HARDWARE SECURITY THROUGH DESIGN OBFUSCATION

by

RAJAT SUBHRA CHAKRABORTY

Submitted in partial fulfillment of the requirements

For the degree of Doctor of Philosophy

Thesis Adviser: Dr. Swarup Bhunia

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2010

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Rajat Subhra Chakraborty

candidate for the Doctor of Philosophy degree *.

(signed) Dr. Swarup Bhunia
(chair of the committee)

Dr. Christos Papachritou

Dr. Francis Merat

Dr. David McIntyre

Dr. Pankaj Rohatgi

(date) March 4th, 2010

*We also certify that written approval has been obtained for any proprietary material contained therein.

To my family and friends.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ACKNOWLEDGMENTS

First of all, I would like to thank my friends at Case Western Reserve University, especially those in the *Nanoscape* research lab for their help and encouragement. Their feedback and intellectual input always helped me in finding new and challenging problems to work on, and many a times they inspired effective solutions for these problems. This Ph.D. is as much their's as it is mine.

My special thanks goes to my thesis advisor Prof. Swarup Bhunia, who has been a "friend, philosopher and guide" over the last few years. His guidance and help has had an immense positive effect on my work. He has taught me never to compromise the quality of my research, and has encouraged me to go the extra distance to ensure that the work meets our mutual high standards. Sincere thanks to Prof. Christos Papachristou, Dr. Pankaj Rohatgi, Prof. David McIntyre and Prof. Francis Merat for agreeing to be in my thesis committee.

I thank Prof. Swarup Bhunia, Prof. Christos Papachristou, Prof. Steven Garverick, Prof. Xinmiao Zhang, Prof. Daniel Saab, Prof. Massood Tabib-Azar, Prof. Mark Buchner, Prof. Vira Chankong, Prof. Catalin Turc, Prof. Wojbor Woyczynski and Prof. David Singer for teaching me courses at Case which I enjoyed immensely. The academic aspect has been an unforgettable part of the entire graduate experience for me. I thank Dr. Francis Wolff for many interesting and intellectually stimulating conversations, and both him and Dr. Lawrence Leinweber for their help in resolving many technical issues.

Finally, words cannot describe my gratitude to my family members - my parents Rita and Pratyush Chakraborty and my wife Munmun for their sacrifice in allowing me to stay thousands of miles away from home while working on my Ph.D. Special thanks to my sister Sharmistha and my brother-in-law Pallab for providing me with a home away from home in the US. I love you all.

Hardware Security through Design Obfuscation

Abstract
by
RAJAT SUBHRA CHAKRABORTY

Security of integrated circuits (ICs) has emerged as a major concern at different stages of IC life-cycle, spanning design, test, fabrication and deployment. Modern ICs are becoming increasingly vulnerable to various forms of security threats, such as: 1) illegal use of hardware intellectual property (IP) or "IP Piracy"; 2) illegal manufacturing of IC or "IC Piracy"; 3) insertion of malicious circuits, referred as "Hardware Trojan", in a design to cause in-field circuit malfunction, and 4) leakage of secret information from an IC. These security threats are accentuated by current IC design practices, such as the widespread use of hardware IP modules to design complex system-on-chips (SoCs). In addition, the economics of electronic manufacturing dictates widespread outsourcing of integrated circuit fabrication to off-shore facilities, which increases the vulnerability to these attacks. In this research, we explore novel hardware design approaches that incorporate a key-based design obfuscation scheme to effectively protect a design against various security threats, while incurring low hardware and computational overheads. *Obfuscation* is a technique that makes comprehending and reverse-engineering a design difficult. To the best of our knowledge, this is the first effort to develop a systematic and provably robust hardware obfuscation approach that enables hardware protection at different stages of the IC life-cycle. Effectiveness of these approaches for protection against IP reverse-engineering and piracy, hardware Trojan and scan-based information leakage is evaluated with benchmark circuits and open-source IP cores. The obfuscation approaches are developed for both *firm* (gate-level) and *soft* (register transfer level) IPs. The principles of the obfuscation approach have been extended to protection of embedded software against piracy and malicious modification. An enhanced secure IC design flow with associated computer-aided design (CAD) tools is also developed.

1. INTRODUCTION

1.1 Background

The spectacular advancement of Electronics in the last half a century has revolutionized the world. The all-pervasive nature of Electronics can be felt in every aspect of human life in this "silicon age". Starting from 1960 when commercial semiconductor manufacturing became a viable business, the industry has grown to have a worldwide revenue of $249 billion in 2008, with a prodigious average annual growth rate of 16.1% between 1975 and 2000 [1, 2]. The key to this rapid advancement in electronics is the aggressive reduction in device dimensions in integrated circuits (ICs) over generations, along with scaling of supply voltage (V_{DD}) and transistor "threshold voltage" (V_{th}), a phenomenon known as "technology scaling". Device integration density (the number of devices per unit area in an IC) has roughly doubled every two years over the last five decades, a trend first predicted in [3], and now famously termed as "Moore's Law". This has enabled a corresponding exponential increase in computing capabilities of ICs.

However, with increasing computing power and integration density, several issues in design, performance and manufacturing of these ICs have cropped up. Increasing power consumption (both due to higher integration density and increasing "leakage" power consumption in the "off-state"), increased cost of testing and verification, and photolithographic complexities in manufacturing nanometer devices (with feature sizes much smaller than the smallest wave-length of light that be used for patterning) are the some of the major issues with IC design and manufacturing in the nanometer regime. To make such design and manufacturing feasible, an IC design house is commonly aided by the following external agencies:

- Electronic design automation (EDA) companies supply the sophisticated software tools that facilitates design, verification and testing of modern ICs.

- Intellectual property (IP) and *standard cell library* vendors, who supply pre-verified, high performance, functional hardware IPs or library cells to the IC design facilities. These help to reduce the design time drastically, improve reliability and yield, and enable meeting hard time-to-market target.

