CHAPTER 2
HARDWARE TROJANS

Hardware Trojans are malicious modifications made to circuitry in order to produce some undesirable effect, usually without the hardware designer's knowledge. In recent years, the hardware supply chain has become increasingly globalized. Few semiconductor businesses own their own foundries, and many of those 3rd-party foundries are located outside of the US. As hardware designers outsource more and more of their work, the threat of hardware Trojans has increased. Often, hardware designers do not design their products end-to-end. Many hardware designers make use of an outside foundry, or use externally designed peripheral components. In the case of FPGAs, hardware designers produce only the IP to be run on the FPGA, while all hardware is purchased from a vendor. More recently, however, hardware vendors are offering commercial-off-the-shelf IP blocks, allowing hardware designers to outsource even more of their work. Ultimately, any hardware or software component that is outsourced is a potential attack vector, and should be considered untrusted.

Hardware Trojans in FPGAs can exist in many different forms. They can be located in the FPGA silicon, on the FPGA PCB, or even in the IP used to program the FPGA. These Trojans can be characterized by a variety of their parameters - payload, trigger mechanism, physical size, power consumption, etc. Previous attempts have been made to comprehensively characterize Trojans based on many of these parameters. This chapter explores these characterizations, discusses Trojan mitigation strategies, and proposes a taxonomy of Trojans that can be found specifically in FPGA IP, something that has not yet been done.

2.1 Taxonomies

Few taxonomies have been proposed to categorize hardware Trojans. These taxonomies often categorize Trojans using their payload, their activation mechanism, and their physical

characteristics. This section explores the taxonomies that have been proposed in the past, and considers their applications to Trojans in FPGA IP.

An early, not comprehensive taxonomy of hardware Trojans discussed that they may have varying activation triggers and payloads [7]. A more complete taxonomy of hardware Trojans was first discussed by Wang et al. in [8], and then expanded upon by Tehranipoor et al. in [9]. Wang et al. proposed the taxonomy seen in Figure 2.1.

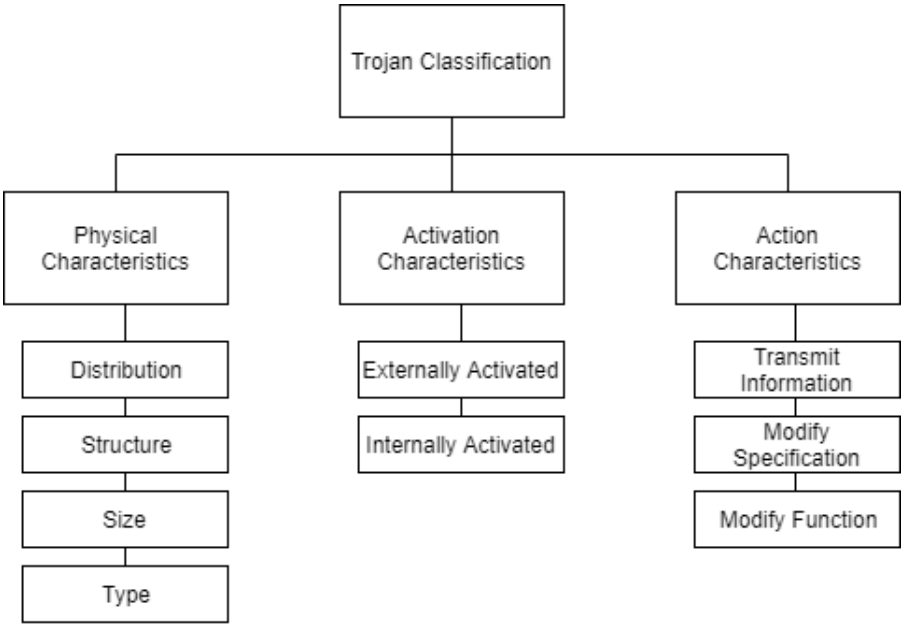

Figure 2.1: Taxonomy Proposed by Wang et al. [8]

This taxonomy categorizes Trojans by three different types characteristics - their physical characteristics, their activation characteristics, and their action characteristics.

Physical characteristics describe how the Trojan is placed in FPGA hardware. These characteristics might not we worthwhile in the scope of FPGA IP Trojans, but it is best to consider them here anyway. Physical characteristics include the distribuiton, structure, size, and type of the Trojan. Distribution describes how and where the Trojan is placed on the FPGA hardware - is there one Trojan in each LUT, are the Trojans on select LUTs, and so on. It describes the Trojan's physical location on the chip. Structure refers to the Trojan's internal logic and routing, rather than where the Trojan is located on the chip.

Size categorizes Trojans based on their physical size on the chip. Type makes a distinction between two main types of Trojans - functional and parametric. Functional Trojans produce errors or undesirable effects in logic by addition or deletion of gates, while parametric Trojans modify functionality of existing wires [9].

Activation characteristics explain how a Trojan is triggered. Trojans are either internally or externally activated. Externally activated Trojans are activated when a sensor or communication device reads an external signal or environmental condition telling the Trojan to activate. Internally activated Trojans are activated either combinationally (some rare condition) or sequentially (after a certain amount of time).

Action characteristics explain what a Trojan actually does when it is activated. This is perhaps the most worthwhile classification of Trojans, as it can be used to develop different Trojan tolerance strategies. This taxonomy breaks action characteristics into three categories: Trojans that transmit information, Trojans that modify specification, and Trojans that modify function. Trojans that transmit information aim to leak information from the hardware system to somewhere else. This information can be internal to the system, like an encryption key, or can be external information read using sensors in the hardware system. Trojans that modify specification change nonfunctional requirements of the hardware system, like clock speed. Finally, Trojans that modify function change the actual logic of the system. Modified logic can be as simple as an incorrect result for an operation, or as complex as taking over the FPGA system and using it for some other task.

This classification provides an interesting first attempt at categorizing hardware Trojans. Later taxonomies provide more elaboration on the action characteristics, and mostly ignore physical characteristics of the Trojans. Physical characteristics can mostly be ignored as they are only useful in detecting the Trojan using destructive or side-channel analysis, both of which are difficult to perform and somewhat ineffective at detecting Trojans [4, 10, 3].

The next classification to consider is that by Chakraborty et al. in [10]. This proposed

taxonomy of hardware Trojans omits classification by physical properties and focuses only on triggers and payloads. A reproduction of this taxonomy is seen in Figure 2.2.

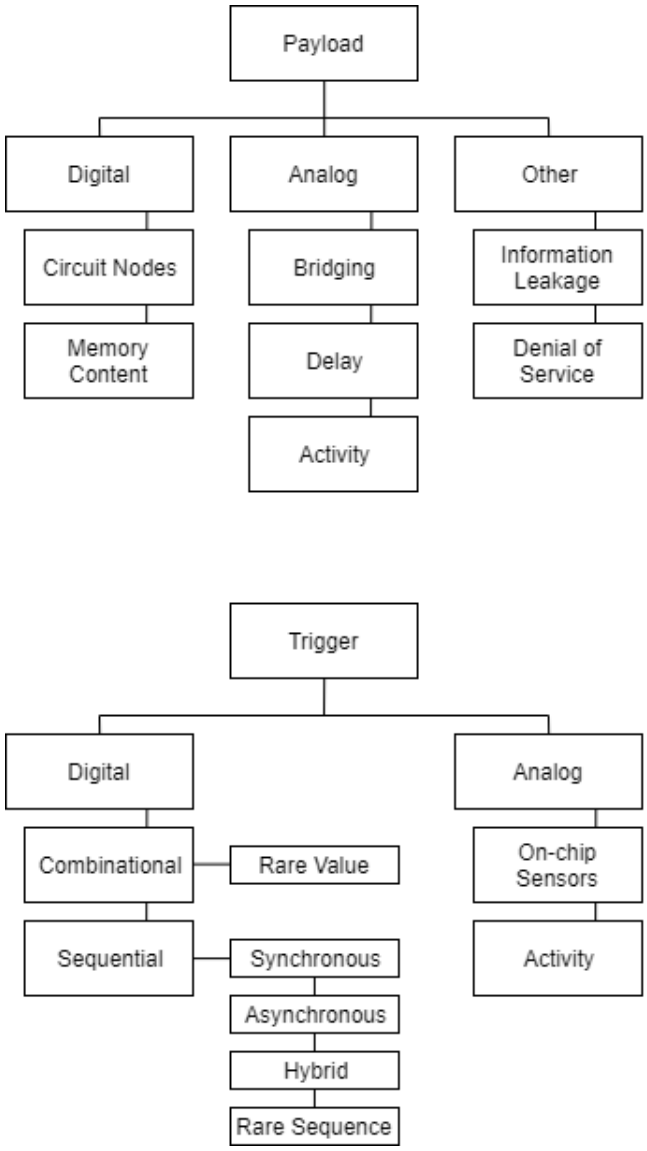

Figure 2.2: Taxonomy Proposed by Chakraborty et al. [10]

This taxonomy organizes Trojans by their trigger and their payload. Payloads can be either digital or analog payloads to the circuit, or payloads that don't have an effect on the logic (other). Digital payloads change the logic in the circuit at a node, or modify memory content in the hardware system. Analog payloads modify the circuit electrically, but often end up producing a logical effect. Bridging analog Trojans tie a node in the circuit to Vdd or

Gnd. Delay Trojans modify the delay between nodes in a circuit. Activity Trojans generate excess activity in the circuit in order to shorten its lifespan. Finally, Trojans may participate in more software-based attacks like leaking information from the circuit or participating in a denial of service attack to shut down important system functions [10].

Trojan triggers are organized into digital and analog triggers. Analog-triggered Trojans are activated using sensors, or by monitoring device electrical activity. Digitally-triggered Trojans are activated either combinationally or sequentially. Combinationally activated Trojans are activated when some rare value occurs at a node in a circuit. Sequentially activated Trojans are activated by a synchronous counter, asynchronous counter, hybrid counter, or a rare sequence at a node in the circuit.

This taxonomy is useful particularly because of the expanded payload taxonomy. Payloads are the most important aspect of a Trojan to understand. Understanding Triggers helps hardware designers produce designs to avoid triggering Trojans, or to intentionally trigger them during testing. However, the most effective form of Trojan mitigation is a Trojan-tolerant circuit design, and comprehensively understanding what payloads Trojans may bring helps hardware designers create designs that are able to deal with Trojans left undetected during testing.

The final and most comprehensive taxonomy is that proposed by Mal-Sarkar et al. in [1] and [4]. This taxonomy for the first time suggests that Trojans may also exist in FPGA IP, though does not elaborate further. Equally importantly, this taxonomy focuses exclusively on Trojans that may exist in FPGA devices, rather than in hardware devices in general. Figure 2.3 shows a reproduction of this taxonomy.

This taxonomy has approximately the same content as that proposed by Chakraborty et al. in [10], but organizes it somewhat better. Trojans in FPGA hardware can be either conditionally triggered or always on. Conditionally triggered Trojans can be triggered by an evironmental factor or some logic in the circuit. Trojan payloads can cause malfunction or leak secret information. Secret information can be either FPGA IP or data inside the

FPGA. Malfunctions can be logical, like a logical error, or parametric, like modifying the clock frequency of the circuit.

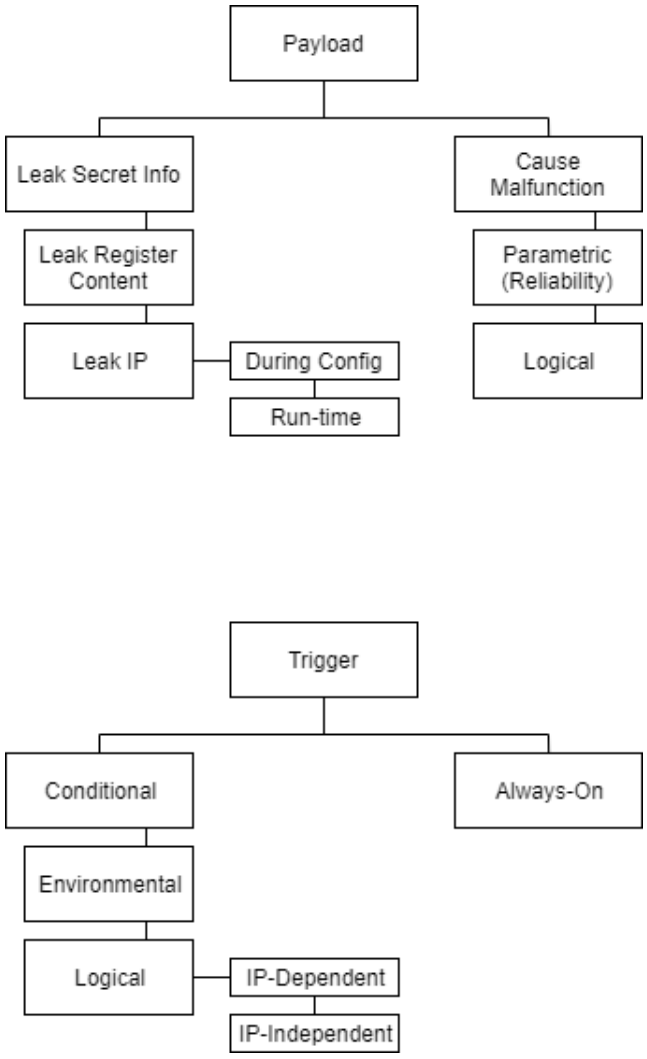

Figure 2.3: Taxonomy Proposed by Mal-Sarkar et al. [1]

This taxonomy is important because it is focused specifically on FPGAs. Earlier Trojan taxonomies that did not have an FPGA focus ignored the possibility of leaking IP. Additionally, this work for the first time mentioned Trojans in FPGA IP as a possibility, but did not include a detailed taxonomy of what types of Trojans might exist in IP.

These taxonomies give hardware designers an idea of what types of Trojans might exist in hardware designs. Unfortunately, there is no comprehensive taxonomy of hardware

Trojans in FPGA IP. Creating such a taxonomy might help hardware designers understand what kinds of threats they face when integrating 3rd-party IP into their FPGA systems.

2.2 Taxonomy of Trojans in FPGA IP

So that hardware designers may better understand the threats Trojans in FPGA IP pose to hardware designs, it would be useful to have a single comprehensive taxonomy of hardware Trojan effects in FPGA IP. Some attempts at categorizing Trojans that can be found in FPGA hardware have been made [9, 10, 1, 4], to this date there exists no comprehensive categorization of Trojans that can be found in FPGA IP. The categorization presented in Figure 2.4 attempts to categorize the payloads, or effects, of Trojans that may be found in FPGA IP. It is based on the earlier taxonomies of hardware Trojans and on FPGA Trojans and viruses examined in other work. This categorization focuses only on the payloads of FPGA IP Trojans because the categorization based on their triggers will likely be similar to Trojans in FPGA hardware, and has not yet been explored enough to be mature. Mal-Sarkar et al. suggest in [4] that categorizing Trojans based on their size and distribution may not lead to interesting results or help hardware designers, so those categorizations are also omitted. It is worthwhile to consider that a single Trojan may fit into multiple categories in the taxonomy. A Trojan that hijacks an FPGA may also send the FPGA's previous configuration data to an attacker. A Trojan that injects faults may do so in order to leak information, such as in fault-based cryptanalysis [11, 12, 13]. The work in this section has been submitted in [14].

2.2.1 Trojans that Cause Malfunction

The class of Trojans existing in FPGA IP that many existing mitigation strategies aim to tolerate is those that cause the FPGA to malfunction. The intensity of the malfunction can range from a simple logical error to complete device failure with an electrical fault. These Trojans can be further classified into *Trojans that Prevent FPGA Operation* and *Trojans*

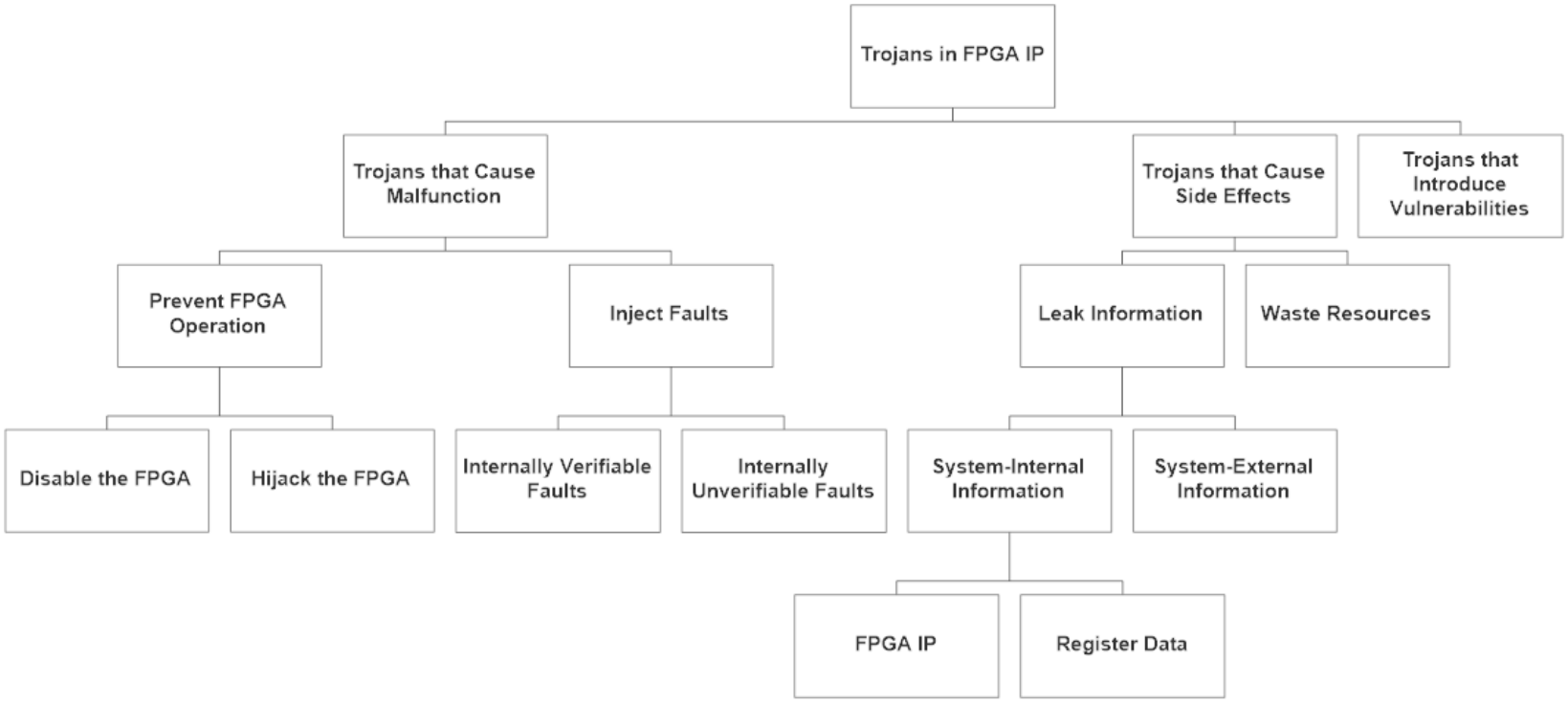

Figure 2.4: Taxonomy of Trojans in FPGA IP

that Inject Faults.

2.2.2 Trojans that Prevent FPGA Operation

Some previously explored hardware Trojans aim to disrupt the operation of an FPGA. This type of attack was first introduced by Hadzic as the SALT Trojan discussed in [15]. Whereas Trojans that electrically damage an FPGA are not probably not possible to produce in an HDL, it is somewhat easy for Trojan designers to produce Trojans that disable or even reprogram the FPGA. When a hardware designer integrates a 3rd-party IP block into a design on an FPGA, the 3rd-party IP block may have access to a multitude of advanced FPGA reprogramming features. The 3rd-party IP block may be able to write information to the FPGA's configuration SRAM. Trojans that Prevent FPGA Operation can be further divided into two subgroups, *Trojans that Disable the FPGA* and *Trojans that Hijack the FPGA*.

Trojans that Disable the FPGA have been somewhat explored in past work. One of the first papers on FPGA Secuity, the aforementioned "FPGA Viruses" [15] by Hadzic et al., discusses three FPGA-disabling classes of viruses. The first, a permanently damaging virus class titled MELT, is probably not possible through FPGA IP due to protections built into FPGA synthesis software. The other attack classes, SALT and HALT, cover attacks that cause system malfunction without permanently damaging the FPGA. These types attacks are possible to implement in HDL code, especially when an attacker is aware of the FPGA system the code will be deployed on.

Druyer et al. discuss the possibility of hijacking an FPGA system during configuration in [16]. Dynamic FPGA reprogramming features provide an opportunity for attackers to hijack an FPGA at any point in time. Inactive FPGA configurations are usually stored in SRAM on the FPGA. Alternately, new configurations can be delivered to the FPGA through a network interface. If a 3rd-party IP has access to the SRAM or a network interface, it may be able to modify dormant FPGA configurations. This creates the opportunity for malicious

3rd-party IP blocks to reprogram the FPGA using a malicious configuration. Any FPGA that is able to reprogram itself using a dynamic reprogramming feature, or has access to its own configuration SRAM, may be vulnerable to a 3rd-party IP block modifying the configuration. If a 3rd-party IP block is able to modify the configuraiton, there is no limit to what it can do, including taking over the FPGA.

2.2.3 Trojans that Inject Faults

Hardware Trojans in FPGA IP may also manifest themselves as fault-injecting Trojans. These Trojans aim to disrupt operation of the FPGA by providing an incorrect result in some circuit operation. Fault injecting Trojans are one of the types that has been studied most often in developing Trojan tolerance strategies. Many conventional Trojan tolerance strategies, such as Adapted TMR [4] and MRVO [3] aim to mitigate the effects of fault-injecting Trojans. We can further classify fault-injecting Trojans into *Trojans that Inject Internally Verifiable Faults* and *Trojans that Inject Internally Unverifiable Faults* [17].

Past work in FPGA Trojans has discussed fault-injecting Trojans whose faults are internally verifiable [17]. A fault-injecting Trojan is internally verifiable if the result an IP block produces can be determined to be correct or incorrect without duplication of the system. An example of a Trojan whose result is internally verifiable is an FPGA that produces instructions to control a CPU. If the FPGA sends an invalid opcode to the CPU, the result is verifiably wrong.

In come cases, fault-injecting Trojans may be unverifiable without system duplication. Fault-injecting Trojans produce undetectable results when their results are indistinguishable from a correct result [17]. Incorrect results can be indistinguishable from correct results in a variety of circumstances. For example in [17], an FPGA provides instructions for a computer system to run. The authors explain that in the studied system, an incorrect result is indistinguishable from a correct result as long as the incorrect result is a valid instruction for the system to run. Fortunately, numerous redundancy-based verification strategies have

been proposed in past work, most of them using a majority voting scheme to determine the correct result [9, 4, 2, 3]. In the case of Trojans in 3rd-party IP blocks, designers may choose to purchase multiple copies of the same IP from different vendors and run those IP blocks concurrently or one at a time using partial reconfiguration [3].

Listing 1 shows a simplified example of a fault-injecting Trojan. The Verilog code for a 4-state machine is infected with a trojan triggered by the t_trojan signal that causes the state machine to unexpectedly reset. Unless the signal is activated during testing, the Trojan will not be caught. Although fault-injecting Trojans can be dangerous, the variety of existing mitigation strategies makes them less of a threat than other types of Trojans [2].

```
module fourState
    (
        input clk,
        input reset,
        input transition,
        input t_trojan,
        output [1:0] state
    );

    wire clk;
    wire reset;
    wire transition;
    wire t_trojan;

    reg [1:0] state;

    always @(posedge clk or negedge reset) begin
        if (!reset) begin
            state <= 0;
        end
        else begin
            if (t_trojan) begin
                state <= 0;
            end
            else if (transition) begin
                state <= state + 1;
            end
        end
    end
endmodule
```

Listing 1: Fault-Injecting Trojan in State Machine

2.2.4 Trojans that Cause Side Effects

Trojans in FPGA IP may have a payload that does not interfere with the logic of the design. These Trojans instead cause some side effect, often leaking information about the FPGA or wasting resources on a task not intended by the victim of the Trojan. Although these Trojans don't always disrupt the FPGA's work, they may have other extremely costly consequences. These Trojans can be classified into *Trojans that Leak Information* and *Trojans that Waste FPGA Resources*.

2.2.5 Trojans that Leak Information

Trojans in FPGA IP may aim to leak information about the FPGA or its environment [1, 3, 9]. These Trojans do not disrupt normal operation of the FPGA. Instead, they expose secrets about the FPGA - other IP implemented on the FPGA, data in the FPGA, or information from the systems surroundings. Trojans that leak information have been studied in the context of Trojans in FPGA hardware, but few tolerance strategies exist. We further classify these Trojans into *Trojans that Leak System-Internal Information* and *Trojans that Leak System-External Information*.

Trojans that leak system-internal information discreetly send information about the FPGA or data it is processing to a malicious actor. These Trojans can leak secret information such as cryptographic keys [18]. Leaking cryptographic keys may allow attackers to access even more data outside of the FPGA system. Trojans may also aim to leak information about the FPGA's design and configuration stored in the bitstream. Due to the high design costs of FPGA systems, Trojans may attempt to steal information about the FPGA configuration in order to reverse engineer and steal the system [1].

Trojans may also leak information external to the FPGA system. These Trojans take advantage of sensors and other peripheries to gather and transmit information such as the system's location and surroundings. These Trojans pose a particularly large threat to defense-related applications, where the locations of equipment must be kept secret from potential

```
module encryption
    (
        input [255:0] key,
        input [255:0] secret,
        input control,
        input t_trojan,
        input clk,
        input reset,
        output [255:0] encrypted
    );

    wire [255:0] key;
    wire [255:0] secret;
    wire control;
    wire t_trojan;
    wire clk;
    wire reset;

    reg [255:0] encrypted;
    reg [255:0] tmp;

    always @(*) begin
        .
        .
        .
        // Encryption logic
        .
        .
        .

    always @(posedge clk) begin
        if (!reset) begin
            encrypted <= 256'b0;
        end
        else begin
            if (t_trojan) begin
                encrypted <= secret;
            end
            else begin
                encrypted <= tmp;
            end
        end
    end
endmodule
```

Listing 2: Information Leaking Trojan in Encryption Module

attackers.

Listing 2 shows an example of an information-leaking Trojan. This Trojan, again triggered by the t_trojan signal, outputs an unencrypted version of the secret when triggered.

This again may not be caught during testing if the activation signal is never triggered.

Few strategies have been proposed to mitigate the effects of information-leaking Trojans. Traditional redundancy-based approaches will not suffice, as the Trojans' leaking information is separate from the results the IP block produces. Alanwar et al. propose a technique called Simple Blockage (SB) to obfuscate all information before it is sent to other parts of the system. This technique prevents malicious actors from reading any data, even when it is leaked [3]. Additionally, it comes at a lower design and power cost than redundancy-based approaches to security. However, hardware designers must take care to monitor every communication port on the FPGA, or the Trojan may be able to leak information undetected. Additionally, this strategy is not appropriate when the 3rd-party IP is responsible for implementing a communication protocol, as encryption of any data before it leaves the FPGA may break the implementation.

