DESIGN FOR TRUST IN BEHAVIORAL VLSI DESIGN

NANDEESHA VEERANNA

Ph.D

The Hong Kong Polytechnic University

2018

The Hong Kong Polytechnic University

Department of Electronic and Information Engineering

Design for Trust in Behavioral VLSI Design

Nandeesha Veeranna

A thesis submitted in partial fulfilment of

the requirements for the degree of

Doctor of Philosophy

July 2017

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signed)

__Nandeesha Veeranna__(Name of student)

I would like to dedicate this thesis to my loving parents, brother, and my friends.

Abstract

The continuous globalization in the semiconductor design and fabrication process of integrated circuits (ICs) is making these extremely vulnerable to malicious modifications. An adversary in a foundry or in an intellectual property (IP) design house, can insert a small malicious circuitry inside the genuine circuit which can lead to a serious catastrophe in the field of operation or leak the secret information.

In addition, the electronic design automation (EDA) industry has tackled the increase in IC design complexity and shorter design cycles by raising the abstraction level from register transfer level (RTL) to behavioral level (C/C++/SystemC). This paradigm shift opens a new window for attackers to insert more powerful hardware Trojan, as a single line in a behavioral program can lead to $7\times$ more gates than at the RT-level [1]. Thus, this thesis addresses the possible security issues concerned to behavioral IPs (BIPS) mapped as hardware accelerators in the heterogeneous system on chip (SoC). Although this thesis mainly extends theories to protect the BIP user from malicious alterations of the BIP, it also investigates methods to protect the BIP vendor from an unlawful usage of the BIP. For this purpose, an open source benchmark suite called Security Synthesizable SystemC benchmark (S3Cbench) consists of multiple BIPs written in synthesizable SystemC which include different types of Trojan was first created. These benchmarks are used as the backbone of this work to create methods to find these Trojans when no golden reference models exist. This thesis then studies the impact of obfuscation on the quality of results (QoR) of the BIPs and propose an efficient method to obfuscate these BIPs without a significant QoR degradation due to the redundant operations

inserted by the obfuscator. A hardware Trojan detection technique which leverages formal verification techniques at the behavioral level is also presented. In particular, property checking techniques to increase source code coverage. This thesis also addresses the detection of hardware Trojan at the system level. C-Based VLSI design has many advantages compared to traditional RTL design. One that this work takes advantages of is the generation of fast cycle-accurate models of complete SoCs. This is used to measure the exact timing at which each BIP mapped as a loosely coupled Hardware Accelerator (HWAcc) slave. This is in turn used to detect if the Trojan has been triggered during the intervals in which the accelerator is not performing any computation. This allows the fine tuning of our proposed circuit called *Trust Filters* which can detect the hardware Trojans at runtime. The last part of this thesis addresses the issues of how to avoid hardware Trojan to be inserted in runtime reconfigurable systems which depend on a configuration bit stream to reprogram their functionality every clock cycle, making them extremely vulnerable to Trojan. The experimental results and implementation analysis demonstrate the effectiveness of our proposed techniques.

List of Publications

Journal Papers:

1. **N. Veeranna**, B. Carrion Schafer, "Hardware Trojan detection in Behavioral Intellectual Properties(IPs) using Property Checking Techniques," *IEEE Transaction on Emerging Topics in Computing*, June 2016, in print.

2. **N. Veeranna**, B. Carrion Schafer, "Trust Filter: Runtime Hardware Trojan Detection in Behavioral MPSoCs," *Journal of Hardware and System Security*, Springer, vol.1, no.1, pp. 56-67, March 2017.

3. **N. Veeranna**, B. Carrion Schafer, "S3CBench:Synthesizable Security SystemC Benchmarks for High-Level Synthesis," *Journal of Hardware and System Security*, Under major revision.

Conference Papers:

1. A. Balachandran, **N. Veeranna**, B. Carrion Schafer, "On Time Redundancy of Fault Tolerant C-Based MPSoCs," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Pittsburgh, Pennsylvania, 11-13 July 2016, pp. 631-636.

2. **N. Veeranna**, B. Carrion Schafer, "Hardware Trojan Avoidance and Detection for Dynamically Re-configurable FPGAs," *IEEE International Conference on Field Programmable Technology (FPT)*, Xi'an, China, 7-10 Dec.2016, pp. 193-196.

3. **N. Veeranna**, B. Carrion Schafer, “Efficient Behavioral Intellectual Properties Source Code Obfuscation for High-Level Synthesis,” in *IEEE Latin American Test Symposium (LATS)*, Bogota, Columbia, 2017, pp.1-6.

4. **N. Veeranna**, B. Carrion Schafer, “Automatic Hardware Trojan insertion in Behavioral IPs during the Obfuscation Process,” **won the third prize in HACK@DAC Hardware Security contest** in *Design Automation Conference (DAC)*, Austin, USA, 2017, poster.

Acknowledgements

I owe immense gratitude to many people who have instructed and advised me in the course of writing this thesis.

First of all, I would like to express my sincere heartfelt thanks to my supervisors Prof. Francis Lau and Dr.Benjamin Carrion Schafer, for their invaluable advice, constant encouragement and precise modification. I admire their knowledge and personality. They not only grants me the opportunity to enter the world of academic research but also gave me the consistent and illuminating instruction in this prolonged three years of study. Without them, I could not have completed this thesis.

My thanks also go to Prof. Michael Tse and Prof. Daniel.P.K. Lun for their invaluable suggestions during the Research Methodology course on writing the research articles and thesis.

I am fortunate to have the chance to work with Anushree Mahapatra, Anjana Balachandran, Dylan Liu, Shaungan Liu, Siyuan Xu, Susmitha, Pandy, and Farah Taher during my Ph.D. study. I am thankful for their share of knowledge and thoughts at work, and the life we have enjoyed together.

I also thank my friends Harish Kumar Narayana, Harun Venkatesan, Virag Raut, Parth Shah, Yuvraj, Suman, Yamuna, Sonal, Tanuja, Bipin, and Atik. My special thanks go to my badminton friends Jerry, Alex, Josephine, Eddie, William, Man Wai, Irene, Wendy, Andrew, Isaac for my interesting and exciting university life in Hong Kong.

At last, my deepest gratitude goes to my parents for their endless love and great confidence in me through these years. They are always my strong backing.

Table of contents

List of figures

List of tables

List of Acronyms

AES	Advanced Encryption Circuit
AHB	Advanced High-performance Bus
ALAP	As Late As Possible
ALU	Arithmetic Logical Unit
AMBA	Advanced Micro-controller Bus Architecture
API	Application Program Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
BIP	Behavioral Intellectual property
CDFG	Control Data Flow Graph
CGRRA	Coarse Grained Runtime Re-configurable Arrays
CPU	Central Processing Unit
CWB	CyberWorkBench
CWM	Combinational With memory
CWOM	Combinational Without Memory
CUT	Circuit Under Test
DFG	Data Flow Graph
DMA	Direct Memory Access
DRP	Dynamically Reconfigurable Processor

DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ESL	Electronic System Level
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GA	Genetic Algorithm
GUI	Graphical User Interface
GPU	Graphics Processing Unit
HW	Hardware
HWAcc	Hardware Accelerator
HDL	Hardware Description Language
HLS	High Level Synthesis
HT	hardware Trojan
IC	Integrated Circuit
IO	Input Output
IP	Intellectual Property
IR	Intermediate Representation
ITRS	International Technology Roadmap for Semiconductors
MPSoC	Multiple Processor System-on-Chip
QoR	Quality of Results
RAM	Random Access Memory
RE	Reverse Engineering
ROM	Read-Only Memory

RTL	Register Transfer Level
SWM	Sequential With Memory
SWOM	Sequential Without Memory
SoC	System-on-Chip
STC	State Transition Control
STP	Stream Transpose Processor
SW	Software
VHDL	Very High Speed IC Hardware Description Language
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

The advent of computers in the last century has revolutionized our society. The Computer is now ubiquitous. Until recently most of the efforts in computer security focused on the software (SW). It was assumed that the underlying hardware (HW) executing the SW was secure. Thus, most of the previous research focused on the development of techniques and methods to detect and avoid malicious SW programs *aka* viruses.

For decades, the underlying hardware used for information processing was considered trusted. With the increase in the complexity of integrated circuits (ICs) and the globalization of design and manufacturing process, the hardware is becoming more vulnerable to malicious activities. The main reason behind this is the insertion of hardware Trojans at different abstraction levels of the IC design process which affects the security and reliability of ICs. Companies were in the past vertically integrated and a single company would often design and manufacture their own ICs. This trend has nevertheless been disrupted due to the complexity of new ICs and the high costs of new fabs. Most IC design companies are now fabless and often rely on third party IPs to tape out their ICs at increasingly shorter design cycles. This opens the question of how trustworthy these ICs really are? A security

breakdown can happen at any stage. In particular, due to the malicious alteration of the IC, also called hardware Trojans.

Hardware Trojans can be defined as malicious modifications of an IC during the design or fabrication phase in an untrustworthy design house or foundry which results in incorrect behavior of an electronic device during run-time. In 2010, the US military discovered a hardware ?backdoor? in the microchips from missile to transponders which they had bought from China. The consequences could have been disastrous if they were left undetected [2]. An FBI investigation in 2004-2006 revealed counterfeit Cisco routers in US defense, finance, and university networks [3]. These fake routers had very low manufacturing quality and high failure rates. The FBI revealed that the US companies were procuring these routers directly from untrustworthy sources in foreign countries for a cheaper price. Similar incidents have been reported in [4–9]. Also, a study conducted by [10] estimated that the semiconductor industry loses up to $4 billion annually because of IP infringement. Hence, hardware security at different levels has become a major area of concern.

In the past, most semiconductor companies were Integrated Device Manufactures (IDMs). These companies used to develop their own semiconductor process technologies, owned and ran their own fabs and sold the finished Integrated Circuits (ICs) [11]. Modern fab buildings and maintenance costs have soared and have reached the order of billions of dollars. Thus, the industries have evolved to a *fabless* model, where specialized fabrication companies manufacture ICs for multiple companies. With the breakdown of Dennard scaling, ICs have started to be tailored to different application domains, aka. domain specific computing. Thus, most ICs are now heterogeneous System on Chips (SoCs) comprised of multiple in-house developed IPs and third party IPs (3PIPs) integrated onto the same die. A simplified flow of the IC design process (also called as VLSI design process) is shown in the Fig. 1.1. From the actual concept of the design to a final fabricated chip, the design has to undergo different steps, which can be classified as different design abstraction levels. An attacker (*e.g.* IP

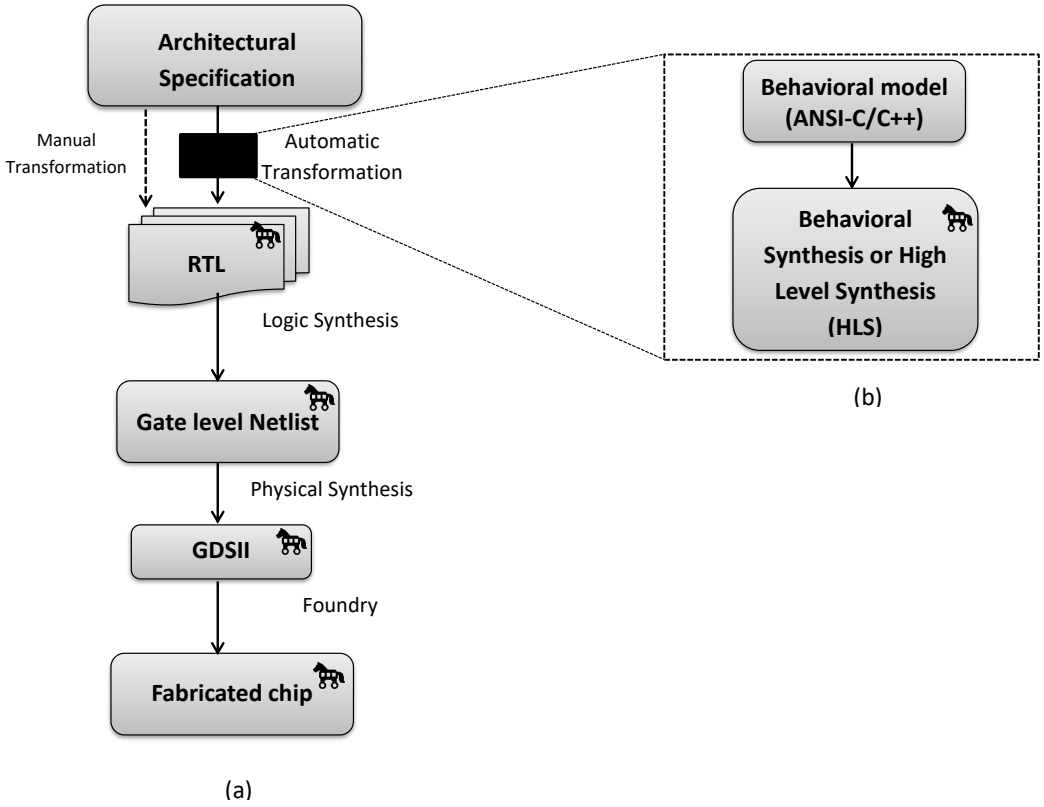


Figure 1.1: VLSI design Process

vendor) can insert the hardware Trojans intentionally in any of these abstraction levels (Fig. 1(a)) with a motive to disable or destroy a complete system at some future time, leak secret information or reduce the performance of the circuit when triggered. Conventional pre-silicon (simulation, code coverage) and post-silicon validation (functional/structural/random patterns) cannot reliably detect hardware Trojans. This is because these techniques aim at detecting defects that cause deviation from functional or parametric specifications and not the additional functionalities or modification in circuit operation triggered by uncommon events [12]. A past mantra, *design for testablility* has to be renamed to *design for trust*.

Fig. 1.1 shows the typical VLSI design process with different abstraction levels and the possibility of hardware Trojan insertion by an attacker at each abstraction level. A plethora of approaches and design methodologies for validating hardware security and trust at these

abstraction levels (Fig. 1.1(a)) have already been proposed in academia. Nevertheless, one area, which still requires much attention, is the behavioral level. In order to further reduce the time-to-market, IC vendors have started relying on High-Level Synthesis (HLS). HLS is a technique to convert an untimed behavioral language description such as ANSI-C of C++ to a register-Transfer-Level (RTL) description (e.g. VHDL or Verilog) that can efficiently execute it (Fig. 1.1(b)). This new VLSI design paradigm shift has also lead to a new market of third-party behavioral IPs (3PBIPs). The widespread use of HLS has triggered the IP design houses to start providing BIPs not only for virtual prototyping but also for the actual design implementation. Although the BIPs market is still in its infancy, some works have shown that the quality of HLS can rival the hand-coded RTL code [13][14]. This opens the door to a wider spread of this new technology. This new and promising design methodology leads to new threats in hardware security that need to be addressed. It is, therefore, imperative to study the trustworthiness of IC designed at the behavioral level. From single accelerator to complete SoC.

1.2 Contribution of this Thesis

This thesis investigates how to efficiently insert different types of hardware Trojans in BIPs and proposes techniques to detect these in stand alone BIPs as well as at the system level in SoCs. In addition, this thesis also proposes a technique to efficiently obfuscate the BIPs for HLS to protect the BIP provider from the illegal re-use of the BIP. The overview of the thesis is shown in the Fig. 1.2. The major contributions of this thesis are summarized as follows:

1. Develop an open source benchmark suite of synthesizable behavioral descriptions with different types of hardware Trojans which are difficult to detect using typical software testing techniques like profiling.

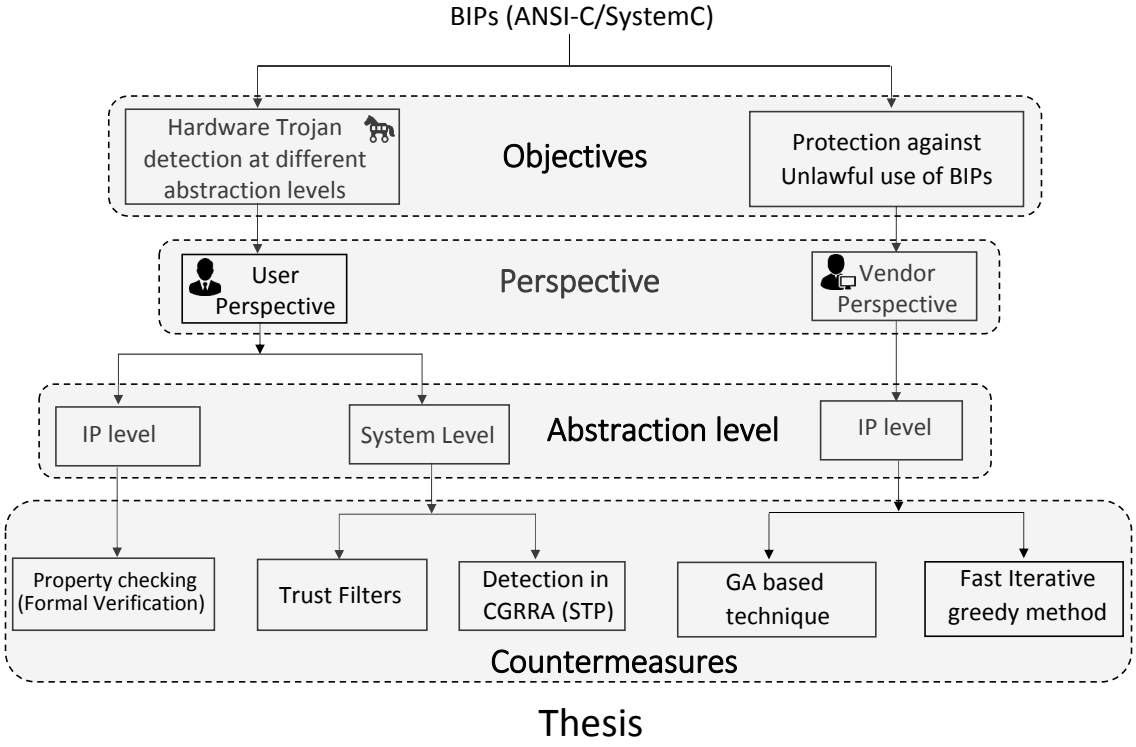


Figure 1.2: Overview of Thesis

2. Study the impact of source code obfuscation on the quality of results of BIPs for HLS and propose a quick and efficient method to maximize the source code obfuscation while minimizing the effect on the quality of results of the synthesized circuit due to redundant operations inserted by the obfuscator, which the HLS process cannot optimize. Two methods have been developed. A Genetic Algorithm (GA) based and a fast greedy heuristic method.

3. Introduce a fully automatic method to increase the coverage in BIPs to aid designers in finding HW Trojans in 3PBIPs using formal verification methods. In particular, property checking at the behavioral level, when the BIP is encrypted and when it is not.

4. Introduce a runtime system level method to detect hardware Trojans in behavioral multi-processor systems. The developed method makes use of accurate fast simulation models of the complete SoC under different workloads and extracts the timing of each

BIP mapped as a hardware accelerator slave in the system. This timing information is in turn used to verify that no hardware Trojan has been triggered at the untrusted accelerator.

5. Propose a mechanism to detect and avoid the hardware Trojan being triggered in runtime reconfigurable field programmable gate arrays (FPGAs), in particular, coarse-grained runtime reconfigurable array (CGRRA), which re-configures itself every clock cycle and which are programmed using HLS.

1.3 Thesis Structure

This thesis is divided into 6 chapters. The motivation and contributions of this thesis are provided in Chapter 1. Chapter 2 introduces how HLS works and describes in detail its three main steps :(1) allocation, (2) scheduling and (3) binding. The review of existing hardware Trojan detection methodologies for different abstraction levels and research gaps are discussed in Chapter 3. This chapter also gives a detailed definition of hardware Trojans, their taxonomy, and their types, and also presents the Synthesizable Security SystemC (S3CBench) benchmark suite developed for this work. This is the first open source benchmark suite of synthesizable behavioral descriptions with a wide variety of hardware Trojans. Chapter 4 is divided into two parts. The first part proposes an efficient method to maximize the BIP source code obfuscation that also minimizes the Quality of result (QoR) degradation due to the redundancy inserted by the obfuscator. In the second part, it presents a technique to detect the hardware Trojan in BIPs using property checking at the behavioral level. Chapter 5 presents the techniques developed to detect the hardware Trojans at the system level (behavioral MPSoCs). Also, it discusses the dynamically re-configurable processors (DRPs) which are used as hardware accelerators in SoCs and possible hardware Trojan scenarios in them. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

High Level Synthesis

High-Level Synthesis (HLS) can be described as the process of converting an un-timed behavioral description into an RTL description which can efficiently execute it. In contrast to traditional RTL design process, which makes use of low-level Hardware Descriptions Languages (HDLs) like VHDL/Verilog, HLS typically accepts as inputs high-level programming languages e.g. ANSI-C or C++, allowing designers to focus on the functional behavior instead of on the implementation details, which are time-consuming and error-prone. This leads to an increase in design productivity and thus reduce development cycles.

Raising the level of abstraction implies that fewer lines of code are required compared to RTL descriptions. This not only leads to shorter design cycles but also fewer bugs and makes it easier to verify and maintain the source code. Fig. 2.1 shows a graph plotting the code size v.s. generated gate size for different designs implemented in ANSI-C and in Verilog. A regression line is plotted onto the same graph for the two cases, where the slope indicates the number of gates per line of code obtained for the C-based design and Verilog case. The ratio between the two slopes indicates roughly the productivity increase. In this case, 28.73/4.20 $\approx$ 7 times [1].

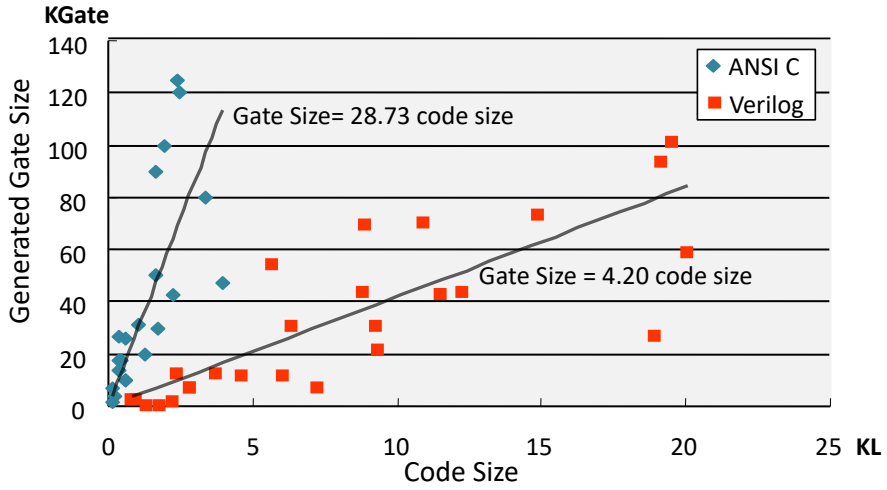


Figure 2.1: HLS productivity graph [1]

2.1 Design Flow

Fig. 2.2 summarizes the typical HLS design procedure. HLS takes as inputs the behavioral description to be synthesized, a set of design constraints and technology libraries of the target (ASIC or FPGA). In the first step, a formal model is created by parsing the high-level descriptions (C/C++/SystemC). This first step checks for syntax errors and creates as output an intermediate representation (IR). The next step performs some technology independent optimizations on this IR. This step is extremely important as large un-optimized circuits can be obtained if the input description is not optimized. Some of these optimizations include dead-code elimination and constant propagation. The output of this step is a Control Data Flow Graph (CDFG) which is taken as input by the three main HLS steps. These three main steps are allocation, scheduling, and binding. All these steps work interdependently.

In particular, allocation specifies the hardware resources that are necessary to implement the different operations in the behavioral description. Scheduling determines for each operation the time at which it should be performed such that no precedence constraint is violated. Binding provides a mapping from each operation to a specific functional unit and from each variable to a register. It should be noted that the final RTL circuit generated by

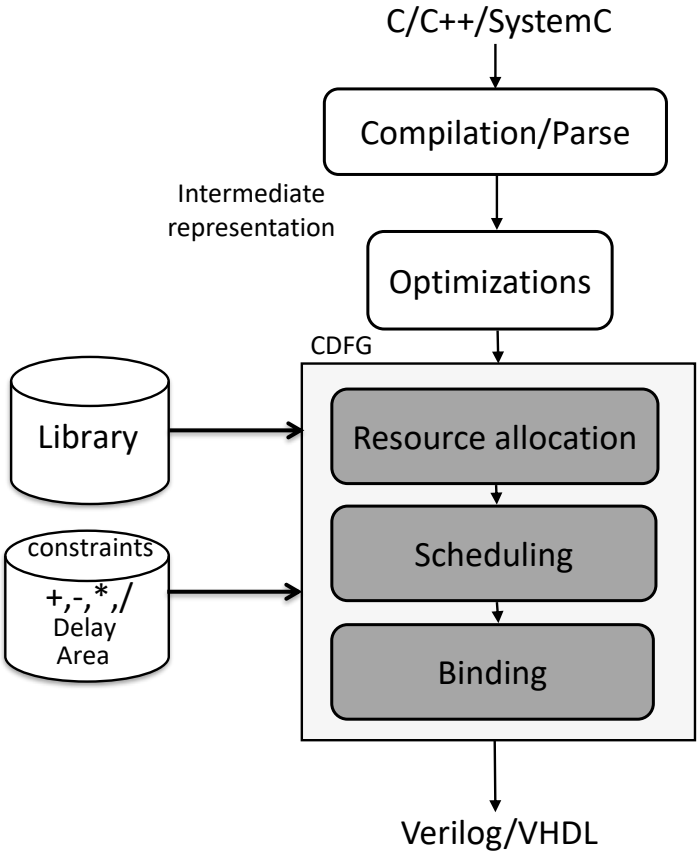


Figure 2.2: High Level Synthesis design flow

this procedure depends on the input specification, the hardware resources available in the library and the synthesis constraints.

2.1.1 Compilation/Parse

HLS has the main advantage that the design and verification can be performed using similar tools to other high-level software programming languages. This implies that the input description is compiled, simulated, and debugged using standard software environments like gcc, gdb and gprof. These attributes make HLS extremely convenient and easy to work with.

It should nevertheless be noted that there are some limitations of what can be synthesized in HLS. Some of the non-synthesizable constructs like dynamic memory allocation and recursion need to be considered when using HLS. Some HLS tools ignore some of these

constructs automatically while others create errors, suggesting the designers to use the synthesizable constructs.

```
int main(){
/* Inputs */
int in1, in2, in3, in4, in5, in6, in7, in8, in9, in10,in11;

/* Outputs */
int out1, out2, out3;

/* Variables */
int a, d, g;

a= in1+in2;
out1 = (a-in3) * 3;
out2 = in4 + in5 + in6;
d = in7 * in8;
g = d + in9 + in10;
out3 = in11 * 5 * g;
}
```

(a)

Adder	1
Multiplier	1

(b)

Adder	8
Multiplier	2

(c)

Figure 2.3: Functional unit allocation example for a ANSI-C code snippet: (a) ANSI-C code snippet (b) Functional unit and the usage constraint set to "minimum operator count"; (c) Functional unit and the usage constraint set to "maximum operator count"

2.1.2 Allocation

Allocation of hardware components (mainly functional units) is the first step after the behavioral description has been parsed. Once the behavioral description is parsed, a functional unit constraint file is generated. This file contains the type and the number of functional units required to map the particular input description. Typically, to extract the highest level of parallelism, the HLS process will try to parallelize the behavioral description as much as possible, and hence use multiple functional units as much as possible. The user then

has the opportunity to overwrite this constraint file manually, setting the maximum number of functional units that the synthesizer can instantiate, once the allocation stage has been completed.

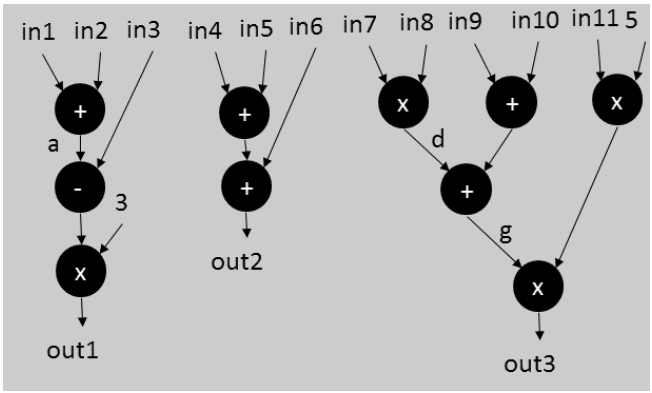
A simple example of a behavioral code snippet is shown in the Fig. 2.3(a), which performs multiple additions and multiplications to compute certain mathematical expressions. Fig. 2.3(c), shows the output of the resource allocation stage that maps individual operation in the source code to separate FUs to maximize parallelism. In this case, 8 32-bit signed adders and 2 32-bit signed multipliers are required. Fig. 2.3(b) shows the minimum number of FUs required to create a hardware circuit that can execute this description, requiring a single 32-bit signed adder and a single 32-bit signed multiplier. This can be either manually set by the user or through a synthesis option to minimize the resources. This example shows one of the advantages of raising the level of abstraction. In the first case, a large number of functional units inevitably lead to larger circuit area but the performance of the circuit will be higher due to decrease in the latency. In the latter case, the area will be reduced, as less FUs are required, but the performance will degrade. In particular, from 1 clock cycle to 6 clock cycles. In a theoretical sense, the maximum operator usage case leads to an approximate $5\times$ (average of the operators) larger in resources and $6\times$ faster in performance design. In the target architecture like FPGAs, this case does not always guarantee a better performance design due to the clock period constraint and also due to the multiplexers cost which is required to share the functional units.

2.1.3 Scheduling

Scheduling follows the resource allocation stage in the HLS process. Scheduling decides the allocation of operations which have to be executed in a particular clock cycle without violating the precedence constraint and data dependencies. Scheduling process basically follows the Data Flow Graph (DFG) restricted by resource and timing constraint as shown in

```
int main(){
int in1, in2, in3, in4,…,in11;
int a, d, g;
int out1, out2, out3;
a= in1+in2;
out1 = (a-in3) * 3;
out2 = in4 + in5 + in6;
d = in7 * in8;
g = d + in9 + in10;
out3 = in11 * 5 * g;
}
```

(a)



(b)

Figure 2.4: Scheduling example: (a) Example codes in ANSI-C; (b) DFG of source code given in (a)

Fig. 2.4. Independent operations can be scheduled at same clock cycle in parallel based on the target frequency constraint and the resources specified in the allocation stage. Chaining can be achieved by directly connecting the operations outputs to inputs of the next operations in the DFG operation. Additional registers are needed if the connection crosses multiple clock cycles. Multi-cycle operations are allowed only if the clock period is too small to accommodate the operation or the functional unit delay is too large.

Many scheduling algorithms have been purposed in the past. Most of them are heuristics as it has been shown that resource constraint scheduling is an $NP-hard$ problem [15–17]. The two most simple scheduling algorithms are As Soon As Possible (ASAP) and As Late As Possible (ALAP). Without violating the precedence, ASAP maps operations to their earliest possible start time while ALAP maps operations to the latest possible start time. ASAP scheduling leads to the shortest schedule if there does not exist any resource constraint, thus leading to an optimal solution.

Fig. 2.5(a) and (b) continues the example shown in the allocation stage and its DFG. In this case, the schedule is restricted to 1 adder and 1 multiplier (minimum resources usage). Fig. 2.5(a) shows the scheduling result of the ALAP schedule and Fig. 2.5(b) of the ASAP

schedule. It can be observed that the ALAP schedule leads to a better result with a latency of 6 clock cycles vs. the 7 clock cycles required by the ASAP schedule.

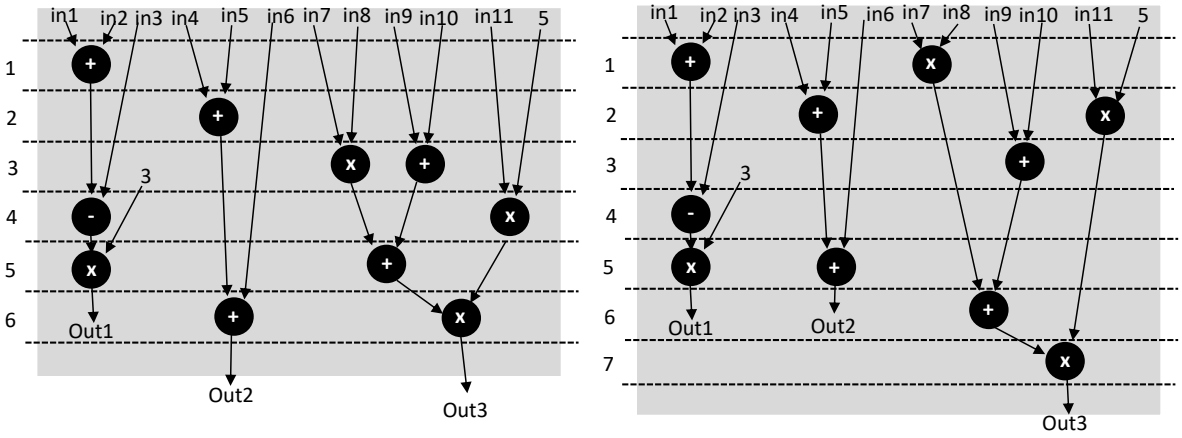


Figure 2.5: Scheduling results of Fig. 2.4 example with one adder and one multiplier constraint: (a) ALAP scheduling; (b) ASAP scheduling

If the constraint is changed to only one adder and one multiplier, only one adder and one multiplier can be executed in each clock cycle. This case can generate two different scheduling results based on whether the scheduler uses an ALAP or ASAP algorithm as shown in Figs. 2.5(a) and (b). As shown in the figure, ALAP schedules faster (6 clock cycles) than the ASAP scheduling algorithm (7 clock cycles). Nevertheless, both the algorithms cannot guarantee an optimal solution. From the above discussion, we can conclude that the choice of the scheduling algorithm has a greater impact on the synthesis result.