- Semiconductor manufacturing companies, which provide the fabrication facilities ("fabs") where the design is actually manufactured and sometimes tested, before being sent back to the design house. The cost of maintaining a cutting-edge fab runs into billions of dollars every year, encouraging most IC vendors (over 250 strong [4]) to follow a "fabless" business model where the design database is outsourced to the fabs for manufacturing [5].

From the above description of prevalent industry practices, it is evident that the control exercised by the IC vendors over the design and manufacturing of their own products is decreasing. The increasing complexity and cost of modern nanometer-scale ICs are the main drivers behind this trend. Reduced control on the IC life-cycle accentuates various security issues associated with ICs. Hence, security of hardware IPs and ICs has emerged as a major challenge in nanometer IC design and test. In the following section, we describe different security threats associated with IC design and test.

1.2 Security Threats

The active participation of various external agents in the design and manufacturing flow has made the entire process highly vulnerable to various security threats. Fig. 1.1 [6] shows the level of "trust" that can be assigned to the various stages of a typical modern IC design flow. The main mechanisms behind these threats can be broadly classified under the following heads:

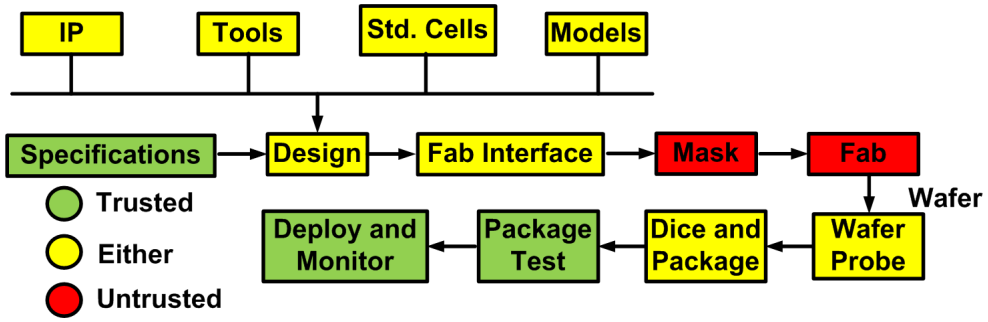


Fig. 1.1. Vulnerability of different stages in a modern IC life cycle with respect to different forms of security attacks [6].

1. **IP piracy**: This is a scenario where the IP is illegally used and/or copied without paying the lawful royalty to the IP vendor [7].

2. **IC piracy**: In this scenario, the manufacturing fab illegally copies and reverse-engineers the design database of an IC sent for fabrication to manufacture illegal copies ("clones") of the IC [8].

3. ***Hardware Trojan* insertion**: The design can be modified in the design house during the design phase or in the fab by malicious insertion, deletion or modification of circuits, referred to as *Hardware Trojans*, which cause the IC to deviate from its intended functional behavior during deployment [6,9]. Typically, these Trojan circuits are *stealthy* by design, which makes it extremely challenging to detect them by traditional post-manufacturing testing [10].

4. **Secret information leakage:** Although the "deploy and monitor" step of the design flow has been shown to be completely trustable in Fig. 1.1, in reality, it has been shown that secret information can be extracted by an adversary from secure ICs with cryptographic functionality in this stage [11]. Such threats increase with increasing *controllability* and *observability* of the internal nodes of the circuit as a result of widespread adoption of "Design for Testability" (DfT) techniques in modern ICs.

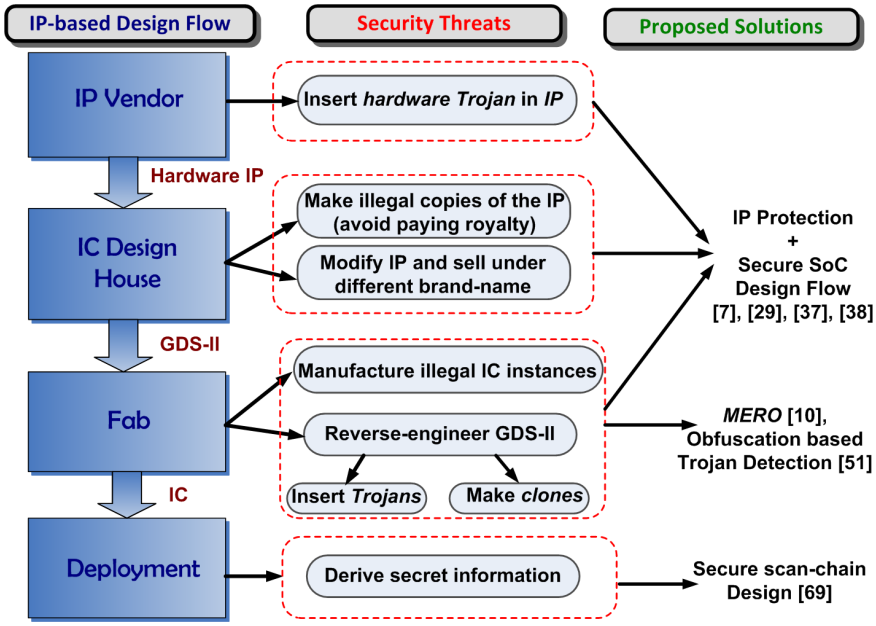


Fig. 1.2. Security threats in a modern IC life-cycle, and corresponding solutions as proposed in this research.

It is worth noting that the "design" stage itself is designated as one of the partially insecure stages of the entire flow. This takes into consideration the possible presence of untrusted personnel in the design house with access to the design, who might sabotage the design to serve other interests. The mitigation of these security threats, with minimal overhead on design, fabrication and testing of ICs, is the main motivation of this research. Next, the proposed research is outlined.