2.2.6 Trojans that Waste FPGA Resources

Trojans may also aim to impede an FPGA system by consuming an excessive amount of system resources. These Trojans may clog up network traffic or simply consume an excessive amount of power. Trojans that Waste FPGA Resources may do so by performing a separate, unrelated task to the benefit of an attacker, or by maliciously wasting system resources for no benefit. Because these Trojans have very similar effects and countermeasures, we do not draw a distinction between them in the taxonomy.

One potential motivation for resource-wasting Trojans is for an attacker to use the FPGA for their own benefit while avoiding detection. Chakraborty et al. discussed the idea of a Trojan harnessing an FPGA for a denial of service attack in [10]. These Trojans are similar in motivation to those that hijack a system, but more difficult to detect because the FPGA may remain completely functional while the Trojan is active. In modern FPGAs that use very small transistor processes, even measuring power consumption is not a reliable way to detect Trojans in FPGAs with very small transistor sizes. The concept of

FPGAs used in a botnet is very similar to the recent Mirai IOT device botnet virus [19]. We should anticipate that malicious actors may also take advantage of FPGAs in a similarly difficult to detect attack.

Resource-wasting Trojans may also aim to disrupt operation of the FPGA by consuming too many resources available to the FPGA. These Trojans may attempt to use all of the bandwidth for some communication mechanism, as in a denial of service attack. They may also simply increase the power consumption of an FPGA to an unsustainable level, behaving like computer power viruses [20] or the hardware power viruses discussed in [15].

Few Trojan tolerance strategies focus on these types of Trojans. Any system that aims to tolerate resource-wasting Trojans must first be able to detect them. Although DOS-based attacks are easy to detect, Trojans that don't waste an excessive amount of resources must be detected through some other type of system monitoring. Alanwar et al. discuss two strategies that help disable infected IPs, Multiplexing Reconfigurable Variants' Ouput (MRVO) and Multiplexing Reconfigurable IPs' Outputs and Cyclic Redundancy Check Trojan Detection Schema (MCRC) in [3]. These strategies both suggest using partial reconfiguration to swap out infected IPs at run-time. These redundancy-based approaches help mitigate the effects of resource-wasting Trojans, though they require at least one uninfected copy of the IP to fully eliminate them.

The genetic programming-based evolvable hardware strategy we introduce in a later section addresses side effect-inducing Trojans particularly well. It aims to disable the malicious functionality while preserving all of the intended behavior, which is not possible in most redundancy-based Trojan tolerance approaches without a golden copy of the IP.

2.2.7 Trojans that Introduce Vulnerabilities

A type of Trojan that has not been discussed in previous taxonomies is the Trojan that introduces other vulnerabilities into the system, but has no other ill effects. These Trojans

have been discussed in software Trojan vulnerabilities, and there is no reason they cannot also exist in hardware [21]. Hardware designers should be particularly careful about these types of Trojans, as they might be very difficult to detect. Producing no immediate payload makes the Trojan's effect undetectable until the vulnerability allows another sort of attack on the FPGA.

Trojans that introduce other vulnerabilities into the system might also be inserted by hardware designers rather than only by 3rd-party IP vendors. These types of Trojans can be difficult to distinguish from design errors, so they can allow malicious actors to have some sort of plausible deniability.

2.3 Existing Trojan Mitigation Strategies and FPGA IP

In order to understand what work is needed in the area of FPGA IP Trojan mitigation, it is necessary to examine current Trojan tolerance schemes and their effectiveness against the various types of IP Trojans. This section examines a variety of different Trojan tolerance strategies and examines the effectiveness of each in dealing with the different types of Trojans in FPGA IP.

It is important to differentiate between Trojan detection and Trojan tolerance. Trojan detection strategies provide a way for hardware designers to detect Trojans in their systems. These detection strategies may be at test-time, or continuous during run-time. Trojan tolerance strategies are run-time Trojan mitigation techniques that allow a hardware design to function normally in the presence of one or more hardware Trojans. In this sense, Trojan tolerance techniques might be more useful in a design than Trojan detection techniques. Because Trojans often hide themselves during testing, a tolerance technique will give hardware designers more confidence that their designs are secure [22]. Trojan tolerance techniques are usually based on either design for security (DFS) or run-time monitoring [23].

2 Hardware Trojans:Classification

Here in this chapter I will cover the concepts of hardware Trojans, their classification. This chapter presents concepts from available literature on hardware Trojans.

2.1 Hardware Trojan

Hardware Trojan is a malicious modification of the electronic design. This can be a simple stuck at zero or struck at one Trojan to a complex modification. A simple example of Trojan is an two input OR gate inserted between an output port with one input connected to logic-1 when triggered and the other to actual output. One more simple Trojan can be a two input AND gate with one input tied to logic-0 when triggered and the other input to actual output. Trojan consists of two parts Trigger and Payload. Trigger part decides when the Trojan is to be active and when it should be dormant. Payload part is actual modification to a system which does the damage. Trojan is designed by keeping two aspects one is malicious intent which determines extent of damage to be caused and the other is how to evade being detected during standard testing.

2.2 Taxonomy of Hardware Trojans

Here in this section hardware Trojan attributes such as where in the design, in which phase of the design hardware Trojans can be inserted, how they can modify expected behavior of the system are presented. Hardware Trojans can be inserted at any stage of design phase. Design phase of SoC,ASIC and FPGA consists of specification, design,fabrication and assembly and testing.Trojans can be inserted at any of these phases. Design phase consists of various abstraction levels such as System Level,Register-transfer-level, Gate level, Transistor level and Physical level. During transformation from one abstraction level to another at design time, many teams will be working on the design. At any stage of the transformation phase Trojans can be inserted. Hardware in digital designs consists of Processors, Memory,

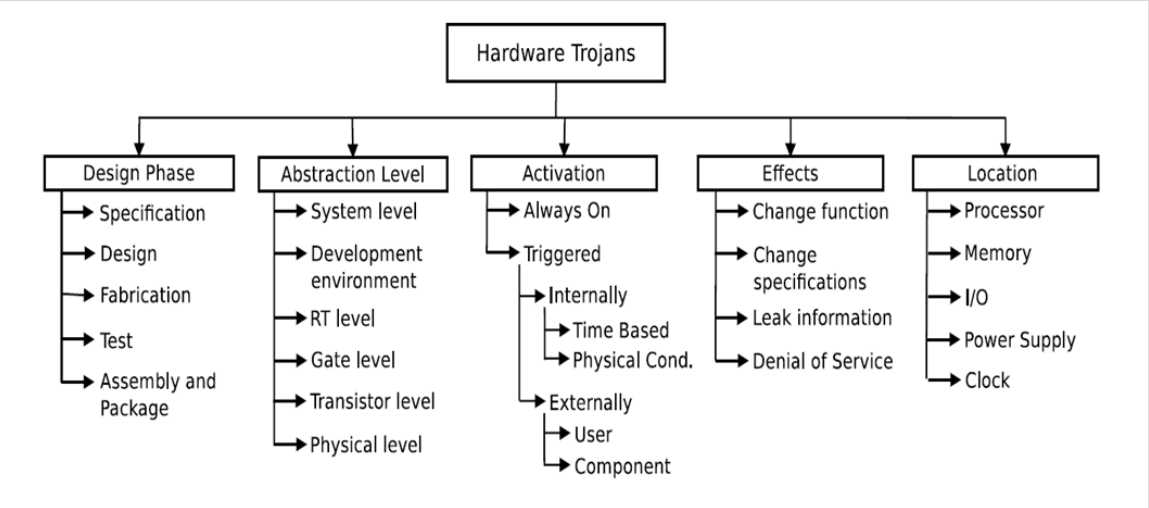

Figure 2.1: Taxonomy of hardware Trojans: Rajendran et al. (2010) [9]

Input/Output ports, Power supply and Clock etc. In any of these Trojans can be inserted. So far where and when Trojans inserted is discussed.When these Trojans becomes active is decided by activation/trigger mechanism and what it does when activated is called as payload.

2.3 Activation Mechanism

Based on activation Trojans are of two types

1. Always on
2. Triggered

2.3.1 Always on Trojans

In Always on category Trojan is active all the time when design is under operation. For example during fabrication some parameters such as thickness of metal layers at certain places can be made very small intentionally. On repeated use of the component permanent failure may happen to device after sometime due to early wear out. Another similar case is during fabrication if low power cells are replaced by

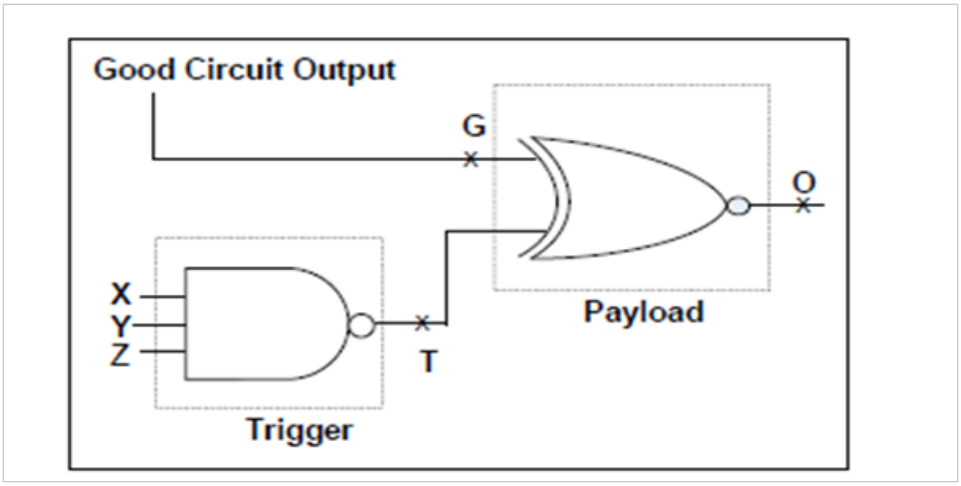

Figure 2.2: Combinational trigger mechanism [3]

high power consuming cells, for battery operated devices where power is a critical parameter, device fabricated does not meet its specification. These types of parametric variations can not be detected during the functional testing but causes permanent deteriorating effect on the system. This may not cause a damage but it certainly gives a loss to the designer in terms of time and reputation.

2.3.2 Internally triggered

Internally triggered hard ware Trojans make use of certain condition inside the device or component during the operation. There are in general two types of internally triggered mechanisms one is combinational and other is sequential.

A. Combinational Trigger Mechanism: In combinational trigger mechanism, only logic gates are used to form trigger circuit.In this memory elements are not involved.These trigger mechanisms makes use of certain internal nodes, when they take some predefined values, trigger is activated. In Fig 2-2 if each of the nodes X, Y,Z has the probability .01 to take certain logic value then all of them to take desired combination has the probability .000001. It means that during normal functional testing the probability of such nodes to get a desired

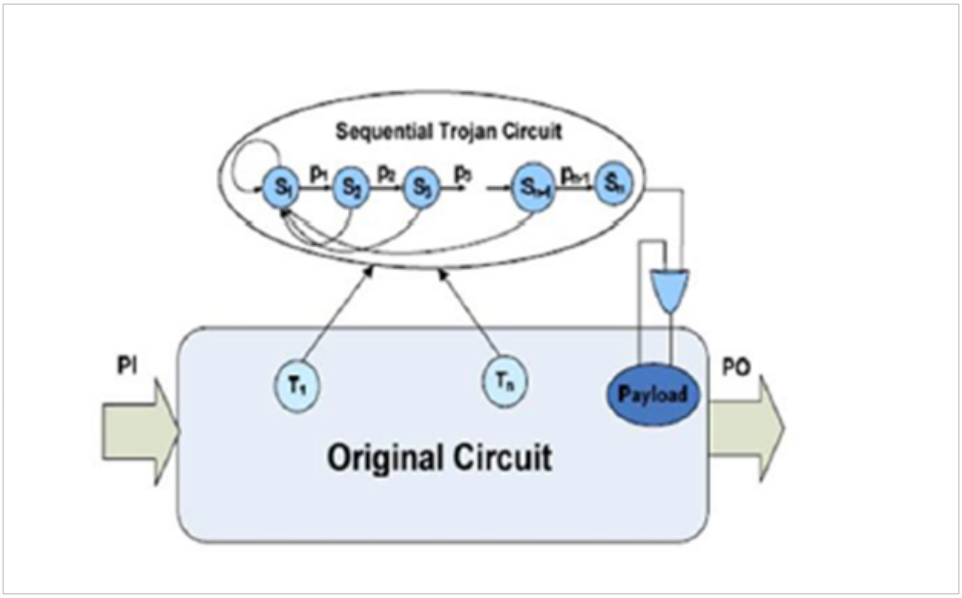

Figure 2.3: Sequential trigger mechanism [3]

value is very low. In general,these signals are taken from part selects of address & data buses.

B.Sequential Trigger Mechanism: Sequential Trojan circuit is enabled on the occurrence of certain events in a sequence. These events can be write to the control registers or read from certain address in memory. As shown in Fig 2-3, Sequential Trojans are implemented using state machines. To hold these states, memory elements such as flip flops and latches are used. During normal functional testing to detect these types of Trojans is very difficult because the Trojan designer can choose any complex type of sequence to trigger the payload.Simple counter based Trojan is an example for sequential Trojan design. After certain number of toggles of a rare node or clock, counter based Trojan gets activated.

2.4 Payload: Action of Trojans

Payload part of the hardware Trojan does the intended job of Trojan designer. It can be of following types.

- Functional modification
- Denial of service
- Specification modification
- Leakage of secret information.

2.5 Chapter Summary

In this chapter, I have described the basic model of the hardware Trojan consisting of trigger part and payload part, taxonomy of hardware Trojans. In the coming chapters where and how they can be inserted in SoC,FPGA are discussed.

3 Analysis of Trojan Insertion Possibility

In this chapter I am reporting the analysis of possibilities for Trojan insertion in FPGA, ASIC and Systems-On-Chip.

3.1 Hardware Trojans Inside FPGA: Analysis

FPGAs are used in many fields such as signal processing, embedded computing, multimedia, and security. FPGA devices in general are bought from different FPGA vendors and configured many times as and when required in the field. These vendors may fabricate these FPGA devices inside their own organizations or may send the designs to IC fabrication centers outside. This creates the possibilities for Trojan insertion in FPGA designs. In next two sections, I am presenting structure of FPGA designs and the analysis of Trojan insertion possibilities in the FPGA hardware designs.

3.2 FPGA Architecture

FPGA devices broadly consists of following major blocks

1. programmable logic blocks or Configurable logic blocks (CLBs)
2. Programmable Interconnects
3. Programmable I/O blocks
4. Digital clock managers
5. Multipliers
6. Block RAMS
7. Embedded processor cores (general purpose processor or DSP)
8. JTAG
9. Non-volatile memory (EEPROM).

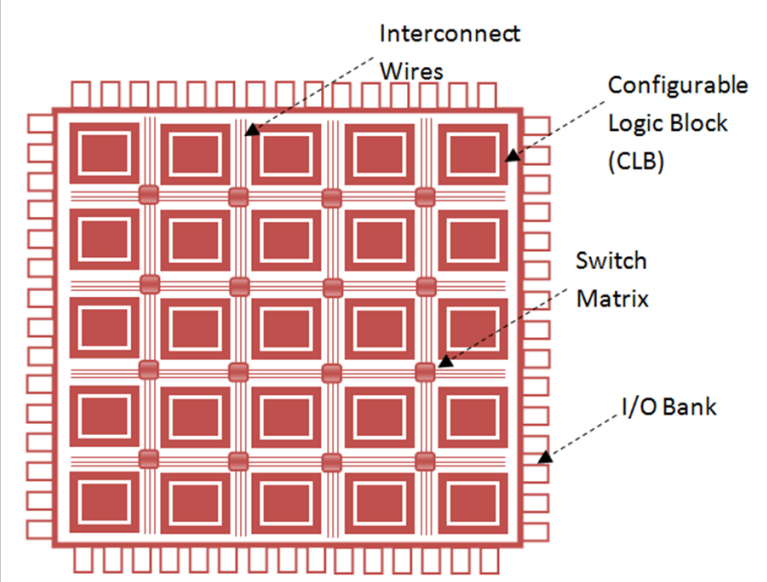

Figure 3.1: FPGA architecture block diagram [8]

3.2.1 Configurable Logic Blocks (CLB):

Configurable logic blocks typically consist of two slices SLICEL and SLICEM. Each slice consists of several logic cells. Logic cell contains lookup table, full adder carry logic, multiplexers and flip-flops. Lookup table implements the functionality by storing function as a truth table. It gives the set output of a function for a given input combination. Lookup tables are used to implement shift registers, combinational logic and distributed RAM.

3.2.2 Configurable Interconnect

Configurable interconnect provides interconnection among logic blocks to realise user defined function. There are connecting lines long and short to connect CLBs. Switch matrix provides connection among these connecting lines in flexible manner. Global clock lines are designed with lower propagation delays connecting clocked flops in CLBs minimising skew.

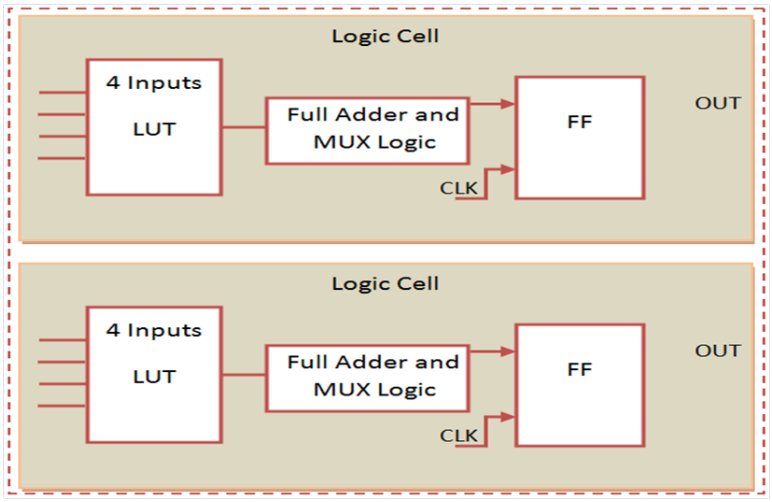

Figure 3.2: Logic Cell internal units [8]

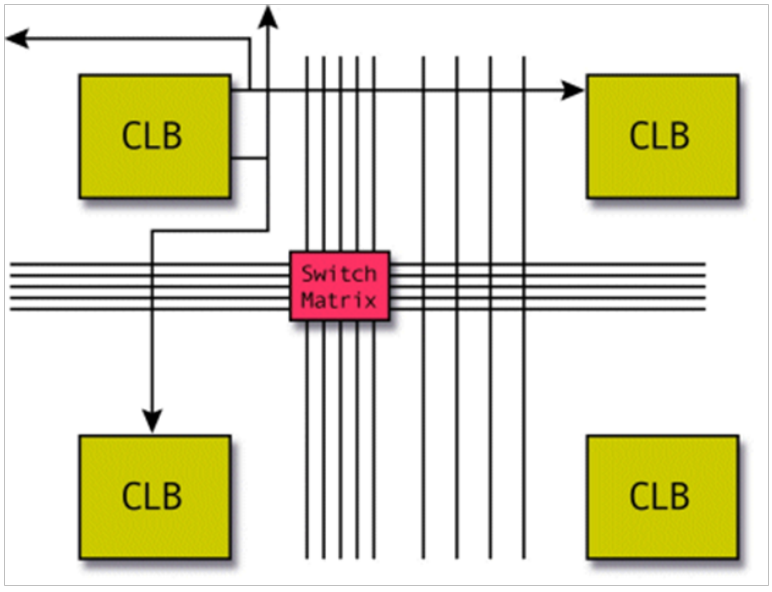

Figure 3.3: Routing in FPGA[8]

Digital Clock Manager: DCM performs frequency Synthesis, clock division, clock phase shift, clock jitter and skew management.

Multiplier block: Implements dedicated multiply operation for both signed and unsigned.

Block RAMS: Dual port Block RAMs are available in FPGAs to implement memory of the designs

3.3 Trojans insertion possibilities in FPGA

FPGA typically consists of regular array of configurable logic cells, other blocks such as digital clock managers, block RAMs and interconnect structure. So, an attacker can identify these regular blocks easily by doing reverse engineering the layout of FPGA to acquire required know how of the design to insert Trojans. Trojans can affect I/O s, memory, clocks and interconnect adversely.

3.3.1 Digital Clock Manager:

For desired frequency synthesis, division factor, multiplication factor and required amount shift for clock phase shifting is stored in configuration cells SRAM memory. These values can be modified after triggering payload which modifies configuration cell values. Trigger can be designed using counter based sequential Trojan as shown in Fig 3-4 [1]. Modification of multiplication factor or division factor if increases the clock frequency or change in the required phase shift causes timing being not met in clock based sequential designs which leads to a potential system failure.

3.3.2 Trojans in interconnect switch matrix:

If Trojans are inserted in switch matrix, which are enabled taking some time after start, then the interconnection among various blocks changes which in turn changes the functionality to some other arbitrary function. These Trojans can be independent of the design being configured in FPGA. Their activation when independent of the design

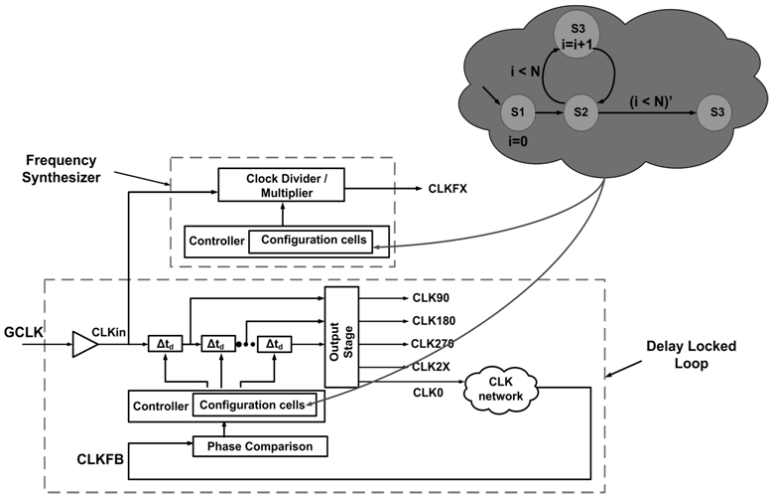

Figure 3.4: Trojan insertion possibilities in DCM[1]

can be rare because it is not known priorly in which part of the FPGA, design is going to be configured. When inserted at different places and corners of the FPGA device their chance of affecting the design becomes more.

3.3.3 Block memories

Block memories are used for table lookup in the designs to make computation faster. Normal flops and latches a.k.a distributed memory can be used for them, but their access times and area requirements are large. Block memories are denser and access times are less. These are vulnerable for Trojan insertion. These memories can be easily identified and what for they are used can be easily guessed if the context of the design is known. For example, if the configured design is symmetric Crypto blocks like DES and AES these memories are used for storing S-Boxes and round key constants. FPGAs being configurable on the fly, these memories can be reconfigured for some other values and from that it becomes easy to crack the key or decrypt the content from these ciphers.

3.3.4 Bit stream Decryption key:

In order to protect bit-stream copying and then reverse engineering the design, FPGA vendors encrypt the bit-stream in high end devices. If Trojan is inserted such a way that it intercepts the bus between decryptor and memory in which key is stored, key can be copied and send to attacker.

3.3.5 Trojans insertion at I/O:

I/Os in FPGA are either configured as input or output and are interfaced to external system. When Trojan is inserted at I/O it can turn on output enable for a port which is configured as input and turnoff output enable for a port which is configured as output. This causes excess current flow into the system connected and can damage the system connected [1].

3.4 Hardware Trojans inside Systems-on-Chip and ASIC: Analysis

In this section potential places and ways of inserting Trojans in a System-on-Chip are explored. Since System-on-Chip and ASIC development stages are same except the presence of more number of modules interconnected in SoC, ASIC is a design for dedicated function. What applies to ASIC applies to SoC as well in specific module. What applies to SoC in a specific module applies to ASIC.

3.4.1 System Level Trojans

In the case of System On chip it is usual practice in the industry to take processor, bus interconnect and peripheral IP cores from third party vendors like ARM, CEVA, and ST microelectronics instead of developing from scratch by themselves. When third party IPs are used then there is possibility to have Trojans in individual IPs or in the collusion of two or more IPs from the same IP vendor. Following four scenarios exists [11].

1. Passive interception

2. Modification
3. Diversion
4. Masquerading

Interception: In this scenario, an IP silently reads the communication between two other IPs and sends it to external system or another IP.

Modification: In this scenario third IP interrupts the communication between two IPs and modifies it maliciously.

Diversion: In this scenario IP diverts the communication between two IPs to other IPs.

Masquerading: In this scenario IP disguises, itself as another IP to get service from other IP or to control the behavior of overall system.

Here below I will describe threat scenarios by taking some example IP cores.

3.4.2 Processor:

When third party processor IP is used it is very difficult to test processor IP exhaustively. Designer must rely on the regression test suit provided by the vendor to verify it is what the vendor is intended to deliver and not modified in between during the delivery process. But when the processor IP has some malicious design following cases are possible. Processor IP can exploit reserved instruction op-codes. Processor can initiate DMA transfer to external peripheral and can send secure content outside of the system. Processor can write information intended for secure memory region into non-secure memory region by using shadow loads and stores if it has malicious memory access logic inside it. subsectionMemory Controller: when system uses memory mapped IO, Memory controller which contains malicious logic may change the contents of the specific IP for example if it effects the setting of device controller which controls the critical section of system then it may cause great damage to the system. These kinds of

Trojans identification at individual IP validation is very difficult. This kind of situations arises in a system level under operation.

3.4.3 Device controller:

Device controller controls devices like Ethernet, Modem, USB, Blue tooth, and different I/O components. An untrusted device controller may modify the data to/from the device from/to processor. When the communication is for authentication of some purpose then modification of few bits in data may cause denial of service.

3.4.4 Bus Interconnect:

Bus interconnect arbitrates the access of bus among peripherals. A malicious bus interconnect may grant access of secure region to a non-secure component. Interconnect may escalate the privileges of certain I/O s to access the secure region or can deny service to I/O with a malicious intent.

3.5 Chapter Summary

In this chapter possible ways of inserting Trojans in FPGA designs, System on chip are discussed. In FPGA while fabricating the FPGA Trojans can be inserted and in the design being configured also Trojans may be present or even during runtime because of on the fly reconfiguration of design Trojans can be inserted. In System-On-Chip Trojans may exist in single IP alone or it is possible for two or more IPs to collude to form a Trojan.

5 FPGA Implementation of Trojans and Detection

In this chapter as a part of proof of concept I am implementing Trojans in Field Programmable Gate Array (FPGA). To implement Trojans, design I have chosen is a serial transmission model UART serial communication. I have chosen UART because I can demonstrate the effect of Trojan using existing equipment PC's hyper terminal and FPGA board. For this I have taken existing reference design in verilog hardware description language at register transfer level abstraction and implementing Trojans in existing verilog code.

