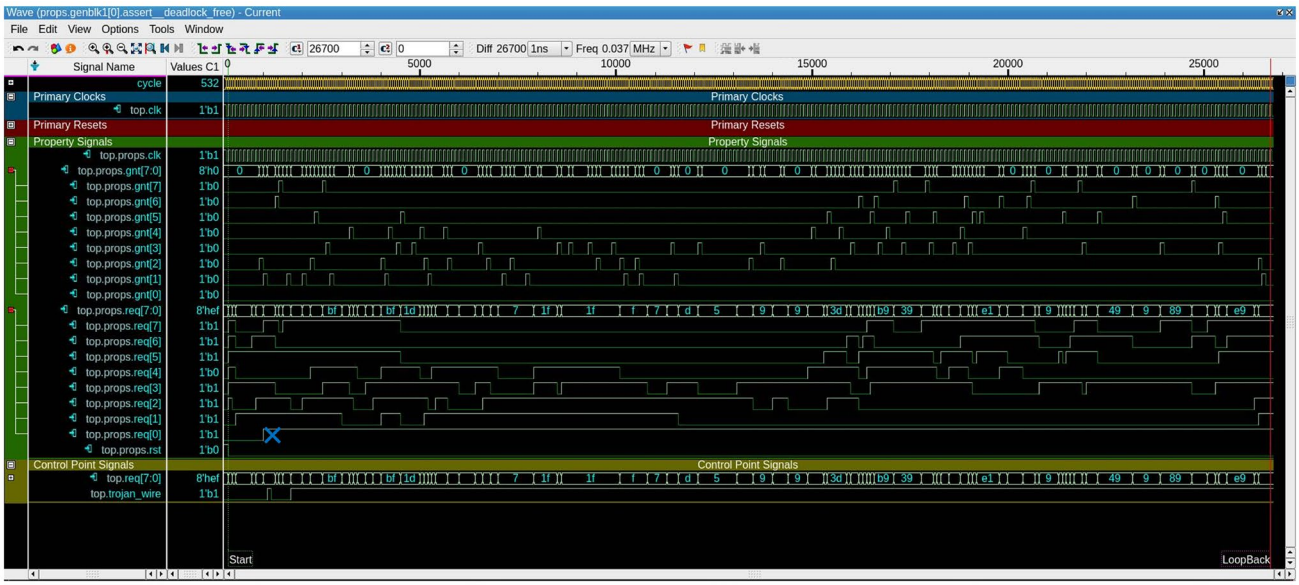

Fig. 11 A failing deadlock freedom property in a random priority arbiter due to the insertion of an internally time-based triggered certain-value Trojan