1.3 Outline of Proposed Research

The main goal of the proposed research is:

To establish "Design for Security" (DfS) *as a new paradigm of system design, through the development of design techniques with low design and computational overhead, that can effectively resist or mitigate the security threats at different stages of the IC life-cycle.* A secondary goal is the development of EDA tools to implement and

automate the design methodology, and to integrate it with the traditional design flow. Fig. 1.2 shows the security threats at different stages of the IC design life-cycle (as described in the previous section), and the corresponding solutions proposed in this research. Notice that the goal of this research is to have a comprehensive solution spanning all stages of the design and manufacturing flow. The DfS techniques proposed by this research are "key-based", where the correct functionality of the system is enabled only after the successful application of the "key". In this respect, it is similar to cryptographic principles, where the encrypted information is only available in an intelligible form after it has been decrypted using a key. Similar to modern cryptographic practices, this work is entirely in consensus with the widely accepted "Kerckhoffs' Principle" [12], which states that a cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

The unifying principle behind all the proposed techniques is *Design Obfuscation*. In simple terms, *Obfuscation* is a technique that makes understanding or reverse-engineering of a design difficult. The adversary usually knows how to try to reverse-engineer the design, but the complexity of the problem of reverse-engineering is exponential or near exponential with respect to one or more design parameters. Traditionally, design obfuscation creates a design which is *functionally identical* to the original design, while being difficult to reverse-engineer [13]. It has been mathematically proven that a successful obfuscation in the traditional sense does not exist, at least for specific programs computing certain classes of functions [14]. However, the proposed approach differs from the traditional approach in a sense that the functionality of the design is actually altered in a different mode of operation prevent piracy and make it difficult to reverse-engineer. Hence, the above mathematical result does not apply to this work. Throughout the work, the robustness of the proposed obfuscation scheme against several attack scenarios has been theoretically analyzed and supported by appropriate simulation results.

In this thesis, *software obfuscation* for embedded applications inspired by the techniques developed for hardware obfuscation has also been explored. The reason

for focusing on embedded software is that traditional software security techniques (both hardware and software), although extremely mature, often require hardware and computational resources which are difficult to provide in resource-constrained embedded systems [15]. Hence, obfuscation based software protection techniques requiring modest computational overhead can be an attractive choice for embedded systems.

1.4 Major Contribution of the Proposed Research

The novelty of the proposed research can be summarized as follows:

- To the best of our knowledge, this is the first research that studies the application of a systematic *design obfuscation* method to achieve hardware security. In particular, it applies key-based obfuscation to (a) develop a piracy-proof SoC design and manufacturing flow; (b) provide protection against hardware Trojans, and, (c) prevent leakage of secret information from a secure IC.

- It targets the development of a secure design methodology that benefits the interests of all parties concerned with IC design and manufacturing, unlike prior research, which addresses the rights of a single party (e.g. the IP vendor).

- The efficacy of the proposed design techniques is supported by accompanying mathematical modeling and theoretical analysis. Hence, the techniques can achieve provably robust performance unlike many previous works in this field based on heuristics and somewhat ad-hoc design choices.

- This research successfully implements the proposed design techniques, with the development of necessary EDA software tools. The techniques are also seamlessly integrated with a conventional IC design flow. The techniques are then applied to benchmark circuits and open-source IPs to validate the theoretical predictions, and to estimate the hardware and computational overheads.

- Finally, embedded software protection utilizing the same principle of key-based functionality obfuscation has been explored and found to be capable of providing high levels of security at moderate code size and performance overhead.

1.5 Organization of the Thesis

In Chapter 2, a piracy-proof system-on-chip (SoC) design methodology based on obfuscation of hardware IPs at as gate-level netlists is presented. The technique is extended to *register transfer level* (RTL) design descriptions in Chapter 3. Chapter 4 develops a design methodology based on obfuscation to provide protection against hardware Trojans. In Chapter 5 we present a statistical test generation methodology to detect hardware Trojans. In Chapter 6, a key-based secure scan design technique to prevent leakage of secret information in an IC is presented. In Chapter 7, embedded software protection by a key-based control-flow obfuscation technique has been explored by extending the key-based hardware obfuscation technique. Finally, in Chapter 8, the work is summarized and future research directions are explored.

2. OBFUSCATION-BASED SECURE SOC DESIGN

2.1 Introduction

Reuse-based SoC design using hardware *Intellectual Property* (IP) cores has become a pervasive practice in the industry. The IP cores usually come in the form of synthesizable Register Transfer Level (RTL) descriptions ("Soft IP"), gate-level designs directly implementable in hardware ("Firm IP") or GDS-II design database ("Hard IP"). The approach of designing complex systems by integrating tested, verified and reusable modules reduces the design time and cost dramatically [16].

Unfortunately, recent trends in IP-piracy and reverse-engineering efforts to produce counterfeit ICs have raised serious concerns in the IC design community [16,17]. IP piracy can take several forms, as illustrated by the following scenarios:

- A chip design house buys an IP core from an IP vendor, and makes an illegal copy or "clone" of the IP. The IC design house then sells it to another chip design house (after minor modifications) claiming the IP to be its own [18].
- An untrusted fabrication house makes an illegal copy of the GDS-II database supplied by a chip design house, and then illegally sells them as hard IP.
- An untrusted foundry manufactures and sells counterfeit copies of the IC under a different brand name [19].
- An adversary performs post-silicon reverse-engineering on an IC to manufacture its illegal clone [17].

These scenarios demonstrate that all parties involved in the IC design flow are vulnerable to different forms of IP infringement which can result in loss of revenue and market share. Hence, there is critical need of a piracy-proof design flow that

equally benefits the IP vendor, the chip designer as well as the system designer. A desirable characteristic of such a secure design flow is that it should be *transparent* to the end-user, i.e. it should not impose any constraint on the end user with regard to its usage, cost or performance.

Existing approaches for hardware protection do not ensure comprehensive protection to IP vendors, chip designers, system designers and end-users [16–24]. They are targeted towards preserving the rights of only *a single* party (mostly IP vendors). In order to benefit all of them simultaneously, an *anti-piracy design flow* is required in which IP vendors, designers and manufacturers take an active part in securing their own rights and ensure that the customer gets an authentic and trustworthy product. Furthermore, existing works in this field primarily focus on protecting the hardware from illegal copy. They do not address the issue of achieving resistance against reverse-engineering during fabrication, test and deployment.