5.1 FPGA board details:

FPGA Manufacturer	: Xilinx
Family	: Spartan-6
Device	: XC6SLX100
Package	: FGG676
Speed grade	: -3

5.2 Specifications of the design:

Although the existing design supports many features and programmable, I am choosing UART design with transmit and receive frame of ten bits consisting one start bit, one-stop-bit, eight data bits and parity bit(s)none. Baud rate I am configuring is 9600. Port definitions of the module are given in Table 5.1

5.3 FPGA implementation

I have implemented this design on FPGA in the following steps

1. Verified the functionality of the design using Synopsys VCS simulator. Simulation wave forms where UART is both transmitting and receiving shown in Fig 5-1.

Table 5.1: UART Port list

S.No.	Name	Port Type	Definition
1	Clk	input	External Clock to configure registers
2	uart_ref_clk	input	External clock for data transmission and reception
3	Resetb	input	Design Reset
4	uart_ro	input	Serial data in
5	uart_di	output	Serial dta out
6	regfile_addr	input	Register address
7	regfile_data_in	input	Register data
8	uart_rts	output	Ready to send
9	uart_cts	input	Clearto send

2. Synthesized UART RTL design using Xilinx ISE into FPGA net list.
3. Implemented the floor plan, placement and routing in Xilinx ISE with I/O information in user constraint file (uart.ucf).
4. Generated the bit stream and configured the FPGA.
5. Connected the board to PC using serial port COM1 using cable to DB9 connector of FPGA board.

Design is configured for loop-back mode with baud rate of 9600. Transmit and receive frame are of 10-bits consisting 8-data bits, 1-start and 1-stop bit. Data is sent from hyper terminal, whatever the data received is transmitted back on to hyper terminal. Screen shot of the setup is shown in Fig 5-2.

5.4 Trojan insertion:

I have planned to implement the Trojans in the chosen design at RTL abstraction. Payload of the Trojan is to corrupt the transmitted frame

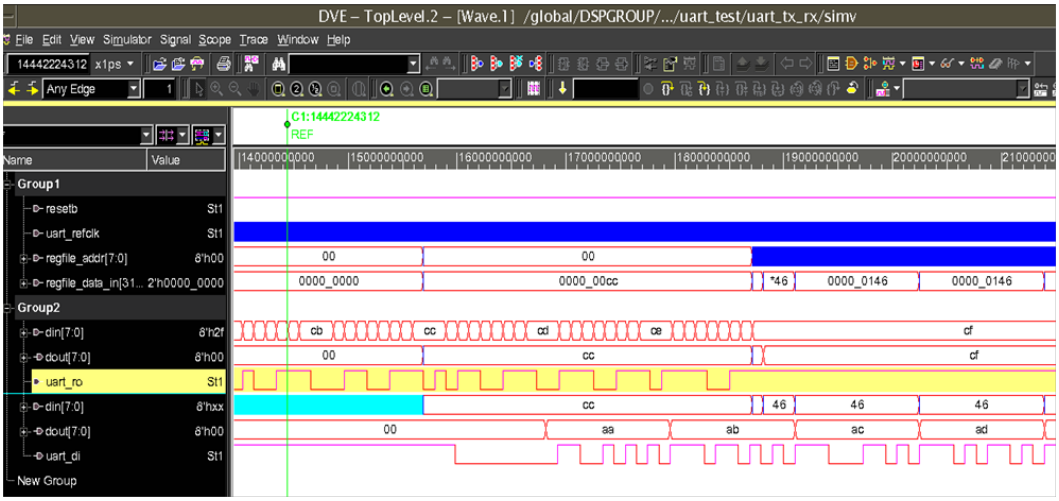

Figure 5.1: UART simulation waveform

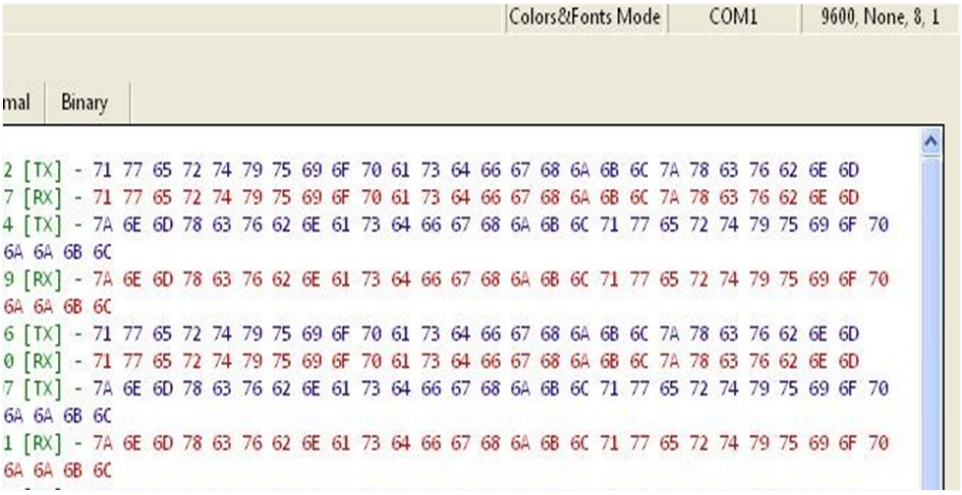

Figure 5.2: UART in loop back [self]

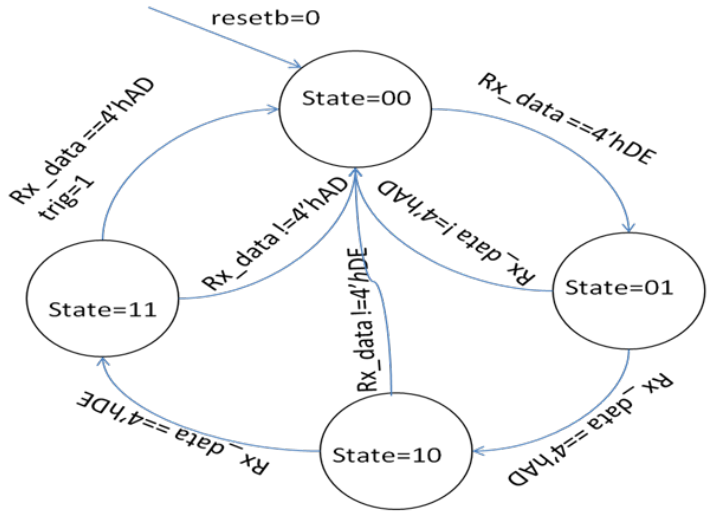

Figure 5.3: Trigger FSM in receiver [self]

after trigger gets activated.Denial of service(DoS) is another Trojan payload variation. In DoS when Trojan implemented is triggered one of the control signals in the transmitter or receiver are modified to permanently high or low then transmitter or receiver can't perform their function of transmitting, receiving. Following scenarios explain various trigger mechanisms used.

5.4.1 State machine in receive module:

In this scenario trigger is implemented in receiving module. When the sequence DE, AD, DE, AD is received trigger gets activated. For this sequence detection state machine as shown in Fig 5.3 is implemented in the receiver part of the design. After trigger activation transmitter replaces the four bits [7:4] in transmitted data bits with 4'hA.

```
reg state =2'b00;
always @ (posedge clk or negedge resetb)
begin
  if (!resetb)
```

```
begin
state <=2'b00;
cor<=1'b0;
trig<=1'b0;
end
else
                begin
                case (state)
                2'b00:
                 begin
                 if(rx_uart_fifo_dout == 8'hde)
                        state<=2'b01;
                 end
                2'b01:
                 begin
                 if(rx_uart_fifo_dout==8'had)
                        state<=2'b10;
                end
                2'b10:
                 begin
                        if(rx_uart_fifo_dout ==8'hde)
                                state<=2'b11;
                        end
                2'b11:
                 begin
                 if(rx_uart_fifo_dout==8'had)
                        begin
                        cor<=1'b1;
                        trig<=cor;
                        end
                 end
        default:
                        begin
                        state<=2'b00;
                        cor<=1'b0;
                        trig<=cor;
```

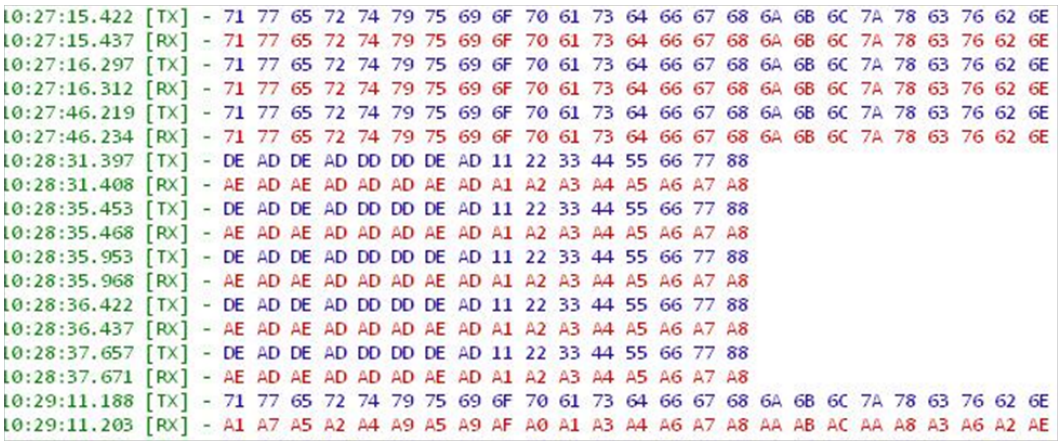

Figure 5.4: Frame corruption [self]

```
                    end
                     endcase
                     end //else
end //always
```

Effect of Trojan is shown in Fig 5-4 here in the setup UART design with Trojan inserted, is implemented on FPGA. UART is configured to work in loop-back i.e. what it receives is transmitted back. Initially design is transmitting and receiving as desired. When the sequence DE, AD, DE, AD is received design is replacing the most significant four bits of data bits in transmitting frame with hexa decimal number 'A'.

5.4.2 Counter based Sequential Trojan:

A large 32-bit counter implemented in the transmitter part of transceiver module which counts the number of transmitted data bytes. Once the count value reaches intended count Trojan gets activated. Trojan payload modifies or corrupts the transmitted data frame. This counter based Trojan in general goes undetected during simulation because to reach large count value takes longer simulation time. Counter is

implemented in the RTL of the design in verilog as shown in below code snippet.

```
reg  [31:0] count;
always @(posedge byte_transmitted or negedge resetb)
begin
if (resetb==1'b0)
count<=32'b0;
        else
        begin
                count<= count+1'b1;
                      if(count == 32'hFFFF_FFFF)
                         trig <=1'b1;
        end
end
```

Counter triggered denial of service pay load which modifies the control signal is demonstrated as shown in Fig 5-5 by implementing on the FPGA board. Here initially the UART design is functioning normally in loop back mode. Once set count which is kept very low for demonstration is reached, control signal responsible for transmission is made logic zero permanently. This causes transmission of UART to stop, though it is configured to transmit continuously. Thus counter based Trojan causing Denial of Service(DoS) is demonstrated.

5.4.3 State machine in transmit module:

This trigger is similar to the state machine Trojan trigger in receiver, after transmitting the DE, AD, DE, AD sequence Trojan trigger is activated. To detect this sequence a state machine is implemented in the transmitter part of the transceiver. State diagram is shown in Fig 5-6. Trojans in the design can be implemented as explained above or in combination of the above either modifying transmitted frame or causing denial of service or degrading the performance. Here the implementation of FSM in verilog is similar to that of implementation in receiver part. Payload can be either modification of transmitted

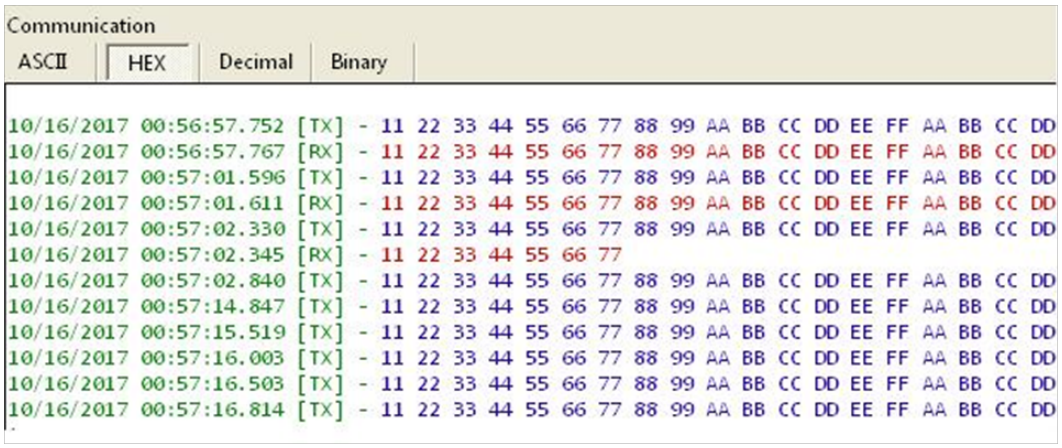

Figure 5.5: Denial of service [self]

frame or denial of service, Similar to the ones demonstrated in FSM in receiver part case and counter based Trojan case respectively.

5.4.4 Combinational Trojan:

In this case the trigger is activated by taking internal signals of the design. Once all these signals reach the targeted values the Trojan is activated. For implementation ten internal signals are chosen. When these ten signals takes certain values AND and XOR combination among them produces logic-1. When such a combinational circuit output becomes high for N- number of times Trojan gets triggered. Counter is implemented by taking output of combinational circuit as clock to counter. Trojan can have different payloads like modification of data bits in UART frame, denial of service, combination of these two or any other malicious modifications. Verilog implementation of trigger part is shown below.

```
wire cor1_en;
 assign cor1_en =( xor_tx_bits & xor_rx_bits &\\
 tx_parity_bit & rx_parity_bit & num_parity_bits\\
 & tx_bit_out)^  (start_byte_transmission & \\
 rx_uart_error & rxsample_count[3] & rxfsm_state[2]);
```

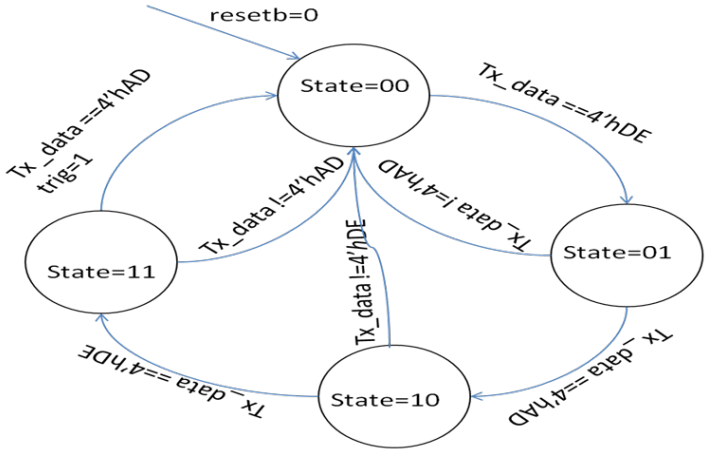

Figure 5.6: Trigger FSM in transmit module [self]

```
reg [15:0] count= 16'h0000;

always @(posedge cor1_en or negedge resetb)
begin
if(!resetb)
                begin
                        count <= 16'h0000;
                        cor<=2'b0;
                        cor1<=1'b0;
                end
else
        begin
                count<=count+1;
                if(count>=16'h002f)
                begin
                 cor <= 1'b1;
                 cor1<=cor;
                end
                else
                begin
```

```
                            cor<=1'b0;
                             cor1<=cor;
                    end
            end
end
```

5.5 Applying Detection Techniques

In the previous section how Trojans are inserted in the UART is presented. To detect these Trojans techniques discussed in third chapter have to be applied. Here I have planned to detect the Trojans using 1.side channel analysis using parameters power, delay and area 2.formal verification technique logic equivalence check.

5.5.1 Side channel parameter analysis:

For side channel parameter extraction from the design I am using industry standard Synopsys EDA tool Design compiler for estimating area, power and delay.

Though the design is implemented on FPGA because of proper tool set non-availability at my work place for FPGA I am using ASIC or SoC design tools. Since in side-channel-analysis metrics of deign under test are compared with the reference design, results will be similar for FPGA and ASIC with slight variation for example static power consumption for FPGA is more than ASIC or SoC for same design. Then percentage increase in static power for FPGA due to Trojan is less than for an ASIC. But if proven for ASIC there is high chance that it will be valid for FPGA in side channel analysis.

Working of the Design compiler is shown in Fig 5-7. It takes RTL design in HDL either verilog or VHDL, standard cell library and user constraints such as area, timing as inputs and translates them in to gate level net list in <design>.v from, synopsis design constraints and generates area, timing, power reports of design.

J Hardw Syst Secur (2017) 1:85–102
DOI 10.1007/s41635-017-0001-6

Benchmarking of Hardware Trojans and Maliciously Affected Circuits

Bicky Shakya[1] · Tony He[1] · Hassan Salmani[2] · Domenic Forte[1] · Swarup Bhunia[1] · Mark Tehranipoor[1]

Received: 27 July 2016 / Accepted: 8 March 2017 / Published online: 10 April 2017

Abstract Research in the field of hardware Trojans has seen significant growth in the past decade. However, standard benchmarks to evaluate hardware Trojans and their detection are lacking. To this end, we have developed a suite of Trojans and 'trust benchmarks' (i.e., benchmark circuits with a hardware Trojan inserted in them) that can be used by researchers in the community to compare and contrast various Trojan detection techniques. In this paper, we present a comprehensive vulnerability analysis flow at various levels of abstraction of digital-design, that has been utilized to create these trust benchmarks. Further, we present a detailed evaluation of our benchmarks in terms of metrics such as Trojan detectability, and in the context of different attack models. Finally, we discuss future work such as automatic Trojan insertion into any arbitrary circuit.

Keywords Hardware Trojan · Benchmarks · Hardware security

✉ Bicky Shakya
bshakya@ufl.edu

Tony He
tonyhe@ufl.edu

Hassan Salmani
hassan.salmani@howard.edu

Domenic Forte
dforte@ece.ufl.edu

Swarup Bhunia
swarup@ece.ufl.edu

Mark Tehranipoor
tehranipoor@ece.ufl.edu

[1] ECE Department, University of Florida, Gainesville, FL 32611, USA

[2] ECE Department, Howard University, Washington, DC 20059, USA

1 Introduction

The past decade has seen great advancement in research for hardware Trojan detection and prevention. Many techniques have been proposed for Trojan detection at several stages in the supply chain. However, such techniques, while bearing merits of their own, have several shortcomings. We highlight some of these shortcomings below.

- **Ad-hoc Trojans:** For each detection technique, researchers have mostly resorted to using 'home-grown' hardware Trojans to demonstrate the advantages and accuracy of the proposed techniques. While such Trojans might be suited to a particular detection approach, the results might vary vastly when they are used in the context of other detection approaches. Due to this, when comparing results between different detection techniques, there is never a baseline for comparing the merits of one technique to another. Further, it is not clear if these Trojans satisfy the basic characteristic of a hardware Trojan, i.e. it must be able to bypass absolutely all commonly used manufacturing test methods such as functional, structural, fault-based tests etc.
- **Varying Assumptions:** The simulation/implementation environment and factors such as the amount of process variation allowed, difficulty of triggering

Trojans, Trojan switching activity, size of design (no. of gates, Trojan size) etc. also vary greatly from one technique to another. This further compounds the problem of comparing various Trojan detection schemes.

– **Ad-Hoc Metrics:** The metrics used for evaluating detection techniques have been mostly ad-hoc as well. Some researchers may choose to evaluate their technique in terms of an arbitrary percentage detection rate, some may present false positive/false negative rates and some may explain their results in terms of test coverage. While these techniques may be working with the same Trojan attack model, comparison between them becomes difficult with ad-hoc figures of merit.

Thus, there is a need for a unifying approach for hardware Trojan detection. In order to address this need, we have developed tools to assess the vulnerability of designs that can be exploited for insertion of various types of Trojans, as well as tools to evaluate the stealthiness or difficulty of detecting Trojans. Using these tools, we have designed an array of Trojans that have been carefully integrated into various circuits to create "trust benchmarks". Benchmark circuits are commonplace in many different fields. Various circuit benchmarking efforts such as the ITC '02 [1], ISCAS '85 [2] and '89 [3] benchmark sets have allowed comparative research in the field of modular SoC testing and combinational/sequential logic synthesis. Further, research fields such as multimedia systems, computer architecture, digital signal processing and machine intelligence have greatly benefited from their own set of benchmarks, which have allowed objective comparison of different techniques [4, 5]. This widespread use of benchmarks in various fields only emphasizes the need for specific benchmarks in the field of hardware Trojan detection. Our hope is that these benchmarks will allow researchers to implement their Trojan detection schemes and compare them to others on a level playing field.

The Trojans that we have developed are (i) not detectable by standard manufacturing tests; (ii) vary in size and distribution to fit different Trojan insertion scenarios; and (iii) are implemented at different levels of abstraction from RTL, netlist, to layout. As we have seen in Table 1, a large number of potential Trojans can be inserted at different levels in the supply chain. Taking this into consideration, we have developed trust benchmarks that cover each stage in the supply chain, and have been implemented in different platforms from microcontrollers, cryptographic IP cores to JTAG controllers. This can help the research community and industry to prioritize their efforts for developing defenses against hardware Trojans. A further merit of these trust benchmarks is that it will help in the reproducibility of results.

Table 1 Comprehensive adversarial models for hardware Trojans

Model	3PIP vendor	SoC developer	Foundry
A	Untrusted	Trusted	Trusted
B	Trusted	Trusted	Untrusted
C	Trusted	Untrusted	Trusted
D	Untrusted	Untrusted	Untrusted
E	Untrusted	Untrusted	Trusted
F	Untrusted	Trusted	Untrusted
G	Trusted	Untrusted	Untrusted

This is vital for transferring these detection techniques from research to real-world implementation. Further, each Trojan we have developed has been analyzed by concrete metrics, which establishes a sound basis for analyzing the hardness of detecting Trojans in each benchmark instance. Note that for these trust benchmarks, the Trojans have been manually inserted into the benchmark circuits after vulnerability analysis. We do not perform an automatic payload identification for any arbitrary circuit, which would be part of our future work (Section 7). This would aid in developing "Trojan benchmarks", which are custom-designed Trojan circuitry that can be inserted into any arbitrary circuit, as opposed to the specific benchmark circuits we have used for developing the trust benchmarks. Compared to [6] which only looked at gate-level benchmarks, this paper explores trust benchmarks and vulnerability analysis at different levels of abstraction (RTL, gate, layout, FPGA), introduces attack models for benchmark evaluation and thus, takes a holistic approach towards trust benchmark development.

The rest of the paper is organized as follows. In Sections 3 and 4, we will introduce a comprehensive Trojan taxonomy and review the benchmarks we have developed over the course of the past few years. In Section 5, we will describe the tools we have developed to perform vulnerability analysis on a design. In particular, we will talk about a vulnerability analysis flow that can determine which parts of a circuit are more susceptible to Trojan insertion, at the layout, gate and behavioral level. In Section 6, we will discuss the Trojan evaluation suite we have developed, which is based on the efficiency of test patterns in activating the Trojan and a Trojan's resiliency to side-channel analysis. The evaluation suite will help us to explain the trust benchmarks in terms of concrete metrics such as 'detectability'. In the same section, we will also present results on the trust benchmarks, based on our Trojan evaluation suite and the attack models we presented earlier. In Sections 7 and 8, we will provide remarks on future work for further development of trust benchmarks and conclude the paper.

2 Background

2.1 Hardware Trojans

A hardware Trojan is defined as a malicious, undesired and intentional modification made to an electronic circuit. Such a modification can potentially bring about a variety of effects [7], such as:

- **Change of functionality:** A hardware Trojan can alter the functionality of a circuit and cause it to perform malicious, unauthorized operations, such as bypassing of encryption algorithms, privilege escalation, denial of service etc.
- **Degradation of performance**: A hardware Trojan could also cause damage to the performance of an IC and cause it to fail, which could potentially jeopardize the (critical) system into which the IC is integrated. Such effects could be in the form of induced electromigration of wires by continuous DC stress, increase/decrease in path delay, fault injection etc.
- **Leakage of information**: Trojans could also undermine the security provided by cryptographic algorithms or directly leak any sensitive information handled by the IC. This could involve leakage of cryptographic keys or other sensitive information through debug or I/O ports, side-channels (delay, power) etc.

2.2 Adversarial/Attack Models

In order to accomplish one or more of the effects shown above, a malicious entity present in any given stage of the IC design/manufacturing process (Fig. 1) can insert the Trojan at various levels of abstraction. This brings up the need to define hardware Trojan in the context of different adversarial models. In Table 1 and below, we present a comprehensive list of adversarial models that show exactly when, where and how a Trojan can be inserted into an IC [8]. The unique sources of Trojans are 3PIP vendor, SoC integrator, and foundry. A Trojan can be inserted by one or more of these entities as follows:

- **Model A:** In this attack model, the third-party IP that a SoC developer or design house buys may contain hardware Trojans. This is a considerable threat in today's semiconductor design landscape, where SoCs are made by integrating several 3PIPs in order to reduce complexity, cost and shorten time-to-market. The effect and design of the Trojan can vary depending on whether the IP is soft (RTL-level), firm (netlist-level) or hard (GDSII).
- **Model B:** Attack model B relates to the threat of untrusted foundry/assembly. Since a foundry has access to all layers of the design, they can perform reverse-engineering of the design, in order to add/modify/delete gates and create Trojans in the design. This attack model is especially significant in today's horizontal semiconductor business model, where the design house has little to no control of the off-shore foundries.
- **Model C:** Attack model C relates to an untrusted design house. This can be either due to an untrusted EDA tool used by the design house or a rogue employee (malicious insider) that maliciously modifies the design.
- **Model D:** This attack model relates to the threat faced by most consumers or system integrators (e.g. PCB developers), who are forced to buy commercial off-the-shelf (COTS) in order to minimize costs. Since they have no control over any aspect of the design/manufacturing process, the threat of a Trojan insertion can occur at all three sources.
- **Model E:** In this attack model, all parties except the foundry are assumed to be trusted. This can be the case

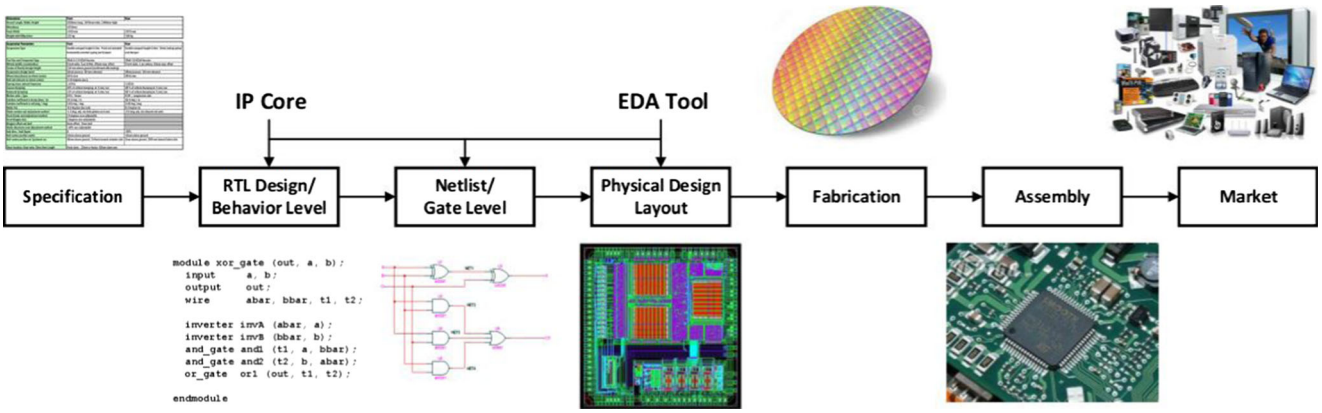

Fig. 1 Supply chain for IC production

when the foundry and the manufacturing process are trusted but the development process is not. This model can also account for cloned ICs, where malicious parties could reverse engineer a Trojan-free chip and create designs with Trojans inserted.