In general, scheduling algorithms can be broadly classified into data-flow-based scheduling (DF-based) and control-flow-based scheduling (CF-based). DF-based scheduling is best suitable for DSP and image processing applications. Scheduling algorithms can be further divided into two types: resource-constrained scheduling and time-constrained scheduling. List scheduling [18, 19] is one of the most widely used heuristics approaches to solve the resource-constraint scheduling problem. In this approach, ready operations are stored in a list according to certain priority function and are scheduled in order into the control state with the resources available. In order to address the time-constrained scheduling problem,

force-directed scheduling [20] heuristic approach is commonly used. In this approach, resource usage can be reduced by minimizing the force on the operations which balances the computations over the available time steps.

CF-based scheduling is mainly used to tackle the control-flow-intensive applications such as controllers and network protocol processors. The earliest approach that dealt with control-flow dominated description is path-based scheduling [21]. Loop-directed scheduling incorporates the depth-first search (DFS) approach to schedule the operations. This approach optimizes the average-case performance and also implicitly accounts for the loop repetition during the DFS. Wavesched [22] is another approach which explores and schedules the operations which are ready in a wave-propagation-like manner. The most recent scheduling algorithms combine the speculative code motions to extract the parallelisms which are not explicitly uncovered in the input description. In [23], the authors have introduced a set of speculative code transformations into a high-level synthesis framework.

The CF dominated descriptions are also described by a significant share of I/O timing constraints for complying to the external circuits. The earliest attempt to address minimum/-maximum timing constraints is relative scheduling [24]. In order to support the relative timings, the authors in [25] introduced behavioral templates which lock a number of operations into certain scheduling templates. A retiming-based approach to schedule the timed VHDL by behavioral code transformation has been employed in [26] without altering the original I/O timings. Many exact scheduling methods, such as the symbolic scheduling [27, 28], the ILP-based scheduling [29, 30] and the constraint-programming-based scheduling allows I/O timing constraints.

Although these scheduling algorithms have a certain level of efficiency in a specific class of applications, they lack the support of various design constraints. For example, DF-based schedulers will fail to effectively handle CF-intensive designs. Also, most of the scheduling techniques [21, 31, 22, 20] are not effective for larger designs because of exponential time

complexity during the worst case and many of them [21, 31, 22] fail to support the relative I/O timings.

2.1.4 Binding

Binding is the last stage in the HLS process. Binding maps each operation to a functional unit and each variable to a register. Each operation is mapped to a specific functional unit inherent in the functional unit constraint file which can execute the operation. One functional unit can be shared by many operations at different clock cycles (resource sharing). In order to achieve this, multiplexers are needed to assign the correct data to the input at the right time and deliver the output to the correct register. Binding is a very important stage in HLS since it affects the routability of the final circuit and hence the wire length and critical path. It has been shown in [32] that binding is an $NP-complete$ problem. Therefore, different heuristics have been proposed in the past.

Some of the existing heuristics to solve the binding problem divides the problem into module binding and register binding and solve independently using clique partitioning techniques [20, 33]. A few other heuristics [34–36] divide the binding problem into module binding and separate register binding along the timestep boundary. Weighted Bipartite Matching [35] is used to solve the subproblems. A few heuristics bind the operations and values one at a time [34, 36]. These approaches suffer a complete ignorance of the interactions between the register and module bindings. In [37], authors have proposed an iterative approach which starts from an arbitrary solution to the binding problem and later make the changes that have the potential for generating a better design. The approach proposed in [38] also starts with an arbitrary design but rips out parts of the design at random, and then reconstruct the design. Optimal designs can also be generated using the ILP approach described in [39] but is not suitable for large input specifications. Also, it is difficult to incorporate circuit-level information in the ILP formulation. A robust binding

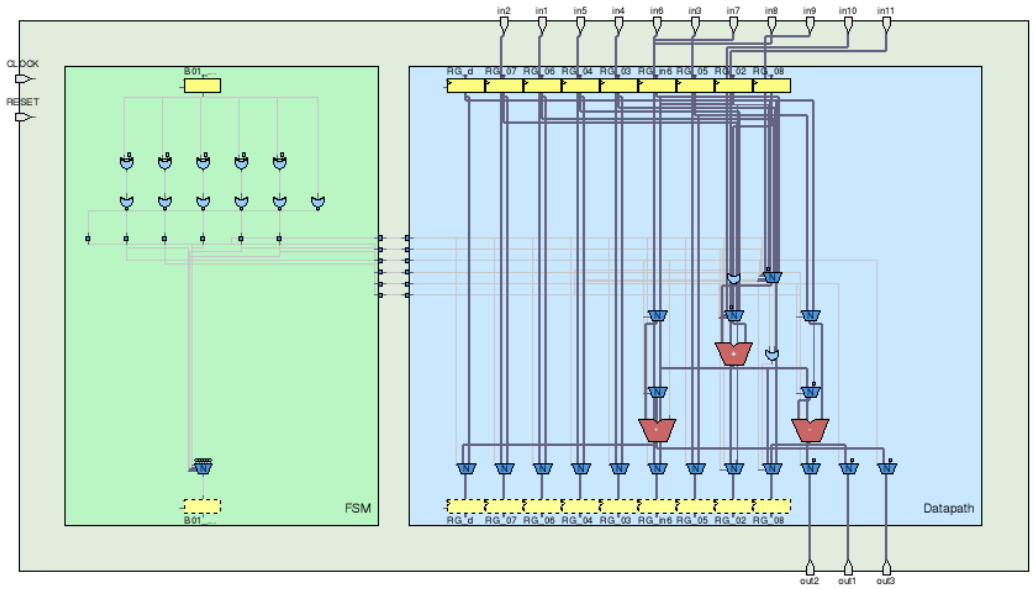


Figure 2.6: RTL netlist generated by HLS tool [40]

algorithm should consider various aspects into the account such as total area of the RTL design which includes module, register, and interconnection areas. The authors in [41–43] have pointed out the importance of including the information of the layout in the HLS tools. A high-performance and area-efficient design can be generated by using the wiring costs obtained from floorplan instead of estimates based on the number of wires. To handle the different objective functions such as area, performance, and testability, a binding heuristic must be adaptable. A dependency between the operator and register binding in the designs should also be considered apart from the adaptability.

2.1.5 RTL Generation

Based on the results obtained from the previously described steps, HLS synthesizes the behavioral description into RTL code (VHDL/Verilog). Fig. 2.6 shows the typical RTL architecture generated by a commercial HLS tool (CyberWorkBench). The RTL code generated is typically divided into a controller (FSM) and a data path. The controller generates the

control signals for the data-path, in order to guarantee the correct execution of the complete circuit. Also, the controller manages the passing of inputs to corresponding functional units at the correct state and generates the control signals for the multiplexers. It also decides the storage of data to either registers or RAM. The generated RTL (VHDL/Verilog) is passed to the logic synthesizer to get the gate-level netlist of the design.

2.2 Commercial HLS Tools

Although HLS is relatively new, its popularity has gained the attention of electronic design automation (EDA) tool vendors and there are numerous commercial HLS tools. Table 2.1 shows the prominent HLS tool vendors, their tool name and the input description supported. Since SystemC (C++ class of hardware modelling) is standardized by IEEE, it is supported by most of the HLS tool vendors. SystemC has the advantage that allows the modelling of hardware related constructs and concurrency. Thus, most of the work done in this thesis uses SystemC as input language.

Table 2.1: Popular HLS tools and their supported high-level languages

Vendor	Tool Name	Supported Languages
Altera	Intel HLS Compiler	C, C++
Cadence	Stratus	C, C++, SystemC
Mentor (Siemens group)	Catapult	C, C++, SystemC
NEC	CyberWorkBench	C, SystemC
Synphony C Compiler	Synopsys	C, System C
Xilinx	Vivado HLS	C, C++, SystemC

2.3 Summary

This chapter has discussed the benefits of HLS and its main steps. The widespread use of HLS has triggered IP design houses to start providing behavioral IPs (BIPs) for HLS not only for virtual prototyping but also for implementing the actual design. Although the market

for these IPs is still in its infancy, BIPs verification poses challenges not addressed yet. It is, therefore, important to study how secure these BIPs are and how to detect any malicious alteration of these IPs.

Chapter 3

Review of Hardware Security Techniques and S3CBench Benchmark Introduction

This chapter reviews the different types of hardware Trojan threats and describes in detail their structure, taxonomy, and how these can be implemented in 3PBIPs. A review of the existing hardware Trojan detection techniques and their limitations are then presented. Finally, this chapter introduces the first open source benchmark suite, developed in this thesis, of synthesizable behavioral descriptions with different types of hardware Trojans called Synthesizable Security SystemC (S3CBench).

3.1 Review of Hardware Security Topics

Fig. 3.1 shows the typical supply chain of semiconductor IC. The globalization of semiconductor design, manufacturing, and distribution in the supply chain raises the question about how secure the final hardware actually is. IC design process involves designing some IPs in-house, buying other IPs from a 3PIP vendor (to reduce the time-to-market), integrate both, and carry out synthesis and verification to generate the layout (GDSII) which is used as a blueprint in the foundry to manufacture the ICs. The manufactured ICs are then tested and

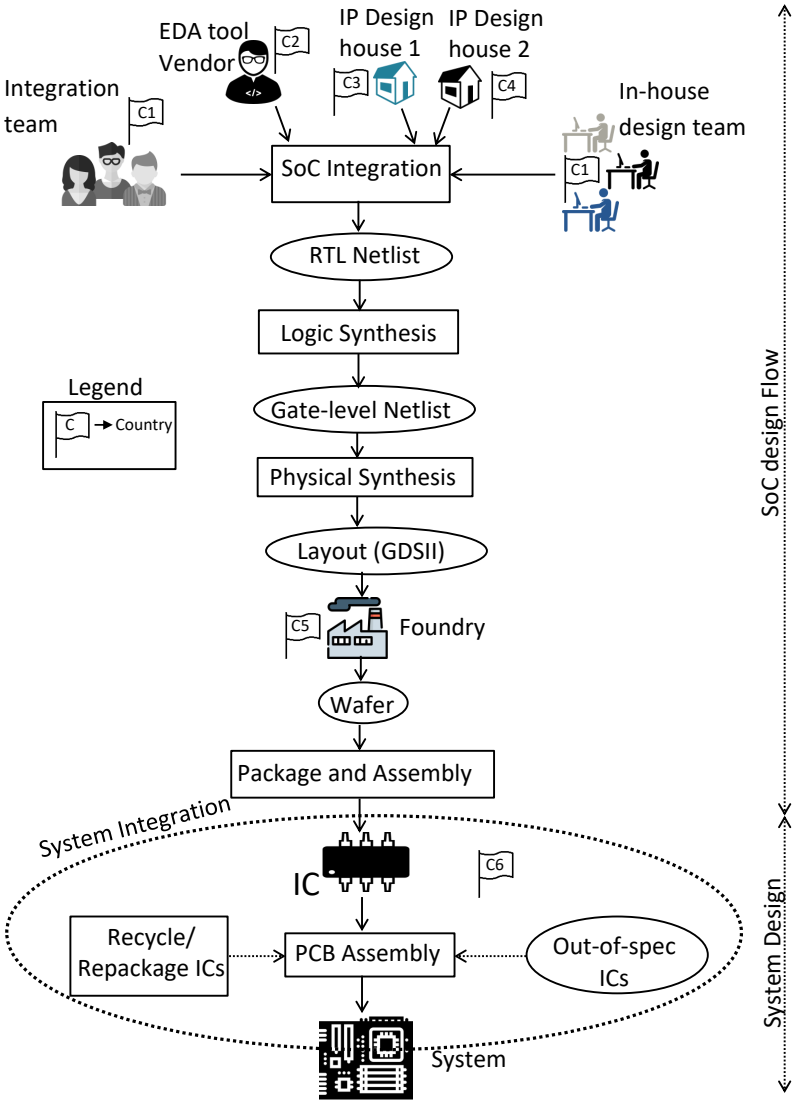


Figure 3.1: Supply Chain of Semiconductor

finally, the fault-free ICs are packaged and deployed. This process implies that there are multiple paths in the supply chain that an adversary can exploit to alter the IC. The following possible hardware security breaches can happen in the supply chain:

1. **Hardware Trojans:** A rogue IP vendor or an adversary in a design house or foundry may modify the existing design to insert malicious alterations that when triggered, disrupt the normal specified behavior of the circuit.

2. **IP piracy and IC overbuilding:** An IP user may illegally sell the IPs procured from the 3PIP vendor to other customers for a cheaper price without the consent of the rightful owner. An untrustworthy foundry may also overproduce the ICs to sell the excess ICs in the gray market.

3. **Reverse Engineering:** A competitor can reverse engineer the IC/IP to extract the hidden internal details. While reverse engineering is not a crime, in fact, it is protected by law, but it is not acceptable to do so without providing credits to the rightful IC/IP owner.

4. **Side-channel attacks:** Side-channel parameters like power consumption, current, time or delay, electromagnetic emission, acoustic information, optical information, etc. are exploited by the adversaries to extract the secret information in the IC, *e.g.* the key in cryptographic chips.

5. **Counterfeiting:** An adversary can sell used chips in the open market by remarking or repackaging the die, by cloning the ICs or by overbuilding it without the legal rights.

Fig. 3.2 shows the systematization of hardware security knowledge based on the different attack scenarios. The first column gives the goals of the attack, the middle column shows the type of attack, and the third column gives the attacker location within the supply chain [44].

Hardware security knowledge in terms of the attacks, countermeasures, and evaluation metrics is presented in Fig. 3.3. The first column gives the possible types of attack, the middle column summarizes the countermeasures, and the last column gives the metrics for evaluation of countermeasures.

These figures highlight the importance of hardware security. The next subsection describes the main security threats and revises the state of the art proposed countermeasures.

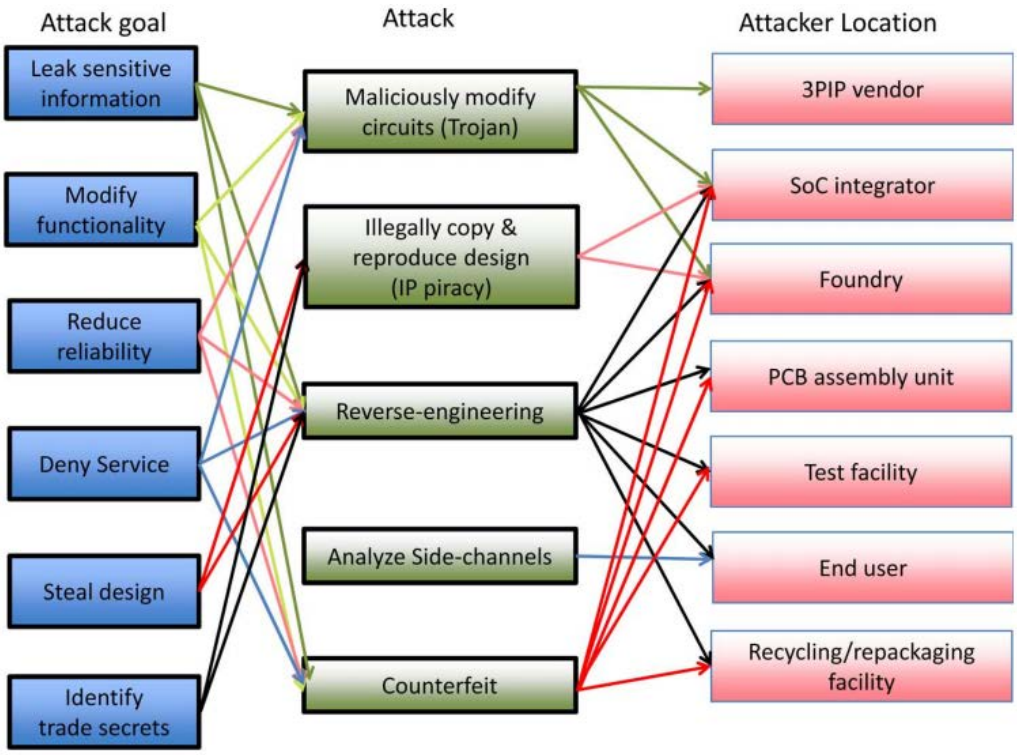


Figure 3.2: Hardware security systematization around the attack method. source: [44].

3.1.1 Hardware Trojans

As discussed in section 1.1, hardware Trojans are the malicious modifications made to the hardware circuit during either the design phase, in the IP design house, or during the fabrication phase at the foundry, which can cause a severe catastrophe at run-time if left undetected. Because of the stealthy nature and the rare activity of hardware Trojans, it is extremely difficult to detect them using the conventional verification or validation approaches.

State of-the-Art Defense Techniques

Most of the existing techniques target the detection of hardware Trojans inserted at the fabrication phase, i.e., in foundry [12, 45]. The possible ways to tackle these types of Trojan are through invasive and non-invasive detection techniques. In the invasive detection methods, the components of the circuit under test (CUT) becomes unusable after they undergo the

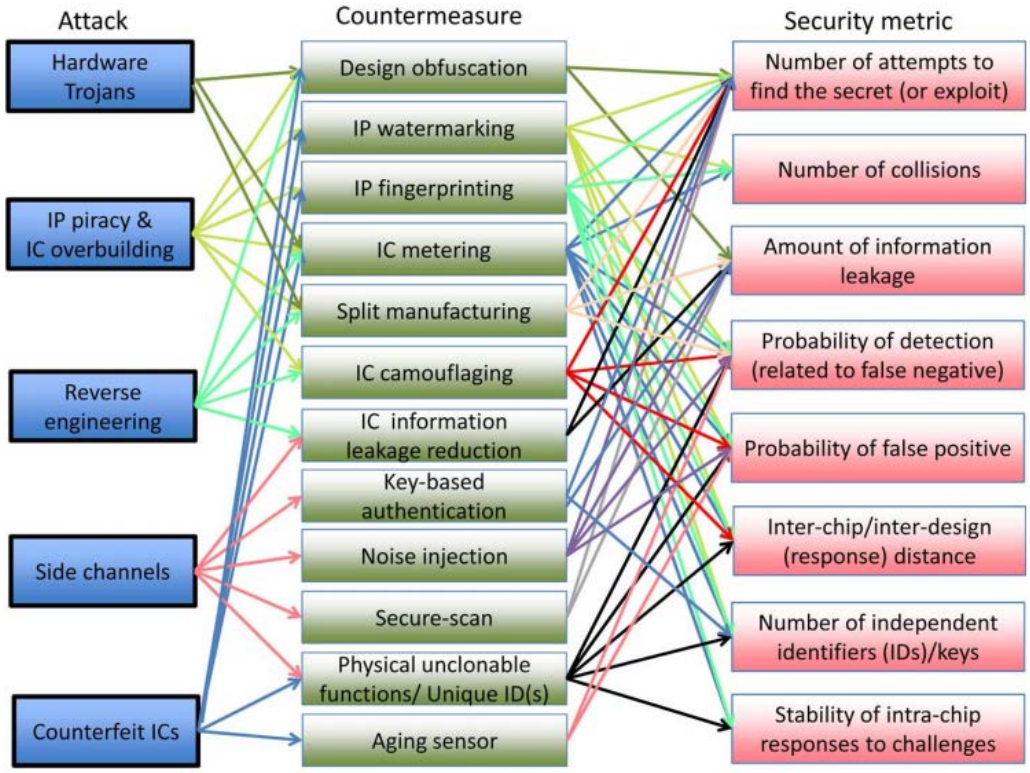


Figure 3.3: Hardware security knowledge in terms of the hardware-based attacks, countermeasures, and metrics for evaluation. source: [44].

detection process. Moreover, the precision measurement equipment used for Trojan detection is extremely costly.

External parametric and functional IC testing are used in non-invasive detection methods. In these methods, CUT is excited with input test patterns and the corresponding output values are measured along with the side channels [46] (e.g., delay, quiescent and dynamic leakage). Functional and statistical test variants are also used to detect Trojans in non-invasive methods. Gate-level characterization [47], path delay measurements [48], transient power analysis [49], thermal profiling [50], or combinations of them. All these techniques assume that the full details of the circuit design are available in addition to the statistical distribution of gate characteristics. The IC's expected characteristic value serves as a reference model to detect the Trojans.

Hardware Trojan detection in 3PIPs and defenses against insider attacks include self-monitoring [51] and static verification [52]. Trojans can be prevented from activation by scrambling the inputs that are supplied to the hardware units at runtime. This avoids the Trojan to procure the information needed to perform malicious action [53]. Authors in [54] proposed that SoC integrator and the 3PIP provider can agree on a set of security-related properties which the SoC integrator can verify.

3.1.2 IP Piracy and IC Overbuilding

An attacker in the integration house may pirate the 3PIP or can instantiate a number of 3PIPs than the acquired license. Also, the attacker in the foundry may pirate the 3PIP using the layout information or overbuild the IC.

State of-the-Art Defense Techniques

Different methods have been proposed to alleviate the problem of privacy and overbuilding. Watermarking, fingerprinting, obfuscation, metering, and split manufacturing.

1. **Watermarking:** In watermarking, a signature of the designer is embedded as an integral part of the design. The designer can claim the ownership later by revealing the watermark of an IC/IP. The types of watermark may include: addition of new states and transitions to the original finite state machine (FSM) [55], [56], [57], addition of a set of design and timing constraints which records the designer's signature [58], embed watermarking at physical design level [59], and FPGA designs [60].

   Graph partitioning is an optimization problem that has many applications in semiconductor design process [61], [62]. The watermark can be encoded as constraints during graph partitioning. The traits of the good watermarking scheme are: 1) it should adhere to the functionality of the circuit; 2) embedding cost should be low; 3) it should

have low implementation overhead; 4)it should have strong authorship proof, and 5) universal, i.e., it should be applicable to any design [58],[63].

2. **Fingerprinting:** Although the watermarking technique is capable of identifying the ownership of IP, it is difficult to track the rogue IP buyer from the unauthorized resold copies. This is due to the fact that the IP instances sold to the different buyer are identical and they have the same watermarks. This problem can be alleviated by embedding a unique and distinguishable mark (fingerprint) into each distributed IP instance [64]. The designer can later reveal the watermark to claim the ownership and signature of the buyer to disclose the piracy source. Fingerprinting can also be applied during different abstraction levels like high-level, logic and physical synthesis [64]. In order to identify whether a particular chip is fabricated at a particular foundry, authors in [65] have employed Kolmogorov-Smirnov statistical test for matching two probability distribution. Fingerprints derived from the SRAM memory cells can also be used to detect the piracy of an IP [66]. First dynamic fingerprinting technique [67] on sequential circuit IPs to enable both the owner and legal buyers of an IP embedded in a chip to be readily identified in the field.

   IC piracy can be overcome by registering the IC fingerprints using physically unclonable functions (PUFs) and comparing the suspicious ICs against the PUF fingerprint database [68]. PUF is a disordered physical system when interrogated by a challenge generates a unique output [69].

3. **Hardware Obfuscation:** Hardware obfuscation in hardware circuits, is concerned with protecting the semiconductor IPs from reverse engineering. A simple way of obfuscating the hardware circuit is by inserting additional gates (XOR/XNOR [70]) or memory elements [71]. The functionality of these hardware obfuscated circuits can be verified only by applying the correct values to the gates and memory.

Another way of achieving the obfuscation of hardware is by obfuscating the FSM of the design. Obfuscation of FSM can be achieved by adding extra states and transitions to it. In the original FSM, some states may be replicated [72], invalid transitions between the states may be added [73], [74], [75], unused states can be utilized [55], [76] or infinite states (also called as black hole states). All these techniques rely on the application of valid key to get the correct functionality.

4. **Hardware Metering:** It is defined as a set of security protocols that enable the design house to achieve post-fabrication control over their ICs. The hardware metering is classified into passive and active. Passive metering gives a unique identification of a chip, or for specifically tagging an ICs functionality so that it can be monitored passively [77]. The identified ICs are matched against a record in the database which reveals the unregistered and overbuilt ICs. Active metering provides an active way for the designers to control, enable, and disable the device.

5. **Split Manufacturing:** In split manufacturing, IC netlist is partitioned into multiple parts and each part is fabricated in a separate foundry. Since no foundry can get the access to full design, it is difficult for the adversary to maliciously modify or clone the design. One way of achieving the split manufacturing is by splitting the layout into front-end-of-line (FEOL) layers and back-end-of-line (BEOL) layers. Each of these layers is separately fabricated in different foundries. After the fabrication, FEOL and BEOL layers are aligned and integrated using electrical, mechanical, or optical methods [78].

3.1.3 Reverse Engineering (RE)

RE in semiconductor industry involves, identifying the technology used in the device [79], extracting the layout/netlist [80] and interpreting the functionality of the design [81]. RE is

legal, in fact, it is protected by the law. In the US, RE is protected by the Semiconductor Chip Protection Act. This law allows RE for teaching, analyzing, and techniques epitome in a hardware circuitry. Similar legislation can also be found in Japan, some European countries, and other countries. Outside the law, RE can be exploited to pirate a design, identify the device technology or fabricate the chip unlawfully without the consent of the rightful owner of the design. Depending on the need of the attacker, he/she can reverse engineer a design to any desired abstraction level (typically to physical design level or gate-netlist or RTL). The target level varies depending on the objective of the attacker. If the attacker has a motive to insert the Trojan, then the target abstraction level can be gate level or RTL.

State of-the-Art Defense Techniques

Camouflaging and obfuscation techniques can be used to mitigate RE. A 3PIP vendor can obfuscate his IP and a SoC integrator can obfuscate his design. A foundry can camouflage the layout of the design.

1. **Camouflaging:** During RE, image processing based technique is used to extract the gate netlist from the layout of a design. In camouflaging, layouts of the standard cells are designed to look alike which deceive the attacker by extracting the incorrect netlist. In [82], authors showed that by camouflaging layout of NAND and NOR cells to look identical, it is difficult for the attackers to extract the netlist with ease.

3.1.4 Side Channel Attacks

Side channel attacks are the most successful attacks in modern cryptographic systems for two main reasons. First, they target the weakness of the implementation of the cryptographic algorithms, not the algorithm themselves [83]. Therefore, the mathematically sound algorithms can become vulnerable to side channel attacks. second, these attacks are non-invasive, i.e., most of the time the attacker will not leave any traces of the attack. The attacker uses

the signals leaked from side channels during system's normal execution, to reveal the secret information. So, it is hard to detect and catch such attacks.

Side channel attacks have two phases, a marrying phase and a data analysis phase. During the marrying phase, the attacker will measure and monitor the system's physical characteristics when the system is running in the normal mode. The physical characteristics can be power consumption [84], current, timing or delay, electromagnetic (EM) emanations [85], acoustic information [86], optical information [87] etc. In the second phase, the attacker will perform data analysis on the collected side channel data to determine the on-chip secret information of interest. As we have already mentioned, side channel attacks are non-invasive, because they do not require to open up the chip. Some of the attacks do not have physical access to the chip either. For example, the EM attacks and acoustic attacks.

State of-the-Art Defense Techniques

1. **Update the secret Key:** The accumulation of side channel information by the adversary can be prevented by frequently updating the secret key [88]. In this method, a predefined sequence of keys along with the second synchronized timings are used to make sure that the sequence of keys is stable for both the exchange parties. If the defender knows the maximum information leakage rate per transmission, the keys can be changed before the amount of leakage information crosses the predefined threshold value [89].

2. **Noise Injection:** By artificially injecting noise, the signal to noise ratio of the side channel information can be reduced. It is very difficult for the attacker to capture the secret key from the noise induced side channel. A dummy circuitry which consumes a random amount of power is added to the system in the noise injection technique to trick the attacker from accessing the key [90]. Using the advanced signal processing techniques, the effect of noise can be alleviated [89].

3. **Leakage Reduction:** Information leakage from the power traces can be reduced by smoothing the power consumption using asynchronous logic [91], differential and dynamic logic [92], current mode logic [93], dual rail with precharge logic [94] etc.

4. **Secure Scan chains:** In this approach, mirror key registers are used in sensitive parts of the circuits to prevent illegal access to the value of sensitive registers in the test mode of operation [95]. Access to the scan chain for the users can be randomized by dividing the scan chains into smaller sub-chains [96]. A new countermeasure based on the dynamic obfuscation of scan data is proposed in [97].

3.1.5 Counterfeiting

A counterfeit IC/chip can be an unauthorized, remarked/recycled die, cloned design obtained through reverse engineering, overproduced chip or failed chip. These ICs can alter the functionality, degrade the performance or impact on the reliability of the chip when it is used in critical applications like e.g. military or aerospace [98]. The main motive of the fake IC vendor is typically financial, but pose a significant risk to the system.

State of-the-Art Defense Techniques

1. **Hardware Metering**

2. **IC Fingerprinting**

3. **Aging Sensors:** Whenever an electronic circuit operates in a functional mode, the transistors integrated on the chip age because of two effects; 1. Negative Bias Temperature Instability (NBTI); 2. Hot Carrier Injection (HCI). According to [46], [99], [100] NBTI and HCI could cause parametric shifts and circuit failures. By incorporating the fast ageing sensors in the circuit [101], recycled ICs can be detected.

4. **IP Watermarking**

The next section will explain in more detail what hardware Trojans are, their main structure and how to implement them at the behavioral level.

3.2 Hardware Trojan

Hardware Trojan can be defined as malicious modifications of an IC during the design or fabrication stage in an untrustworthy design house or foundry, which results in incorrect behavior of an electronic device during run-time. Hardware Trojans can be broadly classified based on either the insertion phase (e.g. specification, design, fabrication), the level of abstraction (e.g. system-level, RT-level, Gate-level, physical level), its activation mechanism (e.g. always on, internally triggered, externally triggered), its effects (e.g. downgrade performance, leak information, change of functionality) and/or its location (e.g. processor, memory, I/O). Several researchers have proposed different taxonomies based on these attributes [102], [45], [103]. Fig. 3.4 shows one popular taxonomy. According to [12], the framework should provide terminology and descriptive name for each class. Thus, the hardware Trojan taxonomy must meet two key criteria:

- *coverage*– it should classify all Trojans; none should exist "outside" the taxonomy; and

- *resolution*– it should distinguish Trojans with significantly different capabilities or required countermeasures.

Surprisingly enough most of the classifications do not include behavioral level at insertion phase level, which is the abstraction level used in this work.

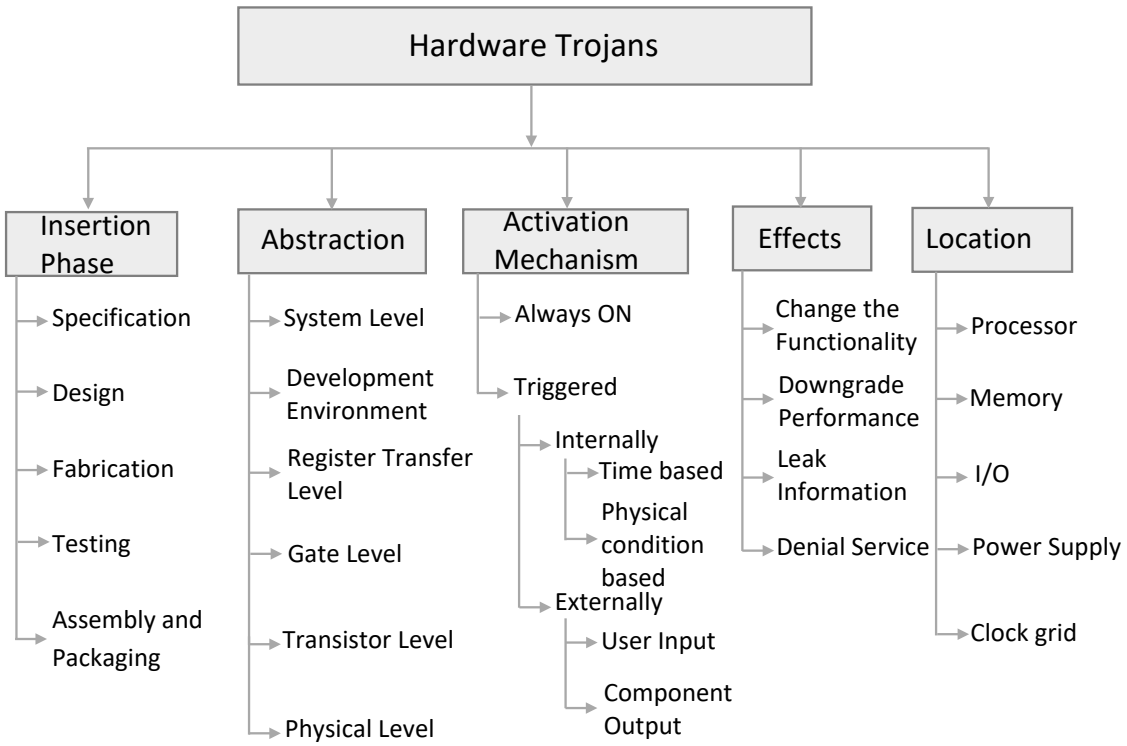


Figure 3.4: Hardware Trojan taxonomy based on different attributes [12]

Table 3.1: Hardware Trojan types overview and types addressed in this work

		Trigger mechanism	
		Combinational	Sequential
Payload	Without memory	✓	✓
	With memory	✓	✓

The fundamental structure of a Hardware Trojan consists of a trigger and a payload mechanism. The trigger mechanism monitors inputs, internal signal or state and stimulates the payload under certain conditions. The payload circuit then modifies the originally intended circuit behavior leading to the malfunction or performance degradation of the circuit. Hardware Trojan are generally triggered under very rare circumstances, which makes them very hard to detect during the verification or testing stages. The trigger mechanisms

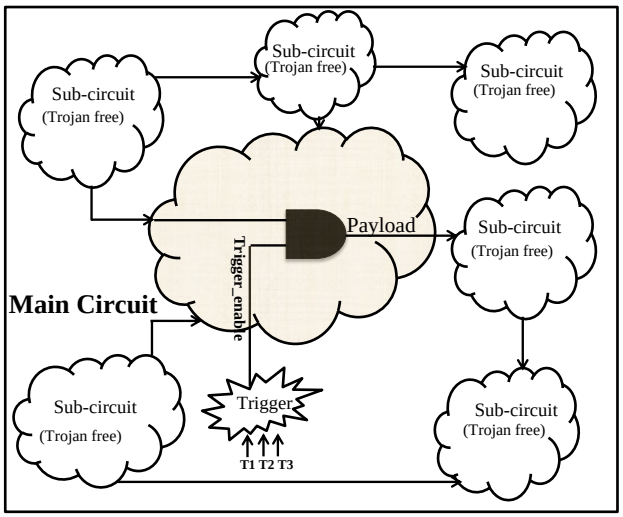


Figure 3.5: Hardware Trojan circuit example showing trigger and payload mechanisms

can be broadly classified as combinational or sequential. The payload mechanism, in turn, can be classified based on whether it can remember its activation status or not (payload with or without memory). Table 5.1 shows on overview of the different trigger and payload mechanisms and indicates the ones targeted in this work.