As mentioned in Chapter 1, *obfuscation* is a technique that transforms an application or a design into one that is functionally equivalent to the original but is significantly more difficult to reverse-engineer [25]. In this chapter, we present ***HARPOON***, a methodology for **HAR**dware **P**rotection through **O**bfuscation **O**f **N**etlist. Such an approach can be used to protect gate-level IPs or "firm IPs". We show that the proposed obfuscation approach can be easily combined with a low-overhead authentication approach. We also present scalability of the approach to complex SoC designs consisting of several IP blocks. In the next Chapter, we provide an extension of the idea to RTL design descriptions. The proposed obfuscation based protection scheme provides anti-piracy and tamper-proof qualities to the hardware at every stage of the hardware design and manufacturing. The following are the major contributions of this work:

- A low-overhead and low-complexity IP design methodology for gate-level IPs aimed towards preventing IP piracy is presented. The proposed design methodology provides simultaneous *obfuscation* and *authentication* of a netlist, thus protecting valuable hardware IPs. The key step to accomplish the obfuscation

goal is to systematically modify the state transition function and internal circuit structure in such a way that the circuit operates in normal mode only upon application of a pre-defined enabling sequence of patterns at primary inputs, referred to as "key" for the circuit.

- A metric is provided to quantify the level of obfuscation and present theoretical analysis to evaluate the effect of design modifications on resulting obfuscation. Such an analysis is used to explore techniques for achieving maximum obfuscation at minimal design overhead.
- we present a design flow for SoCs based on the proposed hardware IP obfuscation technique. This is the first gate-level obfuscation-based design flow capable of providing security at multiple stages of the IC life-cycle.

The rest of the chapter is organized as follows. In Section 2.2, previous work on this topic and the motivation behind the work is discussed. In Section 2.3, a theoretical analysis to evaluate an obfuscation scheme and an obfuscation metric, are presented. In Section 2.4, the proposed obfuscation scheme is described and a design methodology to integrate it in the SoC design and manufacturing flow is developed. In Section 2.5, simulation results for a set of benchmark circuits and the AES encryption/decryption IP core are presented. Finally, conclusions are summarised in Section 2.6.

2.2 Related Work

Hardware IP protection has been investigated earlier in diverse contexts, addressing *licensed* as well as the pre-license *evaluation version* of an IP. Previous work on IP protection can be broadly classified into two main categories: (1) *Obfuscation* based protection, and (2) *Authentication* based protection.

1) Obfuscation based IP Protection: In obfuscation based IP protection, the IP vendor usually affects the human readability of the HDL code [20], or relies on cryptographic techniques to encrypt the source code [21,24]. In [20], the code is reformatted

by changing the internal net names and removing the comments, so that the circuit description is no longer intelligible to the human reader. However, it does not modify the functionality of the IP core, and thus cannot prevent it from being stolen by a hacker and used as a "blackbox" module. In a more common practice widely adopted in the industry [21,24], the HDL source code is encrypted and the IP vendor provides the key to decrypt the source code only to its valid customers. A similar approach has been proposed in [26], where an infrastructure for IP evaluation and delivery for FPGA applications has been proposed based on Java applets. However, this technique may enforce the use of a particular design platform [22, 23], a situation that might be unacceptable to many SoC designers who seek the flexibility of multiple tools from diverse vendors in the design flow. Moreover, none of these techniques prevent possible reverse-engineering effort at later stages of the design and manufacturing flow. Ultimately, the value of such techniques of *making the circuit description difficult to understand* is somewhat questionable if one considers results such as [14] which prove the theoretical impossibility of "black-box obfuscation".

2) Authentication based IP Protection: To protect the rights of the IP vendor through authentication, the approaches proposed are directed towards embedding a *Digital Watermark* in the design [16, 18] which helps to authenticate the design at a later stage. Typically this is inserted by design modifications which result in one or multiple input-output response pair(s) which do not arise during the normal functioning of the IP. Such digital signatures are known only to the IP vendor. Since this digital watermark (or signature) cannot be removed from the IP, it helps to prove an illegal use of such a component in litigation. However, the effectiveness of authentication-based IP protection schemes is limited by the fact that these techniques are *passive* and hence they cannot prevent the stolen IP from being used.

The approaches directed towards preventing the rights of the IC designer, on the other hand, ensure that the design house has a knowledge of *every* IC instance manufactured and sold in the market. It is interesting to note that the need of obfuscation to protect ICs against possible reverse-engineering and copy was investigated long

back in the mid-1970s, coinciding with the commercial release of the first generation microprocessors [27]. However, the scalability of the technique to larger designs by adopting a systematic approach of hardware obfuscation was not explored. Usually, this is implemented by including a *locking* mechanism in the IC [8, 28]. The fabrication facility would require a unique bit sequence provided by the design house to "unlock" the IC. This unique bit sequence is determined by a *physically unclonable function* (PUF) block that is added to the design by the design house. However, such approaches cannot prevent the possibility of reverse-engineering a design to expose its functionality as well as the security scheme. Moreover, they do not address protecting the right of an IP vendor.

2.3 Analysis of Netlist Obfuscation

In order to achieve comprehensive protection of hardware IPs, the proposed approach focuses on obfuscating the functionality and structure of an IP core by modifying the gate-level netlist, such that it both obfuscates the design and embeds authentication features in it. The IC is protected from unauthorized manufacturing by the fact that the system designer depends on input from the chip designer to use the IC. Consequently, the manufacturing house cannot simply manufacture and sell unauthorized copies of an IC without the knowledge of the design house. In addition, by adopting a PUF-based activation scheme, the security can be increased further since it ensures that the activation pattern is specific to each IC instance. In case the IP along with the information required to de-obfuscate it is released in the public domain by a malicious IC design house, the embedded authentication feature acts as a second line of defense and helps in proving illegal usage of an IP during litigation. Finally, the obfuscation remains transparent to the end user who has the assurance of using a product that has gone through an anti-piracy secure design flow.

The obfuscation is carried out in a manner that, for a target design overhead, it offers maximum resistance to any reverse-engineering effort, whether structural or

functional. The possibility of a sophisticated adversary having access to superior computational capabilities and powerful CAD tools cannot be ruled out. Hence, it is important to mathematically analyze the level of obfuscation and ensure that it is practically infeasible for an adversary to reverse-engineer an obfuscated design.