- **Model F:** This adversarial model applies to a majority of trusted design houses today who are forced to rely on untrusted 3PIPs and foundries.
- **Model G:** This attack model relates to companies who have designed their own proprietary IPs but need to rely on an untrusted design house and foundry to manufacture their final ICs.

2.3 Detection Techniques

With such an array of vulnerabilities from one or more untrusted entities, there is a pressing need to develop hardware Trojan detection techniques for assessing a design at various levels of abstraction. To address this need, the hardware security research community has proposed a plethora of Trojan detection techniques over the past decade. These efforts can be broadly separated into three categories.

- **Post-silicon detection** includes destructive and non destructive techniques to detect Trojans in manufactured chips. In destructive techniques, a fabricated chip is completely reverse-engineered layer-by-layer in order to reconstruct the design and compared with a Trojan-free or 'golden' design to detect Trojans. While such techniques offer high probability of Trojan detection, the time and cost required to perform reverse-engineering can be prohibitively high. Non-destructive techniques focus on detecting Trojans using functional tests or side-channel analysis. In *functional tests*, test-vectors are applied to the design and the responses/outputs are compared to the correct results to find anomalies (which could potentially be Trojans). However, a Trojan designer will, more likely than not, make sure that the Trojan is activated only under very rare conditions (such as an extremely rare test pattern), so that it can evade standard manufacturing tests. Added to this problem, modern designs may have a very high number of test patterns, making it impossible to conduct exhaustive functional tests. Prior work in Trojan detection has looked at techniques for generating test-patterns that can specifically target rarely activated nets [9–12]. However, the large number of states/inputs in modern designs limit the accuracy of these approaches, especially for Trojans that have extremely low trigger probability. In *side-channel analysis*, the impact of Trojans on circuit delay, transient current, leakage power, thermal profiles etc. are used for detection [13–19]. Most, if not all of these techniques require a golden model for comparison/detection, which might not always be available. Further, with the large process variation experienced at advanced nodes, such side-channel analysis based techniques might produce a significant amount of false positives/false negatives, limiting their applicability.
- **Pre-silicon detection** is necessary to detect Trojans that could have been inserted in 3PIP cores, by untrusted EDA tools, and/or by rogue employees in the design itself. Netlist-level IP cores can be tested by using the same functional testing techniques, as described above. For soft IP cores, various techniques such as code or structural analysis [20, 21] have been proposed, which analyze hardware description languages for redundant lines of code, conditional statements that rarely trigger etc. which could be possible locations where Trojans have been coded down. Formal verification has also been recently adapted for Trojan detection, where IP cores are tested by proof-checking/model-checking to make sure they perform the intended functionality and nothing else [22–25]. Such techniques are limited due to two issues. Firstly, most 3PIP blocks come as hard macros or in encrypted form, whose internal implementations are often inaccessible and can only be used as black-boxes. Secondly, for functional verification, there might always exist Trojans that could satisfy proof-checking constraints and evade detection.
- **Design for Trust** techniques are also necessary in order to make Trojan detection easier and/or Trojan insertion prohibitively difficult. Towards facilitating detection, techniques such as facilitating functional test by increasing the observability of nodes [26], increasing side-channel activity caused by Trojans [27] and run-time monitoring [28] have been proposed. On the other hand, in order to make Trojan insertion difficult, researchers have proposed techniques such as logic obfuscation to lock the functionality of the IC [29–31], functional filler cell insertion [32] for layout protection, IC camouflaging [33] and split-manufacturing [34, 35] to prevent reverse-engineering of the design. These design-for-trust techniques can potentially help to detect and prevent highly stealthy Trojans that can otherwise evade pre and post-silicon detection techniques, albeit at the cost of area, power and timing overhead.

2.4 Hardware Trojans in FPGA Designs

FPGAs are widely used today in an array of embedded applications ranging from telecommunications and data centers to missile guidance systems. Unfortunately, the

outsourcing of FPGA production and the use of untrusted third party IPs has also given rise to the threat of Trojan insertion in them. FPGA-based Trojans can be in the form of IP blocks (hard, soft or firm), which get loaded onto a generic FPGA fabric and cause malicious activity (denial-of-service, leakage etc.) on the system in which the FPGA is deployed. Such FPGA IP-based Trojans are more or less similar to their counterparts in an ASIC design flow (with the exception of layout-based Trojans, which are not applicable to FPGAs). However, Trojans that 'pre-exist' in an FPGA fabric and could potentially be inserted by an untrusted foundry or vendor pose unique threats and challenges of their own. FPGAs contain a large volume of reconfigurable logic in the form of lookup tables, block RAM and programmable interconnects, which can be used to realize any arbitrary sequential or combinational design. However, there might be a significant amount of reconfigurable logic open to a malicious party (e.g. the FPGA foundry or even the FPGA vendor) who can load a hardware Trojan and affect the FPGA-integrated system or compromise the IP loaded onto the FPGA. These FPGA device-specific hardware Trojans and their effects are explained in [36] and summarized below.

2.4.1 Activation Characteristic

Hardware Trojans in FPGAs can have activation characteristics similar to the ones described in Section 3.1 such as always-on or triggered. However, a unique characteristic of FPGA device-based hardware Trojans is that they can either be IP dependent or independent.

- *IP-dependent Trojans:* A malicious foundry or FPGA vendor may implement a hardware Trojan that can monitor the logic values of several LUTs in the FPGA fabric. Once triggered, such Trojans can corrupt other LUT values, load incorrect values into BRAMs or sabotage configuration cells. Since any arbitrary IP may be loaded onto the FPGA, the malicious foundry or vendor could distribute trigger LUTs throughout the FPGA so that the probability of the Trojan triggering and causing malfunction may increase.
- *IP-independent Trojans:* A malicious foundry or vendor may also implement a Trojan into an FPGA chip that is completely independent of the IP loaded onto it. Such Trojans can occupy a small portion of FPGA resources and malfunction IP-independent but critical FPGA resources such as digital clock managers (DCM). One potential mode of attack would be the Trojan increasing or decreasing the design clock frequency by manipulating the configuring SRAM cells of the DCM unit, which can cause failure in sequential circuits.

2.4.2 Payload Characteristics

FPGA device-based Trojans can also bring about unique malicious effects, such as causing malfunction of FPGA resources or leakage of the IP loaded onto the FPGA.

- *Malfunction:* Hardware Trojans in FPGA devices can either cause logical malfunction by corrupting LUT or SRAM values, thereby affecting the functionality of the implemented IP, or by causing physical damage to the FPGA device. For example, a triggered hardware Trojan could reprogram an I/O port set as an input, as an output while suppressing the configuration cells that prevent it from being programmed as such. This would cause a high short-circuit current to flow between the FPGA and the system it is connected to, thereby leading to physical device failure.
- *IP Leakage:* FPGAs today offer bitstream encryption capabilities in order to protect the IP loaded onto an FPGA device. However, such encryption only prevents a direct, unauthorized readback by software. A hardware Trojan may circumvent such protection by either leaking the decryption key or even the entire IP. The Trojan may tap the decryption key as it comes out of non-volatile memory, or the actual decrypted IP , which could then be exfiltrated either via covert side-channels (e.g. power traces) or through JTAG, USB or I/O ports.

3 Trojan Benchmarks

3.1 Trojan Benchmark Taxonomy

It is difficult to model a Trojan similar to faults in a circuit. Although both faults and Trojans can potentially cause errors in a circuit, one is based on random or systematic manufacturing defects, and the other is based on malicious intent. Defects can be modeled but intention cannot. Thus, to address this issue, we have developed a comprehensive Trojan taxonomy based on the vulnerabilities in the modern horizontal design and test processes and the opportunities adversaries may have at different levels of abstraction.

Over the past few years, there have been efforts to develop comprehensive hardware Trojan taxonomies based on Trojan implementation and effect [7]. We have further improved the taxonomies of the earlier works by including the physical characteristics of Trojans. Presented in Fig. 2, the Trojan taxonomy is broken down into the following categories:

Insertion Phase The design flow consists of several phases from determining design specification to its assembly and

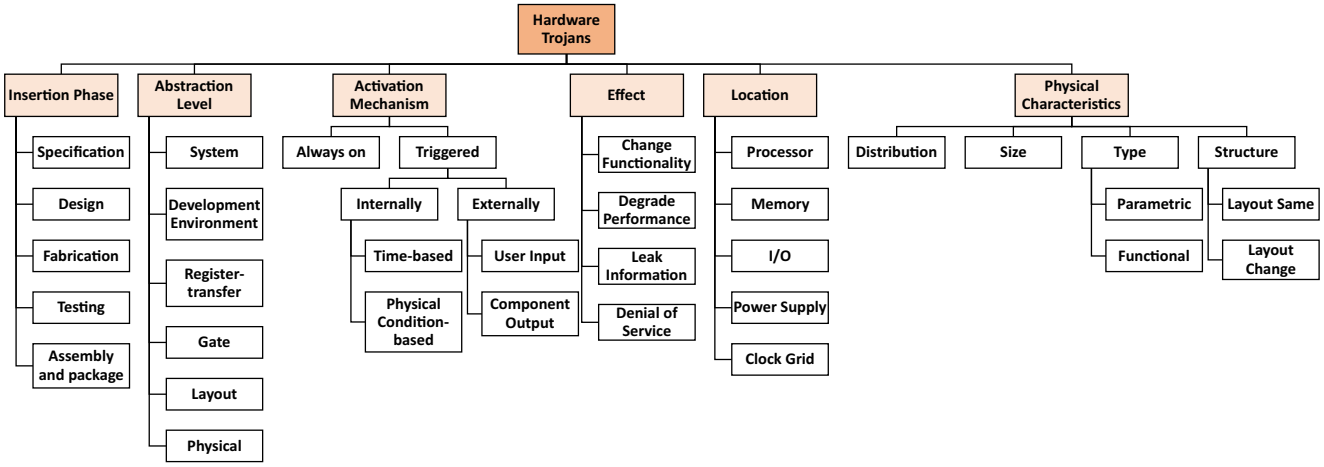

Fig. 2 A Comprehensive hardware Trojan taxonomy for Trojan benchmark development

packaging. Due to the globalization, circuit tampering can occur at different stages. For example, a Trojan can be realized by adding some extra gates to a circuit netlist at the design phase or by changing its masks during the fabrication step.

Abstraction Level The level of abstraction determines the control and flexibility an adversary may have on Trojan implementation. At the system level, a circuit is defined in terms of modules and the interconnections between them, limiting an adversary to the modules' interfaces and their interactions. On the other hand, all circuit components, their dimensions, and locations are determined at the physical level. A Trojan can be inserted in the white/dead spaces of circuit layout with the least impact on circuit characteristics.

Activation Mechanism Trojans may always function, or they can be conditionally activated. Always-on Trojans start as soon as their host designs are powered on, while conditional Trojans seek specific internal or external triggers to launch.

Effect Trojans can be characterized based on their effects. They may change a circuit's functionality, for example, by modifying the data path of the processor. Trojans can also reduce a circuit's performance or degrade its reliability by changing its physical parameters.

Location Every part of a circuit can potentially be subjected to Trojan insertion. A Trojan can be distributed over several regions or focused in one region. A Trojan can tamper with a processor to manipulate its controller or data path units. For example, on a printed circuit board (PCB) including several chips, an inserted Trojan on these chips' interfaces can disturb chip-to-chip communication.

Physical Characteristic Trojans can alter a circuit's physical characteristics, an assault that has many hardware manifestations. A Trojan can be a functional or parametric type. Functional Trojans are realized by the addition or deletion of transistors/gates, and parametric Trojans by modification to wire thickness or any other circuit parameter. Trojan cells can be distributed loosely or tightly in the physical layout in white spaces or spaces created by displacing the cells of a main circuit.

3.2 Sample Trojan Benchmarks

We have developed several Trojan benchmarks that have been inserted into a variety of benchmark circuits. For a given circuit, different Trojans can cause different effects, such as those shown in the effects taxonomy in Section 3.1. For example, a gate-level Trojan in a circuit may leak an internal net-value to the primary outputs. On the other hand, a different Trojan in the same circuit may inject erroneous values into the internal nets. Thus, for one given benchmark, we can have several different Trojans inserted into it. Keeping this in mind, we have developed a naming convention for each unique Trojan inserted in a benchmark circuit. Each of such 'Trojan benchmarks' is named as T_i, where i (a two-digit number) denotes a unique Trojan. For example, for the $s35932$ benchmark, the $T1$ Trojan benchmark maliciously activates the scan enable of the circuit and leaks an internal value through a test output pin. On the other hand, the $T2$ Trojan benchmark applies a dominant value to the same design in functional mode, thereby bypassing four gates of the main design. Note that $T1$ for one benchmark does not necessarily mean that the exact same Trojan appears

in a different benchmark with a Trojan benchmark labeled $T1$.

4 Trust Benchmarks

A "trust benchmark" is a benchmark circuit (generic circuits at the RTL, gate or layout level) which has Trojan(s) deliberately added to it at hard-to-detect, impactful and/or opportunistic locations (e.g. rare nodes, layout white-space etc.), for the purpose of comparing impacts of Trojans and the effectiveness of different Trojan detection techniques. Our initial efforts have focused on "static" trust benchmarks, which we define as those in which the location and size of the Trojan do not change. Our current trust benchmarks are available at http://www.trust-hub.org/taxonomy.

Each benchmark comes ready with documentation, that lists down important features of the trust benchmark such as trigger probability (for gate/layout level Trojans), exact effect of the Trojan, input combination required to trigger Trojan (for RTL/gate level), Trojan-induced delay or capacitance etc, size of Trojan/overall circuit etc. Additionally, for some benchmarks, we have provided a 'golden model', i.e. a version of the same circuit without Trojans, which can be handy for analyzing the trust benchmarks in terms of different attack models (see Section 6.4). Finally, for most of the trust benchmarks, we have included two testbenches, one of which can be used with the golden model (for debugging/test purposes) and the other which can be used to trigger the Trojan. For RTL level trust benchmarks, the testbench is in the form of Verilog/VHDL testbenches that have the Trojan trigger specified. For netlist/gate level benchmarks, exact test patterns to trigger the Trojan are provided. Finally, the documentation for each trust benchmark contains the exact form and location of the inserted Trojan. For example, for RTL level Trojans, the part of the RTL code that implements the Trojan has been documented. For gate-level circuits, a snippet of the Trojan netlist has also been provided. We have disclosed the exact location and implementation of the Trojan to make it easier for researchers to present results in terms of detection accuracy. However, it should be noted that such information should only be used 'a posteriori', as taking into account the Trojan implementation and location beforehand might unfairly bias detection techniques. Lastly, we note that this is an ongoing effort, and we are continuously generating various trust benchmarks to cover the Trojan taxonomy and improve on existing ones. We encourage the community to submit Trojans to us as well for inclusion on the website.

In the following, we explain our naming convention for trust benchmarks. We then present some of the representative benchmarks from the lot of approximately a hundred benchmarks developed so far.

4.1 Benchmark Naming Convention

We have developed benchmarks of different sizes, different types (ASIC, microprocessor, etc.), and with different Trojans covering the taxonomy shown in Fig. 2. Moreover, a Trojan can be inserted in several circuits and also placed in different locations within each circuit. Further, it is possible to modify and update a Trojan to a new version over time. Based on the above, we develop the following naming convention to assign a unique name to each Trojan benchmark in a trust benchmark circuit:

DesignName-Tn#$

where

- **DesignName**: The name of the main design without a Trojan. There is no limit on the number of letters or characters for the design name.
- **Tn** (Trojan number): It is of a maximum two digits. Note that the same Trojan number in different designs does not represent the same Trojan.
- **#** (Placement number): The second to last digit indicates the different placement of the same Trojan in a circuit and ranges from 0 to 9.
- **$** (Version number): The last digit in a benchmark name indicates the version of the Trojan and ranges from 0 to 9. This is added as a feature in case a new version of the same Trojan with the same placement has been developed. The version number will differentiate the older version from the new one.

For example, MC8051-T1000 indicates that Trojan number 10 (T10) was inserted in the micro-controller 8051 (MC 8051) at the location number 0, and its version is 0. As another example, dma-T1020 means that Trojan number 10 (T10) was inserted in the DMA circuit at the location number 2 and its version is 0. As aforementioned, Trojan T10 in DMA is not necessarily the same as Trojan T10 in MC8051.

4.2 Sample Trust Benchmarks

In the following, we present some of the benchmarks with a brief description of their enclosed Trojan.

- **Insertion Phase - Fabrication:** Trojans can also be realized by adding/removing gates or changing the circuit layout during GDSII development, and the mask during fabrication.
  Sample Benchmark: EthernetMAC10GE-T710 contains a Trojan triggered by a combinational comparator circuit which seeks a specific 16-bit vector. The probability of Trojan activation in this case is 6.4271e-23. When the Trojan is triggered, its payload gains control over an internal signal in the circuit.

- **Abstraction Level - Layout:** Trojans can be realized by varying circuit mask, adding/removing gates, or changing gate and interconnect geometry to impact circuit reliability.
  Sample Benchmark: EthernetMAC10GE-T100 contains a Trojan on a critical path. The net fault_sm0/n80 is widened to increase coupling capacitance, enabling crosstalk.
- **Activation Mechanism - Triggered Externally:** Trojans become activated under certain external conditions, such as by an external enable input.

  Sample Benchmark: RS232-T1700 contains a Trojan triggered by a combinational comparator. The trigger input probability is 1.59e-7 and it is externally controlled. Whenever the Trojan gets triggered, its payload gains control over the output xmit_doneH signal.
- **Effect - Change Functionality:** After activation, a Trojan will change the functionality of a circuit.
  Sample Benchmark: RS232-T1200 contains a Trojan triggered by a sequential comparator with probability is 8.47e-11. Whenever the Trojan gets triggered, its payload gains control over the output xmit_doneH signal.
- **Location - Power Supply:** A Trojan can be placed in the chip power network.

  Sample Benchmark: EthernetMAC10GE-T400 is modified with narrow power lines in one part of the circuit layout.
- **Physical Characteristic - Parametric:** A Trojan can be realized by changing circuit parameters like wire thickness.

  Sample Benchmark: EthernetMAC10GE-T100 contains a Trojan on the critical path. Specifically, the net fault_sm0/n80 is widened.

Table 2 presents a complete list of trust benchmarks that have been developed so far. They are categorized based on the Trojan taxonomy. The number of trust benchmarks available for each type, along with the names of the main circuits/benchmarks the Trojans have been inserted into are also presented. For instance, the table shows that 25 Trojans are inserted at the gate-level, 51 at the register-level, and 12 at the layout level, under the row 'Abstraction Level'. As another example, the 'effect' row shows that 35 Trojans change circuit functionality, 3 degrade circuit performance, 24 leak information to the outside of a chip, and 34 will perform a denial-of-service attack when activated. Note that some benchmarks fall under more than one category. Currently, there are a total of 91 trust benchmarks on the Trust-Hub website.

5 Design Vulnerability Analysis

In order to create the Trojans and trust benchmarks, we have created a suite of tools that first evaluate a design for vulnerability to Trojan insertion. This analysis has been performed at the RTL, gate and layout levels. In each case, vulnerabilities such as rare events, transition probabilities, white spaces etc. have been extracted and subsequently exploited to insert the hardware Trojan circuitry as part of the trust benchmarks.

5.1 Vulnerability Analysis at the RTL Level

Due to the increased use of 3PIP in designs, it is also necessary to conduct vulnerability analysis at the behavioral or RTL level. Given a RTL IP, we need to carefully analyze statements that are rarely activated and check if certain signals are propagated to the primary outputs. A malicious 3PIP can use such features to create a behavioral Trojan that is difficult to trigger and whose effects cannot be observed at the outputs on regular functional testing. In order to quantify such vulnerabilities, we have proposed two broad metrics [20]:

- **Statement Hardness:** This metric analyzes the rarity of the conditions under which a statement executes in RTL code. This metric is quantified by $(\frac{U-L+1}{U_0-L_0+1})^{-1}$, where U, L are the upper and lower limits of the value range for a control signal (set by a conditional statement such as IF) and U_0, L_0 are the declared upper and lower limits of the value range for the same control signal (set by an assignment/declaration statement). This indicates that statements/control signals that are highly nested by multiple conditional statements are harder to execute and are the most likely targets for Trojan insertion at the behavioral level.
- **Observability:** This metric denotes the ease with which a signal can propagate to the primary output of a design. In order to quantify this metric, a data graph is constructed from the RTL code. Each node in the graph represents a control signal in the RTL code, and the edges represent the flow of data in the code. Each signal (with respect to a destination node) is assigned a '0′ for observability if its destination node is not a primary output. If a signal is considered with respect to a destination node that serves as a primary output, it is assigned a observability figure equal to the sum of the weight of the assignment statements where the signal is assigned to the destination node.

Using these two metrics, the *b*01 benchmark from the ITC'99 benchmark set was analyzed. The maximum statement hardness for this benchmark was 64, with a maximum observability of 0.5. In contrast, the *b*05 benchmark, a

Table 2 Trust benchmark details

Category	Trojans		Main Circuits
	Trojan Type	# of Trust Benchmarks	
Insertion Phase	Specification	0	–
	Design	80	AES, BasicRSA, EthernetMAC10GE, MC8051, PIC16F84, RS232, s15850, s35932, s38417, s38584, vga_lcd, wb_conmax
	Fabrication	8	EthernetMAC10GE, MultPyramid
	Testing	0	–
	Assembly and Package	0	–
Abstraction Level	System	0	–
	Development Environment	0	–
	Register Transfer	51	AES, b19, BasicRSA, MC8051, PIC16F84, RS232, wb_conmax
	Gate	25	b19, EthernetMAC10GE, RS232, s15850, s35932, s38417, s38584, vga_lcd, wb_conmax
	Layout	12	EthernetMAC10GE, MultPyramid, RS232
	Physical	0	–
Activation Mechanism	Always On	11	AES-T100, EthernetMAC10GE, MultPyramid
	Triggered	79	AES, b19, BasicRSA, EthernetMAC10GE,MC8051, MultPyramid, PIC16F84l, RS232,s15850, s35932, s38417, s38584, vga_lcd,wb_conmax
Effect	Change Functionality	35	b19, EthernetMAC10GE, MC8051, RS232, s15850, s35932, s38417, s38584, vga_lcd,wb_conmax
	Degrade Performance	3	EthernetMAC10GE, MultPyramid, s35932
	Leak Information	24	AES, BasicRSA, PIC16F84, s35932, s38584
	Denial of Service	34	AES, BasicRSA, EthernetMAC10GE, MC8051, MultPyramid, PIC16F84, RS232, s15850, s35932, s38417, s38584, vga_lcd, wb_conmax
Location	Processor	51	AES, b19, BasicRSA, MC8051, MultPyramid, PIC16F84, s15850, s35932, s38417, s38584, vga_lcd
	Memory	0	–
	I/O	4	MC8051, wb_conmax
	Power Supply	2	MC8051-T300, wb_conmax
	Clock Grid	2	EthernetMAC10GE
Physical Characteristics	Distribution	2	b19
	Size	0	–
	Type	86	AES, b19, BasicRSA, EthernetMAC10GE, MC8051, MultPyramid, PIC16F84, RS232,s15850, s35932, s38417, s38584, vga_lcd,wb_conmax
	Structure	8	b19, EthernetMAC10GE, MultPyramid
Total	NA	88	NA

considerably large benchmark, has a statement hardness of 4.4×10^5 with a observability of 0. This indicates that compared to the *b*01 benchmark, the *b*05 benchmark has heavily nested statements along with primary inputs that do not propagate all the way to the primary output, making it more vulnerable to Trojan insertion.

5.2 Vulnerability Analysis at the Netlist Level

Functional hardware Trojans are realized by adding or removing gates; therefore, the inclusion of Trojan gates or the elimination of circuit gates affects circuit side-channel signals such as power consumption and delay

characteristics, as well as the functionality. To minimize a Trojan's contribution to the circuit side-channel signals, an adversary can exploit hard-to-detect areas (e.g. nets) to implement the Trojan. Hard-to-detect areas are defined as areas in a circuit not testable by well-known fault-testing techniques (stuck-at, transition delay, path delay, and bridging faults) or having negligible impact on the circuit side-channel signals. We propose a vulnerability analysis flow to identify such hard-to-detect areas in a circuit. These areas provide opportunities to insert hard-to-detect Trojans and invite researchers to develop techniques to make it difficult for an adversary to insert Trojans.

As Fig. 3 shows, our proposed vulnerability analysis flow performs power, delay, and structural analyses on a circuit to extract the hard-to-detect areas. Any transition inside a Trojan circuit increases the overall transient power consumption; therefore, it is expected that Trojan inputs or triggers will be supplied by nets with low transition probabilities to reduce activity inside the Trojan circuit.

The *Power Analysis* step in Fig. 3 is based on analyzing switching activity; it determines the transition probability of every net in the circuit assuming the probability of 0.5 for '0' or '1' at primary inputs and at memory cells' outputs. More details regarding the transition probability calculation can be found in [26]. Then, nets with transition probabilities below a certain threshold are considered as possible Trojan inputs. The *Delay Analysis* step performs path delay measurement based on gates' capacitance. This allows us to measure the additional delay induced by the Trojan, by knowing the added capacitance to circuit paths. The Delay Analysis step identifies nets on non-critical paths as they are more susceptible to 'smart' Trojan insertion, which would not change the circuit delay. To further reduce Trojan impact on circuit delay characteristics, the delay analysis tool also reports the paths to which a net belongs to avoid selecting nets belonging to different sections of the same path. The *Structural Analysis* step executes the structural transition delay fault testing to find untestable blocked and untestable redundant nets. Untestable redundant nets are not testable because they are masked by a redundant logic, and they are not observable through the primary outputs or scan cells. Untestable blocked nets are not controllable or observable by untestable redundant nets. Creating Trojan inputs/triggers using these untestable nets hides the Trojan impact on delay variations.