Fig. 3.5 shows one of the simplest structures of a hardware Trojan with its trigger and payload mechanisms. The trigger inputs (T1, T2, T3) come from various nets in the circuit. The payload taps signals from the original (Trojan-free) circuit and the output of the trigger. Since the trigger is expected to be activated under rare conditions, the payload output maintains the same value as Trojan-free circuit most of the time. However, when the trigger is activated, the payload output will inject an erroneous value into the circuit and cause an error at the output. In addition, some Trojans may not necessarily impact the function of the circuit, but rather execute a code that is designed to perform a specific function such as sending or receiving information to or from adversary from the outside or to degrade the performance of the circuit.

In the case of behavioral IPs, one can create extremely powerful hardware Trojans with very few lines of code. In [104], it was shown that a single line of C code generates 7 times more gates compared to a single line of RTL code.

3.3 Hardware Trojan in Behavioral IPs

The easiest and straight forward way to create a hardware Trojan at the behavioral level is by using *if-else*, *for loop*, *while loop* or *switch-case* clause. Fig. 3.6 shows an example of such a Trojan which triggers when the Sum of Product (SOP) result of the FIR filter reaches a pre-specified value (TRIGGER). In this case, the payload will set the output to 0. Although a very naïve example, this highlights how easy it is to create hardware Trojans at the behavioral level. One can argue that because of the increase in the level of abstraction in behavioral IPs,

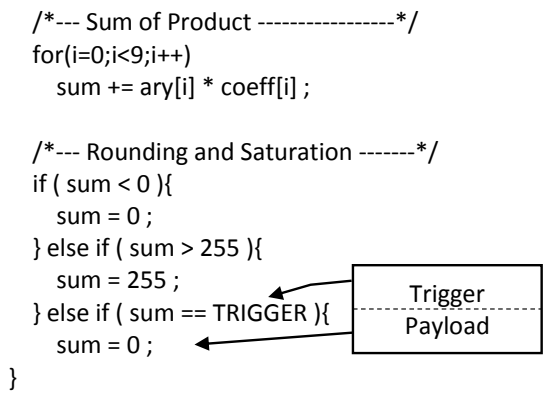


Figure 3.6: FIR behavioral IP HW Trojan example showing trigger and payload mechanisms

it is also easier to read the source code and hence to detect the hardware Trojan by simply reading the source code. At the same time, the complexity of these IPs varies and some of the BIPs are extremely complex e.g. encryption algorithms. Most SoC integrators (IP users) often rely on 3PIPs because they do not have the expertise to develop these complex designs. It is, therefore, reasonable to assume that they do not fully understand every source code line. Moreover, state of the art HLS tools [40] also allows 3PBIPs to be encrypted as shown in Fig. 3.7. In order for an IP vendor to encrypt his/her IPs, he/she is first required to contact the HLS tool vendor, who in turn issues an encryption key to the IP provider. The IP vendor can then specify using synthesis directives in the form or pragmas (comments), which parts of the source code to encrypt. Fig. 3.7 shows that *pragma encryption_start* and *pragma encryption_end* delimit the code section to be encrypted. This allows IP vendors to provide

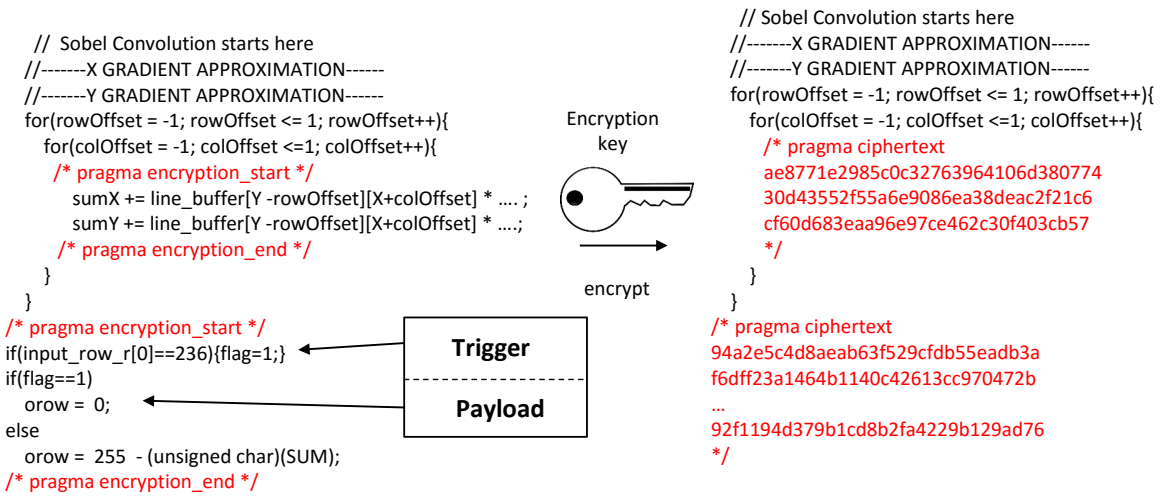


Figure 3.7: Sobel Behavioral IP HW Trojan example with encryption

IP users with some degree of reconfigurability, by e.g. not encrypting the I/O declaration to allow the IP user to set the bit-width of the I/Os. The main idea behind allowing 3PBIPs to be encrypted is to protect the IP vendor from releasing the complete source code, which in turn also affects the price of the IP as it can be sold much cheaper, as the IP user cannot fully re-use the IP. Obviously, the encrypted 3PBIP can only be decrypted and synthesized with that particular HLS tool. In addition, although the RTL code generated by most commercial HLS tools is un-encrypted and hence, the Trojan could be detected here, some vendors, especially FPGA vendors, tightly integrate the HLS tool with their logic synthesis flow so that this is not possible.

It is obvious that encrypting the IP poses significant security threats as hardware Trojans of any complexity can now be easily masked. This thesis thus also deals with the detection of encrypted 3PBIPs.

3.4 S3CBench:Synthesizable Security SystemC Benchmarks for High-Level Synthesis

An open source benchmark suite of synthesizable behavioral descriptions with different types of Hardware Trojan was developed to fully understand the mechanism of hardware

Trojan and in order to investigate efficient methods to detect them. A repository of RT-level benchmarks with different types of Hardware Trojan is available at the Trust-hub [105]. Unfortunately, this benchmark suite misses completely the behavioral abstraction level. Thus, this benchmark suite aims at bridging this gap by providing the first behavioral synthesis benchmark suite in a common language supported by all major HLS vendors (SystemC) which cover most of the hardware Trojan types. The designs have been created in such a way that the Hardware Trojan will always be executed when not triggered, thus, cannot be found using standard software profiling techniques.

3.4.1 S3CBench Overview

S3CBench (Synthesizable Security SystemC) benchmark suite, a freely available security synthesizable SystemC benchmark suite available at [106], consisting of 10 SystemC designs mainly taken from the S2Cbench benchmark suite [107] with different types of Trojan inserted in each one of them. SystemC is a C++ class originally developed to model HW. Its main features are that it has its own data types and that it allows modeling the concurrency by using multiple threads. SystemC has grown in popularity since it was standardized by the IEEE (IEEE 1666 Standard SystemC Language Reference Manual). Since then, it has been extended to allow the modeling of entire VLSI systems using SystemC's transaction level modeling (TLM) extension and to synthesize it into RTL (Verilog or VHDL) using High-Level Synthesis. The main reasons for this work to develop the benchmark suite is SystemC is that all major HLS vendors accept SystemC as its input language. Thus, the benchmarks can be synthesized unmodified with any of these tools. The main objective of S3CBench is to allow researchers to come up with different Trojan detection techniques for behavioral IP protection.

Table 3.2 summarizes the different benchmarks and the type of Trojans inserted in each case.

1. **CWOM:** The trigger mechanism in this type of Trojan is combinational and the payload does not have memory, i.e., the Trojan triggers for a particular input combination and the output malfunctions only while the input combination triggers the Trojan.

2. **CWM:** The trigger mechanism in this type of Trojan is also combinational but the payload has memory, i.e., once the Trojan triggers, the payload is active even after the trigger condition is not active anymore.

3. **SWOM:** The trigger mechanism in this type of Trojan is sequential and the payload does not have memory.

4. **SWM:** The trigger mechanism is also sequential but the payload has memory, i.e., once the Trojan triggers, its effect will last for a prolonged period of time. SWM is also called a *timebomb* Trojan.

The functionality of these hardware Trojans covers a wide range from leaking secret information, denial of service to the malfunctioning of the design. The benchmarks also cover all different trigger and payload mechanisms.

One traditional way of finding Hardware Trojans has been by profiling the source code and analyzing the lines of code not covered. Thus, the benchmarks have been designed in such a way that the hardware Trojan is always executed during normal operation of the design and hence a 100% coverage of the source code which describes the Hardware Trojan is achieved (except for the uart case). This is basically achieved by using a mixture of ternary operators and repeating loop executions.

A brief description of each HW Trojan benchmark is given below:

1.ADPCM: Adaptive Differential Pulse Code Modulation is one of the procedures for converting analog information to binary data. The ADPCM used in this work converts 16-bit Pulse Code Modulation (PCM) samples into 4-bit samples. The types of Trojan inserted, in this case, are SWOM and SWM. For the SWOM case, at regular intervals, the payload is

Table 3.2: Benchmarks and the type of Trojans inserted in each

Bench	Trojan type				Effect
	CWOM	CWM	SWOM	SWM	
ADPCM			✓	✓	mal
AES	✓				leak
BSORT	✓			✓	mal
DECIM				✓	mal
DISP	✓	✓			mal
FIR	✓				mal
INTERP	✓		✓	✓	mal
KASUMI	✓			✓	mal
SOBEL	✓	✓		✓	mal
UART				✓	denial

CWOM:Combinational Without Memory. **CWM**:Combinational With Memory. **SWOM**:Sequential Without Memory.**SWM**:Sequential With Memory.
denial: Denial of service. **leak**: Leak information. **mal**: Malfunction.

activated but does not have memory. In SWM, once the counter reaches a particular count value the Trojan will be triggered and left triggered indefinitely. In both the cases, the Trojan changes the functionality of the design when it triggers.

2.AES: Advanced Encryption Standard (AES) is a symmetric block cipher. The data size of the plain text is 128 bits and the allowed key lengths are 128,192 and 256 bits. The type of Trojan inserted, in this case, is CWOM. The main aim of this Trojan is to leak the secret keys through the output port for a particular rare input combination. Unlike the Hardware Trojans implemented in [105], which directly leak the secret keys through the output, which is extremely easy to detect, in our implementation we extended the work of [108] to leak the secret key. We modified the trigger mechanism compared to their original work by using a given input sequence as the trigger mechanism to leak the data required to calculate the secret key. A secret key leaking mechanism will be discussed in the next chapter.

3.Bsort: Bubble sort takes 8 input data and sorts them in ascending order. The types of Trojan inserted, in this case, are CWOM and CWM. In the CWOM case, based on the particular input trigger combination the payload replaces the original output with a different value. This mechanism will ruin the entire sorting process for that particular sequence. In the SWM case, the counter is incremented each time during the sorting process. Whenever the count value reaches the Trojan trigger condition, the sorting process will be completely disrupted by the Trojan payload indefinitely.

4.Decimation: Decimation is a 5-stage decimation filter. It has 5 FIR filters cascaded together in which the output of each stage is connected to the input of the next stage. The type of Trojan inserted, in this case, is SWM. The Trojan will hold its effect for a particular count interval after the first trigger. During the count interval, the output of the filter is affected by the Trojan's payload.

5.Disp: This algorithm measures the disparity in a stereoscopic image. The input to this design is a stereoscopic image and the output is a disparity estimated image. The types of Trojan inserted, in this case, are CWOM and CWM. In the CWOM case, the Trojan triggers for a particular input pixel combination resulting in a blurring of a corresponding output pixel of the disparity estimated image. For the CWM case, the Trojan triggers for a particular input pixel (trigger ON pixel) and the payload will be active until the input pixel matches with the another trigger pixel (trigger OFF pixel). It should be noted that the output pixels between *trigger ON* and *trigger OFF* pixel are completely blurred. Fig. 3.9 show visually the result of these Trojan.

6.FIR: FIR is a finite impulse response filter. In this case, it is a 10- tap FIR filter designed for 8- bit integer operations. The type of Trojan inserted, in this case, is CWOM. The Trojan triggers for a particular input combination resulting in a payload which outputs wrong results for the particularly applied input.

7.Interp: Interpolation is a 4-stage interpolation filter. The types of Trojan inserted, in this case, are CWOM, SWOM, and SWM. In the CWOM case, the Trojan will trigger whenever the output of the final stage of the FIR filter matches with the specified trigger value. The SWOM case trigger is based on a count value. In this case, the Trojan will trigger at regular count intervals. Similar to SWOM, the trigger mechanism of SWM is also based on a count value but once the trigger is activated the payload effect is long lasting.

8.Kasumi: Kasumi is a block cipher algorithm used in mobile communication systems. It has a 128-bit key and 64-bit input and output. The types of Trojan inserted, in this case, is CWOM and SWM. In the CWOM case, the output is compared with a particular trigger value and the output is overwritten when the output matches with a specified trigger value. In SWM case, the trigger mechanism is based on a count value. After the counter reaches a particular count value the output is overwritten with an already stored value.

9.Sobel: The Sobel is an edge detection 2D filter algorithm that takes as input a bitmap image and outputs a new image (bitmap) consisting of the edges of the original image. The types of Trojan inserted, in this case, are CWOM, CWM, and SWM. In the CWOM case, the Trojan gets activated when the user passes certain input pixel values and the output is overwritten with a new value. In the CWM case, once the Trojan is triggered, it results in the payload to be active for a longer duration, similar to the one in the disparity estimator. In the SWM case, the trigger mechanism is a counter and once it triggers the payload is active indefinitely. In this case, the Trojan only triggers when input images with larger resolutions than the Golden input that never triggers the Trojan are passed. Due to this behavior, it is called as *time-bomb* Trojan.

10.UART: Universal Asynchronous Receiver Transmitter is a communication IP used for serial data transmission. The type of Trojan inserted, in this case, is SWM. This type of Trojan falls under the category of denial of service in hardware Trojan classification. The trigger mechanism is based on a counter. Whenever the counter reaches the particular count

value, the transmission is delayed by certain clock intervals and it also results in the loss of intermediate data during the course of transmission.

3.5 Automatic Generation of Hardware Trojan Trigger Condition

One of the challenges when creating these benchmarks is to find robust trigger conditions for each design. This implies triggering the Trojan only when a rare event happens. Thus, in this work we propose a robust automatic trigger extraction mechanism. The method is composed of 3 steps as follow (also shown in algorithm 1):

Step 1: Taps Insertion and Evaluation. This first steps parses the original Trojan free behavioral description (C_{orig}) and instruments the internal variables with *taps* (line 2). These taps basically print to files the values of each individual variable being written to. Separate taps are created if the same variable is re-used throughout the code. Once the instrumentation takes place, the new behavioral description (C_{taps}) is compiled and executed with the test-vectors provided (TV_{orig}) (line 3). If no test vectors are given then N random vectors are used.

Step 2: Additional Test-vector Generation. This second step analyzes the test vectors in TV_{orig} and extracts combinations of inputs not present. New test-vectors can lead to unique internal variables' values, which in turn can serve as triggers for the Trojan. In order to avoid having to generate new test-vector randomly and verify if they are already present in TV_{orig}, a different approach is taken. For this, the method first sorts $TV_{orig} \rightarrow TV_{sort}$ (line 5). The method than continues by choosing n values not present in TV_{sort} and storing them in TV_{new} (line 6), where n is specified by the user externally. The new values generated are in turn appended to the original test-vectors to create an extended new test-vector $TV_{extend} = TV_{orig} \bigcup TV_{new}$ (line 7).

ALGORITHM 1: Automatic Trigger mechanism extraction

input : C_{orig}, TV_{orig}, n.
C_{orig} : Trojan free Behavioral IP
TV_{orig} : Test-vectors for C_{orig}
n : Number of new Test-vectors
output : $Trigger_X$
$Trigger_X$: Trigger condition for internal variable X

1 **Step 1: Taps insertion and Evaluation.**
2 C_{taps}=insert_taps(C_{orig});
3 $Val\_Taps[i]$=execute(C_{taps}, TV_{orig});
4 **Step 2: Additional Test-vector Generation.**
5 TV_{sort}=sort_TV(TV_{orig});
6 TV_{new}=gen_new_unique_TV(TV_{sort}, n);
7 TV_{extend}= $TV_{orig} \bigcup TV_{new}$;
8 **Step 3: Robust Trigger Condition Extraction.**
9 $Val_{extend}\_Taps[i]$=execute(C_{taps}, TV_{extend});
10 X = find_var_unique_value($Val\_extend_{Taps}[i]$);
11 return($Triggger_X$);

Step 3: Robust Trigger Condition Extraction. This last step, re-executes the instrumented design ((C_{taps})) with the new test-vector (line 9). It then compares the values of the internal variables for the original test-vectors (TV_{orig}) and the new ones ($TV_{extended}$) and selects the test-vectors and the internal signals which have a unique value when simulated with $TV_{extended}$ as compared to TV_{orig} (line 10). The unique value(S) of the internal signal can then by used as the trigger condition of the hardware Trojan. Because multiple internal variables might have unique values, the unique values not created with TV_{orig} are in turn analyzed and the variable that has a single unique value (rare event) or smallest number of unique values, generated when using $TV_{extended}$ is used to trigger the hardware Trojan, with the unique value being the trigger condition.

The entire flow is automated using perl scripts and is extremely fast as it is done at the behavioral (software) level.

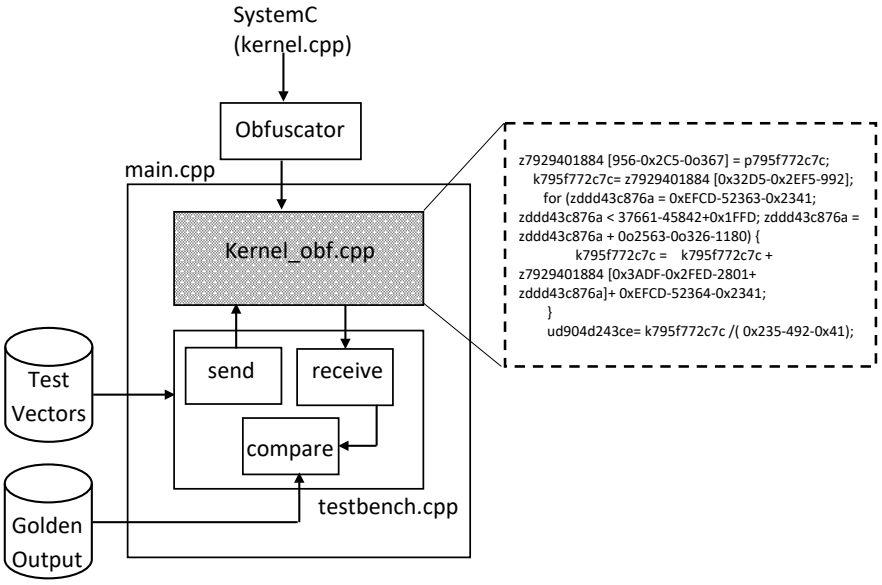


Figure 3.8: Framework of creating the obfuscated benchmark

3.6 Behavioral IP Obfuscation

BIPs normally have different price policies depending on the amount of disclosure of the IP. A BIP consumer can procure the complete source code and would not need the services of the BIP vendor anymore as he would have full control and visibility of the code. Alternative service models include the encryption of the BIP with a predefined set of constraints (e.g. IO bitwidths, synthesis pragmas/attributes), which the BIP consumer cannot modify. Another way is through obfuscation. Obfuscation can be defined as the intentional act of obscuring the functionality of a product to secure the intellectual property innate in the product. Obfuscation is an easy and inexpensive way to protect the IPs from illegal use without the permission from the rightful owner. When obfuscating an IP, a functional equivalent source file is generated, which is virtually impossible for humans to understand and extremely difficult to reverse-engineer. In this work, apart from the un-obfuscated benchmarks presented, we have also included obfuscated version of the benchmarks with hardware Trojans. The obfuscated benchmarks have the same functionality as the un-obfuscated version with the hardware Trojans embedded. This is to demonstrate how the rogue IP vendor in the supply chain can

completely deceive the BIP consumer by inserting the hardware Trojans in the obfuscated IPs, making it even more difficult to detect using the traditional verification approaches. Commercial HLS tools also allow the encryption of these BIPs. This approach was not used in this work as this is tool dependent.

Fig. 3.8 shows an overview of the obfuscated version of the benchmark. As shown in the figure, not every file is obfuscated. Only the synthesizable kernel is, while the files related to the benchmark (TB.cpp, Bench.h) are left unobfuscated. The obfuscator used in this work is Stunnix C/C++ [109]. Adding the obfuscated version of the benchmarks should further highlight the problems of detecting hardware Trojans at the behavioral level. One could argue that previously developed detection methods targeting the detection at the RT-level could be used for the synthesized IP. Two main problems prevent this. First, FPGA vendors provide their own HLS tools (e.g. Xilinx Vivado HLS and Intel's OpenCL SDK). The main problem is that these tools do not generate RTL code as they intend to lock users using only their FPGAs. Secondly, the RTL code generated by commercial HLS tools are often not easy to read as it is machine generated. Most of the hardware Trojan techniques do not actually find a Trojan, but flag a location which could potentially belong to a Trojan , but that could also be a bug or simply un-executed code that the given test vectors is not covering. The next step requires the manual inspection to confirm the existence of Trojan or not. If the behavioral IP is either obfuscated or the RTL is machine generated, this manual inspection is much harder or impossible.

3.6.1 Experimental Results

Each benchmark comes with a SystemC testbench with golden inputs and golden outputs to verify that each design works correctly. The Hardware Trojan never triggers for these inputs. Table 3.3 & 3.4 characterizes each of the designs in terms of they type of Trojan (cols 2 to 4 of Table 3.3), lines of C code of the original design and the *infected* code (cols 5 and 6 of

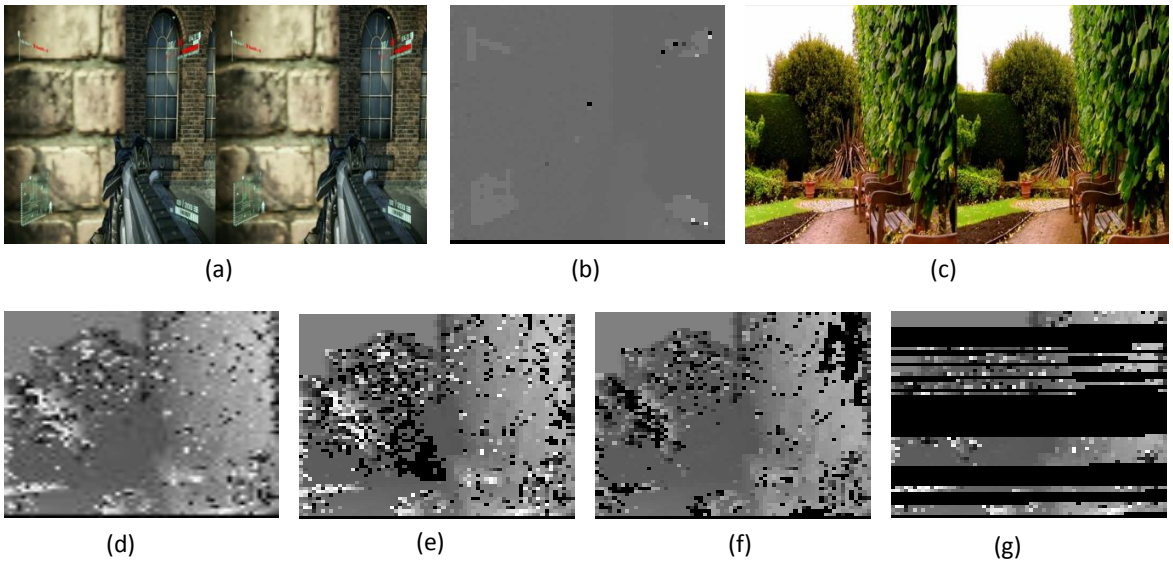


Figure 3.9: Disparity Estimator (a) original input stereo image (1920 × 1080) (b) expected golden output (64 × 64) (c) input stereo image (1920 × 1080) (d) expected output (e) HW Trojan effects (combinational trigger+no memory payload I)(64 × 64) (f) HW Trojan (combinational trigger+ no memory payload II) (64 × 64) (g) HW Trojan effects (combinational trigger+ memory payload) (64 × 64)

Table 3.3), area of the synthesized original circuit and the *infected* version after HLS (cols 7 to 9 of Table 3.3) and finally the coverage with and without the Trojan (cols 5 to 7 of Table 3.4). The reported area is obtained by synthesizing each design with a commercial HLS tool (CyberWorkBench from NEC [40]). The target technology used is Nangate 45nm and the target synthesis frequency 100 MHz.

From the results, it can be observed that a few lines of code (on average 0.5% more) can be transformed into powerful hardware Trojan. On an average, the area of these Hardware Trojans is 2.92% larger, while in most cases does not exceed 1%. One could argue that this extra area introduced by the Hardware Trojan can be used as a marker to detect it compared to the golden, Trojan free design. Although true in nature, the BIP market is extremely small and infancy, and it is difficult to procure golden IPs from another 3PBIP vendor. This leads to difficulty in comparison. One other typical way to detect Hardware Trojan is through coverage reports. Previous work identifies lines of code not executed and assume that this

Table 3.3: Experimental Results comparing benchmark with and without Hardware Trojan for area

Bench	Trigger	Payload	Effect	C orig	C Trojan	Area Orig [μm^2]	Area Trojan [μm^2]	Δ_{Area}
adpcm	Sequential	no memory	SWOM	186	186	3,560	3,570	0.28%
	Sequential	memory	SWM		187		3,602	1.17%
aes	combinational	no memory	CWOM	371	380	36,904	38,326	3.85%
bsort	combinational	no memory	CWOM	78	78	2,455	2,596	5.74%
	Sequential	memory	SWM		78		2,634	7.29%
decim	Sequential	memory	SWM	298	298	71,967	75,516	4.93%
disp	combinational	no memory	CWOM	284	284	72,366	73,671	1.80%
	combinational	memory	CWM		285		78,152	7.99%
fir	combinational	no memory	CWOM	75	75	9,987	10,023	0.36%
interp	combinational	no memory	CWOM	108	108	46,239	46,254	0.03%
	Sequential	no memory	SWOM		108		46,256	0.03%
	Sequential	memory	SWM		109		46,325	0.18%
kasumi	combinational	no memory	CWOM	288	288	65,571	67,572	3.05%
	Sequential	memory	SWM		288		67,895	3.54%
sobel	combinational	no memory	CWOM	173	173	1,762	1,781	1.07%
	combinational	memory	CWM		175		1,823	3.46%
	sequential	memory	SWM		175		1,782	1.13%
uart	sequential	memory	SWM	160	164	1,614	1,724	6.81%
Geomean				177	178	13,057	13,494	
Avg.								2.92%

piece of code is a potential Trojan. Thus, these benchmarks have been written in such a way that when the Hardware Trojan is enabled their code is always executed, except for the uart design. The rest achieves 100% code coverage for the Trojan description. Columns 5 to 7 of Table 3.4 show the source code coverage.

To easily understand the impact of the Hardware Trojan on the results, we have added the images of the results obtained for the disparity estimator case in Fig. 3.9. Fig. 3.9(a) shows the input image and Fig. 3.9(b) the expected golden output, both provided by the BIP vendor. In all of the cases with hardware Trojan, the expected output is always obtained. Fig. 3.9(d) shows the picture that the disparity estimator should generate during the normal operation for another input (Fig. 3.9(c)), which is *not* provided by the IP vendor and hence can trigger the Hardware Trojan. Figs. 3.9(e) and (f) shows the consequence of hardware Trojan with the CWOM case. The effect of the Trojan can be observed with entire regions overwritten by

Table 3.4: Experimental Results comparing benchmark with and without Hardware Trojan for coverage

Bench	Trigger	Payload	Effect	Coverage orig	Coverage with Trojan	Coverage of Trojan code
adpcm	Sequential	no memory	SWOM	100%	100%	100%
	Sequential	memory	SWM		100%	100%
aes	combinational	no memory	CWOM	85.3%	87.1%	100%
bsort	combinational	no memory	CWOM	100%	100%	100%
	Sequential	memory	SWM		100%	100%
decim	Sequential	memory	SWM	100%	100%	100%
disp	combinational	no memory	CWOM	98.2%	98.2%	100%
	combinational	memory	CWM		98.2%	100%
fir	combinational	no memory	CWOM	100%	100%	100%
interp	combinational	no memory	CWOM	96.43%	96.43%	100%
	Sequential	no memory	SWOM		96.43%	100%
	Sequential	memory	SWM		96.67%	100%
kasumi	combinational	no memory	CWOM	98.8%	98.8%	100%
	Sequential	memory	SWM		98.8%	100%
sobel	combinational	no memory	CWOM	100%	100%	100%
	combinational	memory	CWM		100%	100%
	sequential	memory	SWM		100%	100%
uart	sequential	memory	SWM	91.4%	91.9%	50%
Avg.				97.0 %	97.9%	97.22%

it. Fig. 3.9(g) shows the result for a CWM memory case. This is the most powerful Trojan which enables the payload to be active over a longer duration of time. Because of the page limit, we can not discuss the results of every benchmark but encourage the users to use the benchmarks available at [106].

3.7 Summary

This chapter has reviewed the most common hardware security threats and has provided insight about the most recent work done in this area. The chapter has then focused on hardware Trojans, their taxonomy and how they can be inserted in behavioral IPs. In particular, hardware Trojan detection techniques related to our work and their limitations. Most of the hardware Trojan detection techniques rely on trust-hub benchmarks which cover

the wide range of Trojans at different abstraction level but does not cover the behavioral level. Hence, in this chapter, the first security benchmark suite in a behavioral language supported by all major HLS vendors, with different types of Hardware Trojan which produces different effects has been presented. The benchmarks are open source and will continue to be expanded to include more designs in the future.

Chapter 4

Behavioral Intellectual Property (BIP) protection

This chapter addresses the issue of BIP protection from two different angles: The first, from the BIP vendor who needs protection from the unlawful use of the BIP. The second, from the BIP user/consumer to make sure that the 3PBIP purchased is Trojan free. To tackle the first issue, source code obfuscation is used to *hide* the BIP from the consumer, but as it will be shown in this chapter, poses some significant efficiency problems that this work addresses. In the second case, a hardware Trojan detection technique for behavioral IPs through property checking techniques is presented.

4.1 Efficient Behavioral Intellectual Properties Source Code Obfuscation for High-Level Synthesis

The globalization of IC design and manufacturing process poses serious concerns about their trustworthiness and security. It is nowadays virtually impossible to fully design and

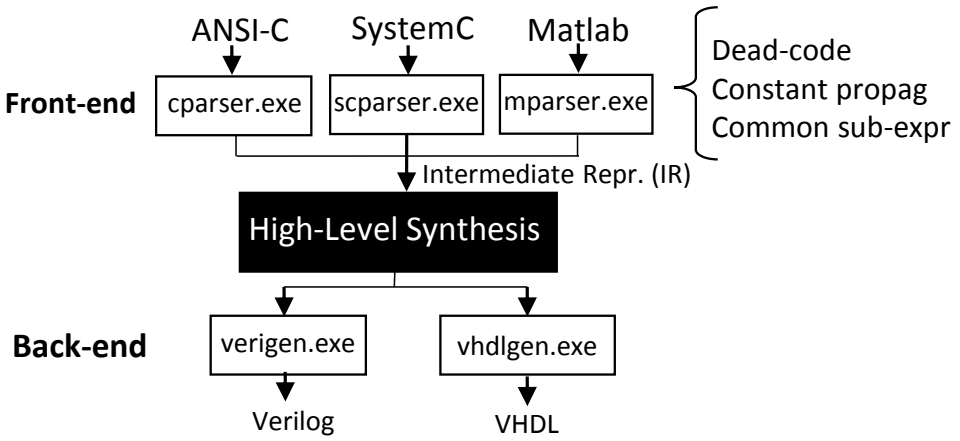


Figure 4.1: Typical HLS flow overview.

manufacture an IC in-house using in-house tools. With the increased time-to-market pressure, companies further rely on third parties for the development of ICs.

Most ICs are now heterogeneous System on Chips (SoCs) comprised of multiple in-house developed IPs and third party IPs (3PIPs) integrated onto the same chip. IC vendors have also embraced High-Level Synthesis (HLS) in order to further reduce the time-to-market. Thus, this has opened the way to the market of third-party behavioral IPs (3PBIP).

One of the main challenges for BIP providers that is hampering the expansion of the usage of these BIPs, is how to protect the IP from being reused illegally. A study conducted by [10] estimated that the semiconductor industry loses up to $4 billion annually because of IP infringement.

Also, BIPs normally have different price policies depending on the amount of disclosure of the IP. A BIP consumer can purchase the complete source code and would not need the services of the BIP provider anymore as he would have full control and visibility of the code. An alternative service model includes the encryption of the BIP with a predefined set of constraints (e.g. I/O bit widths, synthesis pragmas), which the BIP consumer cannot modify. Thus, any future alterations would require a new license agreement and a new purchase. The first type of service is obviously much more expensive than the second, typically 10-100×.