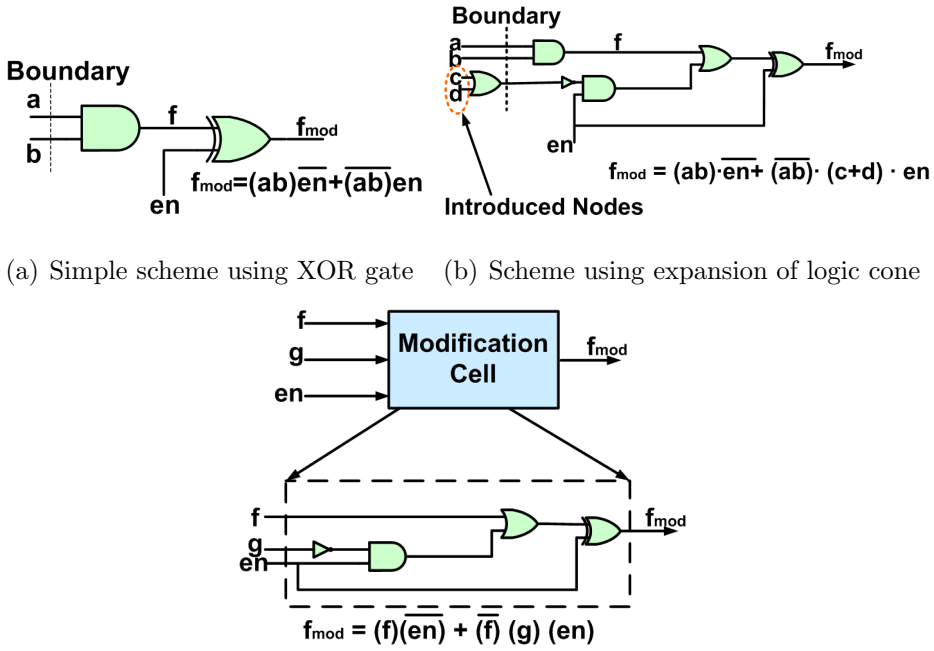
2.3.1 Hardware IP Piracy: the Hacker's Perspective

A hacker trying to determine the functionality of an obfuscated gate-level IP core can take resort to either (1) simulation-based reverse-engineering to determine functionality of the design, or (2) structural analysis of the netlist to identify and isolate the original design from the obfuscated design. The proposed obfuscation approach targets to achieve simulation mismatch for the maximum possible input vectors, as well as structural mismatch for maximum possible circuit nodes. To achieve structural mismatch between the reference and the obfuscated design, both the state transition function as well as the internal logic structure are modified.

Modification Scheme Employing Input logic-cone Expansion

Consider the simple example shown in Fig. 2.1(a). It shows a modified 2-input AND gate. If $en = 0$, it works as an ordinary AND gate; however, if $en = 1$, the original functionality of the AND gate is obfuscated because the output is inverted. Simulation of the simple circuit of Fig. 2.1(a) against an ordinary 2-input AND gate will report 4 possible input vectors with $en = 1$ as failing patterns. To increase the number of failing patterns for this circuit, the input logic cone must be expanded, while ensuring that it continues to function properly when $en = 0$. Fig. 2.1(b) shows an alternative scheme, where the input logic cone has been expanded to include the node c and d. A complete enumeration of the truth-table of the modified circuit will show failures for 13 input patterns (out of the possible 32).

The modification scheme of Fig. 2.1(b) can be generalized to a form shown in Fig. 2.1(c). Here, f is the Boolean function corresponding to an internal node and



(a) Simple scheme using XOR gate (b) Scheme using expansion of logic cone

(c) Modification cell

Fig. 2.1. Schemes for boolean function modification and modification cell.

g is any arbitrary Boolean logic function. It is worthwhile to note that the simple modification scheme of Fig. 2.1(a) is a special case with $g = 1$. As shown, the modified logic function is of the form:

$$f_{mod} = f \cdot \overline{en} + \overline{f} \cdot g \cdot en \tag{2.1}$$

Let us call the function g as the *Modification Kernel Function* (**MKF**). It is clear that for $en = 1$, if $g = 1$ for a given set of primary inputs and state element output state, $f_{mod} = \overline{f}$ and the test pattern is a failing test pattern. To increase the amount of dissimilarity between the original and modified designs, g should evaluate to logic-1 as often as possible. At first glance, the trivial choice seems to be $g = 1$. However, in that case the input logic cone is not expanded and thus the number of failing vectors reported by a formal verification approach is limited. For any given set of inputs, this is achieved by a logic function which is the logical-OR of the input variables.

2.3.2 Obfuscation Metric

A metric to quantify the level of obfuscation is now derived. Let, f be a function of the set P_1 of primary inputs and state-element outputs and g be a function of a set P_2 of primary inputs and state element (SE) outputs. Let $P_1 \bigcap P_2 = P$, $|P_1| = p_1$, $|P_2| = p_2$, $|P| = p$, $P_1 \bigcup P_2 = \Gamma$ and $|\Gamma| = \gamma = p_1 + p_2 - p$. Further, let g be a Boolean OR function with p_2 inputs. Then, for $(2^{p_2} - 1)$ of its input combinations, g is at logic-1. Consider $en = 1$. Then, for all these $(2^{p_2} - 1)$ input combinations of P_2 , $f_{mod} = \overline{f}$, causing a failing vector. Corresponding to each of these $(2^{p_2} - 1)$ combinations of P_2, there are $(p_1 - p)$ other independent primary inputs to f. Hence, the total number of failing vectors when $g = 1$ is:

$$N_{g1} = 2^{(p_1 - p)} \cdot (2^{p_2} - 1) \tag{2.2}$$

For the other "all zero" input combination of P_2, $f = 0$. Let the number of possible cases where $f = 1$ at $g = 0$ be N_{g0}. Then, the total number of failing input patterns:

$$N_{failing} = N_{g1} + N_{g0} = 2^{(p_1 - p)} \cdot (2^{p_2} - 1) + N_{g0} \tag{2.3}$$

In the special case when $P_1 \bigcap P_2 = P = \phi$, N_{g0} is given simply by the number of possible logic-1 entries in the truth-table of f.