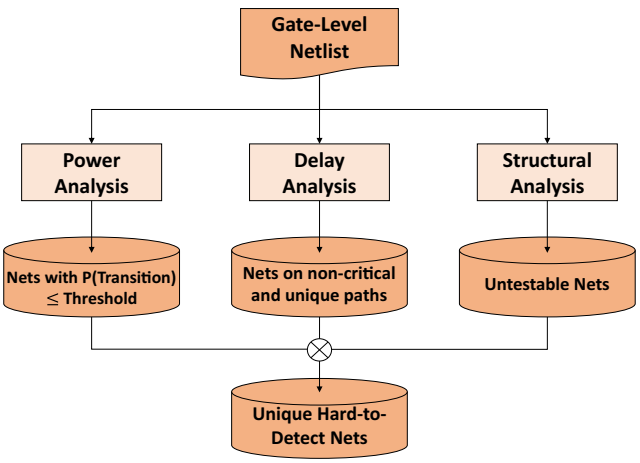

Fig. 3 The vulnerability analysis flow

At the end, the vulnerability analysis flow reports a list of unique hard-to-detect nets in a circuit. This list includes untestable nets with low transition probabilities and also those nets with low transition probabilities on unique non-critical paths. Note that when a Trojan impacts more than one path, it provides greater opportunities for detection. Using unique paths and avoiding shared ones make a Trojan's contribution to the affected paths' delay minimal. This means that the Trojan impact on delay could be masked by process variations. The reported nets are also ensured to be untestable by structural test patterns used in production tests. They also have low transition probabilities so that the Trojans will negligibly affect circuit power consumption. As the nets are chosen from non-critical paths without any shared segments, it would also be extremely difficult to detect Trojans by practical delay-based techniques.

The vulnerability analysis flow can be implemented using most electronic design automation (EDA) tools, and the complexity of the analysis is linear with respect to the number of nets in the circuit. We have applied the flow to the Ethernet MAC 10GE circuit from http://opencores.org, which implements 10Gbps Ethernet Media Access Control functions. Synthesized at 90nm Synopsys technology node, the Ethernet MAC 10GE circuit consists of 102,047 components, including 21,830 flip-flops. The Power Analysis shows that out of 102,669 nets in the circuit, 23,783 of them have a transition probability smaller than 0.1, 7003 of them smaller than 0.01, 367 of them smaller than 0.001, and 99 of them smaller than 0.0001. The Delay Analysis indicates that the largest capacitance along a path, representing path delay, in the circuit is 0.06572 pF, and there are 14,927 paths in the circuit whose path capacitance are smaller than 70% of the largest capacitance, assuming that paths longer than 70% in a circuit can be tested using testers. The Structural Analysis finds that there is not a single untestable fault in the circuit. By excluding nets sharing different segments of one path, there are 494 nets in the Ethernet MAC 10GE circuit considered to be areas where Trojan inputs could be used while ensuring the high difficulty of detection based on side-channel and functional test techniques.

5.2.1 Creating the Trojans

We have also created a separate flow that can create and validate hardware Trojans, given a flattened netlist of a circuit

design [9]. In order to create these Trojans, the flow first applies a random set of test vectors to the design, from which a list of rare nodes are identified. These rare nodes are extracted by calculating the signal probabilities at each node and picking from those nodes whose probabilities are less than a pre-defined threshold (e.g., 0.0001). A hardware Trojan is then created by randomly sampling from these rare nodes. For example, if a 2-trigger hardware Trojan is required, two nodes from the list of rare nodes are randomly picked and an instance of a hardware Trojan is created in a separate file. The same process can be repeated for creating a list of Trojans with an arbitrary number of triggers and also, an arbitrary number of Trojans. Once these Trojans are created, they are functionally validated using Synopsys Tetramax. In this step, we make sure that the Trojans we have created are in fact triggered by selected test patterns (even though those patterns may be extremely rare, which is actually desirable for a Trojan). This step is necessary because an adversary will never insert a functional Trojan that will never get triggered. After validation, the Trojans can be manually inserted into the victim netlist and linked to a desirable payload that can cause malicious alternations to the design.

5.3 Vulnerability Analysis at the Layout Level

In addition to performing vulnerability analysis at the netlist level, we have also implemented a unique flow to identify Trojan-insertion prone areas in a circuit layout [37]. Such a vulnerability analysis flow at the layout level is critical as an untrusted foundry will likely launch a Trojan-insertion attack at the layout level after it gains the entire design from a system integrator. A malicious foundry will likely look for empty regions or 'whitespaces' in a circuit layout, with available routing channels in metal layers above the empty regions. Such whitespaces are very common in many designs today. For example, we have analyzed the ITC99 b15 benchmark layout, which consists of 3296 cells and evaluated the amount of available white space in the design. Figure 4 shows the distribution of whitespace in the benchmark, in units of INVX0, which is the smallest gate in the b15 benchmark layout. The entire layout was divided into grids, with the grid area equal to W^2, where W is the width of the largest cell in the synthesized design library. The layout example shows that there is considerable amount of white spaces in areas closer to the layout boundaries, which averages to about 41.41 units of INVXO. Since such whitespaces are inevitable in any layout design, the threat of Trojan insertion is very realistic as a Trojan size could be as small as a few gates. Further, along with the whitespace, we have also analyzed the average routing channels available above the whitespaces, which might be required to complete the Trojan design. Out analysis shows that for a 9-metal layer implementation, the b15 benchmark layout has 0.84 average unused routing channels available per unit grid.

However, it should be noted that the mere presence of whitespace and routing channels is not a vulnerability that a smart adversary will exploit. In order to bypass detection techniques, a malicious party, whether it is an untrusted foundry with the complete layout or a malicious insider in the design house, will make sure that he/she designs a 'stealthy' Trojan. In order to do this, a vulnerability analysis similar to the one we described in Section 5 must be performed in conjunction with the white space/available routing channel analysis we described above. The first step in doing so is to identify the non-critical paths in the circuit netlist and match those nets to the paths in the layout. A quick analysis of the b15 benchmark circuit shows that, on average, there are about 17 nets per grid whose delay is less than 75% of the critical path of the circuit. Such nets can potentially be exploited to insert a Trojan, as the Trojan-induced capacitance (due to the added wiring connections) and the resulting delay can be evaded from Trojan detection techniques based on path delay. The second step is to insert the Trojan in such a location that its trigger is attached to nets with very low transition probabilities, so that the probability of Trojan activation gets reduced. Figure 5 shows the distribution of nets whose transition probability is less than 10^{-4}, and the delay is less than 75% of the critical path delay, for the b15 benchmark. Clearly, such nets at the layout level, which are also near to layout regions with sufficient amount of whitespace, indicates the vulnerability of the circuit to layout-level Trojan insertion. For example, the nets around row 8 and column 6 in the layout grid in Figs. 4 and 5 have > 10 nets available for implementing a Trojan and its trigger.

6 Trust Benchmark Evaluation

6.1 Trojan Evaluation Suite

Hardware Trojans have a stealthy nature; they rarely activate and make limited contributions to circuit characteristics. To ensure that each Trojan inserted in a trust benchmark does not get activated by test patterns used in production tests, we develop an automatic Trojan Evaluation Suite (TES) to investigate the effectiveness of different types of test patterns at detecting hardware Trojans. TES synthesizes a circuit, generates structural test patterns, applies the patterns to the circuit, and monitors circuit switching activity, including transitions inside Trojan circuits. The flow is implemented using Synopsys tools with additional in-house ones, though it is also possible to develop the same flow with other commercial tools.

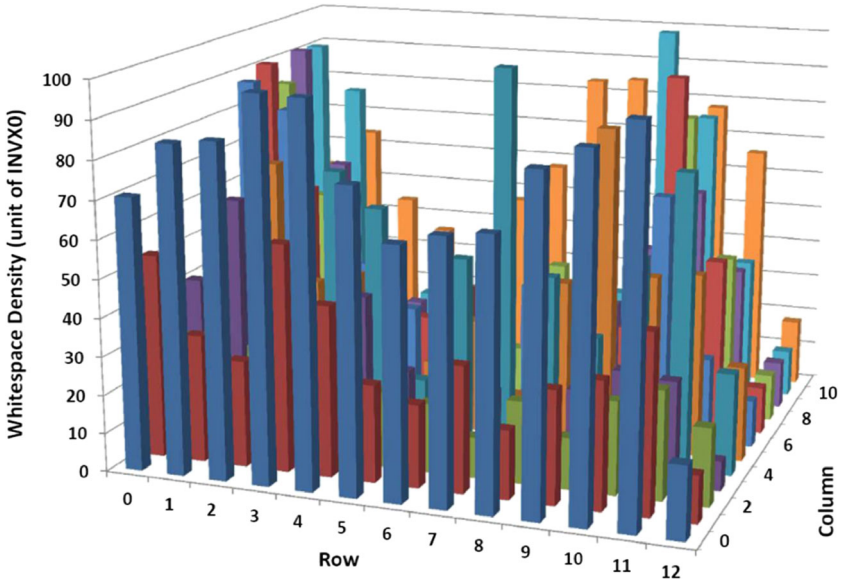

Fig. 4 Distribution of whitespace across the b15 layout

As shown in Fig. 6, the TES flow consists of two main steps: (1) pattern generation and (2) pattern evaluation. To generate structural test patterns, a circuit is first synthesized by the Synopsys Design Compiler. The synthesized netlist is passed to the Synopsys Automatic Test Pattern Generation (ATPG) tool, TetraMax, to generate structural stuck-at-fault (SAF), transition-delay-fault (TDF), and path-delay-fault (PDF) fault test patterns. To generate path-delay fault test patterns, the synthesized netlist is also passed to Synopsys PrimeTime to obtain circuit timing information and critical paths.

To automatically apply patterns and observe switching activity in the main circuit and Trojan circuits during the pattern evaluation step, we develop a program in the Synopsys Verilog Compiler Simulator (VCS) by using Programming Language Interface (PLI) routines. The program fetches patterns generated by TetraMax and applies them to the circuit. Every transition on every net in the circuit is recorded during the simulation, and the test application also determines whether the Trojan is ever activated.

A Trojan can be inserted into a circuit before or after synthesizing the circuit. TES makes the detailed analysis of Trojan circuits possible, and any transition in the Trojan circuit can be recorded. A combinational comparator Trojan, shown in the example in Fig. 7, was inserted into the Ethernet circuit. The Trojan trigger sought a specific 16-bit vector. Whenever the Trojan was triggered, its payload gained control over an internal net. 5218 random functional vectors were applied. Simulation took 104386 ns, and in total there were 106,664,486 transitions in the circuit and

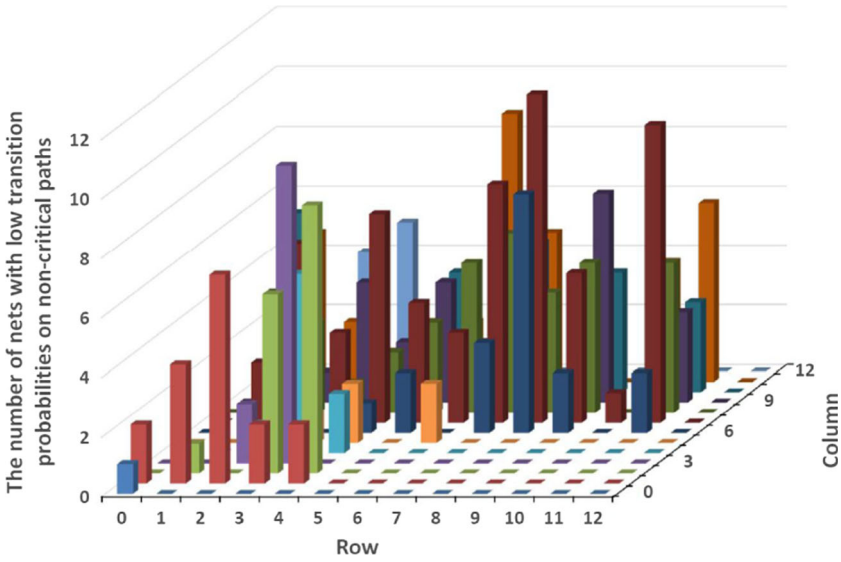

Fig. 5 Distribution of nets with low transition probability and minimal impact on critical path delay in the b15 layout

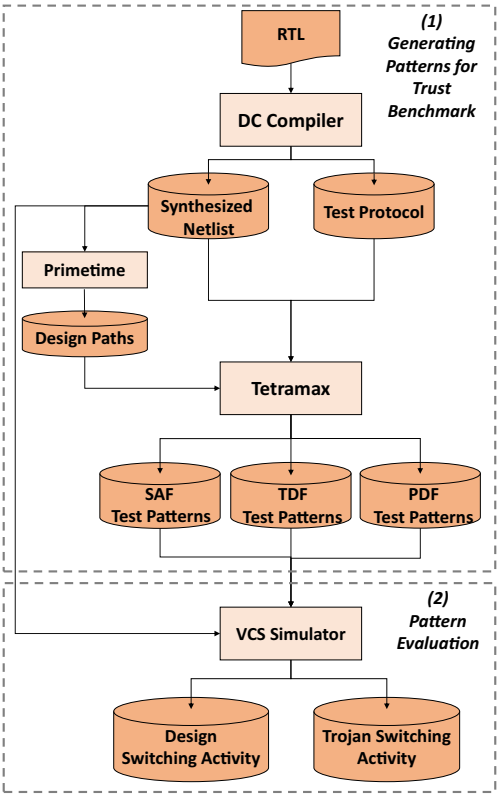

Fig. 6 The Trojan evaluation suite (TES)

4,229 transitions inside the Trojan circuit, though the Trojan never became fully activated (i.e., Trojan's payload never changed the circuit net value).

Without having any knowledge about Trojan insertion, it is possible to use TES to evaluate the effectiveness of a test pattern in generating switching on nets with low transition probability. This information can be used to reduce authentication time by selecting test vectors which create maximum switching on hard-to-detect nets.

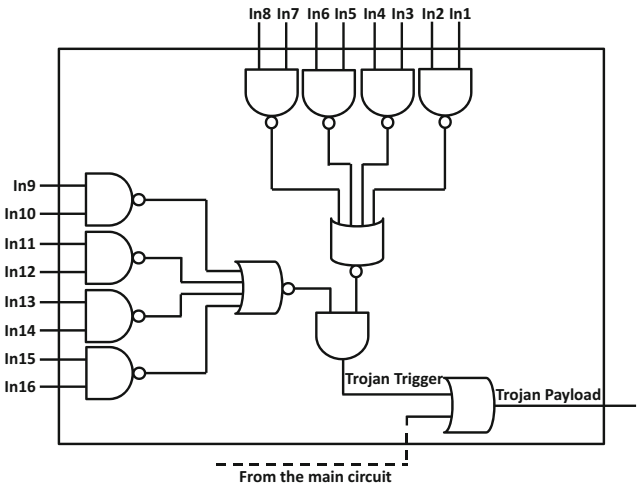

Fig. 7 An example comparator Trojan

6.2 Trojan Detectability

A Trojan's impact on circuit characteristics depends on its implementation. Trojan inputs tapped from nets with higher transition probabilities will aggrandize switching activity inside the Trojan circuit and increase its contribution to circuit power consumption. Furthermore, the Trojan might affect circuit delay characteristics due to additional capacitance induced by extra routing and Trojan gates. To quantitatively determine the difficulty of detecting a gate-level Trojan, a procedure is developed to determine Trojan detectability based on a Trojan's impact on delay and power side-channels across different circuits.

The proposed Trojan detectability metric ($T_{detectability}$) is determined by (1) the number of transitions in the Trojan circuit and (2) extra capacitance induced by Trojan gates and their routing. This metric is designed to be forward-compatible with new approaches for Trojan detection by introducing a new variable, for example a quantity related to the electromagnetic field.

Transitions in a Trojan circuit reflect Trojan contribution to circuit power consumption, and Trojan impact on circuit delay characteristic is represented by measuring the added capacitance by the Trojan. Assuming A_{Trojan} represents the number of transitions in the Trojan circuit, S_{Trojan} the Trojan circuit size in terms of the number of cells, A_{TjFree} the number of transitions in the Trojan-free circuit, S_{TjFree} the Trojan-free circuit size in terms of the number of cells, TIC the added capacitance by Trojan as Trojan-induced capacitance, and C_{TjFree} the Trojan-affected path with the largest capacitance in the corresponding Trojan-free circuit, Trojan detectability ($T_{Detectability}$) at the gate-level is defined as

$$T_{Detectability} = |t| \tag{1}$$

where

$$t = \left(\frac{A_{Trojan}/S_{Trojan}}{A_{TjFree}/S_{TjFree}}, \frac{TIC}{C_{TjFree}} \right) \tag{2}$$

$T_{Detectability}$ at the gate-level is calculated as follows:

1. Apply random inputs to a Trojan-free circuit and obtain the no. of transitions in the circuit (A_{TjFree}).
2. Apply the same random vectors to the circuit with a Trojan and obtain the number of transitions in the Trojan circuit (A_{Trojan}).
3. Perform the Delay analysis on the Trojan-free and Trojan-inserted circuits.
4. Obtain the list of paths whose capacitance are changed by the Trojan.
5. Determine the Trojan-affected path with the largest capacitance in corresponding Trojan-free (C_{TjFree}) and the added capacitance (TIC).
6. Form the vector t (2) and compute $T_{Detectability}$ (the absolute value/modulus of the vector t) as defined in

Eq. (1). Note that Trojan detectability represents the difficulty of detecting a Trojan.

As an example, the comparator Trojan, shown in Fig. 7, is inserted at four different locations, namely TjG-Loc1, TjG-Loc2, TjG-Loc3, and TjG-Loc4 (G represents "gate level"), in the Ethernet MAC 10GE circuit, and Table 3 shows their detectability. The Ethernet MAC 10GE circuit consists of 102047 cells, Column 3 S_{TjFree}, while the Trojan size with 12 cells, Column 5 S_{Trojan}, is only about 0.011% of the entire circuit. TjG-Loc4, in Row 5, experiences the largest switching activity (13484 in Column 4) and relatively induces high TIC (0.00493 pF in Column 6). It is expected that TjG-Loc4 will be the easiest Trojan to be detected due to more impact on circuit side-channel signals, and in turn the detectability of TjG-Loc4 ($T_{Detectability} = 1.07911$ in Column 8) is higher than the others. Although the induced capacitance by TjG-Loc2 (0.00497 pF), in Row 3, is more than the capacitance induced by TjG-Loc1 (0.00029 pF), in Row 2, TjG-Loc1 has more significant contribution into circuit switching activity, 10682 versus 4229 in Column 4. Therefore, TjG-Loc1 has the second largest detectability (0.85166) after TjG-Loc4. Among TjG-Loc2 and TjG-Loc3, although TjG-Loc3, in Row 4, has slightly larger induced capacitance (0.00501 pF), TjG-Loc2 experiences more switching activity (4229 versus 3598 in Column 4). The two Trojans have close detectability where TjG-Loc2 stands above and TjG-Loc3 remains the hardest Trojan to be detected with the lowest Trojan detectability.

6.3 Vulnerability and Detectability

Using the vulnerability analysis flow and the detectability metric proposed in Section 6.2, we have evaluated the vulnerability of the benchmark circuits to Trojan insertion and have presented detectability results for Trojans inserted into these vulnerable circuits. Tables 4 and 5 show detailed analysis of a selected number of gate-level benchmarks. In Table 5, Column 3 indicates that b19 circuit, in Row 2, is the largest circuit in size (62835) among the selected circuits. Table 4 also shows the number of nets with transition probability less than 0.0001 in b19, 4530 in Row 2 and Column 6, is larger than the other circuits, and b19 has considerable number of paths whose capacitances are less than 70% of its critical path' capacitance, 474358 in Column 8. Further, there are eight untestable faults is b19, in Column 9. These provide significant opportunity for implanting Trojans resilient against power and delay side-channel analyses in b19. Table 5 confirms that b19-T100 with $T_{Detectability} = 0.02498$, in Column 8, is the second most difficult Trojan to detect as no transition inside the Trojan is observed, 0 in Column 4, and it induces a small capacitance, 0.00095 pF in Column 6, on a non-critical path, 0.03785 pF in Column 7. s38584-T200, in Row 7, has the lowest detectability, 0.01390 in Column 8; similar to b19-T100, there is no switching activity in s38584-T200, 0 in Column 4, and s38584-T200 induces less capacitance, 0.00041 pF in Column 6, on a shorter path, 0.02984 pF in Column 7, compared to b19-T100. We can also note that trust benchmark $s35854 - T300$ has high detectability (2.84578), as the Trojan gate count is fairly high (731) and the resulting Trojan induced capacitance (TIC) is also high. This makes the Trojan easier to detect through delay side-channels. To summarize, Fig. 8 shows the $T_{detectability}$ metric figures for all the gate-level benchmarks on Trust-Hub, which show varying levels of detection difficulty, based on the aforementioned metric.

6.4 Benchmarks and Attack Models

As we discussed in Section 1, the threat of hardware Trojans can be explained in the context of various adversarial/attack models. In this section, we highlight how each benchmark that we have developed can fit a particular attack model. This is necessary as the motivation behind any Trojan detection/prevention technique cannot be fully understood without specifying its context/scope in terms of an attack model. Below, we present our methodology for classifying the trust benchmarks in the form of a case study on a benchmark. Additionally, we also point out what changes/additions can be made to a benchmark in order to fit it to a particular attack model.

EthernetMAC10GE-T200 The benchmark only comes with the .def (Layout) file and does not include a golden layout. However, there is a golden netlist available. We

Table 3 The detectability of the comparator Trojan placed at four different locations in Ethernet MAC 10GE circuit

Trojan	A_{TjFree}	S_{TjFree}	A_{Trojan}	S_{Trojan}	$TIC(pF)$	$C_{TjFree}(pF)$	$T_{Detectability}$
TjG-Loc1	106,664,486	102,047	10,682	12	0.00029	0.04136	0.85166
TjG-Loc2	106,664,486	102,047	4,229	12	0.00497	0.07211	0.34413
TjG-Loc3	106,664,486	102,047	3,598	12	0.00501	0.04969	0.30403
TjG-Loc4	106,664,486	102,047	13,484	12	0.00493	0.05260	1.07911

Table 4 Design vulnerability analysis of a selected number of Trojan-free circuits

Circuit	Power Analysis					Delay Analysis		Structural Analysis
	# Nets	< 0.1	< 0.01	< 0.001	< 0.0001	Critical path Capacitance(pF)	$< 70\%$ of Critical Path Capacitance	# Untestable faults
b19	70,259	14,482	8,389	5,533	4,530	0.37724	474,358	8
s38417	5,669	589	291	219	69	0.05015	41,901	0
s38584	7,203	817	197	85	30	0.04467	27,689	0
s35932	6,269	0	0	0	0	0.00851	3,156	0

can classify this particular benchmark with different attack models as follows:

- A: No (This benchmark lacks any RTL level description, and going from a gate-level description to a behavioral RTL description is non-trivial. Thus, it does not fit attack model A.)
- B: Possible to adapt (The golden netlist can be used to generate a golden layout, which is required in attack model B, since a design house will always have a golden layout if only the foundry is untrusted.)
- C: Possible to adapt (A separate Trojan can be inserted into the golden netlist at the gate-level, and a layout can be generated. However, the original Trojan-inserted layout would not be needed to fit the benchmark to this attack model.)
- D: No (The RTL level description is not available, due to which a Trojan insertion at the RTL level, emulating an untrusted 3PIP vendor, cannot be performed.)
- E: No (Same as above, due to lack of RTL level description)
- F: No (Same as above.)
- G: Possible to adapt (In order to fit this attack model, a gate-level Trojan can be inserted into the golden netlist and converted to a layout. From the T200 benchmark, the particular Trojan can be observed and inserted into the newly generated layout, thereby emulating both an untrusted SoC developer and an untrusted foundry.)

A count of the number of trust benchmarks which are available on the Trust-Hub website and fit the seven attack models, is shown in Fig. 9. Clearly, most of the benchmarks can be further expanded to emulate the scenario where multiple entities in the supply chain are untrusted.

7 Future Work

Although we have developed a total of 91 trust benchmarks and the corresponding tools, there is still room for more improvement. As part of future work, we plan to explore the following scenarios and develop appropriate solutions.

- **Including additional trust benchmarks:** The set of benchmarks we have developed are only a small set of possible hardware Trojans that can be designed. We plan to update/add more Trojans and trust benchmarks to the Trust-Hub website as part of our ongoing work.
- **Fitting new attack models:** As seen from Fig. 9, there are already a large number of trust benchmarks that can be adapted to fit attack models E, F and G, which would involve inserting Trojan(s) at two or more levels of abstraction into a single benchmark circuit. For example, a design starting at the RTL level can be inserted with Trojan 1 and synthesized to a netlist. After layout generation from the netlist, Trojan 2 can then be inserted at the layout level. This would capture

Table 5 The detectability ($T_{Detectability}$) of a selected number of gate-level Trojans inserted in the circuits in Table 4

Trojan	A_{TjFree}	S_{TjFree}	A_{Trojan}	S_{Trojan}	$TIC(pF)$	$C_{TjFree}(pF)$	$T_{Detectability}$
b19-T100	4,037,383	62,835	0	83	0.00095	0.03785	0.02498
s38417-T100	2,717,682	5,329	59	11	0.00417	0.03234	0.13939
s38417-T200	2,717,682	5,329	1,328	11	0.00531	0.03052	0.41085
s38417-T300	2,717,682	5,329	257	15	0.00046	0.03078	0.04847
s38584-T100	423,986	6,473	705	9	0.00095	0.01504	1.25877
s38584-T200	423,986	6,473	0	83	0.00041	0.02984	0.01390
s38584-T300	423,986	6,473	16	731	0.01246	0.00438	2.84578
s35932-T100	353,304	5,426	354	15	0.00049	0.00699	0.86644
s35932-T200	353,304	5,426	733	12	0.00316	0.00973	1.26280
s35932-T300	353,304	5,426	738	36	0.00050	0.00823	0.37533

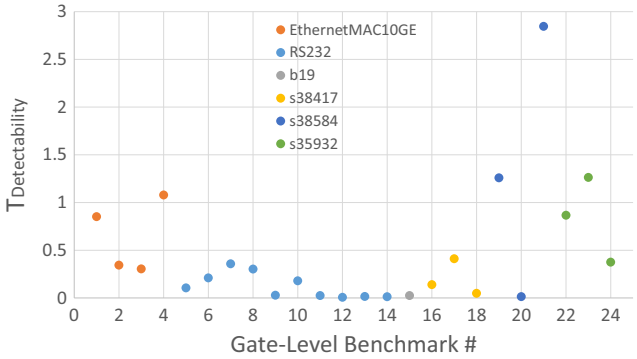

Fig. 8 Detectability metric for gate-level Trojans in trust benchmarks

the essence of attack model F (fabless SoC design house).

- **Larger benchmarks:** Most of the benchmarks currently available are fairly small in size and might not capture the Trojan detection problem in the context of real-life industrial designs, at which point some detection strategies might become completely infeasible. Thus, it would make sense to develop larger trust benchmarks.
- **SoC scenario:** We also plan to put up SoC designs that combine various IPs (at different abstraction levels) into one single design. This emulates a scenario where multiple IP vendors are untrusted, and is representative of the SoC development process today.
- **Dynamic trust benchmark generation:** While the Trojans we have created are inserted only into specific benchmark circuits, Trojan benchmarks, which are arbitrary Trojan circuits (at the RTL, gate or layout level) with an arbitrary number of gates/cells, should ideally be applicable for insertion into any given circuit. The Trojan benchmarks we have developed have been manually inserted into benchmark circuits after performing the vulnerability analysis flow. In order to create dynamic trust benchmarks, we require automatic payload and trigger identification of a circuit at different levels (RTL, gate and layout). While trigger identification is straight-forward (as we have shown in Section 5), automating payload identification for any arbitrary circuit is a non-trivial problem, which must be addressed. Nonetheless, the detectability metric we have proposed would still be useful in characterizing the Trojan benchmarks after they have been inserted into a circuit.
- **Testbench development:** In order to help in the use of the benchmarks, we are constantly adding and refining testbench programs that can help researchers to run the benchmark with or without the Trojan and exactly activate it to observe the effects on delay, power and other side-channels.
- **FPGA device Trojans:** In Section 2.4, we delved into hardware Trojans that can be inserted into the FPGA fabric, while being dependent or independent of the IP on the FPGA. Currently, the FPGA-implementable trust benchmarks on Trust-Hub only cover the threat of an untrusted IP vendor, where the Trojan is a part of the IP. Trojans that are part of the FPGA device (e.g. DCM corruptors, IP leak), which would need to be implemented through the FPGA firmware, is part of our future work.