An additional issue that needs to be addressed, is when a company is interested in purchasing a BIP it often contacts the BIP provider to evaluate the quality of the IP (i.e., area, delay, latency and/or throughput). During the evaluation process, the BIP cannot be made visible as the BIP consumer would not need to end up purchasing the BIP. Thus, mechanisms to protect the IP are required.

Obfuscation is an easy and inexpensive way that is being used for this purpose. When obfuscating an IP, a functional equivalent source file is generated, which is virtually impossible for humans to understand and extremely difficult to reverse-engineer. The obfuscation process typically removes comments, renames variables, and adds redundant expressions. There is a multitude of free or inexpensive obfuscators, which makes this an ideal method to protect Behavioral IPs (BIPs) for High-Level Synthesis (HLS). The main problem when using obfuscators is that commercial HLS tools often make use of dedicated parsers, developed in-house, as front-ends. Fig. 4.1 shows an example of a typical commercial HLS tool. Different parsers parse different input languages (e.g. ANSI-C, SystemC/C+, Matlab or Java) into an intermediate internal representation (IR). This first step is responsible for traditional technology independent compiler optimizations, e.g. dead-code elimination, constant propagation, and common sub-expression eliminations.

The main synthesizer part, in turn, reads this IR and performs the main HLS steps (resource allocation, scheduling and binding) and the backend, in turn, generates either Verilog or VHDL. This flow has some unique advantages that make it extremely popular. Being able to take as input the IR instead of the source code directly allows supporting new languages in the future by simply adding a new front-end parser. The main disadvantage is that each parser needs to be often developed in-house by the EDA company, which might not be an expert in this field, and thus leading to suboptimal designs due to not being able to fully optimize the input description. This is particularly important in the case of obfuscation as

this processes typically inserts multiple redundant operations to make the code less readable, which in turn leads to increase in area, delay, and latency of the synthesized circuit.

This work analyzes the impact of obfuscation on the quality of results (QoR) in HLS, and to propose effective obfuscation methods which do no lead to any area and performance degradations. In particular, the main contribution of this work can be summarized as follows:

1. Study the impact of different level of obfuscation of BIP for different commercial HLS parsers on the QoR.

2. Propose a fast and efficient heuristic approach to obfuscate these BIPs with no overheads.

4.1.1 Motivational Example

Fig. 4.2 shows graphically how the level of obfuscation affects the area of the synthesized circuit when two different parsers from the same commercial HLS tool are used [1]. The results shown correspond to different benchmarks, taken from the open source S2Cbench[107] benchmark suite. Because the commercial HLS tool used supports ANSI-C and SystemC, the benchmark suite was converted to ANSI-C in order to allow the comparison.

As shown in the Figs. 4.2(a)-(f), the increase in the level of obfuscation leads to a monotonically increase in the area for the ANSI-C case, whereas the area stays constant for the SystemC version. This implies that the independent parsers perform a different level of compiler optimizations and hence lead to different synthesis results. It is, therefore, important to develop techniques to maximize the obfuscation of behavioral descriptions for HLS while maintaining the QoR compared to the unobfuscated version.

[1] Tool name cannot be disclosed due to confidentiality agreement

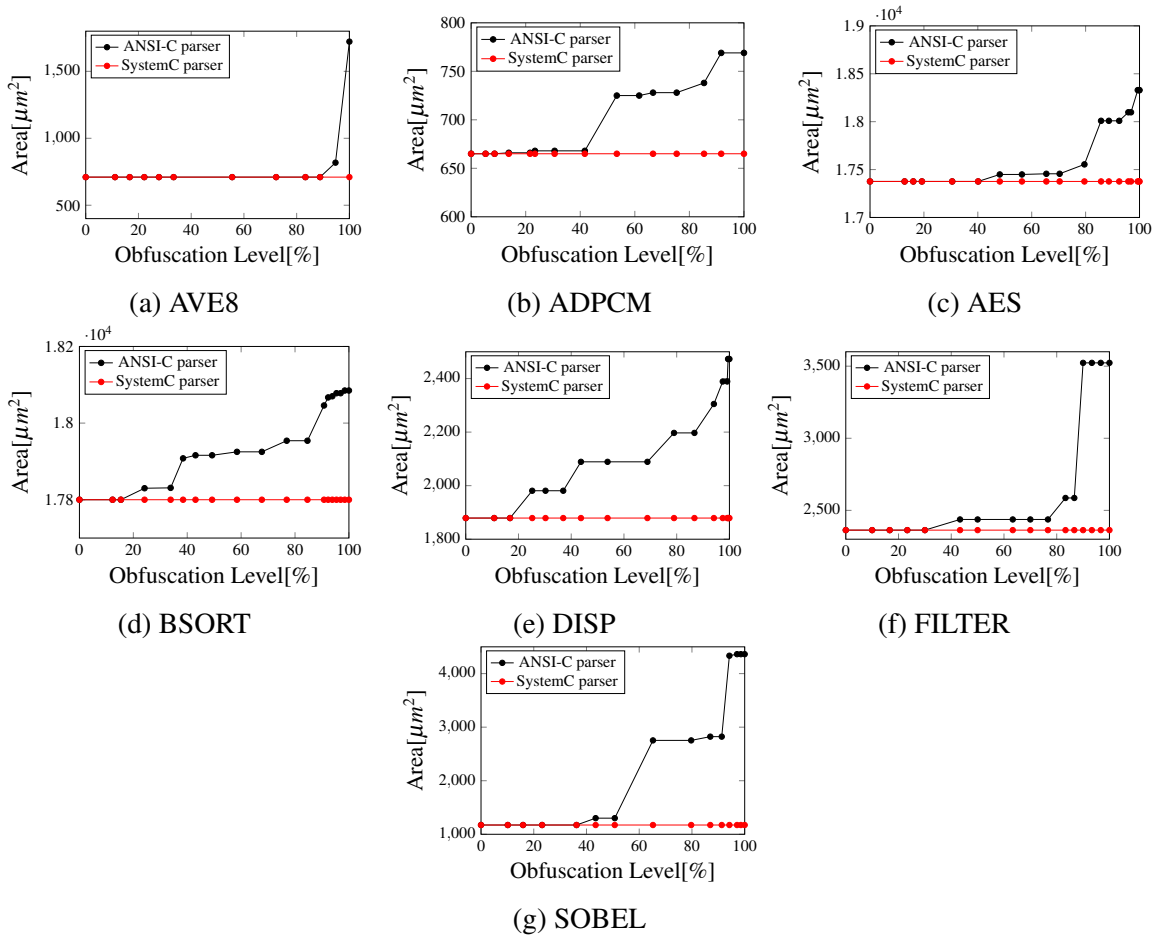


Figure 4.2: Area degradation of different benchmarks [107] with the increase in level of obfuscation

4.1.2 Previous work

In the field of the consumer market, the companies spend a significant amount of time and resources in reverse engineering their adversary's product to understand the internal design details. This is a routine activity which happens in industries ranging from automotive and computer software to electronics. Reverse engineering is not a crime. In fact, it is protected by law. The major threat in reverse engineering is, it can be used in a number of malicious ways. Consider a company A exploits the IP of company B by incorporating in their own products without giving credit or compensation to company A. Although passive IP protection methods like copyrights, patents, and watermarking try to safeguard the rightful

owner, it is not effective because of the degree of enforcement law varies from one part of the world to another. Hence an active approach to IP protection is required, of which obfuscation is one of the vital part [110].

Obfuscation is an easy and inexpensive way for protecting the hardware IPs. Most previous work in the field of hardware IP obfuscation deal with hard IPs [111–117]. In this thesis, we focus on the efficient obfuscation of behavioral IPs, which comes under the category of soft IPs.

Previous work in this area proposed by [74] and [118] represents RTL code as a data flow [74] or state transition matrix [118] graph. The graph is then modified with additional states (which are also called as key states) in the finite state machine representation of the code which should be passed through with the aid of a key sequence [74] or a code word [118]. The IP will start functioning only after the application of correct keys; otherwise, the IP will be stuck in a futile, obfuscated state.

Similar to the software case, soft IPs in hardware can also be obfuscated in terms of readability and intelligibility. The authors in [119] incorporated the techniques such as loop transformation (loop unrolling), statement reordering, conversion of parallel processing to sequential, etc., to make the VHDL source code nearly impossible to read and yet functionally identical to the original source code. Kainth M et al. [120] in their work, took the leverage of control flow flattening [121] [122] approach to break the function and loop into blocks and convert to "switch" statements, which implies the control flow of the program becomes invincible to an attack.

Although the behavioral IP codes are similar to the software codes, the software obfuscation techniques can not be directly portable for hardware obfuscation as it does not take into account the degradation introduced by the *unoptimized* parsers, which in turn lead to unoptimized hardware circuits.

4.1.3 Obfuscation

Obfuscation can be defined as the intentional act of obscuring the functionality of a product to secure the intellectual property innate in the product. Formally:

Definition: O is an obfuscator which can transform a program P into its obfuscated version *O(P)* which has the same functionality F as P, such that the $F\{P\} = F\{O(P)\}$ and such that it is unintelligible for an adversary who is trying to recover P from *O(P)*.

Software obfuscation is different from traditional hardware obfuscation. In software, code obfuscation changes its structures which make the program difficult to understand for the attackers to reverse engineer yet preserving the functionality. Code obfuscation is the most popular alternative to encryption as it does not require any inverse transformation nor the need for an encryption key to decrypt the cipher text.

In the hardware domain, obfuscation is mainly concerned with protecting the functionality of a hardware by changing the micro-architecture of the hardware module to be obfuscated, such that it cannot be reverse engineered. The taxonomy of the hardware obfuscation techniques at different abstraction level (RTL level, gate level, layout) and also some emerging techniques can be found at [110]. This taxonomy does not include the behavioral level, mainly because at this level the micro-architecture has not been fixed and obfuscation at this level is virtually the same as software obfuscation.

As mentioned previously, the obfuscation of BIPs is similar to the obfuscation of software programs. Most of the BIP vendors supply IPs which are either written in ANSI-C or SystemC (C++). This has the additional side benefit that any commercially available software obfuscators can be used to obfuscate these BIPs.

Fig. 4.3 shows a simple example of the obfuscation of a BIP code snippet which computes the average of eight numbers using a commercially available C/C++ obfuscator [109]. As shown in the figure, the obfuscated version of the original code snippet is extremely difficult to understand and thus for the purpose of reverse engineering.

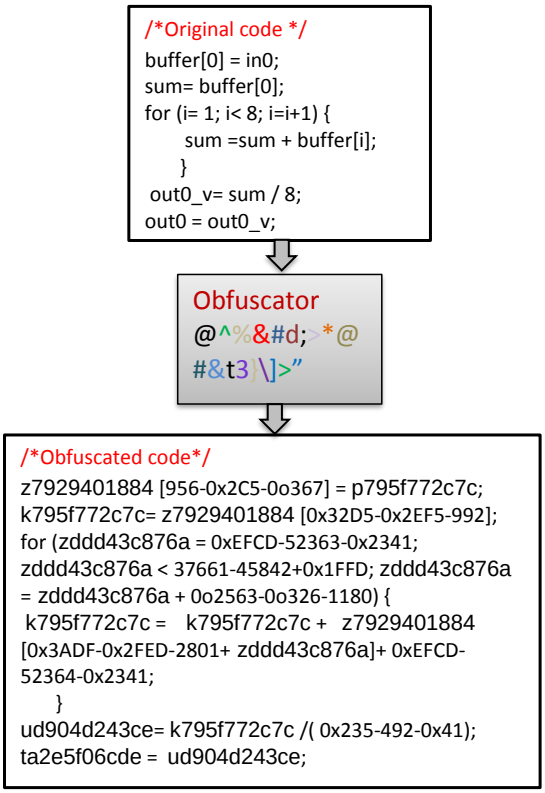


Figure 4.3: Behavioral IP obfuscation example

4.1.4 Typical Obfuscation Process

In this section, a brief description about the techniques used by typical software obfuscators like [109], which can be used to obfuscate BIPs will be discussed. In particular:

1. **Mangle integers and mathematical expressions:** Mangling the integers and mathematical expressions make the attackers difficult to read and analyze the code. If we revert back to the Fig. 4.3, the integer "8" in the for loop has been replaced by "37661-45842+0x1FFD". Also, the "sum" expression, "sum =sum + buffer[i]; " inside the for loop has been replaced by "k795f772c7c = k795f772c7c + z7929401884 [0x3ADF-0x2FED-2801+ zddd43c876a]+ 0xEFCD-52364-0x2341;". Although both the expressions evaluate to the same original value and expression, the mangled version is difficult to understand for the attackers.

2. **Strip spaces:** Writing a program is an art. Usually, any program whether it is a software program or hardware program is written legitimately to make it understandable (readable) for the readers or the programmer itself when he/she wants to read/modify the code again. Trimming the extra spaces inserted in the code is one of the techniques in the obfuscation which accounts for poor readability for the attackers.

3. **Replacing the identifiers and the signals:** This is one of the important features of most of the obfuscator. There are different ways of replacing the identifiers, signals or variables of a program in obfuscation.

   (a) *Using a set of characters such as I and l or set of O and 0:* Replace all the identifiers with the combination of either I and l or O and 0.

   (b) *Use of mix case:* Since C and C++ are case sensitive, a single large identifier is chosen as an identifier to replace all the other identifiers by mixing the case of the identifier.

   (c) *Use of md5:* It is one of the powerful technique to replace the identifier. In this, message direct algorithm is used to replace the identifier with a generated hash value. Eg. "buffer" in the Fig. 4.3 is replaced by "z7929401884" which affects the readability of the code.

   (d) *Use of prefix:* It is one of the simplest and not so efficient ways of replacing the identifiers. In this case, every identifier is prefixed with a certain word. For example, sum (and all the identifiers) in the Fig. 4.3 is changed to _confidential_sum.

4. **Comments:** Deleting the useful comments or the use of nonsensical comments can deceive the hackers from understanding the source code.

Obfuscators typically mix all of these techniques. These techniques work extremely well with mature compilers (e.g. gcc and g++) as these are based on extremely robust parsers

which have been developed for decades. Hence, the binary code (*.exe*) of obfuscated software version and the unobfuscated versions are the same.

As discussed in the motivational example, the parsers of different EDA tools can lead to different results for the obfuscated version of the code. Although the functionality is preserved, the obfuscated version incurs overheads in terms of area and delay because these parsers are not as robust as the software compiler ones, as often the input language is a subset of software languages.

4.1.5 Reason for the overhead

Although the functionality of the obfuscated behavioral IP is equal to that of the un-obfuscated version, the compiler fails to optimize some of the mangled mathematical expression which leads to extra circuit generation after synthesis. For example, in Fig. 4.3 "sum =sum + buffer[i];" has been obfuscated to "k795f772c7c = k795f772c7c + z7929401884 [0x3ADF-0x2FED-2801+ zddd43c876a]+ 0xEFCD-52364-0x2341;". The obfuscated part "0xEFCD-52364-0x2341" is a dead code. It is the responsibility of the compiler to optimize the code using dead code elimination techniques. Failed to optimize will lead to the generation of an extra circuit after the synthesis which in turn increases the latency or critical path. For the above example, the synthesizer may generate some extra adders for the un-optimized expression.

4.1.6 Proposed Method to Minimize QoR Degradation due to Obfuscation

As shown in the motivational example, there is a trade-off between QoR and the level of obfuscation. In this section, we discuss two methods to minimize the QoR degradation due to the obfuscation process. The first method is a meta-heuristic based on genetic algorithm

(GA), which has been shown to lead to very good results for multi-objective optimization problems like this one and the second method is based on a fast iterative greedy method.

4.1.7 GA-based Obfuscation

GA was first proposed by John Holland in [123] to find solutions to problems which are computationally intractable. Due to its modular nature, it can be applied to a wide range of practical problems. In this work, GA is used to find a design with the highest-level of obfuscation with minimum QoR degradation, although by changing the cost function it is easy to adapt the method to find different configurations with unique obfuscation vs. degradation trade-offs. Fig. 4.4 graphically summarizes the two main steps involved in our proposed GA-based obfuscator, where every gene corresponds to a single line in the BIP, which either has to be obfuscated or left unobfuscated.

The inputs to the GA-based explorer are the original un-obfuscated code P, the completely obfuscated code $O(P)$, and *lightly* obfuscated version $LO(P)$, which does not include mangling, as this obfuscation technique has been shown to be the culprit for the QoR degradation. Thus, $LO(P)(QoR) = P(QoR)$. This *lightly* obfuscated version is used as a basis to further obfuscate the BIP or not and is important to avoid any syntax errors when individual lines of code are fully obfuscated.

Step 1: Initial Population Generation: In this first step, an initial random population is generated, each new configuration with random obfuscation level O_L. This is done by randomly parsing the BIP line by line and deciding if the given line should be obfuscated or not. The obfuscation probability is an input parameter to the explorer. In this case, it is set to 30%. This will generate a random pool of parents to start generating the different offsprings.

Each newly generated configuration is synthesized (HLS), the QoR={Area, Delay} extracted and the cost C of this new solution computed using $C = \alpha \times Area - \beta \times O_L$ (scaling the area and O_L) . The values for α and β can be either fixed to find a particular design or

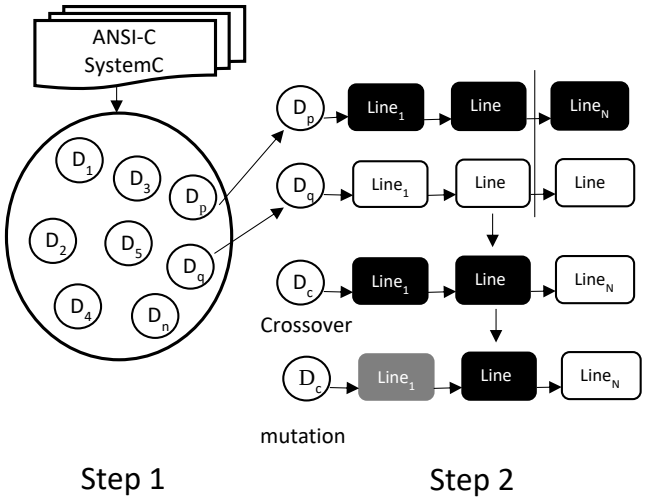


Figure 4.4: Genetic Algorithm (GA) method overview.

can be dynamically updated to explore the entire search space, e.g. $\alpha = 1$ and $\beta = 0$ for the area dominant designs and $\alpha = 0$ and $\beta = 1$ to find obfuscation dominant designs, and $\alpha = 0.5$ and $\beta = 0.5$ when a compromise between QoR degradation and level of obfuscation is sought. Finally, the cost obtained is stored in a cost array C_{cost}. Once k parent designs are generated, the method continues by pruning the $xth\%$ of designs with largest cost in order to reduce the search space and hence the running time. In this case $x = 50$.

Step 2: Offspring Generation: Two main functions are responsible for generating an offspring. Crossover and mutation. Crossover is performed between the two parent designs chosen randomly from within the pruned population. Although there are many techniques to chose parents, this work makes use of *adjacent* designs technique. A random cut-off point is chosen between the two parents and the lines of code obfuscated in each parent are *passed to the child*.

The mutation is performed on the offspring by flipping the lines of code to be obfuscated or not. This method goes from the first line to the last line and computes if the line has to be mutated or not. A random probability P_{rand} is used for this purpose. If P_{rand} is less than a given threshold (10% in this case), then the corresponding line of the BIP is obfuscated.

Finally, the offspring is synthesized and the cost for this configuration computed. The above steps have to be carried till the stopping criterion is met. Where in this work, N=20.

4.1.8 Fast Iterative-Greedy Method

Although the GA-based explorer can find a good set of designs with unique trade-offs, the running time of the explorer can be considerable. Thus, a faster method is needed. In this section, we propose a fast heuristic explorer which generates a single design with better QoR and obfuscation level. The algorithm 2 gives a brief overview of the proposed method. The detailed description of the algorithm is given below:

The inputs to the algorithm are the original code P, the completely obfuscated code $O(P)$, the *lightly* obfuscated code $LO(P)$ (explained in the GA based obfuscation), and the QoR synthesis results of the different variants (area and delay). The output is the version with the highest amount of obfuscation and lowest QoR degradation (best-obfuscated design $BO(P)$). The method is based on three main step, as follows:

Step 1: Extract Potential Degradation Sources: As already explained in the previous section, the mangled integers and expressions are the main culprits for the QoR degradation, as the ANSI-C parser cannot optimize these redundant operations efficiently. Thus, in this step, the completely obfuscated code $O(P)$ is parsed and the lines of code which contains mangled integers and expressions. The line numbers, where these operations take place are stored in the array $array_1$ (lines 1-2).

Step 2: Identify Real Sources of Degradation: Because not all the mangled lines (expressions) contribute toward QoR degradation, in this step, the proposed method analyzes the previously extracted lines stored in $array_1$, and determines which are the lines actually contribute towards any QoR degradation. For this, each line in $array_1$ is unobfuscated separately and the new behavioral description $O(P_{single})$ is synthesized (HLS). The QoR is in turn compared with the unobfuscated original description ($O(P)$) to determine if the

ALGORITHM 2: Fast Iterative-Greedy Obfuscation heuristic

input : $Original\_code \rightarrow P$, $Completely\_obfuscate\_code \rightarrow O(P)$,
$Lightly\_obfuscated\_code \rightarrow LO(P)$,
QoR of P, QoR of O(P)
output : The best-obfuscated design BO(P)

1 **Step 1:** Search for the lines in $O(P)$ which contains mangled integers and expressions
2 $array_1 = search\_lines(O(P))$;
3 **Step 2:** Set i=0;
4 **while** *!End of* $array_1$ **do**
5 Read the file O(P);
6 **if** *Line number of O(P) ==* $array_1[i]$ **then**
7 $O(P_{single}) = Unobfuscate\_line(i, O(P))$;
8 $QoR(O(P_{single})) = HLS(O(P_{single}))$;
9 **if** $QoR(O(P_{single}) < QoR(O(P))$ **then**
10 $array_2 \leftarrow i$;
11 break;
12 **end**
13 **end**
14 i++;
15 **end**
16 **Step 3:** Generate new obfuscated code with least obfuscating lines which degrade QoR;
17 $BO(P) = replace\_obf\_lines(array_2, O(P))$;
18 $QoR(area, delay) = hls(BO(P))$;

particular obfuscated line leads to any degradation or not, by $\Delta_{QoR(O(P)-QoR(O(P_{single})} = \{Area_{O(P)} - Area_{O(P_{single})}, Delay_{O(P)} - Delay_{O(P_{single})}\}$ (lines 8-11). Thus, the result of this step is an $array_2$ containing only the obfuscated line numbers which do contribute to QoR degradation ($\Delta_{QoR(O(P)-QoR(O(P_{single})} > 1$) (line 10). The advantage of this method is that the order of complexity only grows linearly with the number of lines of code (l), and hence $O(n)$. In the worst case, this step will require l synthesis.

Step 3: Replace obfuscation that Degrades QoR: The $array_2$ obtained from the step 2 contains the line numbers of the obfuscated code that contribute significantly towards the QoR degradation. A new obfuscated version of the BIP is in turn generated by substituting the complex expressions in the $array_2$, by simple obfuscation techniques which are known not lead to any QoR degradation. E.g. instead of mangling the integer "8" in the for loop of Fig. 4.3 with "37661-45842+0x1FFD", it can be obfuscated as "5-6+9" or any simple

Table 4.1: Results: Experimental Results

Benchmark	Original		Fully Obfuscated		Genetic Algorithm				Iterative-Greedy			
	Area [μm^2]	Delay [ns]	Area [μm^2]	Delay [ns]	Area [μm^2]	Delay [ns]	O_L [%]	$T_{run}[s]$	Area [μm^2]	Delay [ns]	O_L [%]	$T_{run}[s]$
Ave8	710	1.33	1,722	2.68	710	1.33	88.88	854	710	1.33	94.73	22
ADPCM	665	5.72	669	5.89	666	5.72	99.07	1062	665	5.72	99.07	60
AES	17,376	2.39	18,329	2.39	17,554	2.39	99.83	1,256	17376	2.39	99.83	334
Bsort	17,800	1.68	18,046	2.57	17,917	2.51	98.46	1042	17,800	1.68	98.46	26
Disparity	1,879	3.09	2,571	4.41	1,981	3.29	98.99	2,256	1,879	3.09	99.61	319
Filter	2,367	4.05	3,523	4.67	2,473	4.32	96.67	926	2,363	4.05	96.67	29
Sobel	1,173	2.94	4,363	15.28	1,301	4.04	97.00	959	1,173	2.94	98.55	27
Geomean	2,580	2.72	3,952	4.37	2,662	3.08		1131.7	2,580	2.72		60.19
Avg.							96.98				98.13	

expressions which can be easily optimized by the parser. The same technique applies for the mangled expressions too. The obfuscated revision obtained is the best-obfuscated code $BO(P)$, because in this heuristics, instead of un-obfuscating every mangled integers and expression (line 17) we only obfuscate the lines which actually degrade the QoR value. Finally, the QoR of the newly generated obfuscated code $BO(P)$ is compared with the QoR of the original code P by performing the last HLS (line 18).

4.1.9 Experimental Results and Discussions

Seven synthesizable SystemC benchmarks from the (S2CBench) [107] benchmark suite are used to evaluate our proposed methods. These benchmarks were manually translated into ANSI-C since the commercial HLS tool used in this work has both a SystemC and ANSI-C parser. The obfuscator used is Stunnix C/C++ [109]. Table 4.1 shows the qualitative results (area and maximum delay) for the original (un-obfuscated) code, 100% obfuscated code and the best results found by the GA and iterative greedy method. Table 4.1 also compares both methods based on the running time required to find the solution.

Some important observations can be made from the experimental results. Firstly, by fully obfuscating the BIP without any further considerations, the area and delay on average grow by 35% and 37.7% respectively. This is obviously not acceptable as no IP consumer will accept the price of this overhead. Secondly, the Iterative-Greedy method is much faster than the GA, on average 18×. Finally, both GA and Iterative-Greedy, lead to very good results, with both of the methods allowing to obfuscate on average over 96% of the BIP, without degrading the area nor delay of the synthesized circuit. The Iterative-Greedy can actually achieve the exact same result as the un-obfuscated synthesis.

It should be noted that the unobligated lines of code can, in turn, be obfuscated using a simple variable name substitution, space and comment deletion, which has shown, not to lead to any overheads.

In summary, it can be concluded that the Iterative-Greedy method is very efficient (requiring on average 60 seconds to execute) while leading to BIP descriptions with over 98% of obfuscated lines with no penalties.

4.2 Hardware Trojan Detection in Behavioral Intellectual Properties (IPs) using Property Checking Techniques

In the previous section, we have studied the impact of source code obfuscation on the quality of results of BIPs for HLS and proposed a quick and efficient method to maximize source code obfuscation while preserving the original design characteristics. In this section, we are going to discuss the hardware Trojan detection technique in BIPs using formal verification, in particular, property checking.

The contribution of this work is multi-fold: First, we present a method to detect HW Trojans in encrypted and un-encrypted 3PBIPs given in ANSI-C or SystemC using formal verification methods. In particular, property checking at the behavioral level. Secondly, the

proposed method has been extended to deal with parts of code which are always executed, i.e., ternary operators (explained in detail in the next sections), by automatically re-writing these constructs. Thirdly, we extend a method presented in [108] which leaks the secret key of a block cipher (AES) by creating an HW trojan which executes an *n* number of extra encryptions. Finally, we present a wide cross section of different types of HW Trojan implemented on BIPs to change the functionality of the IP, leak information and another which leads to the denial of service. These BIPs have been made public at the trust hub [105] which already includes designs with HW Trojan at different levels of abstractions, but did not contain any synthesizable BIPs.

4.2.1 Related Work

The main problem with previous hardware Trojan detection approaches is that many proposed methods require a golden Trojan-free IC or functional model [124]. In the case of detecting HW Trojan in third party IPs (3PIP), this is not the case. IP providers only provide a single version of the IP and there is no reference model against which the IP received can be verified. 3PIPs are typically delivered in RTL code (Verilog/VHDL). Thus, previous detection techniques have focused on code coverage [125], although it has been shown that even a 100% code coverage cannot guarantee a Trojan free design [126]. Our work nevertheless deals with this as shown in the next sections.

Most previous work on HW Trojan detection, suggests to source IPs from two different vendors, as it is very unlikely that both IPs are *infected*. In [127], the authors present a system that can detect malicious outputs by duplicating the 3PIPs and comparing their outputs. They also propose a method to avoid collusion between parent and child IPs from the same vendor, by ensuring that two consecutive IPs of the same vendor are never directly connected together. Cui et al. [128] extended this work by proposing a run-time recovery system which rebinds at runtime the IPs from different vendors in case that a malicious output is detected. The

main drawbacks of these methods are that they involve large overheads because all 3PIPs need to be duplicated and also require that all of the IPs are available from different vendors. Moreover, the runtime recovery method proposed in [128] does not clarify how the rebinding of IPs is done at runtime. A similar approach is taken in [129], where two BIPs are sourced from different vendors. The main problem with these approaches is that they all rely on two independent parties providing the same IP, which is not always possible, especially for BIPs, which is a market still in its infancy. Secondly, the overheads, especially in area and power, can be significant as the IP has to be replicated.

In [130], the authors introduce the term *Unused Circuits Identification* (UCI) to identify potential Trojan, for circuit parts not sensitized during the verification stage. Because their method might remove legitimate circuits, it inserts logic to detect if the removed circuits would have been activated and trigger an exception if the hardware encounters this condition at runtime. The drawback of this method is that it implies the need for a processor to deal with this runtime exception.

Formal verification techniques have been also applied for the detection of HW Trojans. In [131], the authors propose that the 3PIP user and the vendor agree upon a pre-defined set of security properties which the IP should satisfy. Rajendran et al. [132] extended this work and uses Bounded Model Checking (BMC) to detect Trojans. The main problem is that this work also relies on the IP vendor and user agreeing upon a set of security properties for the design, which the IP vendor can use to *hide* some of the properties that trigger the HW Trojan from the user. Another problem with these formal verification methods is that they require the IP user to have extensive knowledge of the IP as well as to master RTL-based formal verification tools, which are complicated to use. Also, IPs are often sold encrypted or as gate netlists in order to protect the IP vendor from the illegal re-use of the IP. FPGA vendors allow to parameterize their IPs through their proprietary *GUI* based tools (e.g. Altera's Mega Wizard and Xilinx's Core Generator), but the user has no access to the generated RTL code.

These IPs are typically 10x cheaper, as IP vendors will not be able to re-sell the IP to the user if it releases the source code.

These previous works on HW Trojan detection can be coarsely classified as code analysis techniques and formal verification techniques. Our proposed method makes use of concepts of both categories as it uses SW profiling to detect parts of the code not executed to flag possible HW Trojan and continues by using formal verification methods to create test vectors which force the testbench to execute non-executed code and hence trigger any potential HW Trojan.

Our proposed method relies on *intelligently* inserting assertions in the source code. In previous work on automatically instrumenting the behavioral description to detect errors, every line of code where an operation is performed was instrumented leading to a large number of monitoring signals [133]. Other approaches require the user to manually specify the assertions in the source code and generate On-Chip Monitors (OCM) [134] [135] [136]. The other approach detects and automatically converts behavioral untimed assertions into temporal RTL assertions [137]. Our work is also different from this work because our Trojan detection mechanism is purely pre-silicon based and thus does not require OCM. Also, we do not require the user to specify any assertions beforehand as these are generated automatically by our proposed method.

Hardware Trojan structure in BIPs

The easiest and straight forward way to create an HW Trojan at the behavioral level is by using *if-else* or *switch-case* statements. Fig. 3.6 of Chapter 3 shows an example of such a Trojan which triggers when the Sum of Product (SOP) result of the FIR filter reaches a pre-specified value (TRIGGER). In this case, the payload will set the output to 0. Although a very naïve example, this highlights how easy it is to create HW Trojans at the behavioral level. One can argue that because of the increase in the level of abstraction in behavioral

IPs, it is also easier to read the source code and hence to detect the HW Trojan by simply reading the source code. At the same time, the complexity of these IPs varies and some of the BIPs are extremely complex e.g., encryption algorithms. Most SoC integrators (IP users) often rely on 3PIPs because they do not have the expertise to develop these complex designs. It is, therefore, reasonable to assume that they do not fully understand every source code line. Moreover, the state of the art HLS tools [40] also allow 3PBIPs to be encrypted as shown in Fig. 3.7. In order for an IP vendor to encrypt its IPs, he is first required to contact the HLS vendor, which in turn issues an encryption key to the IP provider. The IP vendor can then specify using synthesis directives in the form or pragmas (comments), which part of the source code to encrypt. Fig. 3.7 shows that *pragma encryption_start* and *pragma encryption_end* delimit the code section to be encrypted. This allows IP vendors to provide IP users with some degree of reconfigurability, by e.g. not encrypting the I/O declaration to allow the IP user to set the bit-width of the I/Os. The main idea behind allowing 3PBIPs to be encrypted is to protect the IP vendor from releasing the complete source code, which in turn also affects the price of the IP as it can be sold much cheaper, as the IP user cannot fully re-use the IP. Obviously, the encrypted 3PBIP can only be decrypted and synthesized with that particular HLS tool. It is obvious that encrypting the IP poses significant security threats as HW Trojans of any complexity can now be easily masked. Our proposed method has therefore been extended to also deal with encrypted 3PBIPs.

4.2.2 Threat Model

The threat model used in this work follows a traditional hardware threat model. We assume that the IP consumer does not have the expertise to fully understand how the IP works and that the IP provider provides a BIP with a testbench that never triggers the HW Trojan. The trigger mechanism that our method detects are based on *if-else*, *switch-case* statements or

for and *while* loops. Under regular verification conditions, the payload of the HW Trojan will never be executed. We believe that this threat model covers most of the scenarios at the behavioral (untimed) level. We also assume that the IP vendor can model the HW Trojan using ternary operators. Ternary operators can be defined as follows:

Ternary Operator. A ternary operator is a conditional operator that provides a shorter syntax for *if-else* statements. The first operand of the statement is a boolean expression. If the expression is true then the value of the second operand is returned otherwise the value of the third operand is returned, e.g. a1? a2 : a3.