The total input space of the modified function has a size 2^{γ}. The *obfuscation metric* (M) is defined as:

$$M = \frac{N_{failing}}{2^{\gamma+1}} = \frac{2^{(p_1 - p)} \cdot (2^{p_2} - 1) + N_{g0}}{2^{p_1 + p_2 - p + 1}} \tag{2.4}$$

The "+1" factor in the denominator is due to the *en* signal. Note that $0 < M \leq \frac{1}{2}$. As an example, for $f = ab + cd$, with $g = a + b$, $M = \frac{13}{32}$. As a special case, consider $\mathbf{p} = \mathbf{p_1} = \mathbf{p_2}$, i.e. the signal g is derived from the same set of primary inputs of f. Then,

$$M = \frac{1}{2}\Big(1 + \frac{N_{g0} - 1}{2^{p_1}}\Big) = \frac{1}{2}\Big(1 + \frac{N_{g0} - 1}{2^{p_2}}\Big) \tag{2.5}$$

In this case, N_{g0} is either 0 or 1; hence $M = \frac{1}{2}$ if $N_{g0} = 1$ and $M = \frac{1}{2}\big(1 - 2^{-p_1}\big)$ if $N_{g0} = 0$. Note that M attains the maximum (ideal) value of 0.5 in this case when

$N_{g0} = 1$. The theoretical maximum, however, is not a very desirable option because it keeps the input space limited to 2^{p_1} possible vectors. Again, if $\mathbf{p} = \mathbf{0}$, i.e., g is generated by a completely different set of primary inputs which were not included in f, then:

$$M = \frac{1}{2}\Big(1 + \frac{N_{g0}2^{-p_1} - 1}{2^{p_2}}\Big) \tag{2.6}$$

Larger values of N_{g0} and smaller values of p_2 for a given p_1 help to increase M. Note that unlike the first case, N_{g0} is guaranteed to be non-zero. This property effectively increases the value of M in case (b) than that in case (a) for most functions. However, in the second case, $M < \frac{1}{2}$, i.e. M cannot attain the theoretical maximum value.

Selection of the Modification Kernel Function (g): Although the above analysis points to the selection of primary inputs or state element outputs to design the **MKF** g satisfying the condition $p = 0$, in practice, this could incur a lot of hardware overhead to generate the OR-functions corresponding to each modified node. An alternative approach is to select an internal logic node of the netlist to provide the Boolean function g. It should have the following characteristics:

1. The modifying node should have a very large fan-in cone, which in turn would substantially expand the logic cone of the modified node.

2. It should not be in the fan-out cone of the modified node.

3. It should not have any node in its fan-in cone which is in the fan-out cone of the modified node.

Conditions (2) and (3) are essential to prevent any *combinational loop* in the modified netlist. Such a choice of g does not, however, guarantee it to be an OR-function and is thus sub-optimal. The effectiveness of this MKF is explored in Section 2.5.

It should be noted that modifying the Boolean function in the form $f_{mod} = f \cdot \overline{en} + h \cdot g \cdot en$ where h is any arbitrary Boolean function not necessarily $\overline{f}$ also results in the expansion in the input logic cone of the node representing f_{mod}; however, the number

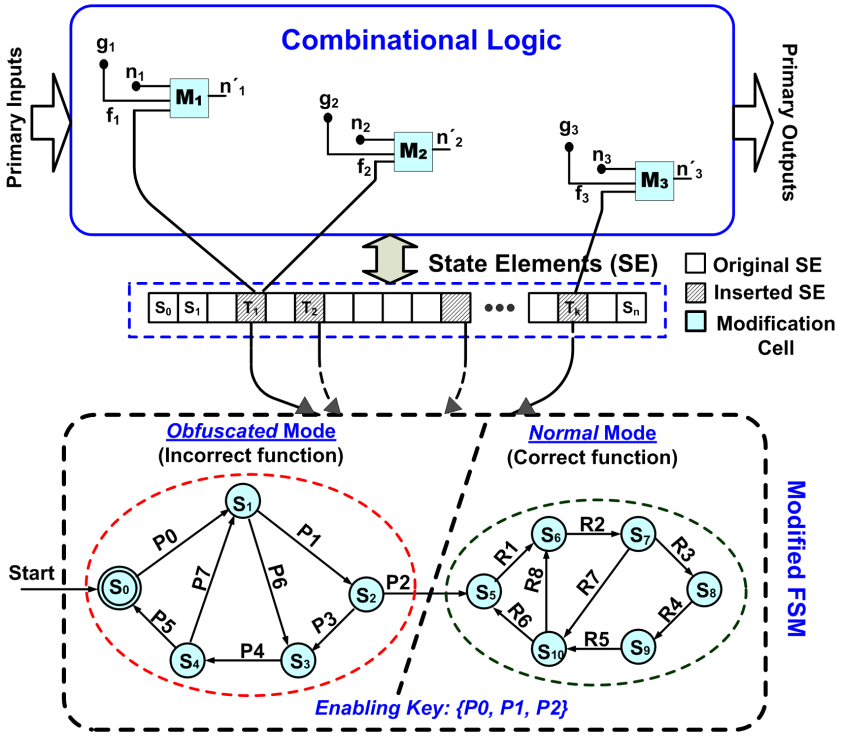


Fig. 2.2. The proposed functional and structural obfuscation scheme by modification of the state transition function and internal node structure.

of failing vectors is expected to be lesser than the modification scheme following eqn. 2.1. This is because if $en = 1$, $g = 1$, $f_{mod} = h$, and $f_{mod} \neq f$ only if $h \neq f$. On average, the number of failing vectors would probabilistically decrease by half for an arbitrary choice of $h \neq \overline{f}$.

2.4 System-level Obfuscation Methodology

In this section, the secure SoC design methodology for hardware protection (based on the analysis of the previous section is presented.