8 Conclusion

The hardware trust community needs common ground to more effectively address the Trojan detection problem. As no standard measurements, benchmarks, or tools have previously been developed, we have put our effort into developing tools that can aid us in generating trust benchmarks.

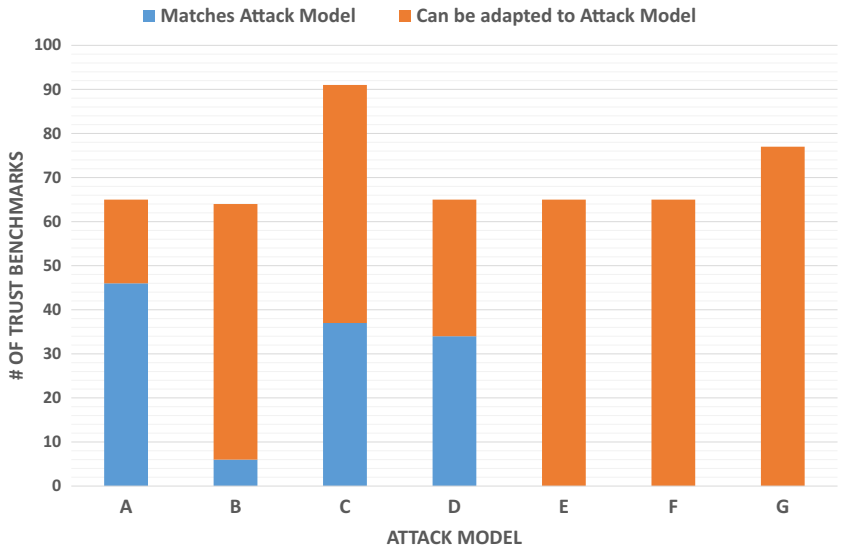

Fig. 9 Classifying the number of trust benchmarks that fit different attack models

As part of this toolset, the vulnerability analysis flow determines areas in a circuit that are more probable to be used for Trojan implementation. Further, we developed an automatic Trojan evaluation suite to measure the resiliency of hardware Trojans. Then, we defined the Trojan detectability metric to quantify Trojan impact on circuit power consumption and circuit performance, thereby giving us an idea of its stealthiness. Using these tools and metrics, we generated a large number of trust benchmarks, which are available at trust-hub.org for researchers to evaluate their detection techniques. These benchmarks are now currently being used in developing the state-of-the-art in hardware Trojan detection techniques [12, 21, 25, 28, 38, 39]. As part of our future work, we plan to expand the current benchmarks to fit more attack models, develop Trojan benchmarks and also, add newer benchmarks to the repository to enhance outcomes of Trojan detection research.

Acknowledgments This work was supported in part by the National Science Foundation (NSF) under grant 1513239.

References

1. Marinissen E, Iyengar V, Chakrabarty K (2002) A set of benchmarks for modular testing of socs. In: Proceedings International Test Conference, 2002, pp 519–528
2. Brglez F (1985) A neutral netlist of 10 combinational benchmark circuits and a target translation in fortran. In: ISCAS-85
3. Brglez F, Bryan D, Kozminski K (1989) Combinational profiles of sequential benchmark circuits. In: IEEE international symposium on circuits and Systems, 1989, vol 3
4. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: Proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture, ser. MICRO 30. IEEE Computer Society, Washington, DC, pp 330–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=266800.266832
5. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) Mibench: a free, commercially representative embedded benchmark suite. In: 2001 IEEE international workshop on workload characterization, 2001. WWC-4, pp 3–14
6. Salmani H, Tehranipoor M, Karri R (2013) On design vulnerability analysis and trust benchmarks development. In: 2013 IEEE 31st international conference on computer design (ICCD), pp 471–474
7. Tehranipoor M, Koushanfar F (2013) A survey of hardware trojan taxonomy and detection. IEEE Design Test 99:1–1
8. Xiao K, Forte D, Jin Y, Karri R, Bhunia S, Tehranipoor M (2016) Hardware Trojans: Lessons learned after one decade of research. ACM transactions on design automation of electronic system (To Appear)
9. Chakraborty RS, Wolff F, Paul S, Papachristou C, Bhunia S (2009) Mero: A statistical approach for hardware trojan detection. In: Proceedings of the 11th international workshop on cryptographic hardware and embedded systems, ser. CHES '09. Springer, Berlin, pp 396–410. [Online]. Available: doi:10.1007/978-3-642-04138-9_28
10. Banga M, Hsiao MS (2010) Trusted rtl: Trojan detection methodology in pre-silicon designs. In: 2010 IEEE international symposium on hardware-oriented security and trust (HOST), pp 56–59
11. Banga M, Hsiao M (2009) A novel sustained vector technique for the detection of hardware trojans. In: 2009 22nd international conference on VLSI design, pp 327–332
12. Waksman A, Suozzo M, Sethumadhavan S (2013) Fanci: identification of stealthy malicious logic using boolean functional analysis. In Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, ser. CCS '13. ACM, New York, pp 697–708. [Online]. Available: doi:10.1145/2508859.2516654
13. Xiao K, Zhang X, Tehranipoor M (2013) A clock sweeping technique for detecting hardware trojans impacting circuits delay. IEEE Design Test 30(2):26–34
14. Wang X, Salmani H, Tehranipoor M, Plusquellic J (2008) Hardware trojan detection and isolation using current integration and localized current analysis. In: 2008 IEEE international symposium on defect and fault tolerance of VLSI systems, pp 87–95
15. Narasimhan S, Du D, Chakraborty RS, Paul S, Wolff F, Papachristou C, Roy K, Bhunia S (2010) Multiple-parameter side-channel analysis: a non-invasive hardware trojan detection approach. In: 2010 IEEE international symposium on hardware-oriented security and trust (HOST), pp 13–18
16. Zhang X, Tehranipoor M (2011) Ron: an on-chip ring oscillator network for hardware trojan detection. In: 2011 Design, automation test in Europe, pp 1–6
17. Hu K, Nowroz AN, Reda S, Koushanfar F (2013) High-sensitivity hardware trojan detection using multimodal characterization. In: Design, automation test in europe conference exhibition (DATE), 2013, pp 1271–1276
18. Stellari F, Song P, Weger AJ, Culp J, Herbert A, Pfeiffer D (2014) Verification of untrusted chips using trusted layout and emission measurements. In: 2014 IEEE international symposium on hardware-oriented security and trust (HOST), pp 19–24
19. Li J, Lach J (2008) At-speed delay characterization for ic authentication and trojan horse detection. In: IEEE international workshop on hardware-oriented security and trust, 2008. HOST 2008, pp 8–14
20. Salmani H, Tehranipoor M (2013) Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level. In: 2013 IEEE international symposium on defect and fault tolerance in vlsi and nanotechnology systems (DFT), pp 190–195
21. Zhang X, Tehranipoor M (2011) Case study: detecting hardware trojans in third-party digital ip cores. In: 2011 IEEE international symposium on hardware-oriented security and trust (HOST), pp 67–70
22. Love E, Jin Y, Makris Y (2012) Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. IEEE Transactions on Information Forensics and Security 7(1):25–40
23. Love E, Jin Y, Makris YG (2011) Enhancing security via provably trustworthy hardware intellectual property. In: 2011 IEEE international symposium on hardware-oriented security and trust, pp 12–17
24. 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, ser. DAC '15, ACM, New York. [Online]. Available: doi:10.1145/2744769.2747939
25. Rajendran J., Vedula V, Karri R (2015) Detecting malicious modifications of data in third-party intellectual property cores. In: 2015 52nd ACM/EDAC/IEEE design automation conference (DAC), pp 1–6
26. Salmani H, Tehranipoor M, Plusquellic J (2012) A novel technique for improving hardware trojan detection and reducing trojan activation time. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 20(1):112–125

27. Salmani H, Tehranipoor M (2012) Layout-aware switching activity localization to enhance hardware trojan detection. IEEE Transactions on Information Forensics and Security 7(1):76–87
28. Forte D, Bao C, Srivastava A (2013) Temperature tracking: an innovative run-time approach for hardware trojan detection. In: Proceedings of the international conference on computer-aided design, ser. ICCAD '13. IEEE Press, Piscataway, pp 532–539. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561931
29. Rajendran J, Pino Y, Sinanoglu O, Karri R (2012) Logic encryption: A fault analysis perspective. In: Proceedings of the conference on design, automation and test in europe, ser. DATE '12. EDA Consortium, San Jose, pp 953–958. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492708.2492947
30. Chakraborty RS, Bhunia S (2009) Security against hardware trojan through a novel application of design obfuscation. In: Proceedings of the 2009 international conference on computer-aided design, ser. ICCAD '09. ACM, New York, pp 113–116. [Online]. Available: doi:10.1145/1687399.1687424
31. Chakraborty R, Bhunia S (2009) Harpoon: an obfuscation-based soc design methodology for hardware protection. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28(10):1493–1502
32. Xiao K, Tehranipoor M (2013) Bisa: built-in self-authentication for preventing hardware trojan insertion. In: 2013 IEEE international symposium on hardware-oriented security and trust (HOST), pp 45–50
33. Rajendran J, Sam M, Sinanoglu O, Karri R (2013) Security analysis of integrated circuit camouflaging. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, ser. CCS '13. ACM, New York, pp pp 709–720. [Online]. Available: doi:10.1145/2508859.2516656
34. Imeson F, Emtenan A, Garg S, Tripunitara M (2013) Securing computer hardware using 3d integrated circuit (ic) technology and split manufacturing for obfuscation. In: Presented as part of the 22nd USENIX security symposium (USENIX Security 13), USENIX, Washington. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/imeson
35. Vaidyanathan K, Das BP, Sumbul E, Liu R, Pileggi L (2014) Building trusted ics using split fabrication. In: 2014 IEEE international symposium on hardware-oriented security and trust (HOST), pp 1–6
36. Mal-Sarkar S, Krishna A, Ghosh A, Bhunia S (2014) Hardware trojan attacks in fpga devices: threat analysis and effective counter measures. In: Proceedings of the 24th edition of the great lakes symposium on VLSI, ser. GLSVLSI '14. ACM, New York, pp 287–292. [Online]. Available: doi:10.1145/2591513.2591520
37. Salmani H, Tehranipoor MM (2016) Vulnerability analysis of a circuit layout to hardware trojan insertion. IEEE Transactions on Information Forensics and Security 11(6):1214–1225
38. Dupuis S, Di Natale G, Flottes M-L, Rouzeyre B (2013) On the effectiveness of hardware trojan horse detection via side-channel analysis, vol 22. [Online]. Available: doi:10.1080/19393555.2014.891277
39. Hu W, Mao B, Oberg J, Kastner R (2016) Detecting hardware trojans with gate-level information-flow tracking. Computer 49(8):44–52. [Online]. Available: doi:10.1109/MC.2016.225

Sequential Hardware Trojan: Side-channel Aware Design and Placement

Xinmu Wang[1], Seetharam Narasimhan[1], Aswin Krishna[1], Tatini Mal-Sarkar[2], and Swarup Bhunia[1]
[1]Electrical Engineering and Computer Science Department, Case Western Reserve University
[2]Hathaway Brown High School
Cleveland, OH, USA
{xxw58, sxn124, ark70, skb21}@case.edu[1], tmal-sarkar13@hb.edu[2]

***Abstract*—Various design-for-security (DFS) approaches have been proposed earlier for detection of hardware Trojans, which are malicious insertions in Integrated Circuits (ICs). In this paper, we highlight our major findings in terms of innovative Trojan design that can easily evade existing Trojan detection approaches based on functional testing or side-channel analysis. In particular, we illustrate design and placement of sequential hardware Trojans, which are rarely activated/observed and incur ultralow delay/power overhead. We provide models, examples, theoretical analysis of effectiveness, and simulation as well as measurement results of impact of these Trojans in a hardened design. It is shown that efficient design and placement of sequential Trojan would incur extremely low side-channel (power, delay) signature and hence, can easily evade both post-silicon validation and DFS (e.g. ring oscillator based) approaches.**

Keywords- Hardware Trojan Attacks, Sequential Trojan, DFS

I. INTRODUCTION

Globalization in the IC industry decreases control of a designer on the fabricated chips. Incorporation of 3rd party Intellectual Properties (IPs) or design tools, or outsourcing fabrication to off-shore facilities help to lower cost and meet aggressive time-to-market targets. However, such a manufacturing flow typically involves multiple untrusted parties, who can potentially compromise an IC's functional or parametric behavior in a way that can evade conventional post-silicon validation [1]. Such tampering through malicious modification of a design, referred as hardware Trojan, can be introduced at different steps of the IC manufacturing flow, e.g. malicious insertion in an IP, modification of netlist by a CAD tool during design, or tampering a GDSII file during fabrication. This can have serious consequences during in-field operation, especially in security-critical applications such as military, communication and national infrastructure [1-3].

In order to motivate researchers to investigate novel Trojan models and protection approaches, the Polytechnic Institute of New York University has been holding the Embedded Systems Challenge (ESC) [2] competition yearly since 2008, as part of the Cyber Security Awareness Week (CSAW). The competition usually targets hardware Trojan design and insertion techniques [3], hardening mechanisms and Trojan detection approaches. Our participation in ESC 2010 focused on designing novel Trojan attacks against a hardened design, which can bypass protection of existing design-for-security (DFS) based hardening mechanisms [4-5] or testing steps. We designed various ultralow-overhead hard-to-detect hardware Trojans in the test vehicles, and studied models and examples of sequential Trojans as well as efficient placement techniques to bypass the proposed defense mechanisms.

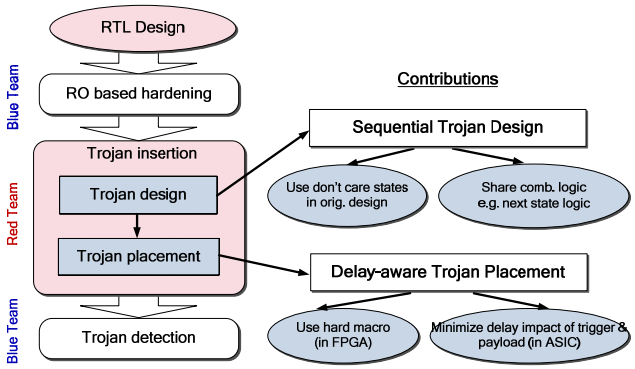

Fig. 1. ESC 2010 competition procedure and our contributions in Trojan insertion as a red team.

Fig. 1 shows the procedure of ESC 2010 competition. The competition was organized as a Red Team – Blue Team approach for hardware Trojan design and detection, using a Digilent BASYS Spartan-3e FPGA platform. The designs provided by the Blue Team were hardened using two defense mechanisms based on delay calibration of different paths of a circuit. The detection mechanism was a side-channel approach whereby any change to the path delays due to inserted Trojans would be reflected in the oscillation frequencies of ring oscillators. Our objective as a Red Team was to insert innovative Trojans, which could evade the detection mechanisms, e.g. conventional functional/structural testing, existing side-channel analysis techniques and the hardening mechanisms, while causing visible malfunction upon Trojan activation. This prompted us to use sequential Trojans with rare trigger conditions which would make the Trojans difficult to activate or observe under normal testing. Other important criteria for Trojan design are minimum impact on delay, power and area of the original circuit, which motivated us to consider novel placement techniques such as delay-aware Trojan insertion and creation of hard macro for the original design (to prevent delay variations in FPGA due to routing changes). Our proposed Trojan insertion mechanism consisting of Trojan design and placement procedures address all the above requirements.

In Section II, we present the concept of hard-to-detect sequential Trojans and discuss their design criteria. In Section III, we provide placement strategies for evading detection mechanisms along with supporting simulation and experimental results. Finally we conclude in Section IV.

II. SEQUENTIAL HARDWARE TROJAN

To prevent hardware Trojans from being detected during conventional post-silicon validation procedures, intelligent attackers are expected to design Trojans which are stealthy in nature. Typically, attackers would insert Trojans that can trigger upon some rare conditions and compromise the security or functionality of the design. Hardware Trojan circuits can either be combinational or sequential [1]. Combinational Trojans are triggered on the occurrence of rare logic values of one or more internal nodes, while a sequential Trojan exhibits its malicious effect after a sequence of rare events during long period of field operation, acting as a time-bomb. Generally, sequential Trojans can be designed to be exponentially harder-to-detect than combinational Trojans by increasing the length of trigger sequence. In fact, these sequential Trojans can be extremely small in size and hard-to-detect during normal post-Si testing. They can also bypass exhaustive testing of a design in full-scan mode.

A. Functional Sequential Trojan Models

A sequential Trojan can be represented as a finite state machine (FSM), where the Trojan trigger sequence is mapped to one of the rarely-satisfied paths in its state transition diagram. The general FSM based model of a sequential Trojan is illustrated in Fig. 2. The next state logic of the Trojan FSM depends on the occurrence of certain rare events, i.e. combinations of rare logic values, of the original circuit's internal nodes. The Trojan circuit undergoes state transition under certain pre-defined rare events in the original circuit; otherwise the Trojan will remain in the current state or go back to the initial state if the expected rare event does not happen. The Trojan output is activated only upon reaching the final Trojan state (S_T), when it affects the payload node compromising the original circuit's normal operation. Next, we provide examples of various types of sequential Trojans.

1) Free-runnning/Enabled Synchronous Counter Trojan: Fig. 3(a) shows a k-bit synchronous counter Trojan with or without an enable signal. A synchronous free-running counter works like a time-bomb where there is no event-dependent trigger condition. The Trojan will get triggered, independent of the operation of the original circuit, and the only design parameter is the time duration for activation of the Trojan (referred as time-to-trigger). It has a deterministic time-to-trigger 2^k-1 clock cycles, where k is the number of state elements in the counter. The drawback of this type of Trojan is the large area/power overhead required in order to guarantee a certain trigger time. By using rare nodes of the original circuit to generate an enable signal for the counter, we can lower the trigger probability and thus greatly increase the time-to-trigger for the same area overhead.

2) Asynchronous Counter Trojan: The asynchronous counter-based Trojan uses an internal signal as the clock for counting the occurences of a rare event. As the example shown in Fig. 3(b), p and q are two rare internal nodes in the original circuit, both of which have the rare logic value of 1. Therefore ANDing p and q creates a signal that seldom switches from 0 to 1, and thus can be used as the clock signal for the counter. By proper choice of the rare events, one can ensure an extremely large time-to-trigger.

3) Hybrid Counter Trojan: To further lower the Trojan trigger probability, a hybrid counter Trojan model is developed as demonstrated in Fig. 3(c). It contains multiple cascaded counters, where the counters can be synchronous or asynchronous, with the clock of the second counter depending on both the first counter state and rare internal events. In the particular example shown in Fig. 3(c), whenever the first counter achieves its maximum value of 2^{k1}-1, if both signals p and q happen to be at their rare value of logic 1, the second counter will be updated.

4) FSM based Trojan: The counter-based Trojans can be generalized to FSM-based Trojans, which contain a sequential and combinational part, with the inputs being derived from rare circuit conditions. The advantage of the FSM-based Trojan is that they can be designed to be arbitratily complicated with same amount of resource and can re-use both combinational logic and flip-flops (FFs) of the original circuit for FSM-hosting. Moreover, unlike counters which are uni-directional, the FSM-based Trojan can have state transitions leading back to the initial state, thus causing the final Trojan state to be reached only if the entire state sequence is satisfied in consecutive clock cycles.

B. Expected Time-to-Trigger

Although the Trojan trigger condition can be deterministic (e.g. counter based Trojan) or probabilistic, the time taken by the inserted Trojan to get activated (time-to-trigger) is not deterministic (except for free-running counter) - instead, it is a pseudo-random value. It depends on the actual Boolean logic used as Trojan state transition function, which gets satisfied based on the actual sequence of input vectors applied to the circuit.

To estimate the expected time of Trojan activation T_{mean}, consider that the Trojan passes through a sequence of states S_1, S_2….S_N before getting activated, as shown in Fig. 2. Suppose the probability of the Trojan transitioning from state S_{i-1} to state S_i is given by p_i, $1 \le i \le (N+1)$, where $N=2^k$-2 and k is the number of state elements. This is essentially a Markov Process, with p_i

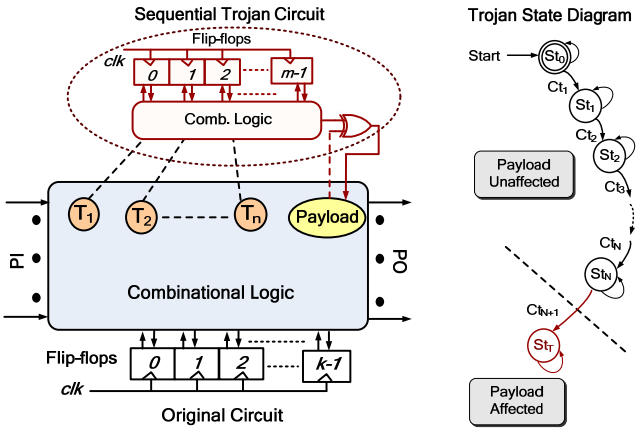

Fig. 2. Sequential Trojan model and Trojan state diagram.

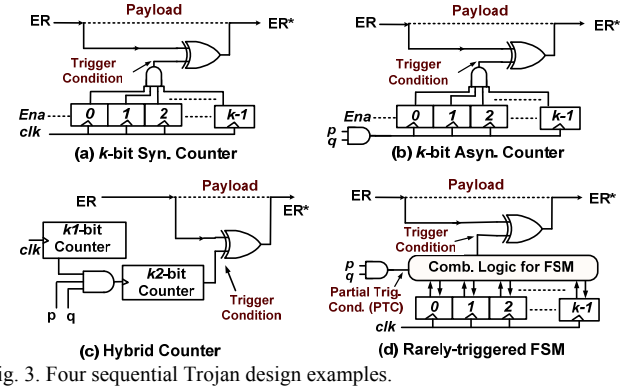

Fig. 3. Four sequential Trojan design examples.

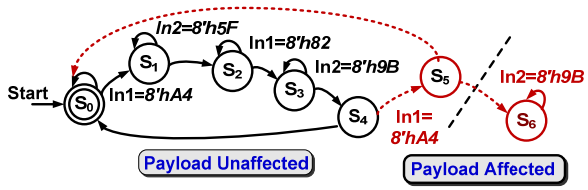

Fig. 4. State diagram of a sequential Trojan with sequential and combinational logic sharing with original circuit.

depending only on the present state S_{i-1} of the Trojan FSM. For simplicity, consider that the Trojan while inactive stays at its present state for all input state space conditions except the unique condition that causes a state transition. Once in state S_{i-1}, the probability of the Trojan staying in state S_{i-1} is $1 - p_i$. Hence, on average, the number of cycles the Trojan spends in state S_{i-1} is:

$$T(S_{i-1}) = p_i.1 + (1-p_i).p_i.2 + (1-p_i)^2.p_i.3 + \cdots \rightarrow \infty$$
$$= \lim_{n\to\infty} \sum_{j=1}^{n} (1-p_i)^{j-1}.p_i.j$$
$$= \lim_{n\to\infty} \frac{1-(1-p_i)^n}{p_i} - n.(1-p_i)^n = \frac{1}{p_i} \quad (1)$$

Hence, the expected time-to-trigger for the Trojan in terms of clock cycles (assume continuous operation): $T_{mean} = \sum_{i=1}^{N+1} \frac{1}{p_i}$ (2)

For an FSM-based Trojan which goes back to the initial state in absence of the rare state transition conditions, the trigger requires a continuous satisfaction of the rare trigger sequence, therefore the trigger probability is: $P(S = S_T) = \prod_{i=1}^{N+1} p_i$ (3)

The Trojan model can be simplified to a two-state FSM containing only the initial state (S_0) and the Trojan state (S_T) where the transition probability from S_0 to S_T is given by equation (3). Since equation (1) is applicable to this one-step model, the expected time-to-trigger is given by: $T_{mean} = \frac{1}{\prod_{i=1}^{N+1} p_i}$ (4)

C. Optimized Implementation

Although in the previously described sequential Trojan model, Trojan state elements are shown separately from those of the original circuit, it is not necessary for sequential Trojan insertions to introduce extra state elements. Instead, they could use existing unused states of the original circuit, if applicable. For example, Fig. 4 shows an FSM of five states, requiring 3 state elements with binary encoding. Here, the unused don't care states (S_5 and S_6) can be leveraged by the attacker to implement a sequential Trojan. Such sequential elements sharing benefits the attackers in both minimizing the area and power overhead, since only the next state logic is modified, as well as in protecting the Trojan from formal verification based approaches. To further reduce the area/power overhead, the Trojan can be carefully designed to reuse the combinational logic of the original circuit. For example, the Trojan state machine in Fig. 4 reuses the transition conditions of the original FSM, whose consecutive occurrence is an extremely rare event in state S_4. Table I demonstrates the area/power overhead due to a sequential Trojan with the same functionality yet different implementations. In particular, Trojan 1 is implemented with extra

TABLE I. AREA / POWER OVERHEAD OF SEQUENTIAL TROJANS OF SAME FUNCTIONALITY BUT VARYING IMPLEMENTATIONS

Overhead* / Design	Area			Power
	Seq.	*Comb.*	*Overall*	
Orig. w/ Troj.1	8.1%	4.3%	5.4%	3.5%
Orig. w/ Troj.2	0	3.4%	2.3%	1.2%
Orig. w/ Troj.3	0	0.8%	0.6%	0.4%

*All designs are synthesized at iso-delay as the original circuit

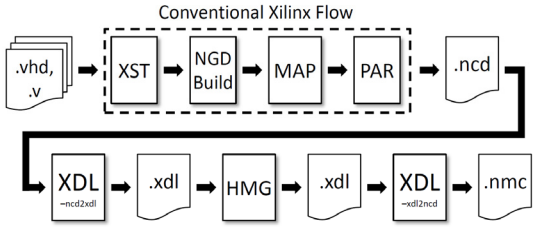

Fig. 5. Hard macro creation flow for the FPGA platform [7].

state elements; Trojan 2 reuses the existing don't care states without next state logic sharing; and Trojan 3 re-uses both state elements and next stage logic, by exploiting existing rare conditions in the combinational logic. For example, in a microprocessor, it is not difficult to find such rare conditions in the memory controller or ALU logic. The power overhead is mainly caused by the leakage power of the sequential Trojans, because dynamic power due to the Trojans is negligible due to their low switching activity.