Ternary operators have the special property that they will always be executed, thus, it is very difficult to detect these Trojan based on code coverage. Our method, as shown in the next subsection deals with this, by parsing the behavioral description in the first stage to automatically expand ternary operations into *if-else* statements.

4.2.3 Proposed Detection Method

Fig. 4.5 shows an overview of our complete proposed flow, where the white boxes indicate steps performed by our method and the black boxes external tools' executions. Our method takes as inputs the encrypted or un-encrypted behavioral IP in ANSI-C or SystemC. Both types of IPs share 6 main steps while for encrypted IPs two extra pre-processing steps are required. The inputs to our flow are the 3PBIP and the testbench with the input stimuli and expected simulation results (golden outputs), which the IP vendor provides to the IP user. By default, we expect that when the 3PBIP is compiled and simulated, that the simulation results fully match the expected golden outputs provided by the IP vendor, hence the HW Trojan has not been triggered. The main steps of our proposed method are as follow:

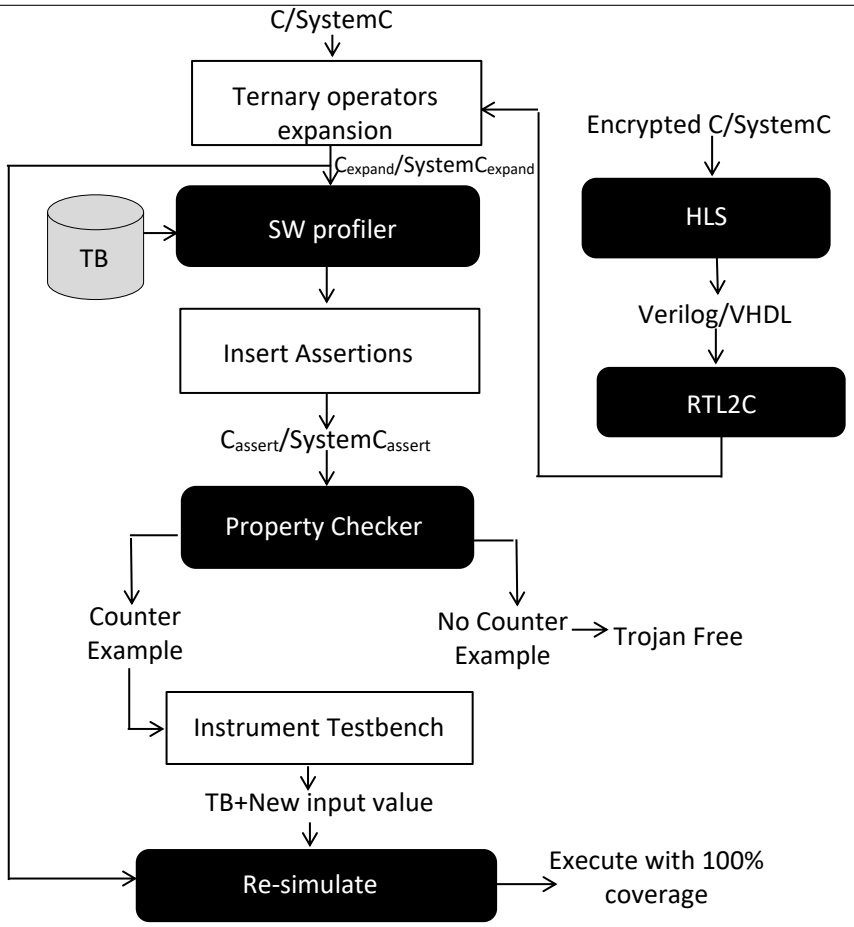


Figure 4.5: Proposed Flow

Step1: Ternary Operator Expansion. Ternary operators can *mask* the presence of an HW Trojan as the source code line is always reported as being executed during profiling. E.g:

sum = (sum == TRIGGER)? 0 : sum;

In this example, the line of code would always be executed and the code coverage report would never detect the threat. Hence, this very first step parses the behavioral description and searches for ternary operators. If found, it expands them into *if-else* conditionals. The input of this stage is thus the original C/SystemC description and the output the expanded version (C_{expand}/$SystemC_{expand}$).

Step2: Software Profiler. Once the description has been expanded, our method continues by profiling the behavioral IP using a SW profiler to identify which lines of the source code have

not been executed during the simulation. At the gate level, circuit parts not sensitized during the verification stage are treated as potential HW Trojan. The equivalent in the behavioral description are parts of the code which have not been executed during the simulation using the testbench provided by the IP provider.

Step3: Assertion Insertion. Our method continues by automatically inserting assertions into the BIP. The inputs at this stage are the behavioral description (C_{expand} or $SystemC_{expand}$) and the profiling report. The output will be an instrumented C or SystemC code (C_{assert} or $SystemC_{assert}$) with assertions, which in turn will be fed to the formal verifier in the next step.

An assertion is basically a statement that something must be true, similar to an if statement. The difference is that an "if" statement does not assert that an expression is true, it simply checks that it is true. The property checker thus detects the possibility of a false condition in the assert function condition as an error and shows the pattern that generates an error. The type of assertion used in this work is the immediate assertion, as it is virtually impossible to build HW Trojan with temporal properties in an untimed behavioral description as there is no clock in the behavioral description.

Currently our method only works with HW Trojans when the trigger condition is either specified in an *if-elsif-else* clause, *switch-case* or *for, while* loops. This is the most natural way of specifying trigger conditions in behavioral code and it is also the trigger mechanism of that most of the RTL HW Trojans available in the open source trust hub [105] (especially if-else). Thus, our method parses the source code and the profiling report identifying the lines of code not executed. If a *block* of code has not been executed, it checks if the first line of this *block* is either an if, switch, for or while loop, where a *block* of code can be loosely defined as one or more lines of code. If it is, it parses the condition in the statement and generates the assertion. If it is not any of these operations, then our method ignores this *block*

```
    /*--- Sum of Product -----------------*/
    for(i=0;i<9;i++)
      sum += ary[i] * coeff[i] ;

    /*--- Rounding and Saturation -------*/
    assert(sum !=TRIGGER)
    if ( sum < 0 ){
      sum = 0 ;
    } else if ( sum > 255 ){
      sum = 255 ;
    } else if ( sum == TRIGGER ){
      sum = 0 ;
  }
```

Figure 4.6: Assertion insertion example

of code and continues. Our approach is scalable and other statements could be supported in the future, as these conditions are included in a library.

Finally, once the entire description is analyzed, a new C/SystemC description is generated with assertions which when executed with the property checker should create counterexamples that will lead to the execution of the conditions not triggered during a regular simulation. Fig. 4.6 shows an example of assertion insertion for the FIR filter described previously to illustrate the assertion insertion. Because the code with the *elseif* clause containing the HW Trojan was not executed during the profiling step, our method inserts an assertion with the opposite condition of the *elseif* condition. This will force the property checker in the next step to find a counter example which would make sum=TRIGGER and hence execute this part of the code.

Step4: Property Checker. Once the BIP is synthesized, our method calls the property checker. The commercial HLS tool used in this work includes this property checker, which makes it easy to integrate into our flow, as the scripts and input files are automatically generated.

Property checking is a formal analysis technique that searches the design's state space without the need of a testbench (simulation) to guarantee that a design property, specified

using a set of assertions, is not violated. If a property is violated, a counterexample is generated. The formal verifier used in this work is based on a traditional model check approach where the input is the behavioral description with properties, e.g. C with assertions and converts this into a state transition representation system and temporal properties. The states are traversed and the verification engine periodically (every cycle) checks the path of this tree to detect the possibility of the existence of a condition that would lead to the assertion to be triggered. If this type of path does not exist, it will be considered that functional specifications are met. If a path exists, it will be considered as a violation (exception) of functional specifications. A counterexample is created in the form of a VCD file containing the sequence of vectors that lead to the property violation. The property checker treats every assertion separately and reports a counter example for each assertion if it exists. Hence, in theory, as many VCD files as assertions are generated.

Step5: Instrument Testbench. If the property checker returns any counter-examples, then our method parses the VCD file with the input vectors and annotates them at the input files of the BIP testbench. In our case, the stimuli of the BIP are stored in separate files with the same name as the I/O port for each I/O.

Step6: Re-simulate:. Once the new test-vectors have been annotated into the stimuli files for each I/O, our method re-simulates the original 3PBIP using these input vectors. There is no need to recompile the behavioral code, as only the test-vectors have changed. The simulation now executes the source code lines, which had not been executed with the original test data provided by the IP vendor. One key issue in this step is that no golden output is available for these new test-vectors, hence, the IP user should understand if the output is reasonable or not. We believe that this assumption is very reasonable because if not, the behavioral IP provider could intentionally provide wrong golden outputs which trigger the HW Trojan. There are different types of code coverage metrics such as branch coverage, line coverage, function coverage etc. involved in the code coverage analysis. In this work, 100% coverage is with

respect to line coverage considering the boundary conditions. The assertions are inserted before the lines of code which are not covered considering the boundary conditions.

Trojan Detection in encrypted IP

In the case that the 3PBIP is delivered in encrypted form, two pre-processing steps are required.

Pre-Step1: High-Level Synthesis. In case that the 3PBIP is encrypted, only that particular HLS tool can decrypt the IP and synthesize it. Hence, the IP is synthesized and the RTL (Verilog or VHDL) is generated. This RTL code is not encrypted anymore as it has to be fed to a third party logic synthesis tool.

Pre-Step2: RTL to C conversion. Once the synthesizable RTL code from the HLS tool is obtained, it is converted back into C code. Work on translating RTL code to C code has been done in the past in academia [138] and industry and there are even numerous commercial products available for this [139, 140]. In our case, the commercial HLS tool used in this work also comes with an RTL to C translator. This is used to convert the generated RTL code into C code, which is in turn used for HW Trojan detection following the 6 steps described previously. It should be noted that there is a big difference between the original manually written C code and the automatically generated by the RTL to C translator. There are no comments and is not as easy to read. Nevertheless, it is functionally equivalent.

4.2.4 Experimental Results

Three case studies are presented, in order to verify our proposed HW Trojan detection method. They cover three of the most important families of HW Trojan. The first changes the functionality of the IP, the second leaks secret information while the third presents a denial of service case. For the first case, a Sobel filter was used in order to visually show the effects of the HW Trojan on the output. Different trigger and payload mechanisms have also been implemented for this case as shown in Table 4.2. Fig. 4.7 shows the results and will be

Table 4.2: Experimental Results

Bench	Trigger	Payload	C orig	C Trojan	Area Orig [μm^2]	Area Trojan [μm^2]	Δ_{Area}	Coverage orig	Coverage after	Fig.4.7
sobel	combinational	no memory	182	186	966	1,028	6%	82.63%	100%	(e)
sobel	combinational	memory	182	188	966	1,073	7.2%	86.57%	100%	(f)
sobel	sequential	memory	182	189	966	980	1.4%	84.32%	100%	(h)
aes	combinational	memory	371	390	36,904	36,984	0.21%	80.73%	100%	
UART	sequential	memory	174	180	1615	1714	5.7%	91.8%	100%	
Geomean			207.9	215.4	2,218	2,328				
Avg.							4.1%	85.21 %	100%	

Table 4.3: False vs. True hardware Trojan Detection Assertions

Bench	Properties	Assertions target Trojans	Assertions Trojans not targeting	Total assertions	Δ_{assert_trojan} [%]
sobel	assert(input_row[0] < 20) assert(input_row[1]<20) assert(input_row[2]>245)	3	1	4	0.75
sobel	assert(colo==512) assert(flag==1)	2	1	3	0.66
sobel	assert(colo==512) assert(rep==512)	2	1	3	0.66
aes	assert(data3[0] == -1460255950)-> 3 times	3	5	8	0.38
UART	assert(count>=20)	1	8	9	0.11
Avg					0.512

described in the next paragraphs. The HLS tool used in this work is CyberWorkBench from NEC [40], which as mentioned previously contains a behavioral level property checker and an RTL to C converter and the software profile used is GNU's gprof. The entire flow was written as a set of Perl scripts. The three designs used in the case studies were taken from the open source Synthesizable SystemC Benchmark suite (S2Cbench) [107] and modified to include the HW Trojan. In particular the Sobel filter, AES and the UART benchmark.

Table 4.2 shows the main characteristics of the designs and their results. Columns 2 and 3 indicate the type of trigger and payload mechanisms. Columns 4 and 5 obfuscating the size in terms of lines of code of the original source code and the modified source code with the Trojan. It can be observed that it requires very few lines of code to implement a powerful HW Trojan at the behavioral level. Columns 6 and 7 indicate the area of the synthesized circuit

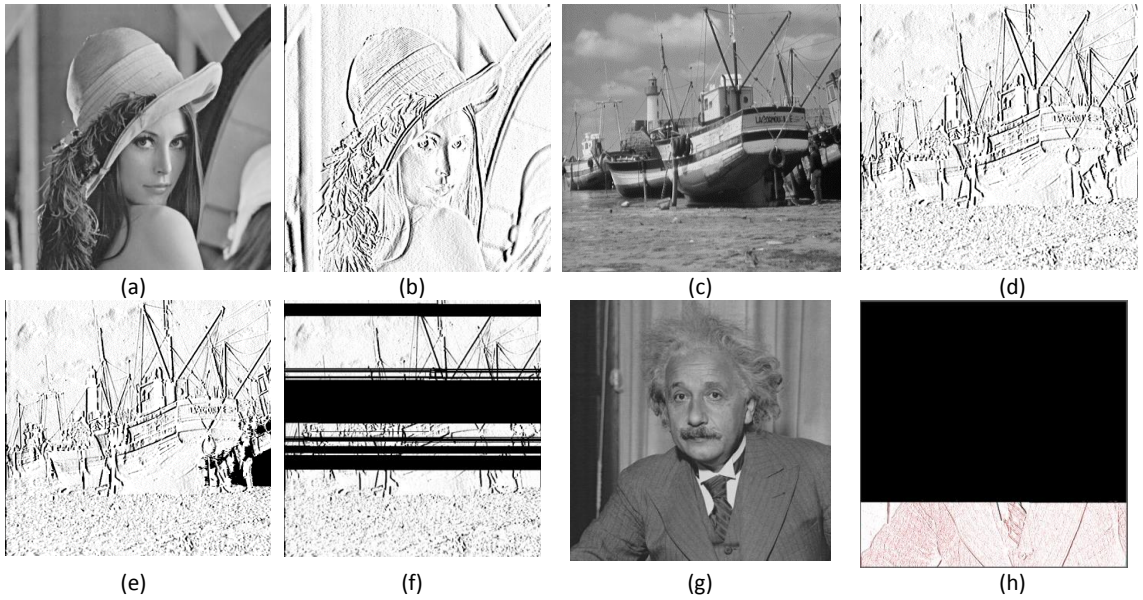


Figure 4.7: Sobel edge detection case study (a) original input image (512 $\times$ 512) (b) expected golden output (c) input image (512 $\times$ 512)(d) expected output (e) HW Trojan effects (combinational trigger+no memory payload (f) HW Trojan (combinational trigger+memory payload) (g) input image higher resolution (600 $\times$ 600)(h) HW trojan effect (sequential trigger+memory payload)

reported by the HLS tool for the two cases (Trojan free and with Trojan) when synthesized using a target frequency of 100 MHz and a 45nm Nangate Opencell technology. Columns 8 and 9 indicate the code coverage before and after our proposed method. Finally, the last column points to the figure in Fig. 4.7 which shows the effect of the Trojan on the result.

Because our method is based on code coverage, it will inevitably insert assertions in parts of the code which are not HW Trojan. Table 4.3 shows the total number of assertions inserted into each design and indicates how many of these assertions involved the HW Trojans built into the designs and how many were unrelated to the HW Trojans. It can be observed that on an average 50% of the assertions involved the HW Trojan. This percentage will obviously strongly depend on the testbench provided to the IP user.

The next subsections describe in detail the HW Trojans built into the test cases and describes in detail how to retrieve that key for the AES example.

Case Study I (Sobel): Change in functionality

Three different types of HW Trojans are built into the Sobel edge detection algorithm, which modifies its functionality when certain input data is received. Each HW Trojan has a different trigger and payload mechanisms, as shown in Table 4.2. Fig. 4.7(a) shows the input image, while Fig. 4.7(b) shows the expected golden output, both provided by the BIP provider. In all of the cases with HW Trojans, the golden output is always obtained (Fig. 4.7(b)), hence the Trojan does not interfere with the functionality of the IP.

Fig. 4.7(c) and (d) show the output that the Sobel filter should be generated when using a different picture with the same resolution. This picture is *not* provided by the BIP provider and will, therefore, trigger the HW Trojan. Fig. 4.7(e) shows the effect of the HW Trojan with a combinational trigger mechanism without memory. When a certain input pixels' values are passed to the Sobel filter the Trojan is activated and the filter output is overwritten with a new value. The Trojan remains active while the inputs are within the given trigger values and the payload gets de-activated when the inputs are outside the given trigger values. Fig. 4.7(f) shows the result of an HW Trojan with combinational trigger and memory payload. In this case, the Trojan is more powerful as once triggered it causes the payload to be active during a longer period of time. Finally, Fig. 4.7(g) shows the original image and Fig. 4.7(h) the effect of a Trojan with the sequential trigger mechanism and memory payload. These type of Trojan are often also called *time-bombs*, and their trigger mechanism is basically a counter. In this case, because the original input image (Fig. 4.7(a)) has a resolution of 512×512, the trigger mechanism was set to trigger after 512×512 input pixels are received. This is why the resolution of Fig. 4.7(g) is also larger (600×600). This example shows how to create simple HW Trojans which trigger when images of different resolutions than the ones used for the IP verification are used. Often images of lower resolutions are used for IP verifications in order to speed up the simulation time[141]. Also, it shows how easy it is to build HW Trojans which will not trigger when static images are used during the algorithmic verification, but

which will trigger when a real-time video is used in the final design. As shown in Table 5.3, the first case has a combinational trigger with a payload without memory and is the smallest in terms of area and also lines of code. It is also the least powerful one, while in order to describe more powerful HW Trojans more lines of code are needed. Also, from Fig. 4.7, it can be clearly observed that the HW Trojans completely change the design functionality. This example is extremely important because most of the modern application rely on image processing ranging from feature detection for face recognition to missile detections. Most of these applications make use of edge detection to further process the data. In case that this stage fails, the functionality of the rest of the system is seriously compromised.

Case Study II (AES): Leak Information

The AES is a symmetric encryption algorithm that was selected by National Institutes of Standard and Technology (NIST) in 2001 as a new cipher. The fixed data size of the plain text in the AES algorithm is 128 bits. The specified key lengths are 128, 192 and 256 bits.

We redesigned the AES encryption circuit from the S2CBench benchmark with a fixed key size of 128 bits and applied the method to leak the secret keys introduced in [108]. In [108], the authors failed to fully explain the algorithm to leak the secret keys. In this work, we extend their work and fully illustrate how to leak the secret keys. In this work, we modified the trigger mechanism compared to their original work by using a given input sequence as the trigger mechanism to leak the data required to calculate the secret keys. After applying our Trojan detection methodology, the trigger and payload mechanisms could be successfully detected. The next sub-section explains in detail how our method is able to leak the secret keys.

ALGORITHM 3: Summary of method to leak secret keys

input : Plain text P, cipher text RK_{10}, RK_{10+x} and RK_{10+x+1}.
output : The initial key K_0.

Step 1: After procuring the cipher encrypted data RK_{10}, the Trojan prolongs the encryption process for 'a' more times to get the cipher text RK_{10+x} and RK_{10+x+1} with Rcon set to 0 after the 10th round of encryption process.

Step 2: From RK_{10+x} manually compute R_{10+x} by following the encryption process

Step 3: Calculate K_{10+x} from RK_{10+x} and RK_{10+x+1} with the equation as below.

$$K_{10+x}=RK_{10+x} \text{ xor } RK_{10+x+1} \quad (1)$$

/* **Step 4:** To calculate K_0 from K_{10+x} */

Step 4-1: Set r=1

Calculate K_{10+x-r} with Rcon set to 0 from the equation

$$K_i = KeyExp^{-1}(K_{i+1}, Rcon = 0) \quad (2)$$

until K_{10} is evaluated i.e., r=a.

for $(r = 1; r <= x; r++)$ **do**
    $K_{10+x-r} = KeyExp^{-1}(K_{10+x-r+1}, Rcon = 0)$
end

Step 4-2: Calculate K_0 with regular Rcon values used for encryption process.

for $(i = 9; i >= 0; i--)$ **do**
    $K_i = KeyExp^{-1}(K_{i+1}, Rcon = regular)$
end

K_0 is the initial key.

Algorithm to Obtain Cipher Key

Unlike other HW Trojans applied to AES circuits in [105], which directly leak the secret keys from the output which should be extremely easy to detect, our extended method calculates the secret keys from the information obtained by the Trojan circuit. Fig. 4.8 shows a high-level diagram of the method, which is based on an online and an offline part. The HW Trojan performs x+1 extra encryptions and this data combined with the plaintext is then used offline to calculate the initial key.

Fig. 4.9 depicts graphically the inverse key expansion method to calculate initial key K_0. In particular, consider RK_x be the output data after the x-th stage round function. Since the number of round functions iteration is 10 in AES with a key of 128 bits, the final cipher text is RK_{10}. Additionally, if the AES encryption circuit performs the round function for 100 times, which is larger than the specified number of rounds, the output data has to be

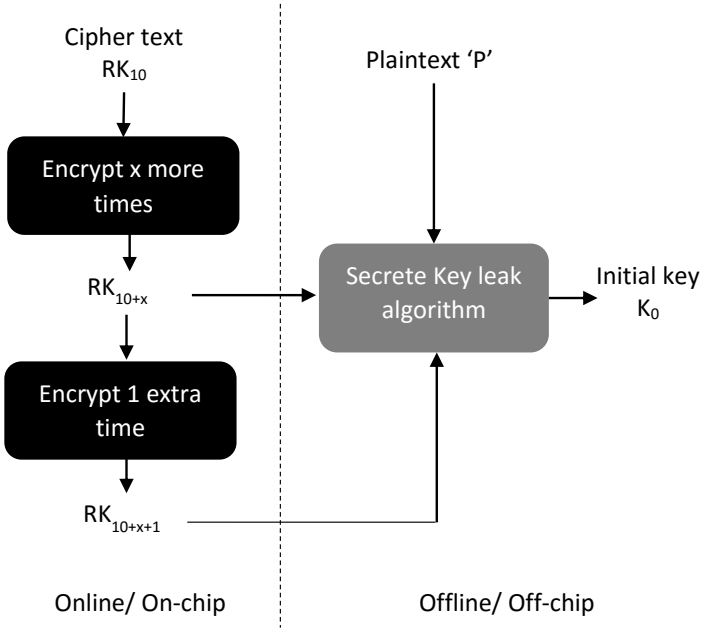


Figure 4.8: Secret Key leaking

RK_{100}. The value of the R_{con} after the 10th round must be set to all zeros till the last round of encryption. i.e., R_{con}=0,0,0,0...0(last). The activation condition chosen for the proposed Trojan circuit is the input sequence of the plain text P. The attack of the suggested Trojan circuit is the execution of an additional a times round function and the output of RK_{10+x} and RK_{10+x+1}. Finally, attackers obtain the cipher key from plain text P, RK_{10+x} and RK_{10+x+1}.

The information consists of plain text P, the ciphertext RK_{10}, RK_{10+x} and RK_{10+x+1}, where the subscript x stands for the extra number of times that the plaintext has been encrypted. At the beginning, the attackers calculate the round key K_x on the round x with RK_{10+x} and RK_{10+x+1}. Finally, the initial key K_0 is calculated with P, RK_{10} and K_x.

Although the explanation of the complete AES algorithm is beyond the scope of this work, algorithm 3 gives the detail description of how to calculate the initial key K_0 with P, RK_{10}, RK_{10+x}, RK_{10+x+1}. During the encryption operation, the final encrypted data will be round key RK_{10}. Whenever the Trojan is triggered, in addition to RK_{10}, RK_{10+x} and RK_{10+x+1} will also be generated at the output (line 1). R_{10+x} is calculated from the obtained RK_{10+x} in the previous stage (line 2). K_{10+x} (The last key of the encryption process) is evaluated from

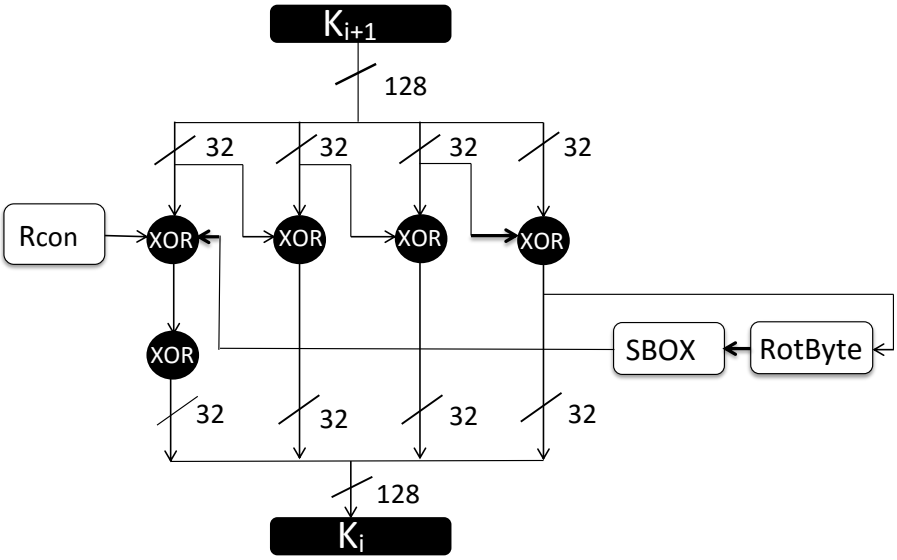


Figure 4.9: Inverse Key Expansion

RK_{10+x} and RK_{10+x+1} (lines 3-4). Once K_{10+x} is obtained, K_{10} is calculated using the inverse expansion function with R_{con}=00 (lines 6-11). Finally, K_0 is obtained from K_{10} with regular *Rcon* values for encryption i.e., *Rcon* = 01 02 04 08 10 20 40 80 1B 36 (lines 13-17) using the inverse expansion function.

Case Study III (UART): Denial of service

Universal Asynchronous Receiver Transmitter(UART) is a communication IP used for serial data communications over a computer or a peripheral device serial port. UART's are commonly used in conjunction with communication standards such as RS-232, RS-422 or RS-485. UART takes bytes of data and transmits the individual bits in a sequential manner. At the receiver side, the second UART device re-assemble the bits to form a complete byte.

The UART design was taken from the S2CBench benchmark suite in SystemC with 1 start bit, 8 data bits and 1 stop bit. The Trojan is inserted at the transmitter block. The function of the Trojan, in this case, is to delay the transmission after certain time based on the counter value. Also, the Trojan will make the receiver lose some of the data when triggered.

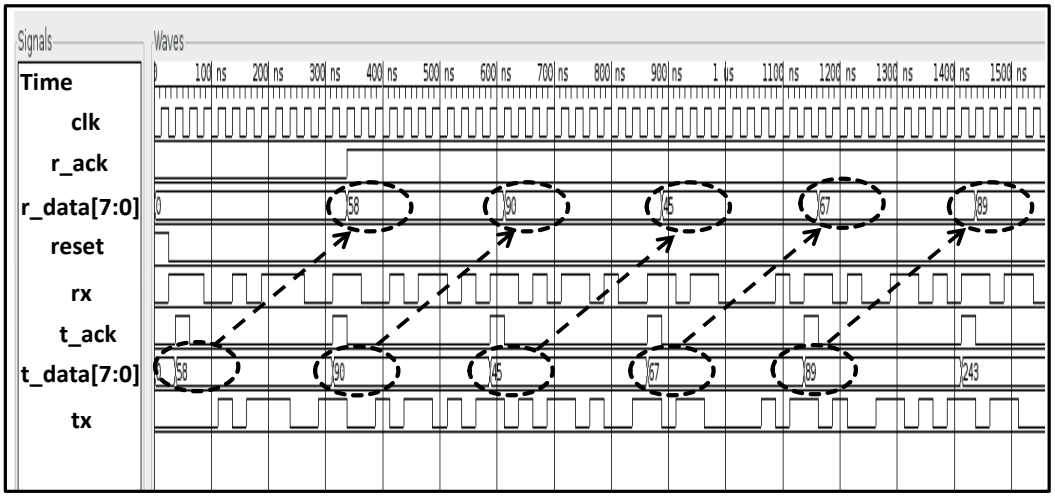


Figure 4.10: UART without Trojan triggered

The activation mechanism, in this case, is internally conditionally triggered as referred from [105].

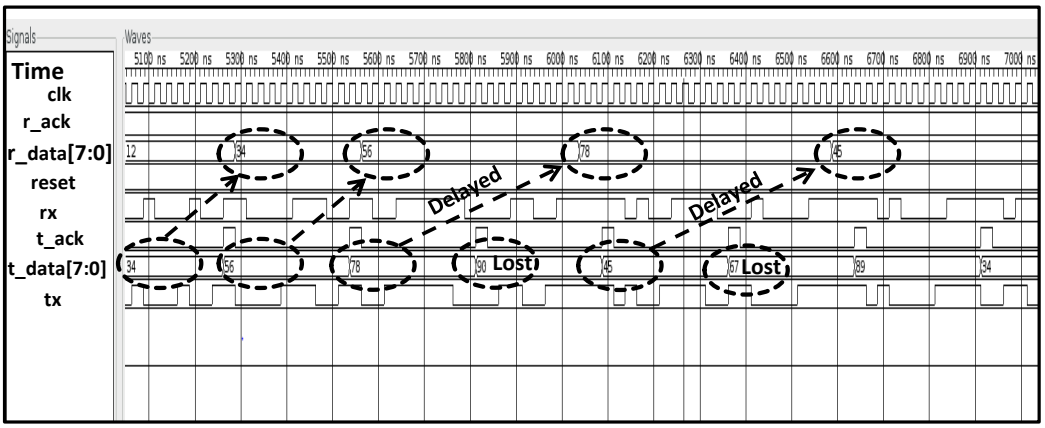


Figure 4.11: UART with Trojan triggered

Fig. 4.10 shows the simulation results for UART in a loopback mode where the transmission is steady throughout. Fig. 4.11 shows the simulation result for the Trojan triggered case. After the count reaches certain value the transmitter delay the transmission by certain clock cycles. Meanwhile, the intermediate data will also be lost during the course of transmission

after the Trojan has triggered. At the receiver side, one can see the loss in transmission of data.

Our Trojan detection method was applied to these IPs in un-encrypted and encrypted modes and in all cases, it could successfully detect the HW Trojan. For the sobel case, because the testbench uses a bmp file as input and as golden output, it was extremely easy to visually see the effects of the HW Trojan and hence to detect it. From the results, it can be observed that in order to design a powerful HW Trojan at the behavioral level only a few lines are needed, even for very sophisticated ones like in the AES case. When synthesized, the area is only on average 4.1% larger than the Trojan free version.

4.3 Summary

This chapter has addressed two issues related to BIP security. In the first part, it has analyzed the impact of code obfuscation of BIPs on the quality of results (QoR) when different parsers are used. As shown, the same BIP can lead to different area and delays as these are responsible for the technology independent optimizations. Thus, two methods were proposed to maximize the amount of obfuscated source code while minimizing the amount of degradation introduced due to the obfuscation process. The first method is based on a GA algorithm and is used as a baseline method used to compare a faster heuristic based on an iterative greedy algorithm. In the proposed fast obfuscation method, nearly 100% of the design is obfuscated while being much faster than the GA baseline obfuscation method.

The second part of this chapter introduced a technique to detect hardware Trojans in BIPs. The third party behavioral IP (3PBIP) market is still in its infancy compared to the more mature RTL IP market. Most of the previous techniques address the issue of 3PIP HW security by instantiating two different IPs from different vendors onto the same system. Other methods rely on golden Trojan free models to detect HW Trojans. This work targets the detection of HW Trojans embedded in 3PBIPs when no golden models are available

using code coverage and formal verification techniques as a pre-processing stage to expand constructs like ternary operators that can mask the Trojan. A fully automated flow was created that can also be used for encrypted IPs and show that our flow works well for a variety of different HW Trojan. Although the proposed method also has some limitations, it is a very good first step into the automatic detection of HW Trojan in 3PBIPs.

Chapter 5

Hardware Trojan Detection at System Level

This chapter introduces a system level method to detect hardware Trojan in third party behavioral intellectual properties (3PBIPs) and in dynamically re-configurable FPGAs. The first part of this chapter, discusses the proposed hardware Trojan detection circuit called *Trust Filters* to detect HW Trojan at runtime. With the help of cycle-accurate simulation model, it is possible to fine tune these filters so that the overall system has no performance degradation. This can be achieved by exploiting the slack time between the time that a slave returns the data to the master and the time that the master sends new data to the slave. The advantages of using behavioral C-based SoC design are multi-fold: (i) It allows the generation of fast cycle-accurate models to measure the exact slack of each BIP mapped as a loosely coupled Hardware Accelerator (HWAcc) slave and (ii) its ability to build the complete SoC using synthesizable Application Programming Interfaces (APIs) and hence allowing the fine tuning of these *Trust Filters*.

The second part of this chapter discusses hardware trojan avoidance and a detection mechanism for dynamically re-configurable FPGAs. Runtime Reconfigurable FPGAs have unique characteristics that make them extremely vulnerable to Hardware Trojan. These

FPGA families reconfigure themselves every clock cycle updating the functionality of the data path. A State Transition Controller (STC) typically holds the configuration code for each of the contexts. This architecture makes these type of architectures very efficient, but also extremely vulnerable to malicious alterations across any of the steps from design to fabrication of the FPGA. Being able to control the STC involves being able to control the functionality of the FPGA and perform any desired function at runtime. Thus, this work specifically targets the detection and especially the avoidance of HW Trojan being triggered in runtime reconfigurable FPGAs and in particular Coarse-Grained Runtime Reconfigurable Arrays (CGRRA).