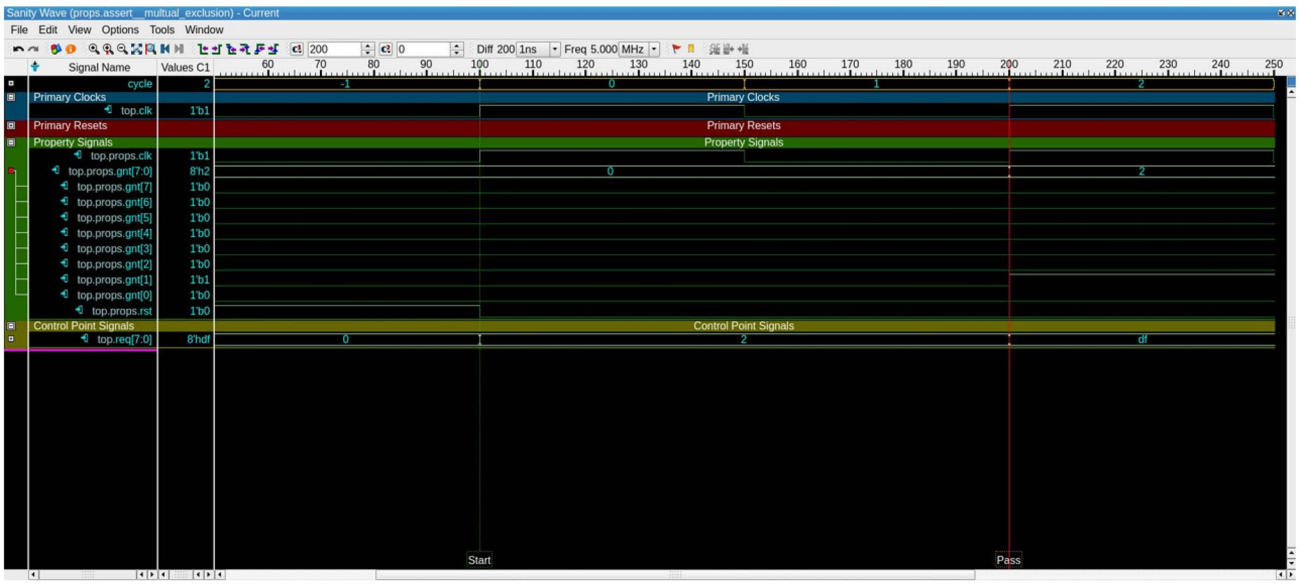

Fig. 12 A sanity waveform for mutual exclusion property that is not affected by the insertion of an internally-time based triggered certain value Trojan

Table 8 Comparison with the related work for hardware Trojans insertion

	Trojan insertion methodology	Invasive	Generic	Scalable	Detection method
[15]	Insertion on the TRNG of an FPGA-based cryptosystem. The payload design consists of a Configuration Look-Up-Table (CFGLUT) and a state machine. The trigger is based on the output of the temperature sensor on board. Accordingly, there's an increase in the FPGA hardware resources utilization.	Yes	No	No	Applying Wavelet Transform on the compromised bitstream.
[26]	Insertion of certain types of hardware Trojans in part of the circuit by using EHW technology. The large scale circuit is decomposed and then redesigned to have a Trojan functionality that manipulated the original truth table of the circuit.	Yes	Yes	Depends on truth table size	Work focuses on Trojan insertion.
[30]	Insertion of a hardware Trojan into reconfigurable network devices through changing the RTL of the system to redirect a frame to an incorrect port, potentially enabling eavesdropping and denial-of-service.	Yes	No	No	Work focuses on Trojan insertion.
[31]	Insertion of hardware Trojans in a floating-point multiplier. The Trojans were implemented by adding new components (multiplexers and registers) to the RTL.	Yes	No	No	Work focuses on Trojan insertion.
[8]	Insertion of hardware Trojan models into gate level netlist. It used an AI-guided framework for automatic Trojan insertion was proposed. The framework aimed to generate a large population of valid Trojans for a given design by mimicking the properties of a small set of known Trojans.	Yes	Yes	Depends the AI-model	Work focuses on Trojan insertion.
Proposed Trojan insertion	Insertion of different types of Hardware Trojan, which leverages the formal verification concepts without the need of changing the DUT.	No	Yes	Yes	Formal verification exit criteria.

properties. An example in the case of sixteen-bit matrix arbiter with internal-time-based trigger and certain-value payload. In this case, sixty-four properties fire out of sixty-five. These numbers depend on the effectiveness of the Trojan location. In some case, this location exists in the cone of influence of a few properties, and the manipulation of this location fires these properties. On the contrary, the manipulation of an effective Trojan location may lead to large number of assertion firings. Based on the results, the matrix arbiter is the most hardware-Trojan-prune arbiter with the highest percentages of firings with respect to the total number of assertions, specially for large sizes. This can be justified due to the high complexity of logic in the case of matrix arbiters, as they keep track of a matrix of state values for each request and priority.

Table 8 shows a comparison with the discussed related work, elaborating how the proposed framework uses non-invasive method of hardware Trojan modeling and insertion, while the discussed previous work tend to apply modifications to the DUT. This table also highlights the scalability and genericity, and detection method of each related work. The Trojan modeling, insertion, and verification achieved by this environment can be used as exit criteria for the functional verification of any system.