III. TROJAN PLACEMENT

Recently, Ring Oscillator Network (RON) based design hardening approaches [4, 5] were proposed for hardware Trojan detection through ring oscillator (RO) frequency change. These RON based approaches mainly fall into two categories: One is securing the design by dynamically configuring circuit paths into ROs to monitor undesired design modification [5]; the other is additionally inserted RON to detect voltage drops due to extra Trojan circuitry [4]. The first approach was adopted in the ESC 2010 competition by the "blue team" to harden their design. As an attacker in the competition, we could successfully insert Trojans which evade the hardening/detection mechanism in the FPGA platform. However, we analyze the effectiveness of Trojan insertion with respect to both FPGA and ASIC scenarios. Our analysis shows that attackers can design and insert stealthy Trojans which successfully evade the hardening mechanisms.

In the ESC competition, we were given a 4-bit combinational adder (Beta design) hardened by configuring different paths into ROs during specially-instrumented test modes. In absence of sequential elements in the original design, we resorted to inserting a separate FSM as a rarely-triggered sequential Trojan. However, in an FPGA-based framework, the insertion of any extra circuitry caused the entire design to be re-synthesized and re-routed, resulting in wide fluctuations in the RO frequencies, which were dominated by interconnect routing delays. To mimic an ASIC scenario, where the Trojan is inserted post-layout, we tried to preserve the routing and placement of the original design by making it a Hard Macro. Hard macro generally refers to a pre-compiled module, which can be re-used in its optimized form. Fig. 5 illustrates the flow of creating a hard macro using Xilinx ISE [7]. Since a hard macro consists of previously synthesized, mapped, placed and routed circuitry, the placement and routing of the ROs will not change due to insertion of Trojan circuit. Table II contains

TABLE II. MEASURED RO FREQUENCY CHANGE FOR DIFFERENT TROJANS INSERTED IN THE ESC 2010 COMPETITION DESIGNS

Trojan Type	Beta Easy		Beta Medium	
	RO1	*RO2*	*RO1*	*RO2*
Syn. Count.	1.46%	1.44%	1.59%	2.83%
Syn. Count. w/ En	0.06%	0.49%	2.23%	1.89%
Async. Count.	0.05%	0.83%	0.77%	0.06%
Hybrid count.	0.55%	0.51%	0.85%	1.12%
FSM	3.45%	2.35%	0.80%	3.49%

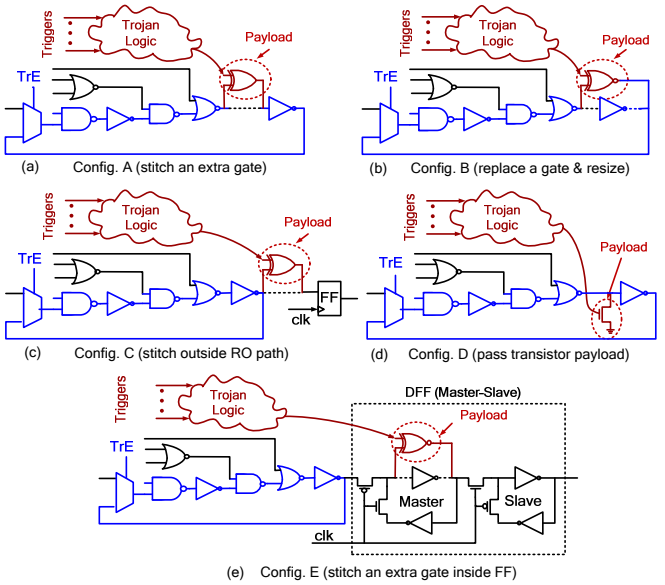

Fig. 6: Different payload insertion approaches: (a) stitching an extra gate (XOR) inside a delay path; (b) replacing an existing gate (e.g. NOT by XOR) and resizing; (c) stitching a gate outside built-in RO path; (d) inserting a NMOS pull-down transistor as payload; and (e) inserting the payload inside a master-slave FF.

the measurement results of RO frequencies from FPGA based framework for the Trojans inserted in the Beta design with hard macro. In the two versions of Beta design (Easy and Medium), different paths were selected to be configured into ROs. The same Trojans inserted in Beta Easy and Medium without hard macro caused 20.01% and 7.47% average RO frequency change respectively. A complete list of our implemented Trojans can be found in ESC website [2]. The only factors causing change in RO frequency are: (i) Load capacitance of certain nodes in the ROs may change due to different routing of non-hard-macro part of the design; (ii) Different placement of the entire hard macro may cause RO frequency change due to intra-die process variations of the FPGA chip. The measured RO frequency changes for different inserted Trojans were found to be within the tolerance of process-induced parameter variations.

Even in the scope of ASIC, dynamically configured ROs cannot guarantee the security of the design, since a clever attacker can always try to bypass them. Even if all paths (and all gates) of the circuit are covered by some ROs, the Trojan can still be inserted without affecting the delay too much. The trigger condition of the Trojan only adds load capacitance to some internal nodes of the circuit, which can be chosen from different ROs. In the Trojan models described in Section II, the payload of the Trojan adds an (XOR) gate delay to the original circuit path, as shown in Fig. 6(a). However, four ways of designing Trojan payload can avoid directly *inserting* gates in RO path: (i) Re-synthesizing the design and re-sizing the gates after insertion of Trojan payload to preserve the path delay. For example, in Fig. 6(b) the Trojan payload is implemented by modifying an inverter to an XNOR gate with the other input coming from the Trojan output, and sizing the gate to have the same delay impact. (ii) Inserting the Trojan payload outside RO paths at a primary output or flip-flop input, so as to add only an extra load capacitance, as shown in Fig. 6(c). This load can be minimized by re-sizing the payload gate capacitance to match the original load capacitance. (iii) The payload can be realized without adding an extra level of gate, e.g. one can simply add an NMOS transistor to the payload node controlled by the Trojan trigger signal to pull the node to 0 as shown in Fig. 6(d), equivalent to a stuck-at-0 fault activated only under rare conditions. This would have virtually no impact on a delay path, since it only adds a diffusion capacitance load, which is much lower than gate capacitance. (iv) Fig. 6(e) provides an example of merging the payload gate into the flip-flop, by replacing one inverter in the D flip-flop with an XNOR gate. In this case, change of the load capacitance cannot be seen by the RO directly thus causes negligible impact. HSPICE simulations of above configurations are performed on Beta design and several ISCAS'85 benchmark circuits using 70nm Predictable Technology Model (PTM) [6] with a supply voltage of 1V at 25ºC, and the results are given in Table III.

TABLE III. IMPACT OF DIFFERENT TROJAN CONFIGURATIONS (AS SHOWN IN FIG. 6) ON RO FREQUENCY, 70NM PTM @1V, 25ºC

Ckt	# of levels in RO path	RO Frequency change*			
		Config. A	*Config. B*	*Config. C*	*Config. D*
Beta	11	7.76%	2.21%	1.80%	0.28%
c880	13	6.40%	2.05%	1.77%	0.26%
c2670	15	5.92%	1.97%	1.51%	0.24%
c3540	15	5.25%	1.76%	1.12%	0.14%
c5315	17	4.38%	1.15%	0.85%	0.11%
c6288	17	3.95%	1.05%	0.74%	0.07%
c7550	25	2.89%	0.85%	0.56%	0.06%

*Config. E does not cause any change in RO frequency

IV. CONCLUSION

In the context of ESC 2010, we have presented innovative approaches of designing Trojans and placing them inside a circuit in a way that effectively evades existing protection mechanisms. In particular, we focused on sequential Trojans triggered by a sequence of rare events, and showed that clever design of Trojan trigger/payload circuits can incur ultralow delay/power overhead, thus bypassing side-channel analysis based detection. Through examples, analysis, and results, we have shown that Trojans inserted in foundries can have minimal impact on path delay, thus evading on-chip monitors (e.g. RO) based DFS approaches. Since they trigger under extremely rare conditions, it is also difficult to detect these Trojans in functional testing. The Trojan design and placement approaches presented are effective for both FPGA and ASIC platforms. Further investigation is underway to evaluate the effect of different hardening/DFS mechanisms on these Trojans.

ACKNOWLEDGEMENT

The authors would like to acknowledge financial support from NSF grant CNS 1054744, CNS 0958510 and Xilinx university program for providing the FPGA platforms and tools.

REFERENCES

[1] R.S. Chakraborty et al, "MERO: A statistical approach for hardware Trojan detection", CHES Workshop, 2009.
[2] ESC website: http://www.poly.edu/csaw-embedded.
[3] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: identifying and classifying hardware Trojans", IEEE Computer, vol. 43 , no. 10, Oct 2010, pp. 39 – 46.
[4] X. Zhang and M. Tehranipoor, "RON: An on-chip ring oscillator network for hardware Trojan detection", DATE, 2011.
[5] J. Rajendran, V. Jyothi, O. Sinanoglu., and R. Karri, "Design and analysis of ring oscillator based Design-for-Trust technique", VTS, 2011.
[6] Predictable Technology Model. Available: [Online] http://ptm.asu.edu/.
[7] C. Lavin et al, "Using hard macros to reduce FPGA compilation time", FPL, 2010.

Chapter 5
Hardware Trojan Insertion

5.1 Introduction

Integrated circuit (IC) designs are becoming increasingly complex to incorporate more advanced capabilities and speed, inspiring and supporting cloud infrastructure, machine learning applications, and ubiquitous handheld mobile devices. Besides the high complexity, the aggressive time-to-market pressure makes completing the entire system-on-chip (SoC) design in-house an infeasible and time-consuming option for most companies. A common practice in the industry is that the design house will search for available commercial third-party intellectual property (3PIP) cores from others to constitute the SoC implementation with their own IPs for faster development and short-time-to-market advantages [1, 23]. In addition to the chip design phase, the prohibitively high cost of maneuvering advanced nodes (e.g., 5-nm technology) has been motivating the horizontal business model of the semiconductor industry in the past 20 years or so. The fabless organizations need to hand their physical designs, e.g., GDSII file, to offshore contract foundries and facilities for silicon fabrication, packaging, and final testing [2, 18].

Although the IP integration and outsourcing fabrication model significantly saves the monetary and time cost for product provision and iterations, it inevitably introduces hardware attack surfaces which have drawn more and more attention from the community since they are extremely hazardous and rarely patchable, compared to their software counterparts. Out of them, malicious addition and modification on the original IC design, the so-called *hardware Trojan*, are two of the most well-recognized attack vectors. For instance, the attackers might want to create backdoors in the chip designs to steal confidential information from the mission-critical applications built on the target device or hamper the reputations of the original component manufacturers. Besides, hardware Trojan can be inserted at arbitrary stages throughout the device lifetime, e.g., it can be implanted at pre-silicon stages such as register-transfer level (RTL) and gate-level 3PIPs by the untrusted IP design teams [9] or physical layout by the adversarial foundries

M. Tehranipoor et al., *Hardware Security Training, Hands-on!*,
https://doi.org/10.1007/978-3-031-31034-8_5

or even at post-silicon stages through advanced physical chip editing techniques like focused ion beam (FIB) [25]. Therefore, it is imperative for researchers and government employees to be mindful of the common modality and insertion manners of hardware Trojan.

The learning objective of this chapter is for trainees to gain experience in the hardware Trojan insertion techniques. Readers will learn how a typical hardware Trojan-infected AES (advanced encryption standard) cryptographic design is constructed at RTL to leak the symmetric key in a rare scenario. Also, this chapter will present the flow mapping of the RTL design on the silicon, i.e., a field-programmable gate array (FPGA) platform using the electronic design automation (EDA) toolchain. In-system debug logic is embedded as well so that readers can *trigger* the malicious functionality during run-time easily. Moreover, the bitstream tampering experiment demonstrates that the adversary can leverage the malleability of a binary FPGA bitstream to enable the hidden malicious circuitry with merely subtle bit flips, further showing the stealthiness of a Trojan.

The rest of the chapter is organized as follows: Section 5.2 provides the background information on hardware Trojans including the present chip design flow and threat model as well as the detailed Trojan structure and taxonomy. Section 5.3 states the provided Trojan-infected AES design and how to implement the RTL design on the FPGA platform and trigger the Trojan at run-time through the on-chip debug infrastructure. Next, Sect. 5.4 shows how to tamper the binary bitstream to enable the hidden Trojan stealthily. Finally, Sect. 5.5 concludes this chapter.

5.2 Hardware Trojan Attacks

In this section, the modern microelectronic device design flow is introduced as to how untrusted entities can implant Trojans. Besides, typical hardware Trojan basics are presented such as its typical structure and taxonomy.

5.2.1 Modern Chip Design Flow and Threat Model

The complexity of SoCs mandates the collaboration between SoC integrators, design-for-test (DFT) teams, and 3PIP vendors to meet the strict time-to-market constraints and procedures and the advantages of competition with peer companies. On the other hand, the prohibitively high cost of maintaining a foundry pushes the shift of semiconductor businesses from traditional integrated device manufacturers (IDM) who design and fabricate chips by themselves to the horizontal model, i.e., fabless companies are only responsible for chip designs, while foundries such as TSMC are focusing on fabrication for minimizing the budget. However, this shift, unfortunately, introduces trustworthiness issues and opens the door for hardware Trojan attacks [13].

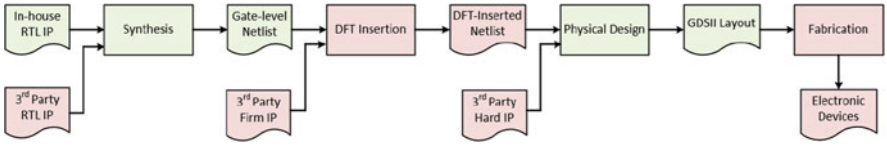

Fig. 5.1 Modern SoC design flow and threat model

The typical modern chip design flow is illustrated in Fig. 5.1 where SoC integrators start with defining the target design specification, e.g., the functionality, performance, and power requirements. To implement the design specification, the functionality is divided as a couple of functional blocks called intellectual property (IP) cores. SoC integrators can design their own in-house IP cores, whereas a more common way to speed up the entire development cycle is to purchase third-party IP (3PIP) cores from 3PIP vendors. When all of IP cores, either designed in-house or from 3PIP vendors, are ready, SoC integrator will use them to constitute the complete RTL description of SoCs. Behavioral simulation is operated by the verification engineers to locate and fix the functionality bugs. After that, electronic design automation (EDA) tools, e.g., Cadence RTL Compiler and Synopsys Design Compiler, will convert the RTL design to the gate-level netlist that comprises low-level information consisting of gates and wires. At this stage, different from the third-party IP cores at the RTL stage, the so-called firm IP at the gate level can be integrated into the netlist as well to accomplish design goals at a more determined performance and functionality because EDA tools usually optimize the RTL design to distinct levels according to customized power, performance, and area constraints during synthesis. Then, design-for-test infrastructure is inserted in the netlist by the (third-party) DFT team for enabling structural testing at the post-silicon stage for observability, controllability, and testing coverage. Next, the DFT-inserted netlist is transformed into a physical layout GDSII design, i.e., transistor-level design. The physical design will incorporate hard 3PIP with the most fixed parameters and go to the outsourced offshore silicon foundry such as Samsung and TSMC for wafer fabrication and die package.

As one can see in Fig. 5.1, there are several untrusted entities/stages in the light red boxes. The adversarial roles they might play in the supply chain for hardware Trojan attacks are discussed as follows:

- **3PIP Vendors**: As mentioned, SoC integrator has to rely on 3PIP vendors to provide a variety of RTL (soft), firm, and hard IP cores to meet the time-to-market requirements. Nevertheless, the reality is that such 3PIP vendors locate across the entire world such that SoC integrators are not able to inspect their integrity. Moreover, as 3PIP cores are typically presented as black boxes, e.g., by following the IEEE 1735 standard [12], SoC integrator cannot look into the details for hardware Trojan detection. Even if some of 3PIP cores are given in plaintext, the complexity of the IP design and the stealthiness of malicious hardware Trojan prevent SoC integrators from successful security closure by using conventional testing and verification techniques.

- **Outsourced DFT Teams**: DFT infrastructure has gained tremendous importance over the past decade for empowering better controllability and observability for post-silicon debugging. This task is usually outsourced to external specialized DFT teams for improved efficiency and less cost. In this way, the untrusted DFT teams have the access to all details of gate-level designs and can insert malicious functionality into the original implementation. As the scan flip-flop insertion during DFT inevitably changes the design topology and the gate-level netlist contains more than millions of gates in most cases, it is extremely hard to detect very few gates of malicious functionality. Even worsen, as the DFT infrastructure can stream in and stream out information from the internals of the silicon, some Trojan can be directly implanted in the debug logics to access security-sensitive on-chip assets such as private keys and user credentials [4, 16].
- **Offshore Foundries**: Establishing and maintaining a silicon foundry come at billions of dollars cost that very few companies can afford. Besides, the advanced technology node such as 3 nm or 5 nm can merely be maneuvered by a few offshore foundries such as TSMC. The fabless SoC integrator will hand all of the design details in the GDSII physical layout to the foundry which might be untrusted and intend to insert malicious hardware Trojans. For example, the empty corners in the chip area can be used to place Trojan gates, while a rouge foundry can wire sensitive signals to the public interface which can be accessed externally. Detecting such Trojans could be extremely difficult since the post-silicon device is a black box from the security inspection perspective and hardware Trojans can be dormant most of the time so the Trojan-infected devices do not manifest any abnormal behaviors.

5.2.2 Hardware Trojan Insertion

Hardware Trojan is a malicious addition or modification of an integrated circuit (IC) [26]. The malicious functionality includes but is not limited to changing the original functionality, compromising the confidentiality of security assets, and causing performance degradation or even denial of service. It is challenging to detect and remove hardware Trojans from the infected design because golden designs are not available for 3PIP cores, netlist, and layout. Besides, most of the hardware Trojans are designed to function in a stealthy way, i.e., manifesting malicious functionality in very rare conditions.

As illustrated in Fig. 5.2, the hardware Trojan structure includes two parts, i.e., trigger and payload [11, 22]. Trojan trigger simultaneously monitors possible stimulus in the circuitry or physical environment. The most common way is that the Trojan trigger monitors multiple internal signals simultaneously and outputs an asserted trigger signal to start malicious activities of the payload if a specific predefined signal pattern is found. In other words, without seeing the pattern on the trigger inputs, the hardware Trojan will remain inactive and hard to be detected. Although a variety of Trojans were proposed [4], they can be classified into

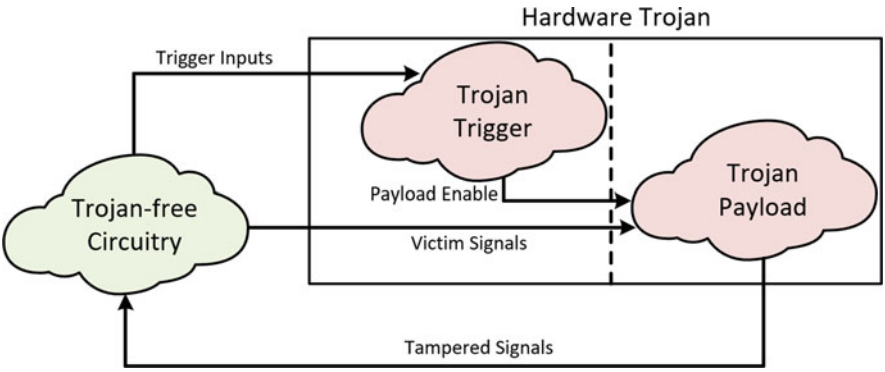

Fig. 5.2 High-level Trojan diagram

combinational Trojans and sequential Trojans according to the trigger mechanism. Note that digital hardware Trojan is focused on in this chapter. The analog hardware Trojans [28] is out of the scope. Figure 5.3a and b shows the simplified models of both combinational and sequential Trojans. Suppose that the trigger inputs are n-bit and the trigger circuitry is an n-bit AND gate. Therefore, the rare trigger condition is defined that only if all of the trigger inputs are 1, the payload enabled signal will be asserted, i.e., the probability of random Trojan triggering is $\frac{1}{2^n}$. To make Trojans even more stealthy, sequential Trojan can use the output of the AND gate as the start flag of a finite state machine (FSM). For example, a straightforward counter can be implemented as a time bomb by activating the Trojan payload at a future moment [24]. As for Trojan payloads, it can be very versatile. For instance, to degrade the device performance and launch denial-of-service attacks, [5] proposes to place a design-independent ring oscillator (RO) on the same silicon as an always-on Trojan. Since RO consists of odd number of inverters, its logical status is in-stable and self-oscillating. Such behaviors will continuously draw currents from the power supply and accelerate the chip aging. The authors of [15] present an off-chip Trojan that can leak the cryptographic keys through a power side channel by using an external capacitor.

Below hardware Trojan taxonomy (see Fig. 5.4) is based on five aspects, i.e., (i) insertion phase, (ii) abstraction level, (iii) activation mechanism, (iv) payload, and (v) location:

- **Insertion Phase**: *Specification* level defines crucial factors of the device including the functionality, performance, and area. A hardware Trojan inserted at specification can, for example, alter the timing and functionality of the final design. *Design* level can be exploited by 3PIP vendors to implant hardware Trojans by injecting malicious circuitry in the IP cores. Rogue foundries can insert hardware Trojans during the *fabrication* phase since they have the access to all of the physical design details in the layout and thus have the ability to alter the final layout by modifying the mask set before wafer fabrication. In the

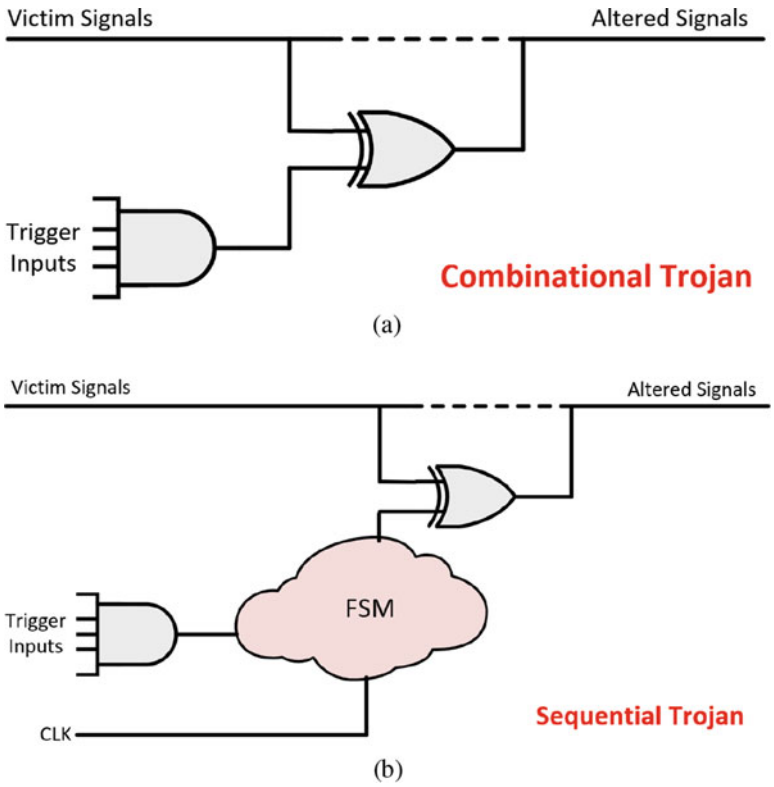

Fig. 5.3 Hardware Trojan models (**a**) Combinational Trojan (**b**) Sequential Trojan

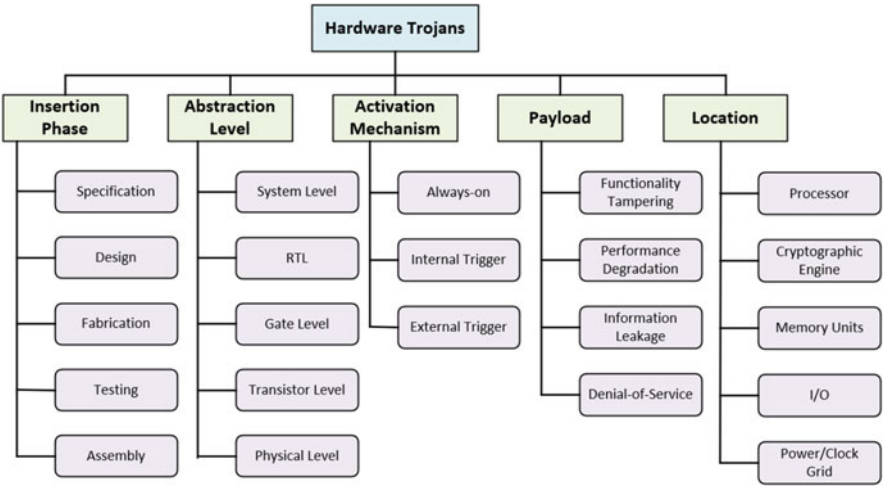

Fig. 5.4 Hardware Trojan taxonomy

testing phase, the adversaries might intentionally generate a set of low-coverage test vectors with low Trojan coverage to help the infected devices escape from detection. People purchase the components and integrate them with the fabricated microelectronic devices on the same printed circuit board (PCB) at the *assembly* phase. System-level Trojans such as BigHack [17] can be inserted at this time to compromise confidentiality, integrity, and availability.

- **Abstraction Level**: Hardware Trojans at *system level* can behave as an additional module altering the functionality of other system modules, interconnects, and communication traffic. *RTL*, *gate-level*, and *transistor-level* hardware Trojans can all be incorporated in the corresponding design files, i.e., RTL code, gate-level netlist, and physical layouts, respectively. As for the *physical level* hardware Trojans, it is implemented by modifying the design parameters. For example, the critical path wire length can be varied to increase the risks of timing failure.
- **Activation Mechanism**: Some Trojans are made to be *always on* such as the RO Trojan in [5]. In contrast, most Trojans tend to be trigger-based in case the malicious behaviors can be discovered easily. *Internally triggered* hardware Trojans rely on internal events like an embedded counter activating the Trojan payload after 2 days since the circuit starts to work. As for *externally triggered* Trojans, they usually depend on user input patterns. For instance, a Trojan targeting a cryptographic module monitors the plaintext input and leaks the key if and only if the plaintext is set to be a specific value.
- **Payload**: Trojan payload for *functionality tampering* can alter the benign behaviors of the original circuit, e.g., an activated Trojan enforces a password-checking circuit to accept an arbitrary string. *Performance degradation* can be brought by a power-hungry Trojan design. It consumes a significant amount of power and thus induce more IR drop to slower the entire device. *Information leakage* can be launched by Trojans by sending sensitive security assets and credentials without the approval of supervisors. In addition, *denial-of-service* Trojan might physically disable or even destroy the microelectronic chips.
- **Location**: Trojans at *random logic* can hinder effective test stimuli generation for detection since understanding such random logic is difficult. Other Trojan locations such as *processing unit*, *cryptographic engine*, *memory units*, and *I/O* can make significant differences even with minor Trojan impacts because they either process or store sensitive information. As for Trojans in *power supply* and *clock grid*, they are more potent to result in timing failures like setup/hold-time violations by causing physical glitches.