5.1 Hardware Trojan detection in behavioral MPSoC

5.1.1 Introduction

Most of the complex ICs now are complex heterogeneous Multi-Processor SoCs (MPSoCs), composed of multiple in-house and third party IPs (3PIPs). Commercial HLS tools have thus, been extended to include complete verification environments and SoC design capabilities in order to allow designers to design and verify complete systems at the behavioral level. Therefore, it is important to study how secure these behavioral SoCs are and how to detect any malicious alterations present.

5.1.2 Threat Model

The threat model used in this work assumes that a 3PBIP is instantiated in a SoC as a slave to perform a dedicated task (also called hardware accelerator-HWacc). This work also assumes that the test-vectors provided by the IP vendor do not trigger the hardware Trojan and that the output of the BIP for the given test-vector always match the reference output provided. Otherwise, the 3PBIP vendor would expose himself easily to the IP user. This work also

Table 5.1: Hardware Trojan types overview and types addressed in this work

		Trigger mechanism	
		Combinational	Sequential
Payload	With memory	✓	✓
	Without memory	✗	✗

assumes that the foundry can alter the design before tape-out by inserting an HW Trojan. This implies that the hardware Trojan can be inserted by the 3PBIP provider, or by any third party company downstream the design and fabrication process. Thus, a runtime detection method with low area and performance overhead is required. It should be noted that the IP provider could try to hide the HW Trojan by providing a testbench that triggers the HW Trojan, but that the reference[1] output also provided by the same IP vendor does not report as erroneous. We assume that the IP user is able to detect these cases either during the IP integration or verification process. It should be noted that our proposed detection mechanism can only deal with hardware Trojans which alter the functionality of the IP, and in cases where the latency is fixed (number of clock cycles requires to generate an output) the trust filter can also detect cases when the performance is degraded, as it knows the latency that the Trojan free IP has. Our proposed method can not detect other cases where e.g. the Trojan increases the power consumption of the IP. Security of the trust filter can also be challenged because a rogue adversary in a foundry can insert hardware Trojan or tamper the trust filter. This can be alleviated by incorporating the technique mentioned in [142].

5.1.3 Contributions

The contribution of this work can be summarized as follows:

[1]Reference output and the golden IPs should not be confused. The reference outputs are the set of test vectors provided by the IP vendor for the preliminary verification of the IP. Golden model is a complete IP to which any IP of the same functionality can be compared.

1. Introduce the concept of *Trust Filter* for heterogeneous MPSoCs to detect HW Trojan at runtime.

2. Develop different types of *Trust Filter* and use the slack time between computations at each slave [2] to determine which type of filter to use at each slave.

3. Accurately measure the area overhead and performance penalty introduced by these filters by building complete cycle-accurate models of the complete SoCs.

Although this work can also be extended to RTL it is almost impossible to simulate complete SoCs at the RTL to fine tune the *Trust Filters*.

5.1.4 Background and Related Work

An overview of most common and relevant HW Trojan attacks and countermeasures can be found at [124] and [143]. The vital issue with these previous works is that many of the proposed methods need a golden Trojan-free IC or functional model. The problem is that IP distributors only provide a single version of the IP without any golden or trusted model for verifying the sourced IP. Customarily, 3PIPs are distributed in HDL (VHDL or Verilog). Thus, most of the previous Trojan detection techniques have focused on code coverage [125], in spite of proving that even a 100% code coverage cannot guarantee a malicious free design [126].

Eric et al. [144] showed that by assigning 3PIP vendor the task of constructing compliance proofs for their hardware IP, consumers (SoC integrator) can make sure that the HW they purchase operates within the parameters they have chosen as provable security properties. This is basically an agreement between a SoC integrator and a 3PIP vendor on a predefined set of properties which can be verified by SoC integrator. Waksman et al. [145] instead of discovering the malicious logic in the design, which they believe - is an extremely hard

[2] This work makes indistinguishable use of the terms HWAccs, slave, and BIP to designate the computationally intensive task synthesized using HLS.

problem - made the backdoor design problem itself intractable to the attacker. They have scrambled the inputs that are supplied to hardware units at runtime, making it infeasible for malicious components to acquire the information they need to perform malicious actions. Beaumont et al.[146] have developed SAFER PATH architecture, which uses instruction, data fragmentation, program replication and voting to create the computational system that is able to operate safely in the presence of active hardware Trojans. However, both of these works [145], [146] lead to systems with considerable area and performance overheads.

Rajendran et al. [127], in their work to detect HW Trojan in 3PIPs, suggest to source the IP from different vendors assuming that it is virtually impossible that both IPs are infected. The system can reveal pernicious outputs by duplicating the 3PIPs and analyzing their outputs. They also propose a method to avoid collusion between IPs procured from the same vendor, by ensuring that two consecutive IPs of the same vendor are never directly connected together. Cui et al. [128] extended this work by introducing a run-time recovery system which rebinds at runtime the IPs from distinct vendors in case that a vicious output is detected. The main shortcomings of these detection techniques are that they involve large overheads because all the 3PIPs needs to be replicated and also demands that all of the IPs are available from different vendors. Furthermore, the run-time recovery method suggested in [128] does not clarify how the re-binding of IPs is done run-time. Also, it does not investigate into the exploration of efficient distinct vendor allocation procedure of similar operations as it affects the final delay and area of scheduling.

Waksman et al.[147] have presented FANCI, a tool that flags suspicious wires in a design which have the potential to be malicious. It identifies nearly unused circuits by static Boolean function analysis. Zhang et al. [148], [149] proposed VeriTrust to detect the hardware Trojans at the design stage. The uniqueness of their approach is a detection technique which is insensitive to hardware Trojan implementation styles. The common feature of all these techniques is that they all apply to pre-silicon hardware Trojan detection. In case, that the

hardware Trojan is inserted at a later stage, e.g. at the foundry, these methods cannot be used and runtime detection methods are required to complement them.

Moreover, most of these previous works on HW Trojan detection is done at the IP level isolating the IP, and do not consider system level design issues. Also, at the system level, there is a possibility that IPs can collude, e.g. a master sends the trigger information to a slave where the payload is activated. In [150], the authors propose a task scheduling based approach that an MPSOC designer can take to protect MPSOCs against malicious modifications by sourcing IPs from different vendors, which leads to the unacceptable area and power overheads and is often not even feasible, especially for BIPs.

Our work is different in many dimensions. Firstly, it targets the detection of HW Trojans built into the slaves of heterogeneous MPSoC triggered by a master (microprocessor). These slaves act as loosely coupled HWAccs and are given as BIPs in ANSI C or SystemC. Secondly, our proposed method makes use of the slack time during the bus operation in MPSoCs to detect the HW Trojan, thus in many cases leading to no performance penalties. Finally, as mentioned previously, our proposed detection methods mainly works at runtime and not at the pre-silicon level (in theory it can also be used at the pre-silicon level, but it cannot guarantee to find the hardware Trojan as it is virtually impossible to find the trigger condition). Formally, the objectives of this work can be formulated as follows:

Problem Formulation: Given N synthesizable behavioral descriptions $BD = \{C_1, C_2, \ldots, C_n\}$ to be mapped onto a shared bus MPSoC as loosely coupled HWaccs, where $HWaccs = \{BIP_1 \leftarrow C_1, BIP_2 \leftarrow C_2, \ldots, BIP_n \leftarrow C_n\}$, with a potential hardware Trojan. Find the minimum number of clock cycles at which each HWacc is inactive in the system, i.e., $BIP_1 = slack_1$ and given two types of Trust Filters $TF = \{typeI, typeII\}$, instantiate the most appropriate filter at each HWacc, e.g. $(BIP_1 \leftarrow typeI)$, $(BIP_2 \leftarrow typeII)$. *TypeI* is a fully autonomous filter which uses the idle time (slack time) to detect hardware Trojans and *typeII* is a manual filter, which requires the master to explicitly start the detection procedure.

The decision to use *typeI* filters or *typeII* filters is given by: Use *typeI* if $t_{BIP_1} < slack_1$ else *typeII*, where $t_{BIP_1} = t_{read1} + t_{compute1} + t_{compare1}$, with t_{read1} being the number of clock cycles it takes for a HWacc to read the data to compute a new output, and $t_{compute1}$, the latency in clock cycles to produce a new results once a new input has been read and $t_{compare1}$ the number of clock cycles required to compare the new output with the reference output stored in the trust filter. This problem is explained in detail in section 5.1.6.

5.1.5 Trojan Detection Method

Fig. 5.1 shows the target MPSoC platform used in this work. It consists of M number of masters (typically embedded processors) and N slaves. The slaves are all BIPs which are mapped onto the MPSoC as loosely coupled HWaccs. In order to build larger systems, the BIPs are considered independent concurrently executing applications, which periodically repeat themselves. Each BIP is assumed to have its corresponding testbench (TB) also passed as an input to our proposed flow. Thus, a system S contains M processors and N tasks to be mapped onto these processors, where the HW/SW partitioning has been done a priori mapping the TB to the master and the BIP as a slave, hence $N = \{BIP_1, BIP_2, \ldots, BIP_N\}$ and $M = \{TB_1, TB_2, \ldots, TB_M\}$. The masters and slaves are interconnected through a memory mapped shared bus (i.e., AHB/AXI bus). This standard bus follows the following 5 high-level step protocol:

1. the master requests bus access for writing to the corresponding slave to the bus arbiter;
2. the arbiter grants access to the master and thus writes data to the corresponding slave, otherwise, it waits until bus access is granted;
3. once the slave receives the data from the master, it computes the result and after finishing the computation waits for the master's reading request;

4. the master requests bus access for reading the result from the corresponding slave to the bus arbiter;

5. the master reads the data from the corresponding slave if the bus access is granted, otherwise it has to wait until the bus is available or give up reading.

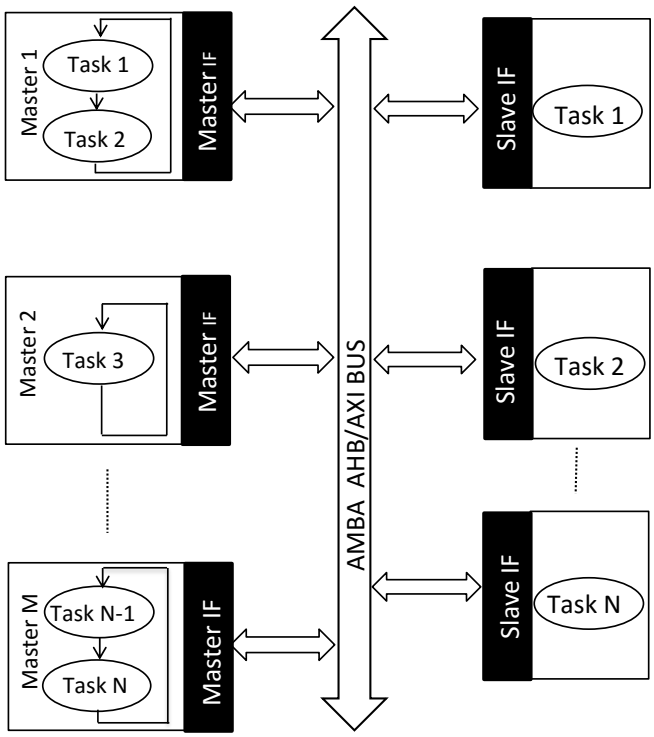


Figure 5.1: Target heterogeneous MPSoC Platform

At the heart of our proposed method is the measurement of time that each HWAcc mapped as a slave onto the heterogeneous MPSoC remains idle waiting for data from the master. This idle time is called *slack*. In order to measure this slack accurately, a cycle-accurate simulation of the entire SoC is required. Previous work on system-level design space exploration traditionally makes use of approximately timed high-level models to figure out the best possible system architecture. In these cases, the communication part is traditionally modeled using transaction-level models, hence they cannot accurately determine the communication overheads. The *slack* measured in our case is in turn used by the trust filter to pass a pre-defined

set of test vectors to the slave and compare the computed output with reference outputs also stored in the trust filter. Thus, the system will not have any performance penalty if the slack time available is large enough to accommodate this test phase.

The pre-defined test vectors and reference outputs used to detect the presence of a hardware Trojan are taken from the test-vectors included in the testbench provided by the BIP provider. IPs typically come with a testbench and reference outputs to verify that the IP follows the specifications indicated in the data sheet. These test-vectors can obviously not trigger any hardware Trojan, if present, as the IP vendor would immediately expose himself. Thus, this work assumes that these test-vectors can be used to test the correct functionality of the IP. Building complete behavioral SoCs has two unique advantages: First, it allows to use the system-level design tools available in state-of-the-art HLS tools like bus interface generators and secondly, the use of cycle-accurate model generators to accurately measure the slack of each slave. The proposed flow can be decomposed into two main parts. The first generates the entire SoC and builds an executable cycle-accurate model of the system in order to measure the slack of each slave while the second inserts different types of trust filters into the SoC depending on the slack reported in the previous stage. The next subsections describe these two stages in detail.

5.1.6 Behavioral MPSoC Generation

One of the uniqueness of this work is the generation of completely synthesizable C-based MPSoCs by using a bus generator provided by the commercial HLS tool used in this work [40] and its cycle-accurate model generator. Fig. 5.2 shows an overview of this stage. This first stage can be further subdivided into 4 main steps:

Step 1: Bus interface annotation. The very first step in our proposed flow is the modification of the BIPs to include the bus interfaces. As mentioned previously, commercial HLS

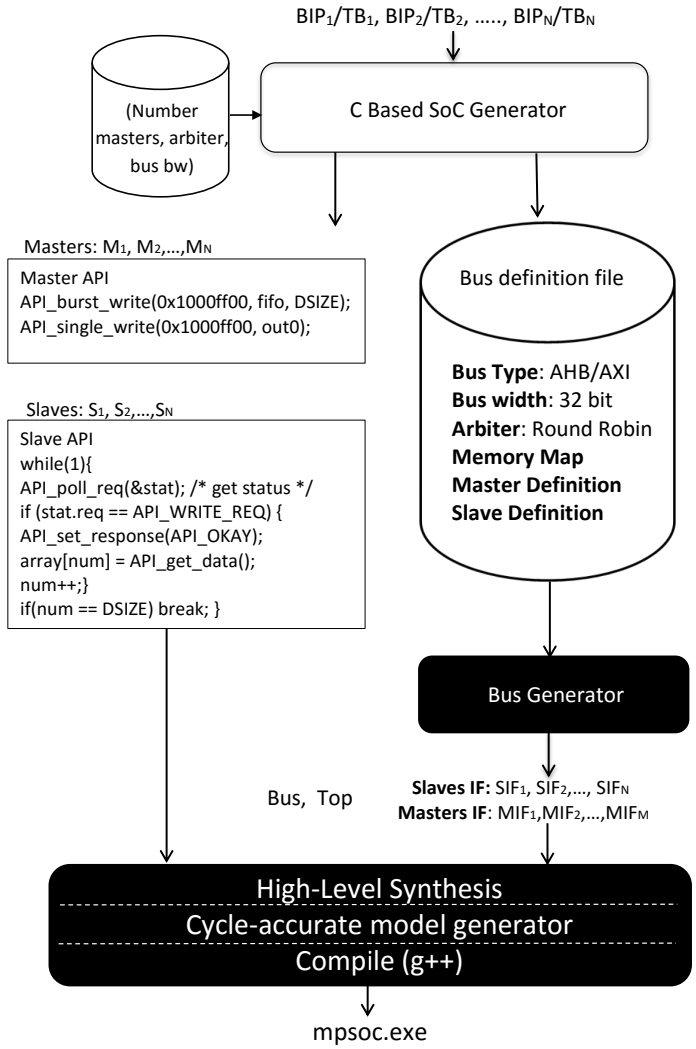


Figure 5.2: Design flow overview

tools now provide bus interface generators in the form of synthesizable APIs. These are function calls to read and write from the bus but will be synthesized into the bus interface circuit. This approach has numerous advantages over traditional RTL design methods. For example, the bus is completely abstracted away and the designer can easily modify the bus type by only using a different API. In our case, an AMBA AHB bus interface is chosen. The BIPs are all automatically instrumented using a simple perl script.

Step 2: Bus Generator. Once all the BIPs have been instrumented with the bus interface API, the next step automatically generates the bus definition file (bdef) for the bus generator used in the work. This bus generator takes as inputs all the instrumented BIPs and the bus

definition file and generates synthesizable ANSI-C descriptions for the bus interfaces, the bus itself and a top module. The bdef includes the bit-width of the bus, the arbiter (fixed or round robin) the number of masters and slaves and their memory map. In this work, the bus bit-width is set to 32-bit and the arbiter to round robin. Once the bus definition file is created, the bus generator is called and the aforementioned synthesizable system generated.

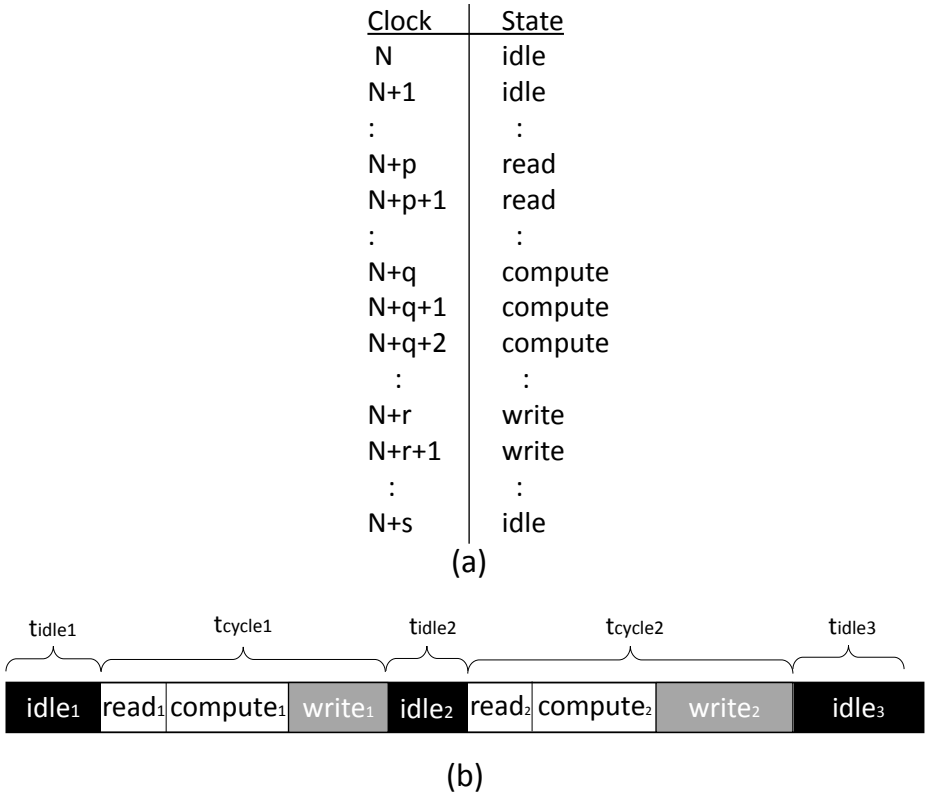


Figure 5.3: Slave slack estimation example; (a) report generated after simulation. (b) timing chart of report.

Step 3: Cycle-accurate model generation. Because HLS is a single process synthesis method, each of the BIPs, bus interfaces, and bus generated in step 2 have to be synthesized

separately. The results of the HLS is then passed to the cycle-accurate model generator, which creates a cycle-accurate SystemC model of the entire system.

Step 4: Slaves' Slack Estimation. Once the cycle-accurate model has been generated, our method continues by automatically instrumenting the SystemC files of each slave in the cycle-accurate model in order to report the number of cycles in which each slave is active or not. This automatic instrumentation reports, when the model is executed, the clock cycles in which the slave is waiting for data from the master, the clock cycles at which it is reading the data sent by the master, cycles doing its intended computation and finally cycles sending data back to the master in a text file. Fig. 5.3(a) shows an example of a report generated for each of the slaves to be protected. Fig. 5.3(b) shows the timing diagram that can be constructed from this report. The cycle through which the slave goes periodically repeats itself as follows: idle $\rightarrow$ read data sent by the master $\rightarrow$ perform computation $\rightarrow$ return data back to master $\rightarrow$ idle. The complete cycle takes t_{cycle} to be completed. It should be noted that $t_{cycle1} \neq t_{cycle2}$, with $t_{cycle1} = t_{read1} + t_{compute1} + t_{write1}$, where $t_{read1} = t_{read2}$, $t_{compute1} = t_{compute2}$ and $t_{write1} \neq t_{write2}$. The time taken to read data sent from the master (t_{read}) does not change between executions once the master initializes the communication and has been granted control over the bus. Similarly, the time taken to compute the results ($t_{compute}$) also does not change once the slave has read all the data sent by the master (the slaves used in this work do not have data-dependent loops). The only element that changes between two executions is the write back time (t_{write}). This is because the slave has to wait for the master to regain access to the bus in order to send the data back to it. This can take the different number of clock cycles depending on the bus availability. Moreover, the idle (t_{idle}), times between executions also change due to this exact same effect, hence $t_{idle1} \neq t_{idle2} \neq t_{idle3}$ and in particular $t_{idle2} < t_{idle1} < t_{idle3}$. Thus, once the simulation finishes, our method reads the report files of each slave and outputs the smallest idle time ($t_{idlesmallest}$) for each of them,

which is the same as the maximum slack available, $slack_{slave_i}$=$floor(slack_1,...,slack_q)$. In the example given in Fig. 5.3(b) $t_{idle2} = SL_{BIPn}$.

It should be noted that the way that the tasks are assigned to the different processors, does affect the slack of each slave. In this work, we, therefore, assume that the task scheduler has been programmed to balance the load across the system and that tasks are equally distributed across the processors.

5.1.7 Trust Filter

Once the minimum slack for each slave has been determined, our method continues by automatically inserting a *Trust filters* at each slave, with two options available:

$TF = \{typeI, typeII\}$. Thus, at the end of this stage, each HWacc will have a trust filter assign to it, e.g. $(BIP_1 \leftarrow typeI), (BIP_2 \leftarrow typeII), \ldots, (BIP_N \leftarrow typeII)$. By default, a trust filter is inserted for each slave, although the designer can manually overwrite this and specify at which slaves to insert the trust filters as some slave can be considered *safe*, e.g. in-house developed and hence do not need to be checked for hardware Trojan.

As the name indicates, trust filters are reliance circuits which are connected to the slaves and which can detect hardware Trojan at runtime. The trust filters do not avoid the activation of the Trojan and only detect these. Based on the type of slave and the available slack time obtained in stage 1, our method generates two different types of trust filters.

Type I - Autonomous Trust Filter: This is a fully autonomous trust filter, which uses the slack time between each transaction to verify if a hardware Trojan has been triggered. Thus, the slack time reported in the previous stage should be large enough to allow the filter to pass the test vectors to the slave, read the results and compare the results with the reference outputs stored in the filter. An interrupt signal is raised if the outputs do not match the reference outputs. In particular, if $t_{BIP_i} < slack_i$, where $t_{BIP_i} = t_{read\_tf\_i} + t_{compute\_tf\_i} + t_{compare\_tf\_i}$, with $t_{read\_tf\_i}$ being the number of clock cycles it takes for a HWacc to read the data to

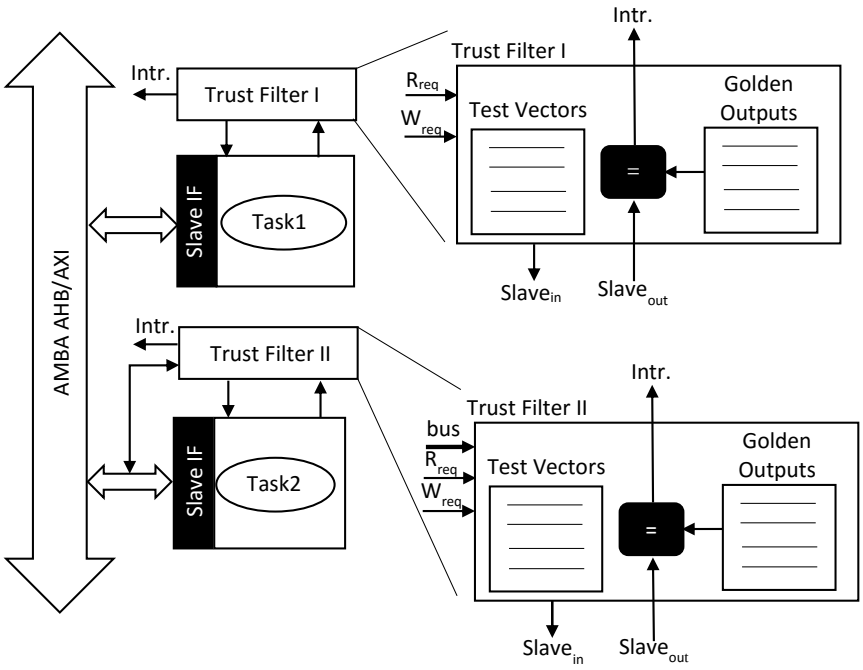


Figure 5.4: Trust Filter attached to slaves

compute a new output, $t_{compute\_tf\_i}$ the latency in clock cycles to produce a new results once a new input has been read and $t_{compare\_tf\_i}$ the time needed to compare the output with the reference output stored in the filter. After HLS, all of these three timing elements are known, and hence, the type of filter can be accurately assigned.

Fig. 5.4 shows the internal structure of this filter, where type I has 3 inputs and type II has 4 inputs. The inputs for this filter type are the read and write request from the AMBA

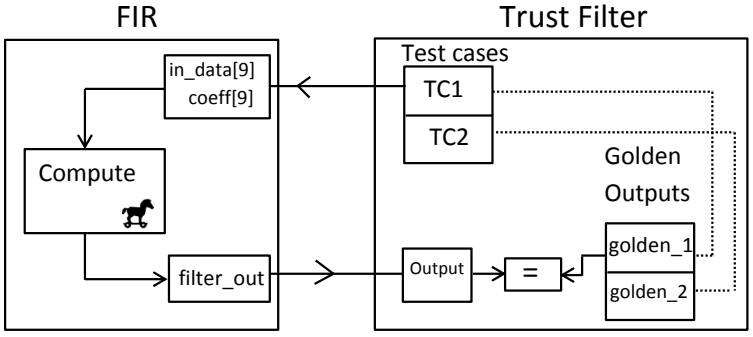


Figure 5.5: An example to demonstrate the number of test-cases needed for the trust filter

bus (R_{req}, W_{req}), and the output generated by the slave. The R_{req} is used to monitor when to start the verification. Based on the AMBA AHB bus protocol, the slave will generate a R_{req} pulse when sending data to the master. The slave will then remain in idle mode waiting for new data from the master once the data has been sent, indicated by a W_{req} pulse. Type I trust filters do not require to monitor the W_{req} signal as stage 1 reports that the filter has enough time to conduct the validation. Nevertheless, to avoid any timing problems, the trust filter also monitors the W_{req} and potentially halt the verification process in case that the master starts sending data before the trust filter has finished.

Type II - Manual Trust Filter: For some of the slaves' applications it can not be guaranteed that the available slack time between two transactions is sufficient for the particular slave to do the computation on the input data sent from the trust filter (e.g. aes case), with $slack_i < t_{BIP_i}$. Also, for some of the applications (e.g. sobel) the slave stores intermediate results in a buffer, which are needed in the next computation cycle. Applying the trust filter test vectors to the slave would completely flush this data, hence making the next computation's result invalid. In order to deal with these type of cases, this second type of trust filter is activated manually from the master. E.g. in the sobel case, after a full picture frame has been computed, the master sends a *start_trojan_check* signal in order to start the verification. This type of filter has an extra input compared to type I as it can be seen from the Fig. 5.4 (data bus). In order for the master to start the verification, it writes to the memory mapped slave the *start_trojan_check* code (in this case $0X8000$) and the filter starts the verification. When finished, it returns a finished signal to the master so that this can resume normal operation. This second type of filter has obviously the drawback of the performance penalty because it has to manually start the verification process and wait for it to finish, but on the other hand, has the advantage that it can guarantee 100% that the verification will be conducted successfully. In type I filters, our detection method relies on the slack estimation of stage 1.

In the case that the workload changes over time, the slack time could also change and hence the filter might never have the time to perform a full verification.

It should be noted that the designer can parameterize the behavior of the filters in case that these have not finished the verification cycle and a new write request from any master arrives. The designer has the option to set the filters up in such a way that the slave will not acknowledge the write request until the trust filter has finished or it can be set up in such a way that it will stop the verification process and deal with the master's request immediately. Both options have the strong and weak points. The first guarantees that the slave will always be verified. The drawback is that it might lead to a potential performance degradation, while the latter, does never lead to any performance degradation, but might lead to not being able to evaluate if the hardware Trojan has been triggered.
Our method automatically parses the BIP testbench and extracts two test-vectors that lead to different output results. The generation of the trust filter is fully automatic through a set of perl scripts. The inputs and outputs are as follows:

Trust filter generator Inputs: Slave latency, test vectors, reference outputs, type I or II, bus bandwidth.

Trust filter generator Outputs: Synthesizable ANSI-C code with embedded test-vectors.

According to Table 5.1, the types of Trojan addressed in this work have a payload with memory (Combinational with memory (CWM) and sequential with memory (SWM)). Hence the effect of Trojans lasts for a significant number of clock cycles. Intuitively, two test-vectors are necessary for the trust filter to detect the Trojans. Consider an example of FIR filter which is used as a slave as shown in Fig. 5.5, where TC1 is the first test-case inputs, TC2 is the second test-case inputs, golden_1 is the output corresponding to TC1, golden_2 is the output corresponding to TC2, in_data and *coeff* are the slave inputs, and *filter_out* is the slave output. The trust filter sends the first test-case (TC1) to the slave and the slave in-turn sends the output (filter_out) back to the trust filter. The trust filter compares the obtained

Table 5.2: Complex System Benchmarks and Hardware Trojan Type Description

Bench	Trigger/payload/filter	S1	S2	S3	S4	S5	S6	S7
sobel	Trigger	comb	seq		comb	comb		seq
	Payload	mem	mem		mem	mem		mem
	Filter type	II	II		II	II		II
md5c	Trigger					comb	comb	
	Payload					mem	mem	
	Filter type					I	I	
fir	Trigger				comb		comb	comb
	Payload				mem		mem	mem
	Filter type				I		I	I
interp	Trigger						comb	
	Payload						mem	
	Filter type						I	
bsort	Trigger		comb	seq	comb	seq	comb	
	Payload		mem	mem	mem	mem	mem	
	Filter type		I	I	I	I	I	
kasumi	Trigger			comb	comb	comb		
	Payload			mem	mem	mem		
	Filter type			I	I	I		
aes	Trigger							comb
	Payload							mem
	Filter type							II
ave8	Trigger						seq	comb
	Payload						mem	mem
	Filter type						I	I
adpcm	Trigger			comb		seq		
	Payload			mem		mem		
	Filter type			I		I		
slaves		1	2	3	4	4	5	5
masters		1	1	2	1	2	2	3

result from the slave with the golden output already stored in the trust filter. The trust filter raises the interrupt signal to indicate the presence of Trojan if there is any mismatch in the value between the obtained output and the golden output. If the output matches the stored golden output then the trust filter sends the second test-case (a unique test-case which should lead to a different output than the first test case). This second test case is important to cover the case that the Hardware Trojan happens to generate the exact same output as the first test vector. Although very unlikely, as the number of output combinations is 2^{bw}, where bw is the bitwidth of the output.

Consider {golden_1, golden_2} = {71, 129} are the outputs for the two unique set of test-cases TC={TC1, TC2} stored in the trust filter. Suppose the Trojan is activated in the slave and FIR filter output (filter_out) is equal to 71 (accidentally matches with the gold_1, which is highly unlikely) then the trust filter is unable to detect the presence of Trojan in the slave (FIR). Since the Trojan payload (CWM and SWM) will persist for the subsequent cycles (we consider that it has memory), the FIR filter generates the same output (i.e., filter_out = 71) for the second test-case (TC2) as well. This leads to the mismatch of the filter_out with the second stored golden output (golden_2 = 129). Hence, by incorporating the second test-case (TC2) in the trust filter, the rare escape of the Trojan can be avoided.

5.1.8 Experimental Results

Experimental Setup

Different computational intensive application, amiable to hardware acceleration, were selected and grouped together into complex systems in order to test our hardware Trojan detection method. Table 5.2 shows the different system configurations ranging from systems with a single master and single slave (S1) to more complex systems with 3 masters and 5 slaves (S7). The slaves were taken from the Synthesizable SystemC benchmarks suite (S2CBench) [107], where sobel is a 3x3 edge detection filter, md5c is message digest algo-

rithm for cryptographic applications, fir is a 9 tap FIR filter, interp is a 3 stage interpolation filter, bsort is a 8 element sorting algorithm, kasumi is a block cipher, aes is an Advanced Encryption Standard specification, ave8 is an average of 8 number algorithm and adpcm is an adaptive differential pulse coding decoder encoder.

To verify that our proposed method can efficiently detect hardware Trojan, different types of Trojans with different trigger mechanisms (combinational and sequential), all *with a payload with memory* were inserted in each of the slaves with trust filters attached, as shown in Table 5.2. This table also shows the type of filter used for each of the slaves. It can be seen that type II filters are only used for the sobel and aes cases. For the rest of the cases, the type I filter could be used. In the case of the sobel filter the verification is done after one full image has been filtered and in the aes, every time a value has been encrypted/decrypted. All the Trojans with combinational trigger mechanisms are triggered by the master by sending the trigger combination through the bus. The hardware Trojans inserted into these synthesizable benchmarks are similar to the ones in [105]. Some of the common payload mechanisms used in this work include functionality changes, secret key leakage, denial of service and the manipulation of the output signal. The HLS tool used in this work is CyberWorkBench from NEC [40], which includes a bus generator and cycle-accurate model generator. The experiments are conducted on an Intel i7-4790 @3.60GHZ CPU and 8 GB memory. The target HLS frequency in all cases set to 100MHz. The target technology used, Nangate 45nm and each benchmark were synthesized using the default synthesis options of the HLS tool.