5 Conclusion and Future Work

The work described in this paper outlines how to formally confirm the correctness of various types of arbiter circuits. This verification process includes ensuring general requirements that apply to all kinds of arbiters, such as freedom of deadlocks, guaranteeing exclusive access to grants, and maintaining appropriate timing between requests and grants. Additionally, the process covers verifying specific protocol-related properties for different arbiter types, such as Round Robin, matrix, fixed priority, and random priority arbiters.

Beyond the formal validation of diverse functional criteria, we introduce a systematic method for non-invasive hardware Trojans insertion without modifications to the DUT. These Trojans come in different forms with distinct triggers and payloads. Our proposed formal verification environment demonstrates its capacity to identify these Trojans by assessing various sets of properties. Based on our experimental results, we find that the insertion of a Trojan consistently fires at least one of the system properties. In some cases, all properties fire with counterexamples.

This environment considerably simplifies the process of validating security aspects for verification engineers. Moreover, we conclude that this approach can be broadly applied to other systems with modifications to the set of properties that define proper system behavior. Moving forward, our future efforts will involve extending our current framework to evaluate hardware encryption systems, such as AES. This extension will encompass not only functional aspects, but also security aspects, including confirming secure paths to and from private keys. Furthermore, we will apply our proposed method of Trojan modeling and insertion to these systems as well. For the applicability of this future work, we need to have the RTL of such encryption systems available along with its proven fully-covering set of properties.

Funding Open access funding provided by The Science, Technology & Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

Availability of data and materials Data-sets generated during and/or analyzed during the current study are available from the author upon reasonable request.

Declarations

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

References

1. Al-Anwar A, Alkabani Y, El-Kharashi MW, Bedour H (2013) Hardware trojan protection for third party IPs. In: 2013 Euromicro Conference on Digital System Design. IEEE, pp 662–665
2. Alfke P, Xilinx (1998) Efficient shift registers, LFSR counters, and long pseudo-random sequence generators. https://docs.xilinx.com/v/u/en-US/xapp052. Accessed 1 Feb 2024
3. Anwar MW, Rashid M, Azam F, Naeem A, Kashif M, Butt WH (2020) A unified model-based framework for the simplified execution of static and dynamic assertion-based verification. IEEE Access 8:104407–104431
4. Auerbach G, Copty F, Paruthi V (2010) Formal verification of arbiters using property strengthening and underapproximations. In: Formal Methods in Computer Aided Design, pp 21–24. IEEE
5. Bagga S, Gupta R, Jose J (2023) Modelling and analysis of confluence attack by hardware trojan in noc. In: Emerging Electronic Devices, Circuits and Systems: Select Proceedings of EEDCS Workshop Held in Conjunction with ISDCS 2022, pp 231–246. Springer
6. Becker DU (2012) Efficient Microarchitecture for Network-on-chip Routers. Stanford University
7. Cheng J, Fleming ST, Chen YT, Anderson J, Wickerson J, Constantinides GA (2021) Efficient memory arbitration in high-level synthesis from multi-threaded code. IEEE Trans Comput 71(4):933–946
8. Cruz J, Gaikwad P, Nair A, Chakraborty P, Bhunia S (2022) Automatic hardware trojan insertion using machine learning.