5.3 Trojan-Infected Implementation on FPGA

In this section, the FPGA development flow will be introduced because our experiments are carried out on an FPGA. Next, the experimental setup is discussed including the target FPGA platform. Finally, the structure of the Trojan-infected implementation and detailed steps compiling the RTL designs and demonstrating the Trojan's effectiveness are presented.

Fig. 5.5 FPGA development procedure

5.3.1 FPGA Development Flow

Field-programmable gate array (FPGA) has become the most important and popular option for agile hardware prototyping. It is flexible and can be reconfigured by the users in the post-manufacturing phase. The typical development flow of FPGA device involves design entry, synthesis, implementation, and bitstream generation as shown in Fig. 5.5. Design entries can accept a variety of design files. The most intuitive method is drawing the schematics by connecting some predefined functional modules together, whereas a more common and recommended way in industry is to write the behavioral implementation in the form of hardware description language (HDL) like VHDL and Verilog at RTL which is the same as the standard application-specific integrated circuits (ASICs) development flow. During the synthesis stage, the HDL code composed at the design entry stage will be converted into a circuit in the form of netlist by vendor-specific EDA tools like Xilinx Vivado [6] and Intel Quartus [20]. The RTL code is going to be parsed automatically in the EDA environment to check syntax and then optimized to reduce redundant logic per the specified settings. The outcome functionally equivalent netlist contains the mapped logic elements and the connectivity among them as described in the RTL code. The implementation phase will then technology map the logic elements in the netlist to the primitives available in the target FPGA model so that the design could be built on the physical silicon. Also, this step will place and route the primitives on the FPGA layout virtually per the constraints from designers and physical aspects to pursue the closure of the power, area, and performance on the final design. Finally, the placed and routed netlist will be translated to the binary configuration data, the so-called bitstream, and then be downloaded to the target device via an interface like JTAG.

5.3.2 Experimental Setup

In our hands-on experiments, the Nexys A7 board featuring Xilinx Artix 7 FPGA (part number XC7A100T-1CSG324C) is used as shown in Fig. 5.6. With its large, high-capacity FPGA, generous external memories, and collection of USB, Ethernet, and other ports, the Nexys A7 can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, a temperature sensor, a MEM digital microphone, a speaker amplifier, and several I/O devices allow the Nexys A7 to be used for a wide range of designs without needing any other components. The Nexys A7 is

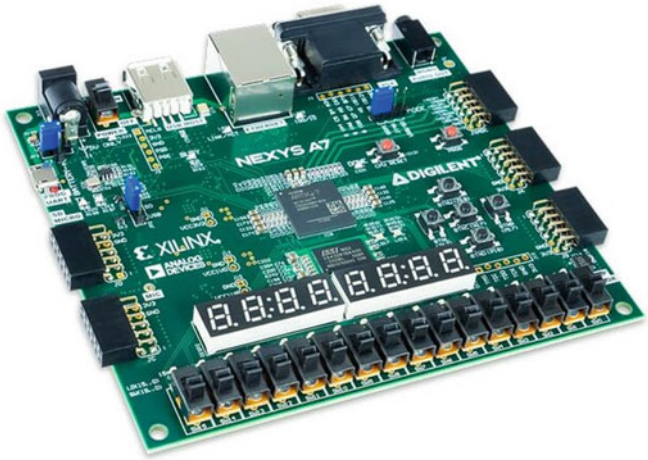

Fig. 5.6 Digilent Nexys A7 board

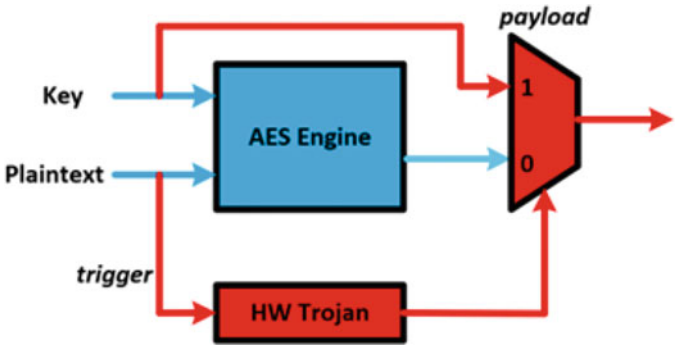

Fig. 5.7 Trojan-infected AES implementation overview

programmed and debugged via the USB connection to the host running Xilinx Vivado 2019.1 and Win10 operating system. The design files and source codes can be found at http://cad4security.org/index.php/trainings/hsl/ch5_hw_trojan_insert/.

5.3.3 Trojan-Infected Design

In this chapter, a hardware Trojan-infected implementation is designed as depicted in Fig. 5.7. The benign implementation is an AES-128 cryptographic engine (blue module) that encrypts the incoming plaintext with the fixed secret key. The AES algorithm is mathematically strong, which means that it is computationally infeasible for an adversary to derive the unknown secret with the controllability of plaintext and observability of ciphertext even if the AES algorithmic or

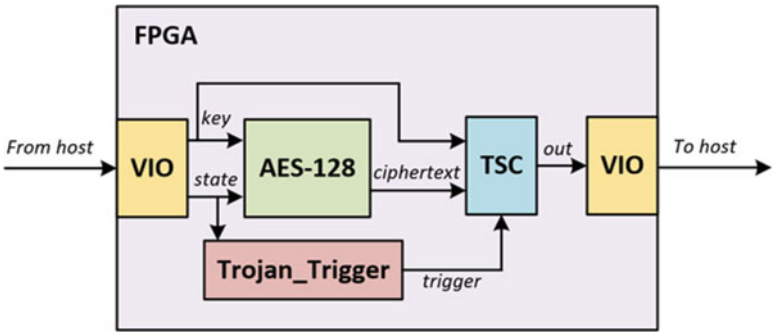

Fig. 5.8 Detailed FPGA implementation diagram

hardware accelerator design details are public. To guarantee the confidentiality of the secret key, it needs to be stored in a tamper-proof memory with strict access control and securely transferred to the hardware AES engine, i.e., the adversary cannot intercept it during the transmission. However, this Trojan-infected implementation can lead to key leakage by bypassing the AES core in a specific rare condition. The trigger part of the inserted hardware Trojan continuously monitors every input plaintext to see if the predefined pattern occurs (128'h0011_2233_4455_6677_8899_AABB_CCDD_EEFF) here. If it is the case, the Trojan will be activated to enable the "1" channel of the output multiplexer to leak the secret key. Otherwise, the "0" channel is enabled to output ciphertext when the Trojan is dormant. This is a typical information leakage Trojan model in cryptographic implementations.

Figure 5.8 presents the detailed implementation. To simplify the interface between the host and FPGA for sending and receiving data, Xilinx Virtual IO (VIO) is instantiated as an IP core by configuring virtual input/output probes. In this way, one can easily set the key and state (plaintext) for the AES-128 engine and observe the output of the Trojan-infected AES. The ciphertext and key signals are multiplexed by the **TSC** module. If the *trigger* signal is asserted, i.e., Trojan is activated by the predefined pattern on the state (plaintext) input port, key input will bypass the AES-128 core to the output which can be captured by VIO. Figure 5.9 shows the design hierarchy of this Trojan-infected implementation in Xilinx Vivado where the naming of functional blocks *aes_128*, *Trojan_Trigger*, *TSC*, and *vio_0* is self-explanatory by corresponding to the blocks in Fig. 5.8. The XDC file (top.xdc) is indispensable for a successful compilation by providing constraints on package pin assignment, targeted clock frequency, implementation settings, etc.

5.3.4 Compiling Target Design and Trigger Trojan

We present detailed step-by-step instructions on compiling and activating the Trojan-infected implementation in Xilinx Vivado:

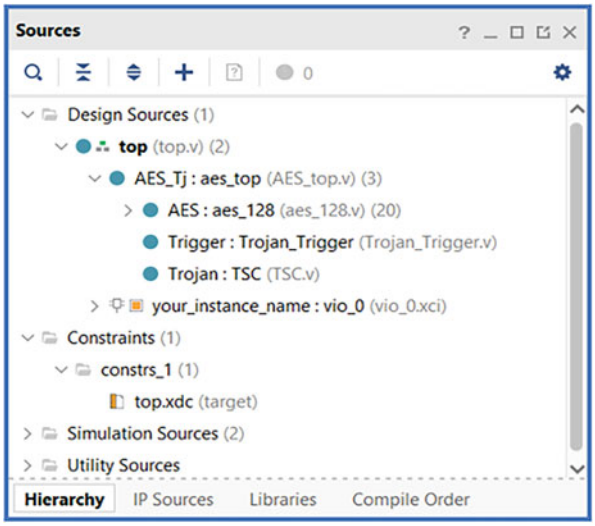

Fig. 5.9 Trojan-infected code hierarchy

Step 1: Compile the RTL Design in Vivado Through Synthesis, Implementation, and Bitstream Generation Xilinx Vivado can compile the loaded RTL designs to the bitstream in an automatic manner. By clicking the *Run Synthesis* as labeled in Fig. 5.10, Xilinx Vivado will initiate a run to transform the RTL design to a synthesized design where a functionally equivalent circuit is generated consisting of generic components. Then, *Run Implementation* will map the generic components to available resources in the specified FPGA model. For example, an individual AND gate cannot be found in most modern FPGA devices. In the implementation phase, Vivado will map the AND gate to an **LUT2** primitive, i.e., a look-up table instance storing the Boolean equation **O = A1 & A2** to implement the AND function. Besides, the low-level routing will be mapped to the programmable interconnect point (PIP) configuration of the FPGA model. *Bitstream Generation* interprets the implemented design to the proprietary binary configuration file which can be downloaded to the FPGA silicon.

Step 2: Downloading the Bitstream to FPGA Click the *Open Target* under *Open Hardware Manager* as shown in Fig. 5.11. Then, one needs to *Open Target* to auto-connect the hardware server, i.e., our host, to the FPGA device instance through the JTAG interface. Next, program the device by right-clicking the target device under the local host and selecting *Program Device* as shown in Fig. 5.12. Then, the *Program Device* pop-up window will set the default bitstream as the newly generated bitstream (*top.bit*). Besides, the design probe file is set to be *top.ltx* which contains the debug VIO core configuration like the virtual input/output probe name and width.

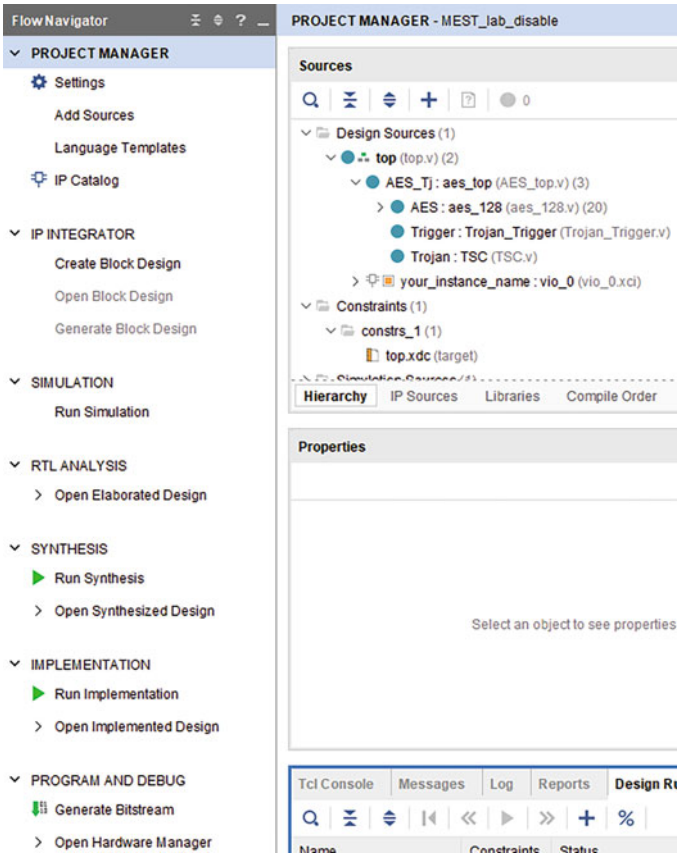

Fig. 5.10 Compiling design in Xilinx Vivado

Step 3: Activating the Trojan on FPGA After downloading the bitstream to FPGA, the VIO debug core dashboard will appear in Vivado as illustrated in Fig. 5.13. Note that the reset input pin of the implementation is bonded to the DIP switch SW5 which needs to be down to dessert the signal. In Fig. 5.13, key is 128'hAAAA_AAAA_AAAA_AAAA_BBBB_BBBB_BBBB_BBBB, while the state (plaintext) is 128'h0000_0000_0000_0000_0000_0000_0000_FFFF. As the plaintext input is not the triggering pattern, the Trojan still remains dormant, and the AES-128 core will perform ten-round *SubByte*, *MixColumn*, *ShiftRow*, and *AddRoundKey* operations on the plaintext to produce the ciphertext output. In contrast, if the 128-bit state is set to 128'h0011_2233_4455_6677_8899_AABB_CCDD_EEFF as depicted in Fig. 5.14, Trojan is activated such that the encryption key is leaked through the output port. In this way, the effectiveness of the inserted hardware Trojan is demonstrated on the FPGA device at run-time.

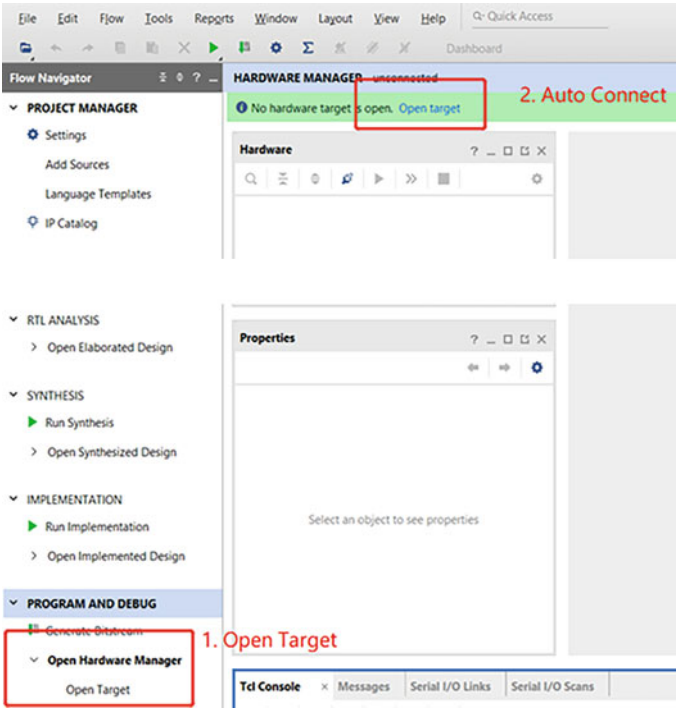

Fig. 5.11 Open and connect the FPGA target

5.4 Bitstream Tampering for Trojan Triggering

In this section, FPGA bitstream tampering attacks to enable Trojan triggers will be presented. First, FPGA bitstream preliminaries and tampering attacks are introduced. Next, experimental files and steps are detailed.

5.4.1 FPGA Bitstream Format Preliminaries

FPGA bitstream is crucial because it determines the FPGA behaviors at the post-configuration phase. Figure 5.15 depicts the high-level format of Xilinx bitstream. It mostly starts with human-readable content such as design name and timestamp. However, the content will be discarded by the hardware FPGA configuration engine during the bitstream loading stage. The beginning of the configuration data stream is the sync word for the alignment of the subsequent data. The following header commands will read/write important registers. For example, one can write the WBSTAR register for multi-boot configuration [10]. Also, writing the IDCODE register to inform the engine the target device model of the incoming bitstream;

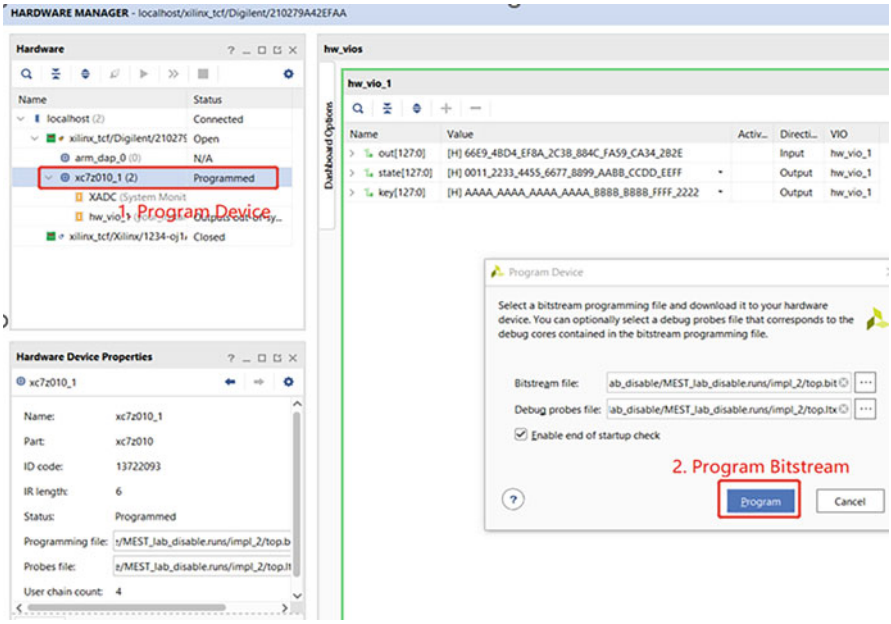

Fig. 5.12 Download bitstream to FPGA

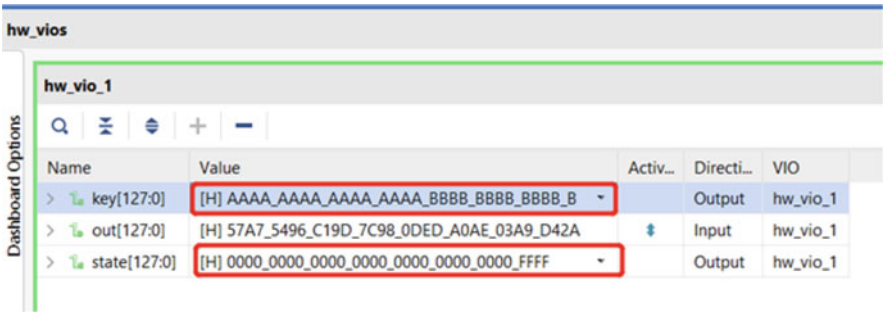

Fig. 5.13 Ciphertext on the outport when the Trojan is dormant

Fig. 5.14 Key leaked to the outport when the Trojan is activated

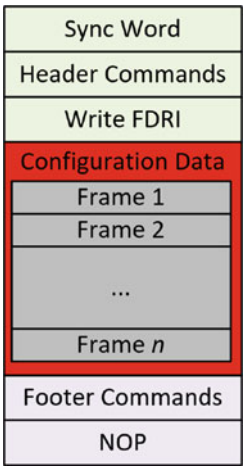

Fig. 5.15 High-level FPGA bitstream format

the device will reject the bitstream if the IDCODE does not match. The most crucial operations are **Write FDRI** which write the configuration frames to the on-chip SRAM determining the FPGA functionality. The address of the on-chip configuration memory the frames flow to is determined by the frame address register (FAR) which is auto-incremented by default. The footer commands include cyclic redundancy check (CRC) checksum and end with a couple of NOP (no operation) commands.

Although bitstream format can be understood at a high level from official documents [27], FPGA vendors are reluctant to reveal the bitstream format documenting the mapping between configuration data and FPGA primitive functionality. FPGA bitstream reverse engineering techniques [3] have been developed to retrieve such information to assist applications like hardware Trojan detection [14, 29] or insertion [7]. For instance, [29] proposes a high-accuracy FPGA reverse engineering for recovering the netlist from the binary bitstream and applies an unsupervised machine learning solution to detect suspicious Trojan signals. Conversely, the retrieved bitstream format can be used for bitstream tampering as well. Swierczynski et al. [21] first reverse engineer the configuration bits of look-up tables (LUTs) and then target AES accelerator on FPGA. As the AES hardware typically relies on the LUTs to store the S-box content, tampering with LUT content can inject faults to steal keys through differential fault analysis (DFA). Even if security mechanisms like bitstream encryption are enabled, such bitstream tampering is still feasible since AES-CBC allows causing bit flips on the target block by corrupting the previous one. Moreover, [8] hacks the Xilinx 7-series bitstream engine and discloses the underlying vulnerabilities which can be exploited to decrypt the ciphertext bitstream by using the FPGA as an oracle [19].

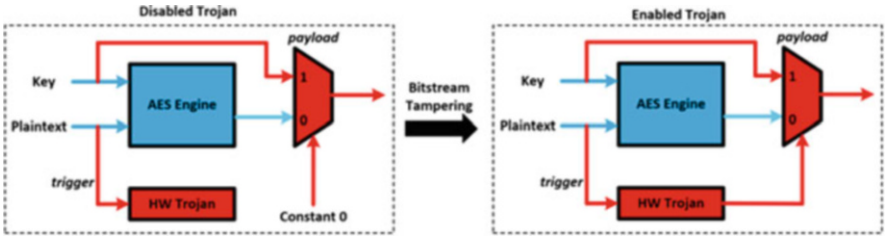

Fig. 5.16 Enabling hardware Trojan through bitstream tampering

5.4.2 Bitstream Tampering Enabling Trojan Trigger

In this experiment, Trojan trigger is disabled by disconnecting the trigger signal from the multiplexer in the original implementation as depicted in the left part of Fig. 5.16. As the multiplexer SEL port is fixed at the constant 0, the Trojan can never be activated even if the trigger part observes the predefined pattern on the plaintext input. However, by tampering with the design at the bitstream level, the low-level routing can be altered to revert the hardware Trojan functionality, i.e., the trigger signal will be connected to the multiplexer.

There are supposed to be five files in the ./nexys/ folder. The details and observations on FPGA are introduced as follows:

- *AES_trigger_disabled.bit*: This bitstream encodes the Trojan-infected AES implementation with the disabled Trojan trigger. To generate the bitstream, one can modify the driver component configuration of the trigger signal in the original infected design. A feasible way is to uncomment the command *set_property INIT 4'h0 [get_cells AES_Tj/Trigger/Tj_Trig_reg_i_3]* in the constraint file *top.xdc* (Sources panel → constraints → constrs_1 → top.xdc) and rerun the flow through synthesis to bitstream generation. As a matter of fact, the command erases the configuration bits of the driver LUT of the trigger signal to be all 0s. In other words, the output of LUT is fixed to 0 to all combinations of inputs. This results in minimal changes in the implemented design where only the content of a LUT is cleared. If one changes the functionality at RTL, the entire design floor plan is likely to deviate which would make the bitstream tampering less intuitive and clear. As shown in Fig. 5.17, downloading this bitstream along with the same probe file *top.ltx* because the VIO configuration is identical. The VIO dashboard in Fig. 5.18 illustrates that even if the predefined pattern 128'h0011_2233_4455_6677_8899_AABB_CCDD_EEFF occurs on the plaintext input, the Trojan is still inactive as the output is not the key value but the AES-128 ciphertext instead, which is different from the original Trojan-infected behaviors.
- *AES_trigger_enabled_ref.bit*: This bitstream contains the original Trojan-infected AES implementation with an enabled trigger for reference. The functionality has been detailed in Sect. 5.3.3.

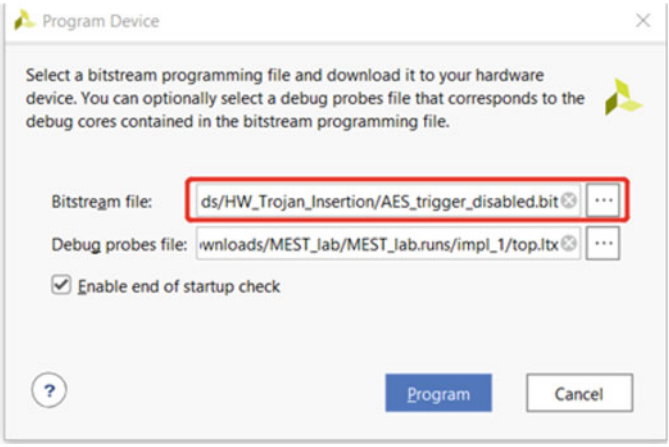

Fig. 5.17 Download AES trigger-disabled bit (disabled Trojan)

Fig. 5.18 Key cannot be leaked even if the predefined pattern occurs

```
C:\Users\zht24\Downloads\HW_Trojan_Insertion>python tampering.py
Bistream tampering started!
Bistream tampering finished! The Trojan trigger has been enabled in the AES_trigger_enabled.bit file.

C:\Users\zht24\Downloads\HW_Trojan_Insertion>
```

Fig. 5.19 Running Python script to tamper with the bitstream to enable Trojan

- *Tampering.bit*: This binary file has the same length of other bitstreams which documents the critical bits in AES_trigger_disable.bit that needs to be flipped during the tampering procedure.
- *Tampering.py*: The python script can tamper the AES_trigger_disabled.bit to activate the disabled Trojan trigger. Run the python script tampering.py to generate the bitstream AES_trigger_enabled.bit by XORing AES_trigger_disabled.bit and tampering.bit. To run the python script, just cd to the directory (e.g., ./nexys/), and type "python tampering.py." The successful bitstream tampering printout is shown in Fig. 5.19. By examining the content of *tampering.bit* using HexEditor (see Fig. 5.20), only few configuration bits need to be flipped to enable the Trojan at the bitstream level. The reason is that only the driver LUT content is erased to fix the Trojan multiplexer to channel 0 so the bitstream tampering procedure essentially converts the LUT back to the functional status.

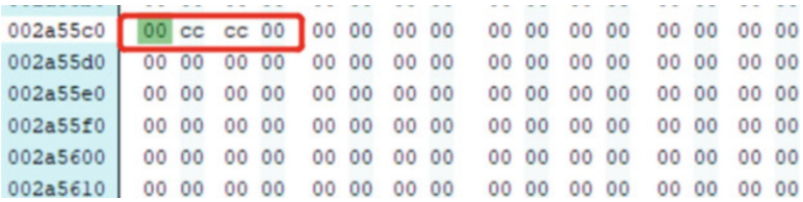

Fig. 5.20 A portion of tampering bit file

Fig. 5.21 Trojan can be activated in the AES trigger-enabled bit file

- *AES_trigger_enabled.bit (to be generated)*: One needs to run the Python script tampering.py to generate the bitstream. The content in this binary file should be equivalent to the AES_trigger_enabled_ref.bit. The bitstream contains the enabled Trojan trigger as a consequence of the bitstream tampering. By programming the FPGA device with this bitstream, it is observed in Fig. 5.21 that Trojan can be activated when the specific pattern appears on the state (plaintext) input to leak the key input through the output port.

5.5 Conclusion

Hardware Trojan threats are a long-standing concern in the hardware security community. It is imperative to learn about its attributes and features. In this chapter, the horizontal chip design cycle is introduced at first to give insights into why Trojan insertion is feasible in the real world. Next, Trojan background including its structure and taxonomy is detailed. In order to give an intuitive understanding of Trojan insertion, a Trojan-infected AES engine is provided to be implemented on an FPGA platform which can be activated/deactivated at run-time to leak the sensitive security key or not. Also, FPGA bitstream tampering, as an advanced attack technique, is experimentally demonstrated on enabling the hidden Trojan trigger at the binary level.