In terms of how the tasks were mapped onto the different masters, a simple load balancing approach is taken, assigning an equal number of tasks to each master.

It should be noted that the proposed method has not been prototyped on an FPGA. Our work makes use of cycle-accurate SystemC models for the entire MPSoC. These models mimic the behavior of the MPSoC cycle-accurately and hence lead to extremely accurate

Table 5.3: Experimental Results

System	Area of MPSoC [μm^2]	Total Area TF [μm^2]	Area Overhead [%]	Cycle accurate simulation without TF[s]	Cycle accurate simulation with TF[s]	Δ_{sim} [%]	performance penalty on HW[%] (over 10,000 cycles)
S1	6,889	445	6.45	251	258	2.7	0.3
S2	36,692	631	1.71	311	323	3.71	0.3
S3	43,302	893	2.06	465	481	3.32	0
S4	44,296	1203	2.71	506	523	3.2	0.3
S5	49,024	1246	2.54	743	768	3.25	0.3
S6	82,753	1523	1.84	853	881	3.17	0
S7	92,770	1902	2.05	964	1023	5.7	0.63
Avg.			2.76			3.57	0.26
Geomean	40,620	1,012		526	546		

TF:Trust Filter. **HW**:Hardware. Δ_{sim}: Difference in the cycle accurate simulation time.

results. The work also inserts these filters completely automatically into any of the slaves that want to be monitored, hence the setup effort is negligible. The method is fully automatic.

Experimental Results and Discussion

Table 5.3 shows the results of the area, cycle-accurate simulation running time and final SoC performance penalty for the different master-slave combinations. It is seen that area of the trust filter is negligibly low compared to the area of total MPSoC. When simulating the entire system, the cycle-accurate model simulation running time will also be affected, as the complexity of the entire system increases. Columns 5-7 show the runtime differences, indicating that on average the simulation time is increased by 3.57%. Finally, column 8 shows the performance penalty in the SoC in % of extra clock cycles required to execute all the tasks on the silicon extracted from the cycle-accurate simulation due to the trust filters. Systems with the only Type I filters do not have any performance penalties as the slaves' idle time is used for the verification. Only systems with Type II filters show a minor penalty which requires on average 0.26% extra clock cycles. Obviously, in Type II filters the master *manually* determines when to perform the verification, thus, this value could increase or decrease based on the verification frequency.

In order to fully characterize the proposed system, Table 5.4 shows the worst case performance penalty analysis of the manual trust filter (type II) for the different systems. In this

Table 5.4: Worst Case Performance penalty analysis of manual trust filter for different systems

System	Performance Penalty in [%] over 10,000 cycles for different test intervals			
	after each testcase	after 10 testcases	after 50 testcases	after 100 testcases
S1	100	10	2	1
S2	78.3	8.3	1.78	1.08
S3	100	10	2	1
S4	88.78	9.26	2.07	1.2
S5	99.5	10.3	2.3	1.3
S6	100	10	2	1
S7	96.96	11.46	2.38	2.16
Avg.	94.79	9.90	2.07	1.24

case, each slave has attached the manual trust filters and we measure the performance degradation of this configuration when the master requests for different intervals the verification of each slave. From Table 5.4 it can be observed that this verification is performed right after a single test-case is sent, after 10, 50 and 100. For each case, the performance overhead varies between 100% to 9.90%, 2.07% and 1.24% depending on the interval at which a verification is requested. As expected, the higher the interval, the lower the performance overhead. The worst case would be for the S1 system as it only has a single slave. Although the performance overhead can be significant, depending on the verification request interval, this method assures 100% Trojan detection. One alternative to the manual trust filter when extremely secure systems are required is, as mentioned in the previous section, to instantiate type I autonomous filter, which ignore any master request until the verification has finished.

One case that this work does not contemplate is the case in which the attacker can alter the trust filter before it is either taped-out or at the foundry. One easy way to circumvent the proposed trust filters method would be to change the stored outputs in the Trust Filter so that the Trojan may evade the detection. This is a potential weakness of any pre-silicon hardware Trojan detection technique. Most detection method can be potentially breached if it is assumed that the fabrication process can modify the countermeasures before manufacturing.

Anyway, for this particular case, one easy way to partially detect that the trust filter has been manipulated is to insert a standard built-in-a-self check mechanism, similar to the BIST (Built-in-Self-Test) of VLSI circuits. Also, we can incorporate the technique mentioned in [142]. We leave this for our future work.

Overall we believe that our hardware Trojan detection method presented in this work is very efficient and have shown that it leads to small area and performance overheads.

5.2 Hardware Trojan Detection in Dynamically Re-configurable FPGAs

ICs are becoming extremely complex being most of the cases Systems-on-Chip (SoC). These SoCs include microprocessors, embedded memories, different types of external interfaces and a set of Hardware Accelerators (HWaccs), all interconnected through a shared bus, bus hierarchy or even a Network on Chip (NoC). SoC designers are often system integrators, integrating a set of in-house and third party IPs (3PIP) onto the SoC and then outsourcing the fabrication to another company.

One particular architectural novelty in state of the art complex SoCs, is the use of runtime reconfigurable architectures for the HWaccs mapped as slaves on these heterogeneous SoCs, also called Reconfigurable SoCs (RSoCs). Fig. 5.6 shows one example. These are typically coarse-grained runtime reconfigurable arrays (CGRRA), which are runtime reconfigurable. This means that their functionality can be updated at runtime. This architecture makes these systems extremely efficient as different tasks can be mapped onto a single CGRRA, but at the same time makes them extremely vulnerable to malicious alterations that can control the functionality being executed on them at runtime. Hence, methods to secure these CGRRAs need to be investigated.

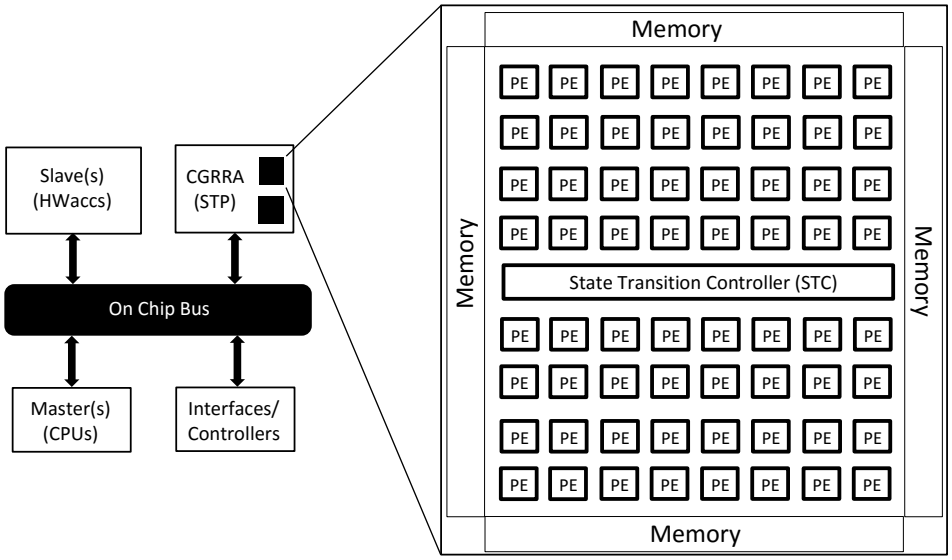


Figure 5.6: Coarse Grain Runtime Reconfigurable Array (CGRRA) IP in a Reconfigurable SoC

This section focuses on Hardware Trojan detection in CGRRA IPs embedded into RSoCs. As illustrated later on when the threat model is explained in detailed, this work assumes that a rogue designer can insert a hardware Trojan in the controller part of the CGRRA to change its configuration. The contribution of this work can be summarized as follows:

1. Propose a methodology to avoid any Hardware Trojan to be executed on a CGRRA by fully utilizing the State Transition Controller's (STC) memory and hence not allowing *space* for the Hardware Trojan.

2. Present a method to avoid the CGRRA to load the Hardware Trojan from external memory by *pinging* the CGRRA at regular intervals.

3. Detect the existence of Hardware Trojan in case that the CGRRA can access external Programmable Read Only Memory (PROMs) with new STC configurations.

4. Present extensive experimental results of different RSoCs validating our method making use of cycle-accurate SoC models.

5.2.1 Related Work

Hardware Trojans can also be inserted in reconfigurable devices like FPGAs. FPGAs are particular interesting targets as they can be altered by manipulating the corresponding bitstream which configures the device. The first successful real-world FPGA hardware Trojan insertion demonstration into a commercial product can be found at [151]. The authors manipulate the FPGA bitstreams to alter the AES-256 algorithm in such a way that it turns into a linear function which can be broken into 32 known plain text-cipher text pairs. After the manipulation, the attacker is able to obtain all the user data from the cipher texts. This work highlights the practical relevance of bitstream modification attacks that became realistic due to FPGA bitstream manipulations. The work in [152] and [153] proposes other methodologies for hardware Trojan detection in reconfigurable systems like FPGAs. All these previous works are related to reconfigurable systems that are static, i.e., the configuration remains the same during its execution, but not dynamically reconfigurable ones like the one presented in this work. CGRRA have unique challenges that have not been addressed yet in this previous work. To the best of our knowledge, this is the first work which addresses the hardware security issues in dynamically reconfigurable systems.

5.2.2 Stream Transpose Processor

The target architecture used in this work is a Coarse-Grained Runtime Reconfigurable Array (CGRRA) and in particular the Stream Transpose Processor (STP) from Renesas Electronics [154]. It should be noted that without any loss of generality, our proposed method can be applied to any runtime reconfigurable architecture, as they are mostly based on a similar architecture (especially the CGRRA ones) [155].

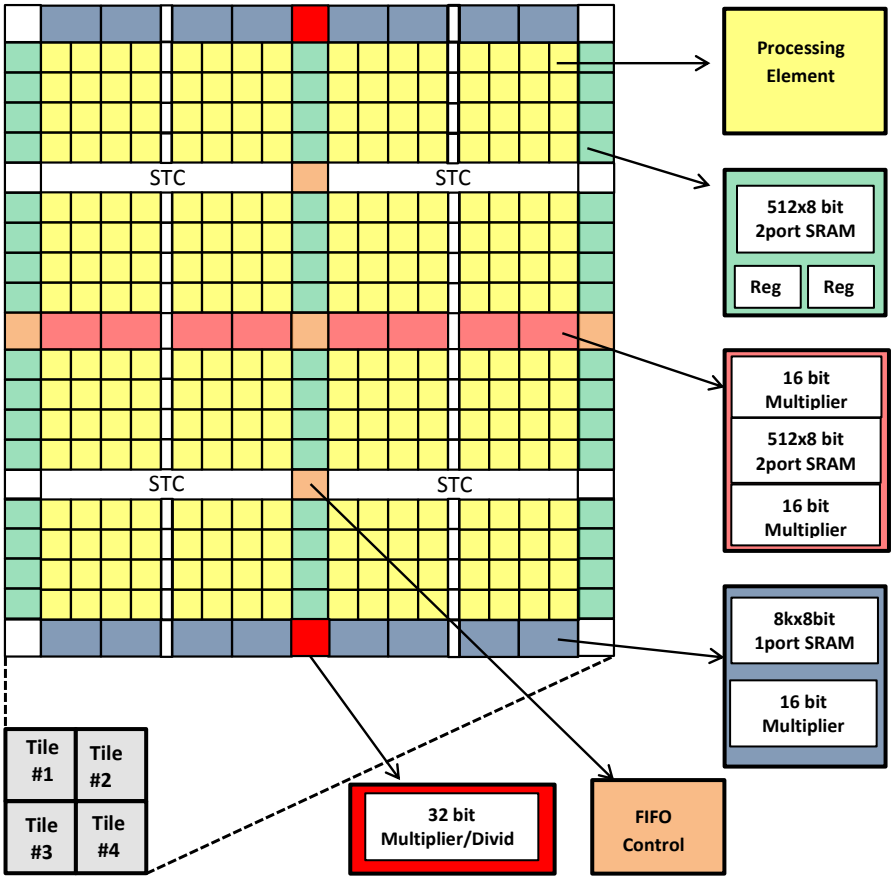


Figure 5.7: STP tile structure

5.2.3 STP Architecture

The STP is a coarse-grained, runtime re-configurable core that can be integrated into any SoC as a loosely coupled HWaccs (it is sold as an IP and not as a stand-alone FPGA). The primary unit of STP is called *tile*. The latest STP consists of four tiles. Fig. 5.7 shows the structure of a single tile, where each tile consists of 64 Processing Elements (PEs), the STC, on-chip embedded memory blocks and 16-bit multiplier units. VMEM is an 8-bit, 512-word memory with two input and output port and HMEM is an 8-bit, 4096-word memory unit with one input and an output port. The STP is a coarse-grained FPGA because is can perform bytes size operations. The data path consists of an array of PEs and memories. Fig. 5.8 shows the structure of a PE which has an 8-bit arithmetic logic unit (ALU), a data manipulation unit (DMU) for 8-bit shift operation and 1-bit logic operations, two 8-bit register file units

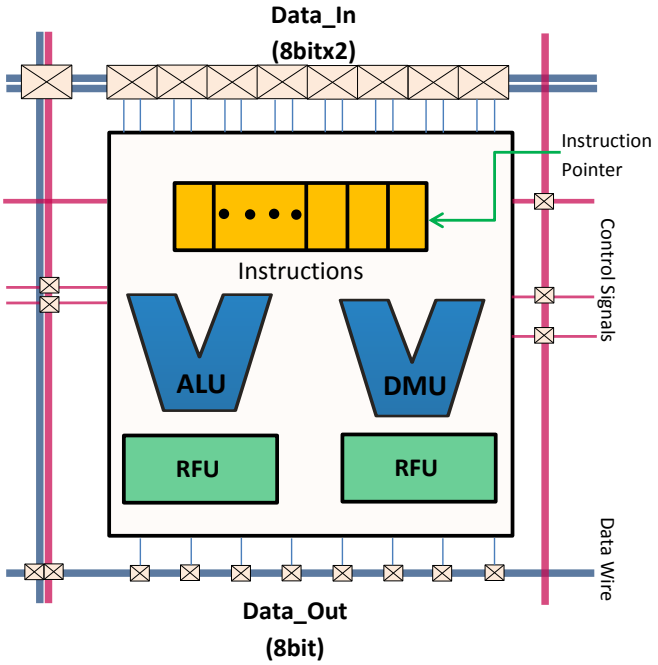


Figure 5.8: Architecture of Processing Element (PE)

(RFUs) which hold 8-bit data. The ALU and DMU can be combined to extend the operand to a 16-bit adder or 4-to-1 multiplexer.

One unique architectural feature that makes this architecture extremely flexible and efficient is that the data path is reconfigured every clock cycle in less than 1ns. A State Transition Controller (STC) located in the center of the CGRRA holds a set of *contexts* with the information on how the datapath should be configured. The latest version of the STP can hold up to 64 different configuration codes (contexts), but can also load more contexts from external, off-chip, memories through its DMA controller. This is important because it will impact our Hardware Trojan avoidance and detection method.

5.2.4 STP Design flow

One of the uniqueness of the STP is how it is configured. Because of the architectural details described in the previous subsection, it makes sense to configure it using High-Level Synthesis instead of traditional logic synthesis. In RTL, the hardware is manually laid out spatially on the circuit, parallelizing the application to be executed on it using a Hardware

Description Language (HDL) e.g. VHDL or Verilog. In contrast, in HLS a behavioral description (e.g. C or C++) is given as an input and it is parallelized based on the target underlying architecture, thus, it is better suited for this architecture. HLS can be defined as the process of converting an un-timed high-level behavioral description into an RTL description (VHDL/Verilog) that can execute it. HLS is composed of three main steps: (1) Resource allocation, (2) scheduling and (3) binding. In the first step, the number and type of resources (e.g. Functional Units) are extracted. These are in turn scheduled based on the target frequency, the resources' delays and their number and lastly different operations are mapped to the different resources. The target architecture typically obtained after HLS is an FSM and a data-path, where the FSM generates the control signals for the data (e.g. muxes). The FSM generated after HLS can be mapped onto the STP's STC while the data-path onto the PEs, making HLS extremely suitable for this architecture.

Fig. 5.9 shows the flow diagram of the STP configuration process. It takes as inputs a behavioral description given in ANSI-C and a set of constraints (e.g. target clock frequency, and technology libraries). It then performs HLS on it and then the typical FPGA physical design steps: Technology mapping, placement and routing. This back-end part also generates the code for the STC which in turn configures the data-path. As many contexts as states of the FSM are generated, where each context is loaded every clock cycle reconfiguring the STP's data-path in less than 1ns.

One of the byproducts of the synthesis process is an RTL simulation model which will be used in our experiments to test the effectiveness of our proposed detection method.

5.2.5 Hardware Trojan in DRPs

The architectural uniqueness of CGRRAs and in particular the STP used in this work makes it extremely vulnerable to hardware Trojan. In particular, the fact that the configuration is stored in the STC controller makes it susceptible to being manipulated so that the CGRRA

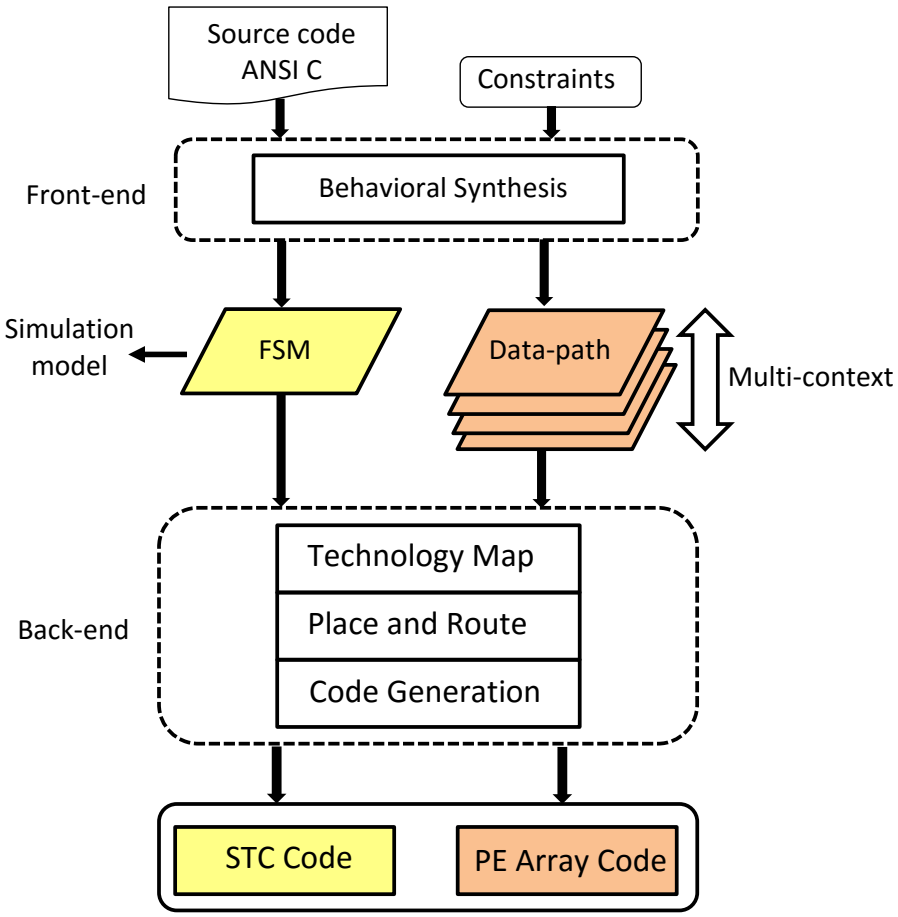


Figure 5.9: Configuration flow of STP

executes any other function not originally intended for. The threat model used in this works assumes that the Hardware Trojan is placed onto the RSoC by either the CGRRA vendor, hardware designers, the RSoC physical design team or at the manufacturing stage. It is assumed that at any of these stages the STC can be manipulated.

The trigger mechanism used in this work is very simple. The hardware Trojan is automatically triggered when the CGRRA is not being actively used by monitoring the DMA's read and write signals. Thus, right after the CGRRA returns the computed data, the hardware Trojan is triggered. Once any of the masters send new data to the CGRRA, the Hardware Trojan will immediately stop to serve the new request and the Hardware Trojan will resume its activity once the data has been returned to the master. We believe that this is the most

difficult scenarios of all, as the Hardware Trojan will never manifest itself under normal operations, simulation nor during the testing stage.

The payload, in this case, targets at heating up the RSoC and increasing the power consumption by actively switching all the PEs. Depending on the thermal characteristics of the IC, this could lead to the IC being burnt in the worst case, while in the best case it would consume more power and create a hotspot, which in turns would affect the reliability of the IC. The threat model assumes that the attackers know about the STC configuration and make use of unused context in the STC's memory to implement its behavior.

The STP, as mentioned previously, also allows through its DMA port to access the external memory to load more configurations onto the STC. This is important for the case that the application requires more than 64 contexts or in order to execute different applications onto the STP. It can intuitively be noticed that this can lead to a serious security problem as the hardware Trojan once triggered could load any application onto the STP. Thus, this work considers this case too.

5.2.6 Trojan Avoidance and Detection method

The proposed method is divided into two parts. The first part tries to avoid the actual insertion of the Hardware Trojan in the system, while the second part avoids the Hardware Trojan being loaded from external memory and if it is not possible to avoid the hardware Trojan to be triggered, detects its presence. These two methods are described in detailed as follows:

Hardware Trojan Insertion Avoidance: As described previously, the STC contains the configuration code (contexts) for the data-path part of the STP. For the STP family targeted in this work, the maximum number of context that it can hold is 64. This means that any unused context poses a potential threat as it can be used to host the Hardware Trojan. Thus, the idea behind the Hardware Trojan avoidance used in this method is based on making sure that for every application mapped onto the STP, it always make use of all the 64 contexts.

Fig. 5.10(a) shows an example where only 4 contexts are being used (C_{used}) out of 10 hypothetical maximum number of contexts (C_{max}) which can be used (4/10), in this case $C_{used} = \{C_0, C_1, C_2, C_3\}$. The rest of the contexts $C_{(4..9)}$ are unused and hence can be exploited by the attacker to implement the Hardware Trojan shown in Fig. 5.10(b). In this example shown, all the PEs are used to switch in order to consume more power and hence drain the battery faster and to generate a hot-spot. Part of the 64 PEs of the Hardware Trojan is devoted to jumping back to C_0 once a request from the master is received. This is important, as it avoids being detected. It should be noted that the Hardware Trojan does not need to use all the remaining contexts to be effective. One context is enough to implement its functionality. In this case, the Hardware Trojan just needs to stay in this context once triggered, switching all the PEs simultaneously.

Our proposed avoidance method, hence, requires making sure that all the context are fully utilized for any application. Thus, the maximum number of contexts should always be used $C_{used}/C_{max} = 1$ with $C_{used} = C_{max}$ (in the STP case 64) independently of the application mapped onto the STP. Listing 1 shows an example of how this is achieved. A snippet of a FIR filter computing the sum of products of some data and the filter coefficients is shown. When the main computational loop is fully unrolled, the latency of the circuit should be 1 (depending on the target synthesis frequency), which means that a single context is used. Thus, $C_{used} = 1$. This means that our method has to *fill* $C_{filled} = C_{max} - C_{used}$=64-1=63 remaining contexts in order to fully utilize the STC memory. This is achieved through a context adjustment function (context_adjust), called once the computation has finished.

Although behavioral descriptions do traditionally not have the notion of the clock and hence it is difficult to create a delay of a fixed number of clock cycles, most of the commercial HLS tools extend the C/C++ syntax with synthesis directives in the form of pragmas. The delay loop is composed of three important elements, which are highlighted in Listing 1. The first is that in order to guarantee the sequential execution of the delay loop a synthesis

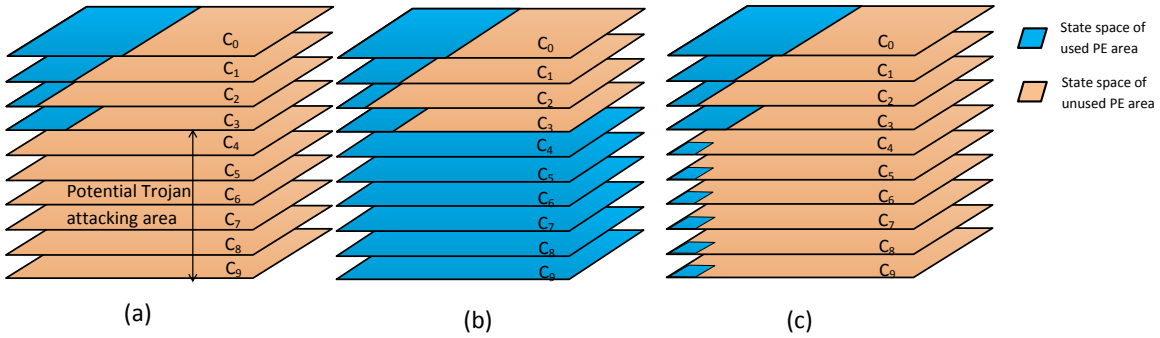


Figure 5.10: Utilizacion of PEs. (a) Typical usage (b) once HW Trojan is triggered (c) with proposed method

directive to not unroll the loop has been specified (in most cases, by default, HLS tools unroll all the loops in order to parallelize the loop body). The second is a simple logic operation as the loop body to avoid the synthesizer to optimize the complete logic away. Because the logic is very simple, it ensures that each loop iteration is executed in exactly one clock cycle and hence consumes a single context. The additional benefit is that it only consumes a single PE for the loop counter and hence has very little power overhead. Finally, the write request signal is also checked every clock cycle in order to exit the delay loop if a new write request from the master arrives. Fig 5.10(c) shows the results graphically.

If the STP can only use the contexts from within the STC, then this proposed method should be sufficient to guarantee that no Hardware Trojan can be inserted in the system. The main problem comes when the STP is allowed to load new configurations from external memories. This configuration is very usual to make the RSoCs even more flexible. In this case, the method just introduced is not sufficient as the Hardware Trojan could be loaded externally into the STP. Because it is virtually impossible from preventing externally stored configurations to have a Hardware Trojan (e.g. the PROM used to store them can be exchanged when the system is deployed) another method is required to avoid a hardware Trojan to be loaded into the STC from external memories and in case that it is loaded to at least mitigate the event that a Hardware Trojan is present by detecting it or from preventing it from being executed.

```
  :              :
for(i=0;i<TAPS;i++)
    sum += ary[i] * coeff[i];
// Context adjustment function
context_adjust(CONTEXTS, a);
}
void context_adjust(int contexts, int a){
    //pragma unroll=0
    for(int i=0;i<contexts;i++){
        a=a&0x1b;
        if(write_req)
         return;
    }
}
```

Listing 5.1: Context adjustments function

Hardware Trojan Detection: The CGRRA used in this work is mapped as a loosely coupled HWacc slave onto a shared memory mapped bus, which is a typical configuration in current RSoCs. This work assumes that the Hardware Trojan is built into this slave so that the Trojan is only active when the master does not require the slave to do any computation. When in idle mode, the CGRRA will load an STC configuration stored in an external PROM through its DMA port. The Trojan in this case, similar to the previous example will simply make use of all the PEs of the CGRRA. Once the master requests a new computation from the slave, this will load back from the external PROM the original design, compute the result and once in idle mode trigger the Hardware Trojan again. Similar to the previous case, this makes it extremely difficult to detect the Hardware Trojan. The only hint that anything might not work well is that the computation latency increases due to having to load contexts from outside memory.

The main idea behind our proposed detection method is to avoid the Hardware Trojan to be loaded into the STC. Knowing the STC memory size is 40 Kbytes and given a typical DMA channel capacity of 8Mbytes/s, it can be estimated that the time required by the CGRRA to load a new configuration is 500μs. This implies that if the master requests the CGRRA to perform any new computation within a 500 μs interval, the Hardware Trojan, to avoid being detected would reconfigure the CGRRA with its original design and hence the Hardware Trojan would never have enough time to be loaded into the STC. Obviously, this interval would need to be adjusted based on the final SoC implementation where the CGRRA is used. Instead of sending a full computation request to the CGRRA and in order to save power, the master only *pings* the slave. This avoids having to perform a full computation of the algorithm mapped onto the CGRRA, requiring less power at the same time.

```
if(MSB == 0)
    sum = fir(data, coef);
// Decrypt function if MSB=1
else
   return(decrypt_ping(data));
// Context adjustment function
context_adjust(CONTEXTS);
}
int decrypt_ping(int data){
z(0..7) = dat(0..7) ^  dat(8..15) ;
z(8..15) = dat(8..15) ^  dat(16..23);
z(16..23) = dat(16..23) ^  dat(24..31);
z(24..31) = dat(24..31) ^  key;
return z;
}
```

Listing 5.2: Context adjustments function

In case that the Hardware Trojan acts differently and continues loading the Trojan after the master has *pinged* the CGRAA, the *pings* itself is used to detect if a Hardware Trojan has

been triggered or not. In order to detect if the master wants the slave to perform a regular computation or to reply a pinged signal, the MSB value of the first transmitted byte is used to discriminate these two cases. an MSB=0 implies regular operation, while a MSB=1 implies that a ping signal was sent. This *ping* does not just return the data sent by the master, because this method could easily be compromised if the attacker knew about it, e.g. the attacker could build a Hardware Trojan that switches all the PEs, but at the same time implements the logic to return the *pinged* signal in parallel. To avoid this, the master encrypts a piece of data (4 bytes) and sends it to the CGRRA. The protocol followed between the master and the slave is straight forward. Listing 2 shows a snippet of the encryption method. The data is decrypted by the slave and sent decrypted back to the master. If the data is correct then the master knows that the CGRRA has not been corrupted. The encryption can be made more complicated, but in this case a simple method based on doing a pairwise logic XOR of consecutive bytes of the data sent by the master, including a last logic XOR with the key stored in the CGRRA is used. This key is only available to the master and hardcoded in the CGRRA. This method is safe enough because the STP can be reprogrammed anytime by the final user with the final encryption/decryption mechanism. Thus, the attacker does not have access to it during the SoC design and manufacturing stages.

We believe that these two methods should make CGRRA based designs much safer. To further increase security, these methods can be combined together. The experimental results section shows the effectiveness of these methods.

5.2.7 Experimental Results

Different computational intensive applications, amiable to hardware acceleration were selected from Synthesizable SystemC benchmarks suite (S2CBench)[107] and converted to ANSI-C because the commercial STP synthesis flow used in this work only accepts ANSI-C as input language. Table 5.4 shows the different benchmarks used and their complexity in

Table 5.4: Benchmark characteristics

Bench	lines	mul	add/sub	loops	arrays	funcs
ave8	24	0	7	1	1	0
fir	54	7	7	1	2	1
sobel	87	0	26	8	3	2
bosrt	52	0	0	2	2	0
kasumi	221	0	44	21	13	5
interp	91	10	14	5	5	1

terms of the number of lines of code, the number of operators, loops, arrays and functions. These applications mainly come from the signal processing and security domain. A standard I/O interface was included in all of them to read and write data from the shared bus using a synthesizable API provided by the STP vendor.

The CGRRA used in this work is the STP (DRP-II) from Renesas Electronics [156] which can have a maximum of 64 contexts and 4 tiles in it. *Musketeer* is the Integrated Development Environment (IDE) from Renesas Electronics used to synthesize the input code from ANSI-C to the code to configure the STP. This tool includes a HLS front-end as well as a physical design back-end. Each benchmark is synthesized with the default options set and a target HLS frequency of 50MHz. 50MHz might seem a low frequency, but this was the highest frequency which did not lead to any multi-cycle operation for all of the benchmarks.

Table 5.5 and Table 5.6 characterize our proposed methods in terms of area (number of PEs used and number of contexts for each application mapped onto the STP) and energy consumption respectively for the following three cases: case1-Trojan avoidance, case2-Trojan detection and case1&2, the combination of case1 and case2 for maximum protection. PE_{max} represents the maximum number of PEs used throughout all of the contexts, represented mathematically as follows:

$$PE_{max} = Max\{PE_{C_0}, PE_{C_1}, \ldots, PE_{C_{N-1}}\} \tag{5.1}$$

In the ave8 benchmark, column 2 indicates 10 (C_{19}), which represents that among the required 23 contexts to execute the benchmark, given in column 4, the context number 19 (C_{19}) requires the maximum number of PEs among all the other contexts. In this case $PE_{max} = 10$. PE_{all} in column 3 represents the total sum of all the PEs from all the generated contexts, represented mathematically as follows:

$$PE_{all} = \sum_{i=C_0}^{C_{N-1}} PE_{\{C_i\}} \tag{5.2}$$

E.g. for the ave8 benchmark, column 3 indicates that the total number of PEs used across all the 23 contexts is 107. Obviously, these PEs are shared across contexts and the total number of available PEs is constant and equal to 256 PEs (4 tiles, each consisting of 64 PEs). It should be noted that for case 1 and case 1&2 the total number of context are always equal to the maximum number of contexts that the STC can hold. In this case 64, as shown in columns 6 and 10.