arXiv preprint arXiv:2204.08580. https://arxiv.org/abs/2204.08580. Accessed 1 Feb 2024
9. Dally WJ, Towles BP (2004) Principles and Practices of Interconnection Networks. Elsevier
10. Danesh W, Banago J, Rahman M (2021) Turning the table: using bitstream reverse engineering to detect fpga trojans. J Hardw Syst Secur 5:237–246
11. Das S, Karfa C (2022) Formal modeling and verification of starvation freedom in nocs. In: Artificial Intelligence Driven Circuits and Systems: Select Proceedings of ISED 2021, pp. 101–114. Springer
12. EDA S (2023) Questa®PropCheck. https://verificationacademy.com/products/questa-propcheck. Accessed 1 Feb 2024
13. Elmiligi H, Gebali F, El-Kharashi MW (2016) Multidimensional analysis of embedded systems security. Microprocess Microsyst 41:29–36
14. Golubcovs S, Mokhov A, Bystrov A, Sokolov D, Yakovlev A (2019) Generalised asynchronous arbiter. In: 2019 19th International Conference on Application of Concurrency to System Design (ACSD), pp 3–12. IEEE
15. Guo X, Dutta RG, Jin Y, Farahmandi F, Mishra P (2015) Pre-silicon security verification and validation: A formal perspective. In: Proceedings of the 52nd Annual Design Automation Conference, pp 1–6
16. Govindan V, Chakraborty RS, Santikellur P, Chaudhary AK (2018) A hardware trojan attack on fpga-based cryptographic key generation: impact and detection. J Hardw Syst Secur 2:225–239
17. Hamlet JR, Mayo JR, Kammler VG (2019) Targeted modification of hardware trojans. J Hardw Syst Secur 3:189–197
18. He J, Guo X, Meade T, Dutta RG, Zhao Y, Jin Y (2019) Soc interconnection protection through formal verification. Integration 64:143–151
19. Ikram S, Barner C, Sweeney J, Ellis J, Dufresne B (2015) Formal verification of a multistage arbiter. SNUG Boston, Synopsys
20. Islam SA, Sah LK, Katkoori S (2020) A framework for hardware trojan vulnerability estimation and localization in rtl designs. J Hardw Syst Secur 4:246–262
21. JayashankaraShridevi R, Ancajas DM, Chakraborty K, Roy S (2017) Security measures against a rogue network-on-chip. J Hardw Syst Secur 1:173–187
22. Kailas K, Paruthi V, Monwai B (2009) Formal verification of correctness and performance of random priority-based arbiters. In: 2009 Formal Methods in Computer-Aided Design, pp 101–107. IEEE
23. Khan AA, Mir RN, Din N-U (2022) Adaptive hybrid arbiter design for real-time traffic-aware scheduling. Circuit World 48(2):185–203
24. Kukimoto Y. Introduction to Formal Verification. https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html (Last Accessed 9 June 2023)
25. Kumar VB, Deb S, Gupta N, Bhasin S, Haj-Yahya J, Chattopadhyay A, Mendelson A (2020) Towards designing a secure risc-v system-on-chip: Itus. J Hardw Syst Secur 4:329–342
26. Liu L, Wang T, Wang X (2021) Method of implanting hardware trojan based on ehw in part of circuit. J Electron Test 37:279–284
27. Liu Y, He J, Ma H, Zhao Y (2019) Hardware trojan detection leveraging a novel golden layout model towards practical applications. J Electron Test 35:529–541
28. Moein S, Khan S, Gulliver TA, Gebali F, El-Kharashi MW (2015) An attribute based classification of hardware trojans. In: 2015 Tenth International Conference on Computer Engineering & Systems (ICCES). IEEE, pp 351–356
29. Moein S, Subramnian J, Gulliver TA, Gebali F, El-Kharashi MW (2015) Classification of hardware trojan detection techniques. In: 2015 Tenth International Conference on Computer Engineering & Systems (ICCES). IEEE, pp 357–362
30. Mukherjee R, Govindan V, Koteshwara S, Das A, Parhi KK, Chakraborty RS (2020) Probabilistic hardware trojan attacks on multiple layers of reconfigurable network infrastructure. J Hardw Syst Secur 4:343–360
31. Nikhila S, Yamuna B, Balasubramanian K, Mishra D (2019) Fpga based implementation of a floating point multiplier and its hardware trojan models. In: 2019 IEEE 16th India Council International Conference (INDICON), pp. 1–4. IEEE
32. Rajendran J, Dhandayuthapany AM, Vedula V, Karri R (2016) Formal security verification of third party intellectual property cores for information leakage. In: 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), pp 547–552. IEEE
33. Rathmair M, Schupfer F, Krieg C (2014) Applied formal methods for hardware trojan detection. In: 2014 IEEE International Symposium on Circuits and Systems (ISCAS), pp 169–172. IEEE
34. Sepúlveda 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: 2018 IEEE 23rd European Test Symposium (ETS), pp 1–6. IEEE
35. Shakya B, He T, Salmani H, Forte D, Bhunia S, Tehranipoor M (2017) Benchmarking of hardware trojans and maliciously affected circuits. J Hardw Syst Secur 1:85–102
36. Slimane MB, Hafaiedh IB, Robbana R (2017) Formal-based design and verification of soc arbitration protocols: A comparative analysis of tdma and round-robin. IEEE Des Test 34(5):54–62
37. Tang W, Su J, Gao Y (2023) Hardware trojan detection method based on dual discriminator assisted conditional generation adversarial network. In: J Electron Test, pp 1–12
38. Tebyanian M, Mokhtarpour A, Shafieinejad A (2021) Sc-cotd: Hardware trojan detection based on sequential/combinational testability features using ensemble classifier. J Electron Test 37(4):473–487
39. Veeranna N, Schafer BC (2016) Hardware trojan detection in behavioral intellectual properties (ip's) using property checking techniques. IEEE Trans Emerg Topics Comput 5(4):576–585
40. Veeranna N, Schafer BC (2017) Trust filter: Runtime hardware trojan detection in behavioral mpsocs. J Hardw Syst Secur 1:56–67
41. Weber M (2001) Arbiters: design ideas and coding styles. SNUG Boston, Silicon Logic Engineering, Inc.
42. Yilmaz Y, Aniello L, Halak B (2021) Assure: A hardware-based security protocol for resource-constrained iot systems. J Hardw Syst Secur 5(1):1–18