2.4.1 State Transition Function Modification

The first step of the obfuscation procedure is the modification of the state transition function of a sequential circuit by inserting a small Finite State Machine (FSM). The inserted FSM has all or a subset of the primary inputs of the circuit as its inputs (including the clock and reset signals) and has multiple outputs. At the start of operations, the FSM is reset to its initial state, forcing the circuit to be in the *obfuscated* mode. Depending on the applied input sequence, the FSM then goes through a state transition sequence and only on receiving N specific input patterns in sequence, goes to a state which lets the circuit operate in its *normal* mode. The initial state and the states reached by the FSM before a successful initialization constitute the "pre-initialization state space" of the FSM, while those reached after the circuit has entered its normal mode of operation constitute the "post-initialization state space". Fig. 2.2 shows the state diagram of such a FSM, with $P0 \rightarrow P1 \rightarrow P2$ being the correct initialization sequence. The input sequence $P0$ through $P2$ is decided by the IP designer.

The FSM controls the mode of circuit operation. It also modifies selected nodes in the design using its outputs and the *modification cell* (e.g. M_1 through M_3). This scheme is shown in Fig. 2.2 for a gate level design that incorporates modifications of three nodes n_1 through n_3. The **MKF** can either be a high fan-in internal node (avoiding combinational loops) in the unmodified design, or the OR-function of several selected primary inputs. The other input (corresponding to the *en* port of the modification cell) is a Boolean function of the inserted FSM state bits with the constraint that it is at logic-0 in the *normal* mode. This modification ensures that when the FSM output is at logic-0, the logic values at the modified nodes are the same as the original ones. On the other hand, in the *obfuscated* mode, for any FSM output that is at logic-1, the logic values at the modified nodes are inverted if $g = 1$ and logic-0 if $g = 0$. Provided the modified nodes are selected judiciously, modifications at even a small number of nodes can greatly affect the behavior of the modified system.

This happens even if the *en* signal is not always at logic-0. In the implementation, the number of outputs of the inserted FSM was supplied by the user as an input parameter. These outputs are generated as random Boolean functions of the state element bits at design time with the added constraint that in the *normal* mode, they are at logic-0. The randomness of the Boolean functions adds to the security of the scheme. Such a node modification scheme can provide higher resistance to structural reverse-engineering efforts than the scheme in [29].

2.4.2 Embedding Authentication Features

The proposed obfuscation scheme allows us to easily embed authentication signature into a gate-level design with negligible design overhead. Such a embedded signature acts as a *digital watermark* and hence helps to prevent attack from trusted parties in the design flow with knowledge of initialization sequence. Corresponding to each state in the *pre-initialization state space*, a particular pattern is made to appear at a sub-set of the primary outputs when a pre-defined input sequence is applied. Even if a hacker arranges to by-pass the initialization stage by structural modifications, the inserted FSM can be controlled to have the desired bit-patterns corresponding to the states in the *pre-initialization state space*, thus revealing the watermark. For post-silicon authentication, scan flip-flops can be used to bring the design to the obfuscated mode. Fig. 2.3 illustrates the modification of the state transition function for embedding authentication signature in the *obfuscated mode* of operation.

To mask or disable the embedded signature, a hacker needs to perform the following steps, assuming a purely random approach:

1. Choose the correct inserted FSM state elements (n_p) from all the total state elements (n_t). This has $\binom{n_t}{n_p}$ possible choices.

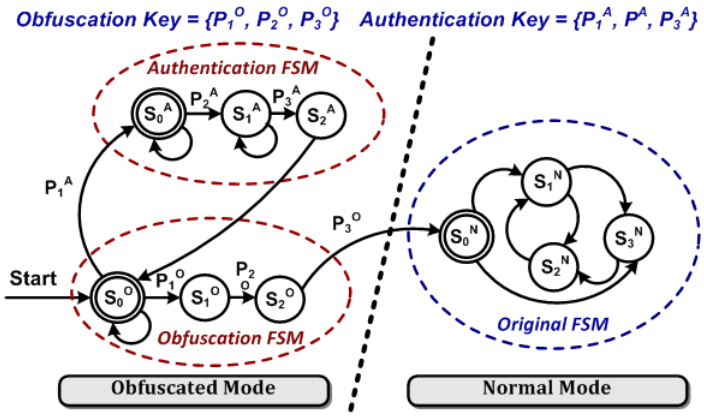


Fig. 2.3. Modification of the initialization state space to embed authentication signature.

2. Apply the correct input vector at the n_i input ports where the vectors are to be applied to get the signature at the selected n_o output ports. This is one out of 2^{n_i} choices.

3. Choose the n_o primary outputs at which the signature appears from the total set of primary outputs (n_{po}). This has $\binom{n_{po}}{n_o}$ possibilities.

4. For each of these recognized n_o outputs, identify it to be one among the possible $2^{2^{(n_i+n_p)}}$ Boolean functions (in the *obfuscated* mode) of the n_i primary inputs and n_p state elements, and change it without changing the normal functionality of the IP.

Hence, in order to mask one signature, the attacker has to make exactly one correct choice from among $N = \binom{n_t}{n_p} \cdot 2^{n_i} \cdot \binom{n_{po}}{n_o} \cdot 2^{2^{(n_i+n_p)}}$ possible choices, resulting in a masking success probability of $P_{masking} \cong \frac{1}{N}$. To appreciate the scale of the challenge, consider a case with $n_t = 30$, $n_p = 3$, $n_i = 4$, $n_o = 4$ and $n_{po} = 16$. Then, $P_{masking} \sim 10^{-47}$. In actual IPs, the masking probability would be substantially lower because of higher values of n_p and n_t.