Table 5.5: Experimental Results: Area Overhead

Benchmark	Original			Trojan Avoidance (case 1)		Trojan Detection (case 2)		Trojan Avoidance & detection(case 1& 2)	
	Area [PEs]		# contexts						
	Maximum PEs utilized in a single context(PE_{max})	Total area (PE_{all})		Area [PEs] (PE_{all})	# contexts	Area PEs (PE_{all})	# contexts	Area [PEs] (PE_{all})	# contexts
Ave8	10(C_{19})	107	23	148	64	151	27	188	64
FIR	15(C_{40})	236	46	254	64	280	50	303	64
Sobel	23(C_9)	119	17	166	64	163	21	207	64
Bsort	30(C_5)	105	11	158	64	149	15	207	64
Kasumi	29(C_{28})	212	33	243	64	256	37	284	64
Interp.	40(C_{20})	301	28	337	64	345	32	378	64
Avg.	25	180	26	218	64	224	31	262	64

Our proposed method obviously also incurs in extra power overheads. Thus, it is important to measure these. The main problem is that STP synthesis tool does not report any power figures. Thus, in order to estimate the power consumption, a single PE based on Fig. 5.8 was modeled in RTL (Verilog) and synthesized targeting a Xilinx Virtex-4 FPGA. The main reason for targeting this FPGA is that this FPGA family and the technology libraries

Table 5.6: Experimental Results: Energy Overhead

Bench	Energy Consumption				
	Original (in nJ)	Case 1(in nJ)	Case 2(in nJ)	Case 1&2(in nJ)	with Trojan(nJ)
Ave8	0.70	0.794	0.79	0.84	3.36
FIR	0.96	1.10	1.05	1.19	3.44
Sobel	1.40	1.54	1.52	1.64	3.78
Bsort	1.68	1.82	1.75	1.86	4.24
Kasumi	1.18	1.34	1.27	1.41	3.68
Interp	2.10	2.24	2.19	2.32	4.46
Avg	1.33	1.47	1.42	1.55	3.36

used for the STP are both 90nm [157]. We then measured the average power consumption using Xilinx's Power Estimator (XPE) [158]. It should be noted that the power calculated using this method is not too accurate, but serves as an indicator of the relative differences between the different implementations. The energy reported is the total energy within 64 clock cycles (matching the total number of contexts that the STC can hold).

Table 5.6 shows the energy consumption results of the original design without any hardware Trojan protection mechanism (column 2), with only case1 method (column 3) with case2 (column 4) and lastly with both protection methods (column 5). It can be observed that our proposed methods on average increase the energy consumption by 9.5% and 6.3% respectively, while on average 14.2% for the combined approach (case1&2). To put things in better context, the last column shows the energy consumption of the hardware Trojan implemented on the STP switching all of the PEs in each of the 64 contexts, which has an average energy consumption overhead of 60.4%.

Our proposed methods will also affect the critical path and thus the maximum frequency. Fig. 5.11 shows the increase in the critical path delay for all 3 scenarios (case1,2 and 1&2) when compared to the critical path of the original design without any hardware security measure. On average case 1 method only degrades the critical path by 1.9%, while case 2

degrades it by an average of 6.1%. This is mainly due to the insertion of a mux when our method reads the MSB to decide if the actual design has to be executed or if the incoming data is a *ping* from the master which has to be decrypted. When combining both methods (case 1&2) the critical path increases on average by 7.8%. We believe that this increase is reasonable, especially considering the added security to the system.

The previous results described our proposed detection methods quantitatively. It is nevertheless also required to proof qualitatively that our proposed method works. It should be noted that it is impossible to demonstrate that case1 works as it is a method designed to avoid a hardware Trojan to be inserted into the system. Hence, only case2 is tested. In order to do so, we implemented a complete SoC instantiating a CGRRA as a memory mapped HWAcc slave in it. As mentioned in section II and shown in Fig. 5.9, one of the by-products of the design flow is an RTL (Verilog) simulation model of the CGRRA application mapped onto the STP. This simulation model is inserted as a component in a SoC built out of a single master and the CGRRA connected through am AMBA AHB bus. In order to build this SoC a system generator called CybusM, within NEC's CyberWorkBench [40] is used. This bus generator takes as inputs the bus type (i.e., AHB, AXI), arbiter type (round robin used in this case), bus bit width (32 bit in this case), a number of masters, number of slaves, and their memory map and generates a cycle-accurate model of the entire system. This cycle-accurate model is given in SystemC or RTL. In this case, as the simulation model generated by Musketeer is in RTL, this second option is chosen. This allows us to measure the effectiveness of our proposed detection techniques. Hence, a Hardware Trojan that tries to access external memory when the CGRRA is not processing any data is instantiated.

When instantiating case2 Trojan avoidance and detection method and having the master *pinging* the HWacc every certain interval the simulation results of the cycle-accurate model showed that the Hardware Trojan could not be loaded from external memory and the correct output was always received by the master.

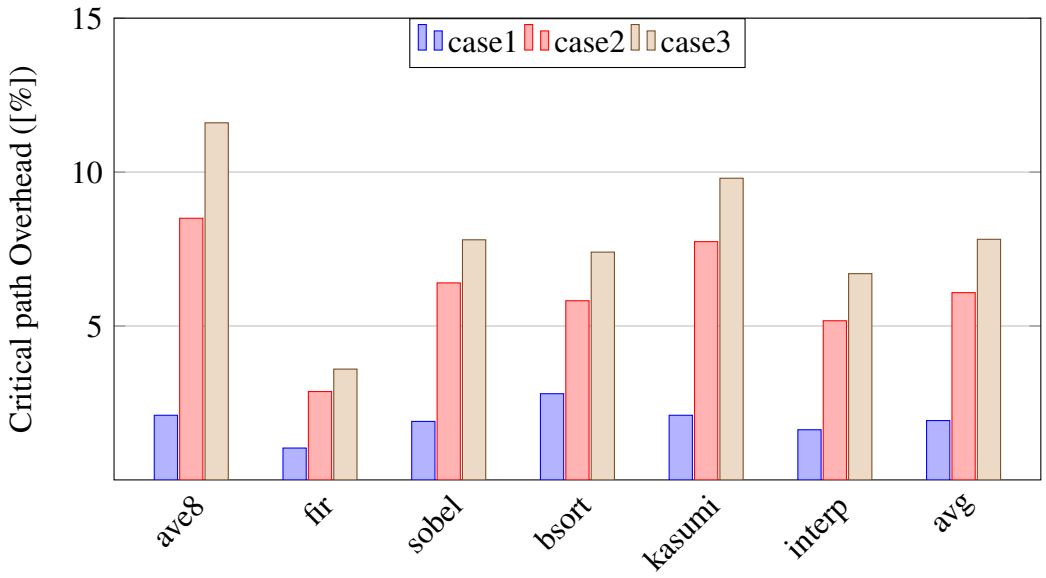


Figure 5.11: Critical path overhead comparison between original designs and case1,case2 and case3 methods.

From these results, we believe that our hardware Trojan detection method for CGRRA is very efficient and that it can cover a wide range or trigger and detection mechanisms with low power/energy overheads and limited critical path delay increase.

5.3 Summary

This chapter has presented techniques to detect hardware Trojans at the system level. The first part of the chapter has presented a novel area efficient circuit to detect hardware Trojans in MPSoCs. Most of the previous hardware Trojan detection techniques address the issue of 3PIP hardware security by instantiating two different IPs from different vendors onto the same system, which is extremely expensive due to the area overhead or not possible as two IPs might be unavailable from different vendors. Other methods rely on golden Trojan free models to detect hardware Trojans. This work targets the detection of hardware Trojans embedded in 3PBIPs when no golden models are available. This work leverages state-of-the-art HLS tools to build complete SoCs at the behavioral level allowing us to measure the idle time of each slave and using this to test for hardware Trojans.

The second part of this chapter has presented a hardware Trojan avoidance and detection method for runtime reconfigurable FPGAs, in particular, a coarse-grained runtime reconfigurable array (CGRRA) mapped as a slave on a memory mapped bus SoC. CGRRAs have unique capabilities that make them extremely powerful and efficient, but also extremely vulnerable to hardware Trojan. This work covers cases that STC, which holds the configuration data for the data-path, cannot be re-programmed at runtime, and the case that it can by accessing an external memory. In both the cases, our proposed method has shown to be very effective and requiring minimum design effort at a very low power overhead.

Chapter 6

Conclusions and Future work

6.1 Conclusions

Hardware security has become an extremely important topic in VLSI design. The globalization of the design and fabrication process has opened the window to the malicious manipulation of the design process. Thus, much work has been proposed to address this issue. Some include IP watermarking, fingerprinting, IC metering, IC camouflaging and split manufacturing to alleviate the threat of IC counterfeiting, reverse engineering, IP piracy and IC overbuilding. Although the behavioral level is quite new and still in its infancy, the security issues related to behavioral IPs have not been addressed properly. This thesis is a step towards making behavioral IC design more secure.

This thesis first discussed an open source synthesizable security SystemC benchmark suite (S3CBench) developed which includes different types of hardware Trojans. The benchmark is developed in such a way that it is difficult for the typical software profiler to point out the Trojan in code coverage report. The Trojans have different trigger and payload mechanisms and have different effects on the circuit. In particular, functionality changes, information leakage and performance degradation of the original circuit.

The BIP protection issue was addressed from two different angles. The first to protect the BIP provider and the second the BIP consumer. Obfuscation is one of the techniques used to protect the source code from illegal use without the permission of rightful IP owner. In this thesis, it was observed that obfuscating the source code can severely impact on the quality of results (QoRs) of BIPs for HLS. Thus, we have proposed a quick and efficient method to maximize the source code obfuscation while retaining the original design characteristics. In the next stage, we have studied different hardware Trojan detection techniques proposed for semiconductor IPs at various level of VLSI abstraction. Most of the previous techniques addressed the issue of 3PIP hardware security by instantiating two or more different IPs from different vendors onto the same system. Other methods rely on golden Trojan free models to detect the hardware Trojans. For BIPs, we have proposed a fully automatic method to detect the presence of hardware Trojans using formal verification methods. In particular, property checking at the behavioral level.

Finally, this thesis has addressed the issue of hardware Trojan detection at the behavioral system design level. With the advent of HLS, commercial tools now allow the creation of complete SoCs at the behavioral level. Thus, methods to ensure their security are needed. This thesis in particular leverage two main features of C-based design: (i) It allows the generation of fast cycle-accurate models, which this thesis uses to measure the exact slack of each BIP mapped as a loosely coupled Hardware Accelerator (HWAcc) slave and (ii) its ability to build the complete SoC using synthesizable Application Programming Interfaces (APIs). With this in mind, this thesis proposed a small hardware Trojan detection circuit called *trust filters*, which exploits the slack time of the slave to detect HW Trojan at runtime. Finally, CGRRAs have unique capabilities that make them extremely powerful and efficient, but also extremely vulnerable to hardware Trojan. We have also proposed hardware Trojan avoidance and detection method for these CGRRAs which are mapped as a slave on a

memory mapped bus SoC. Our proposed method has shown to be very effective and requiring minimum design effort at a very low power overhead.

6.2 Future Work

Although HLS is finally being widely adopted, the design of complete SoCs at the behavioral level is still in its infancy. This poses many security concerns that need to be addressed in the future. Short term future work will address other system-level security issues. This thesis has mainly addressed the issue of hardware Trojan detection for hardware accelerators mapped as loosely coupled slaves. Other areas that need to be investigated are the collusion of different modules within the SoC and covert timing. Long-term research, on the other hand, will address novel IC design materials and security concerns with their use. We are reaching the end of Moore's law and much work is being done to replace traditional CMOS logic with another type of devices like carbon nano-tubes and memristive devices. Especially for memristors that can hold and process information, it is therefore important to study how trustworthiness they are and any potential security issues that can come along with their use.

Bibliography

[1] K. Wakabayashi and B. C. Schafer, ""All-in-C" Behavioral Synthesis and Verification with CyberWorkBench," in *High-Level Synthesis*. Springer, 2008, pp. 113–127.

[2] A. Rawnsley. (2011) "Fishy chips: Spies want to hack-proof circuits," *Wired*. [Online]. Available: http://www.wired.com/dangerroom/2011/06/chips-oyspies-want-to-hack-proof-circuits/

[3] J.H. Follett. (2008) "CRN Cisco Channel at Center of FBI Raid on Counterfeit Gear,". [Online]. Available: www.crn.com/networking/207602683

[4] S. Adee, "The hunt of kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[5] J. Kumagai, "Chip Detectives," *IEEE Spectrum*, vol. 37, no. 11, pp. 43–48, Nov 2000.

[6] Australian Government DoD-DSTO. (2008) "Towards Countering the Rise of the Silicon Trojan". [Online]. Available: http://dspace.dsto.defence.gov.au/dspace/bitstream/1947/9736/1/DSTO-TR-2220%20PR.pdf

[7] Defense Advanced Research Projects Agency(DARPA). (2007) "TRUST in integrated circuits(tic) - proposer Information Pamphlet". [Online]. Available: http://www.darpa.mil/MTO/solicitations/baa07-24

[8] Semiconductor Industry Association (SIA). (2008) "Global billings report history(3-month moving average) 1976-March 2009". [Online]. Available: http://www.sia-online.org/galleries/Statistics/GSR1976-March09.xls

[9] Defense Science Board Task Force. (2005) "Study on High performance microchip supply". [Online]. Available: http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf

[10] SEMI. (2008) "Innovation is at risk as semiconductor equipment and materials industry loses up to $4 billion annually due to IP infringement". [Online]. Available: http://www.semi.org/en/Press/P043775

[11] *FABLESS:The Transformation of The Semiconductor Industry*. CreateSpace Independent Publishing Platform, 2014.

[12] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, Oct 2010.

[13] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[14] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is high level synthesis ready for business? A computational finance case study," in *2014 International Conference on Field-Programmable Technology (FPT)*, Dec 2014, pp. 12–19.

[15] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 10, no. 4, pp. 464–475, 1991.

[16] S. Govindarajan, "Scheduling algorithms for high-level synthesis," *Term paper ECE*, vol. 834, 1995.

[17] Z. Baruch, "Scheduling algorithms for high-level synthesis," *ACAM Scientific Journal*, vol. 5, no. 1-2, pp. 48–57, 1996.

[18] A. C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in *23rd ACM/IEEE Design Automation Conference*, June 1986, pp. 461–466.

[19] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical evaluation of some high-level synthesis scheduling heuristics," in *28th ACM/IEEE Design Automation Conference*, June 1991, pp. 686–689.

[20] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, Jun 1989.

[21] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 85–93, Jan 1991.

[22] G. Lakshminarayana, K. S. Khouri, and N. K. Jha, "Wavesched: a novel scheduling technique for control-flow intensive designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 5, pp. 505–523, May 1999.

[23] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 302–312, Feb 2004.

[24] D. Ku and G. D. Micheli, "Relative scheduling under timing constraints," in *27th ACM/IEEE Design Automation Conference*, Jun 1990, pp. 59–64.

[25] T. Ly, "Scheduling Using Behavioral Templates," in *32nd Design Automation Conference*, 1995, pp. 101–106.

[26] M. Münch, N. Wehn, and M. Glesner, "An Efficient ILP-based Scheduling Algorithm for Control-dominated VHDL Descriptions," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, no. 4, pp. 344–364, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/268424.268428

[27] S. Haynal, "Automata-based symbolic scheduling," Ph.D. dissertation, University of California, Santa Barbara, December 2000.

[28] A. Vijayakumar and F. Brewer, "Weighted control scheduling," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, Nov 2005, pp. 777–783.

[29] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, Apr 1991.

[30] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 9, pp. 1266–1278, Sep 1993.

[31] S. Bhattacharya, S. Dey, and F. Brglez, "Performance Analysis and Optimization of Schedules for Conditional and Loop-intensive Specifications," in *Proceedings of the 31st Annual Design Automation Conference*, ser. DAC '94. New York, NY, USA: ACM, 1994, pp. 491–496. [Online]. Available: http://doi.acm.org/10.1145/196244.196477

[32] B. M. Pangrle, "On the complexity of connectivity binding," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 11, pp. 1460–1465, Nov 1991.

[33] C.-J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, July 1986.

[34] M.C.McFarland, "Allocating registers, processors, and connections," Dept. Elect. Eng., Carnegie-Mellon Univ., Tech. Rep, Aug.1981.

[35] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *27th ACM/IEEE Design Automation Conference*, Jun 1990, pp. 499–504.

[36] K. Kucukcakar and A. C. Parker, "Data path tradeoffs using mabal," in *27th ACM/IEEE Design Automation Conference*, Jun 1990, pp. 511–516.

[37] G. Krishnamoorthy and J. A. Nestor, "Data path allocation using an extended binding model," in *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, Jun 1992, pp. 279–284.

[38] T. A. Ly and J. T. Mowchenko, "Applying simulated evolution to high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 3, pp. 389–409, Mar 1993.

[39] C. H. Gebotys and M. I. Elmasry, "Optimal synthesis of high-performance architectures," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 3, pp. 389–397, Mar 1992.

[40] NEC CyberWorkBench. (2015). [Online]. Available: www.cyberworkbench.com

[41] B. M. Pangrle, F. O. Brewer, D. A. Lobo, and A. Seawright, "Relevant issues in high-level connectivity synthesis," in *28th ACM/IEEE Design Automation Conference*, June 1991, pp. 607–610.

[42] J.-P. Weng and A. C. Parker, "3D scheduling: high-level synthesis with floorplanning," in *28th ACM/IEEE Design Automation Conference*, June 1991, pp. 668–673.

[43] A. C. H. Wu, V. Chaiyakul, and D. D. Gajski, "Layout-area models for high-level synthesis," in *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, Nov 1991, pp. 34–37.

[44] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug 2014.

[45] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, Jan 2010.

[46] F. Koushanfar and A. Mirhoseini, "A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 1, pp. 162–174, March 2011.

[47] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan horse detection using gate-level characterization," in *2009 46th ACM/IEEE Design Automation Conference*, July 2009, pp. 688–693.

[48] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 51–57.

[49] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan Detection using IC Fingerprinting," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 296–310.

[50] K. Hu, A. N. Nowroz, S. Reda, and F. Koushanfar, "High-sensitivity hardware Trojan detection using multimodal characterization," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 1271–1276.

[51] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building Stealthy and Malicious Hardware," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 64–77.

[52] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 159–172.

[53] A. Waksman and S. Sethumadhavan, "Silencing Hardware Backdoors," in *2011 IEEE Symposium on Security and Privacy*, May 2011, pp. 49–63.

[54] E. Love, Y. Jin, and Y. Makris, "Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, Feb 2012.

[55] Y. M. Alkabani and F. Koushanfar, "Active Hardware Metering for Intellectual Property Protection and Security," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 20:1–20:16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362923

[56] A. Cui, C. H. Chang, S. Tahar, and A. T. Abdel-Hamid, "A robust fsm watermarking scheme for ip protection of sequential circuit design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 678–690, May 2011.

[57] C. H. Chang and A. Cui, "Synthesis-for-testability watermarking for field authentication of vlsi intellectual property," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 7, pp. 1618–1630, July 2010.

[58] F. Koushanfar, I. Hong, and M. Potkonjak, "Behavioral Synthesis Techniques for Intellectual Property Protection," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 3, pp. 523–545, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1080334.1080338

[59] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Robust IP watermarking methodologies for physical design," in *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, June 1998, pp. 782–787.

[60] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "FPGA fingerprinting techniques for protecting intellectual property," in *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference (Cat. No.98CH36143)*, May 1998, pp. 299–302.

[61] G. Wolfe, J. L. Wong, and M. Potkonjak, "Watermarking graph partitioning solutions," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, June 2001, pp. 486–489.

[62] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey," *Integr. VLSI J.*, vol. 19, no. 1-2, pp. 1–81, Aug. 1995. [Online]. Available: http://dx.doi.org/10.1016/0167-9260(95)00008-4

[63] F. Koushanfar and Y. Alkabani, "Provably secure obfuscation of diverse watermarks for sequential circuits," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2010, pp. 42–47.

[64] A. E. Caldwell, H.-J. Choi, A. B. Kahng, S. Mantik, M. Potkonjak, G. Qu, and J. L. Wong, "Effective iterative techniques for fingerprinting design ip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 208–215, Feb 2004.

[65] J. B. Wendt, F. Koushanfar, and M. Potkonjak, "Techniques for foundry identification," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[66] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, Sept 2009.

[67] C. H. Chang and L. Zhang, "A blind dynamic fingerprinting technique for sequential circuit intellectual property protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 1, pp. 76–89, Jan 2014.

[68] M. Rostami, M. Majzoobi, F. Koushanfar, D. S. Wallach, and S. Devadas, "Robust and Reverse-Engineering Resilient PUF Authentication and Key-Exchange by Substring Matching," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 37–49, March 2014.

[69] U. Rührmair, S.Devadas, and F. Koushanfar, *Security Based on Physical Unclonability and Disorder, Introduction to Hardware Security and Trust*. Springer.

[70] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," in *2008 Design, Automation and Test in Europe*, March 2008, pp. 1069–1074.

[71] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Logic Barriers," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 66–75, Jan 2010.

[72] Y. Alkabani, F. Koushanfar, and M. Potkonjak, "Remote activation of ICs for piracy prevention and digital right management," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2007, pp. 674–677.

[73] R. S. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, Oct 2009.

[74] ——, "RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation," in *2010 23rd International Conference on VLSI Design*, Jan 2010, pp. 405–410.

[75] ——, "Hardware protection and authentication through netlist level obfuscation," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2008, pp. 674–677.

[76] F. Koushanfar and G. Qu, "Hardware metering," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, June 2001, pp. 490–493.

[77] F. Koushanfar, Farinaz, G. Qu, Potkonjak, and Miodrag, *Intellectual Property Metering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 81–95. [Online]. Available: http://dx.doi.org/10.1007/3-540-45496-9_7

[78] Intelligence Advanced Research Projects Activity (IARPA). (2011) Trusted integrated circuits program. [Online]. Available: https://www.fbo.gov/utils/view?id=b8be3d2c5d5babbdffc6975c370247a6

[79] Chipworks. (2012) Intel's 22-nm tri-gate transistors exposed. [Online]. Available: http://www.chipworks.com/blog/technologyblog/2012/04/23/intels-22-nmtri-gate-transistors-exposed/

[80] R. Torrance and D. James, "The state-of-the-art in semiconductor reverse engineering," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 333–338.

[81] Defense Advanced Research Projects Agency (DARPA). (2012) Integrity and reliability of integrated circuits (IRIS). [Online]. Available: http://www.darpa.mil/Our_Work/MTO/Programs/Integrity_and_ReliabilityofIntegratedCircuits-.

[82] R. Cocchi, L. Chow, J. Baukus, and B. Wang, "Method and apparatus for camouflaging a standard cell based integrated circuit with micro circuits and post processing," Aug. 13 2013, uS Patent 8,510,700. [Online]. Available: https://www.google.com/patents/US8510700

[83] P. Rohatgi, *Improved Techniques for Side-Channel Analysis*. Boston, MA: Springer US, 2009, pp. 381–406. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-71817-0_14

[84] P. Kocher, J. Jaffe, and B. Jun, *Differential Power Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. [Online]. Available: http://dx.doi.org/10.1007/3-540-48405-1_25

[85] P. Rohatgi, *Electromagnetic Attacks and Countermeasures*. Boston, MA: Springer US, 2009, pp. 407–430. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-71817-0_15

[86] D. Genkin, A. Shamir, and E. Tromer, *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 444–461. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44371-2_25

[87] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, *Simple Photonic Emission Analysis of AES*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 41–57. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_3

[88] P. Kocher, "Leak-resistant cryptographic indexed key update," Mar. 25 2003, uS Patent 6,539,092. [Online]. Available: https://www.google.com/patents/US6539092

[89] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[90] M. Joye, *Basics of Side-Channel Analysis*. Boston, MA: Springer US, 2009, pp. 365–380. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-71817-0_13

[91] S. Moore, R. Anderson, R. Mullins, G. Taylor, and J. J. A. Fournier, "Balanced Self-Checking Asynchronous Logic for Smart Card Applications," *Journal of Microprocessors and Microsystems*, vol. 27, pp. 421–430, 2003.

[92] K. Tiri, M. Akmal, and I. Verbauwhede, "A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards," in *Proceedings of the 28th European Solid-State Circuits Conference*, Sept 2002, pp. 403–406.

[93] F. Mace, F. x. St, I. Hassoune, J. d. Legat, and J. j. Quisquater, "A Dynamic Current Mode Logic to Counteract Power Analysis Attacks," in *In The Proceedings of DCIS 2004*, 2004, pp. 186–191.

[94] M. Stanojlović and P. Petković, "Strategies against side-channel-attack," in *Proceedings of the Small Systems Simulation Symposium*, 2010, pp. 86–89.

[95] B. Yang, K. Wu, and R. Karri, "Secure Scan: A Design-for-Test Architecture for Crypto Chips," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2287–2293, Oct 2006.

[96] J. Lee, M. Tehranipoor, C. Patel, and J. Plusquellic, "Securing Designs against Scan-Based Side-Channel Attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 325–336, Oct 2007.

[97] A. Cui, Y. Luo, and C. H. Chang, "Static and dynamic obfuscations of scan data against scan-based side-channel attacks," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 2, pp. 363–376, Feb 2017.

[98] F. Koushanfar, S. Fazzari, C. McCants, W. Bryson, P. Song, M. Sale, and M. Potkonjak, "Can eda combat the rise of electronic counterfeiting?" in *DAC Design Automation Conference 2012*, June 2012, pp. 133–138.

[99] K. Uwasawa, T. Yamamoto, and T. Mogami, "A new degradation mode of scaled p+ polysilicon gate pmosfets induced by bias temperature (bt) instability," in *Proceedings of International Electron Devices Meeting*, Dec 1995, pp. 871–874.

[100] P. Heremans, R. Bellens, G. Groeseneken, and H. E. Maes, "Consistent model for the hot-carrier degradation in n-channel and p-channel mosfets," *IEEE Transactions on Electron Devices*, vol. 35, no. 12, pp. 2194–2209, Dec 1988.

[101] X. Zhang and M. Tehranipoor, "Design of on-chip lightweight sensors for effective detection of recycled ics," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 1016–1029, May 2014.

[102] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M. S. Hsiao, J. Plusquellic, and M. Tehranipoor, "Protection Against Hardware Trojan Attacks: Towards a Comprehensive Solution," *IEEE Design Test*, vol. 30, no. 3, pp. 6–17, June 2013.

[103] J. Rajendran, E. Gavas, J. Jimenez, V. Padman, and R. Karri, "Towards a comprehensive and systematic classification of hardware trojans," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 1871–1874.

[104] M. A. Coussy, Philippe, *High-Level Synthesis - From Algorithm to Digital Circuit*. Springer, 2008.

[105] trust hub. (2010). [Online]. Available: https://www.trust-hub.org/resources/benchmarks

[106] S3CBench. (2016). [Online]. Available: https://sourceforge.net/projects/s3cbench/

[107] B. Carrion Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 53–56, 2014.

[108] M. Yoshimura, A. Ogita, and T. Hosokawa, "A smart Trojan circuit and smart attack method in AES encryption circuits," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2013, pp. 278–283.

[109] "Stunix c/c++ obfuscator," http://stunnix.com/, accessed: 2017-01-15.

[110] D. Forte, S. Bhunia, and M. Tehranipoor, *Hardware Protection through Obfuscation*. Springer, 2017.

[111] A. R. Desai, M. S. Hsiao, C. Wang, L. Nazhandali, and S. Hall, "Interlocking Obfuscation for Anti-tamper Hardware," in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW '13. New York, NY, USA: ACM, 2013, pp. 8:1–8:4. [Online]. Available: http://doi.acm.org/10.1145/2459976.2459985

[112] M. Brzozowski and V. N. Yarmolik, "Obfuscation as intellectual rights protection in vhdl language," in *Computer Information Systems and Industrial Management Applications, 2007. CISIM '07. 6th International Conference on*, June 2007, pp. 337–340.

[113] R. S. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, Oct 2009.

[114] F. Koushanfar, "Provably Secure Active IC Metering Techniques for Piracy Avoidance and Digital Rights Management," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 51–63, Feb 2012.

[115] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299. [Online]. Available: http://doi.acm.org/10.1145/948109.948149

[116] Z. Guo, M. Tehranipoor, D. Forte, and J. Di, "Investigation of obfuscation-based anti-reverse engineering for printed circuit boards," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[117] F. Koushanfar and Y. Alkabani, "Provably secure obfuscation of diverse watermarks for sequential circuits," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2010, pp. 42–47.

[118] A. Desai, M. Hsiao, C. Wang, and c. Nazhandali, "Interlocking obfuscation for anti-tamper hardware," in *In: Proceedings of the eighth annual cyber security and information intelligence research workshop. ACM*, 2013, pp. 1–4.

[119] M. Brzozowski and V. Yarmolik, "Interlocking obfuscation for anti-tamper hardware," in *In: Proceedings of the eighth annual cyber security and information intelligence research workshop. ACM*, 2013, pp. 1–4.

[120] M. Kainth, L. Krishnan, C. Narayana, S. G. Virupaksha, and R. Tessier, "Hardware-assisted code obfuscation for fpga soft microprocessors," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 127–132.

[121] C. Wang, "A security architecture for survivability mechanisms," Ph.D. dissertation, University of Virginia, 2000.

[122] T. Laszlo and A. Kiss, "Obfuscating C++ programs via control flow flattening," in *Sectio Computatorica*, Aug 2009.

[123] J. Holland, *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[124] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M. Hsiao, J. Plusquellic, and M. Tehranipoor, "Protection Against Hardware Trojan Attacks: Towards a Comprehensive Solution," *IEEE Design Test*, vol. 30, no. 3, pp. 6–17, June 2013.

[125] M. Banga and M. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *HOST*, June 2010, pp. 56–59.

[126] J. yang Jou and C. nan Jimmy Liu, "Coverage analysis techniques for HDL design validation," in *APCHDL*, 1999.

[127] J. Rajendran, H. Zhang, O. Sinanoglu, and R. Karri, "High-level synthesis for security and trust," in *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, July 2013, pp. 232–233.

[128] X. Cui, K. Ma, L. Shi, and K. Wu, "High-Level Synthesis for Run-Time Hardware Trojan Detection and Recovery," in *DAC*, 2014, pp. 157:1–157:6.

[129] A. Sengupta and S. Bhadauria, "Untrusted Third Party Digital IP Cores: Power-Delay Trade-off Driven Exploration of Hardware Trojan Secured Datapath During High Level Synthesis," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 167–172. [Online]. Available: http://doi.acm.org/10.1145/2742060.2742061

[130] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building Stealthy and Malicious Hardware," in *IEEE Symposium on Security and Privacy*, ser. SP '11, 2011, pp. 64–77.

[131] E. Love, Y. Jin, and Y. Makris, "Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition," *IEEE Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, Feb 2012.

[132] J. Rajendran, V. Vedula, and R. Karri, "Detecting Malicious Modifications of Data in Third-party Intellectual Property Cores," in *DAC*, 2015, pp. 112:1–112:6.

[133] B. Schafer, "Source Code Error Detection in High-Level Synthesis Functional Verification," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[134] M. Ben Hammouda, P. Coussy, and L. Lagadec, "A design approach to automatically synthesize ANSI-C assertions during High-Level Synthesis of hardware accelerators," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, June 2014, pp. 165–168.

[135] J. Curreri, G. Stitt, and A. George, "High-level synthesis techniques for in-circuit assertion-based verification," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[136] M. Ben Hammouda, P. Coussy, and L. Lagadec, "A Design Approach to Automatically Generate On-chip Monitors During High-level Synthesis of Hardware Accelerator," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, ser. GLSVLSI '14. New York, NY, USA: ACM, 2014, pp. 273–278. [Online]. Available: http://doi.acm.org/10.1145/2591513.2591521

[137] A. Ribon, B. Le Gal, C. Jego, and D. Dallet, "Assertion support in high-level synthesis design flow," in *Specification and Design Languages (FDL), 2011 Forum on*, Sept 2011, pp. 1–8.

[138] N. Bombieri, H.-Y. Liu, F. Fummi, and L. Carloni, "A method to abstract RTL IP blocks into C++ code and enable high-level synthesis," in *DAC*, 2013, pp. 1–9.

[139] Carbon Design Systems. (2015) Carbon Model Studio. [Online]. Available: www.carbondesignsystems.com/

[140] Aldec. (2015) DVM. [Online]. Available: www.aldec.com

[141] B. Carrion Schafer, A. Trambadia, and K. Wakabayashi, "Design of Complex Image Processing Systems in ESL," in *ASPDAC*, Taiwan, 2010, pp. 809–814.

[142] Y. Cao, C. H. Chang, and S. Chen, "A cluster-based distributed active current sensing circuit for hardware trojan detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 12, pp. 2220–2231, Dec 2014.

[143] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan Attacks: Threat Analysis and Countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, Aug 2014.

[144] E. Love, Y. Jin, and Y. Makris, "Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 1, pp. 25–40, Feb 2012.

[145] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Security and Privacy (SP), 2011 IEEE Symposium on*, May 2011, pp. 49–63.

[146] M. Beaumont, B. Hopkins, and T. Newby, "SAFER PATH: Security architecture using fragmented execution and replication for protection against trojaned hardware," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 1000–1005.

[147] A. Waksman, M. Suozzo, and S. Sethumadhavan, "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. New York, NY, USA: ACM, 2013, pp. 697–708. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516654

[148] J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for Hardware Trust," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–8.

[149] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "VeriTrust: Verification for Hardware Trust," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, July 2015.

[150] C. Liu, J. Rajendran, C. Yang, and R. Karri, "Shielding heterogeneous mpsocs from untrustworthy 3pips through security-driven task scheduling," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2013, pp. 101–106.

[151] S. Pawel, F. Marc, K. Philipp, M. Amir, and P. Christof, "Interdiction in Practice – Hardware Trojan Against a High-Security USB Flash Drive," 2015. [Online]. Available: https://eprint.iacr.org/2015/768.pdf

[152] D. M. Shila, V. Venugopalan, and C. D. Patterson, "FIDES: Enhancing trust in reconfigurable based hardware systems," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, Sept 2015, pp. 1–7.

[153] A. Al-Anwar, Y. Alkabani, M. W. El-Kharashi, and H. Bedour, "Hardware Trojan detection methodology for FPGA," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, Aug 2013, pp. 177–182.

[154] M. Motomura, "STP Engine, a C-based Programmable HW Core featuring Massively Parallel and Reconfigurable PE Array: Its Architecture, Tool, and System Implications," in *Proc. Cool Chips XII*, 2009, pp. 395–408.

[155] H. Amano, "A survey on dynamically reconfigurable processors," *IEICE Transactions on Communnication*, vol. E89, pp. 3179–3187, December 2006.

[156] Renesas. STP. [Online]. Available: http://www.renesas.com/products/soc/asic/programmable

[157] Xilinx. (2010). [Online]. Available: http://www.xilinx.com

[158] ——. (2010). [Online]. Available: http://www.xilinx.com/products/technology/power/xpe/license-virtex-4.html