Hala Ibrahim received the B.Sc. degree (First Class Hons.) in electronics and communications engineering from Alexandria University, Alexandria, Egypt, in 2017. Hala is currently pursuing the M.Sc. degree from Ain Shams University, Cairo, Egypt, focusing on hardware security. She has publications in Network-on-Chips and UVM testbench environment for interconnection routers. Her current research interests include digital verification, formal verification, hardware security, and hardware Trojan modeling.

Haytham Azmi is a Researcher at the Microelectronics Department, Electronics Research Institute, Cairo, Egypt. His research interests include the design and implementation of digital integrated circuits and the design and development of electronic design automation tools to improve the continuous integration of electronic circuits at the micro- and nanoscale technologies.

M. Watheq El-Kharashi received the B.Sc. (First Class Hons.) and M.Sc. degrees in computer engineering from Ain Shams University, Cairo, Egypt, in 1992 and 1996, respectively, and the Ph.D. degree in computer engineering from the University of Victoria, Victoria, BC, Canada, in 2002. He is currently a Professor of computer organization and a chair of the Department of Computer and Systems Engineering, Ain Shams University; and also an Adjunct Professor with the Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada. He has published 170 articles in refereed international journals and conferences and authored two books and eight book chapters. His current research interests include advanced system architectures, especially networks-on-chip (NoC), systems-on-chip (SoC), and secure hardware. More specific interests include hardware architectures for networking (network processing units) and security; advanced microprocessor design, simulation, performance evaluation, and testability; and computer architecture and computer networks education.

Mona Safar received the Ph.D. degree (2011), M.Sc. degree (2007), and B.Sc. degree (First Class Hons.) (2004) in computer engineering from Ain Shams University, Cairo, Egypt. She is currently an Associate Professor in the Department of Computer and Systems Engineering, Ain ShamsUniversity. Her research interests include reconfigurable computing, application specific architectures, embedded systems, computer-aided design and verification, and electronic system level design.