2.4.3 Choice of Optimal Set of Nodes for Modification

To obfuscate a design, an optimal set of nodes need to be chosen to be modified, so that maximum obfuscation is achieved under the given constraints. The practical level of obfuscation is estimated by the amount of verification mismatch reported by a *Formal Verification* based equivalence checker tool. Formal equivalence checker tools essentially try to match the input logic cones at the state-elements and the primary outputs of the reference and the implementation [30]. Hence, nodes with larger fanout logic cone would be preferred for modification since that will in turn affect the input logic of comparatively larger number of nodes. Also, large input logic cone of a node is generally indicative of its higher *logic depth*; hence, any change at such a node is likely to alter a large number of primary outputs. Thus, in determining the suitability metric for a node as a candidate for modification, both these factors need to be considered. The following metric is proposed as the *suitability metric* for a node:

$$M_{node} = \left(\frac{w_1 \cdot FO}{FO_{max}} + \frac{w_2 \cdot FI}{FI_{max}}\right) \times \frac{FO \cdot FI}{FI_{max} \cdot FO_{max}} \tag{2.7}$$

where FI and FO are the number of nodes in the fan-in and the fan-out cone of the node, respectively. FI_{max} and FO_{max} are the maximum number of fan-in and fan-out nodes in the circuit netlist and are used to normalize the metric. w_1 and w_2 are weights assigned to the two factors, with $0 \leq w_1, w_2 \leq 1$ and $w_1 + w_2 = 1$. The values $w_1 = w_2 = 0.5$ were chosen because they gave the best results in terms of obfuscation, as shown in the next section. Note that $0 < M_{node} \leq 1$. Because of the widely differing values of FO_{max} and FI_{max} in some circuits, it is important to consider both the sum and the product terms involving $\frac{FO}{FO_{max}}$ and $\frac{FI}{FI_{max}}$. Considering only the sum or the product term results in an inferior metric that fails to capture the actual suitability of a node, as observed in the simulations.

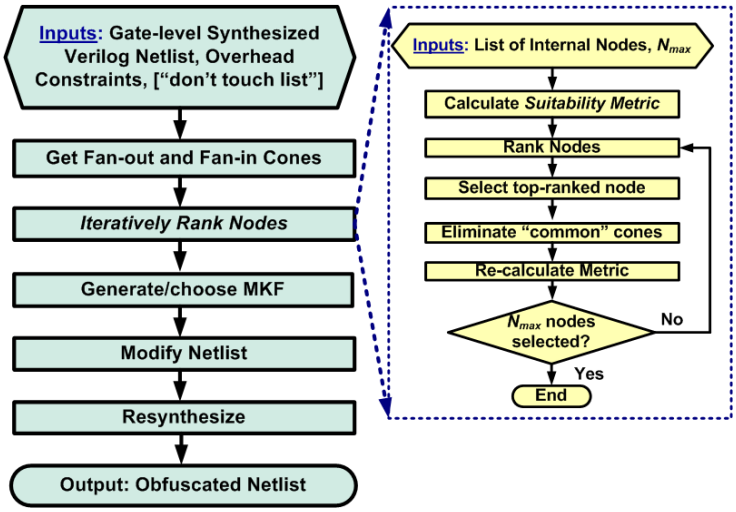


Fig. 2.4. Hardware obfuscation design flow along with steps of the iterative node ranking algorithm.

2.4.4 The *HARPOON* Design Methodology

The overall hardware obfuscation design flow is shown in Fig. 2.4. First, from the synthesized gate-level HDL netlist of an IP core, the fan-in and fan-out cones of the nodes are obtained. Then, an iterative ranking algorithm is applied to find the most suitable N_{max} modifiable nodes, where N_{max} is the maximum number of nodes that can be modified within the allowable overhead constraints. The ranking is a multi-pass algorithm, with the metric for each node being dynamically modified based on the selection of the node in the last iteration. The algorithm takes into account the overlap of the fan-out cones of the nodes which have been already selected and eliminates them from the fan-out cones of the remaining nodes. On the completion of each iteration, the top ranking node among the remaining nodes is selected, so that selection of N_{max} nodes would take N_{max} iterations. In this way, as the iterations progress, the nodes with more non-overlapping fan-out cones are assigned higher weight. The superiority of this iterative approach over a single-pass ranking approach was observed for all the benchmark circuits considered.

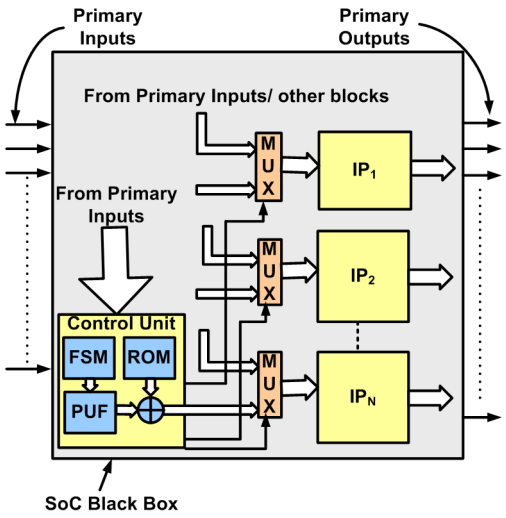


Fig. 2.5. SoC design modification to support hardware obfuscation. An on-chip controller combines the input patterns with the output of a PUF block to produce the activation patterns.

A "*don't touch*" list of nodes can be optionally input to to direct the software tool not to modify certain nodes, e.g. nodes which fall in the critical path. In large benchmarks, it was observed that there were sufficient nodes with high fanouts, such that skipping a few "*don't touch*" nodes still maintains the effectiveness of node modification algorithm in achieving functional and structural obfuscation. For each node to be modified, proper **MKF** (g) is selected either on the basis of its fan-in cone size, or by OR-ing several primary inputs which were originally not present in its input logic cone. The FSM is then integrated with the gate-level netlist and the selected nodes are modified. The modified design is re-synthesized and flattened to generate a new gate-level netlist. The integrated FSM and the modification cells are no longer visually identifiable in the resultant netlist. This re-synthesis is performed under timing constraint, so that it maintains circuit performance.

The IP vendor applies the hardware obfuscation scheme to create a modified IP and supplies it to the design house, along with the activating sequences. The design house receives one or multiple IPs from the IP vendors and integrates them on chip.

To activate the different IPs, the designer needs to include a low-overhead *controller* in the SoC. This controller module can perform the initialization of the different IP blocks in two different ways. In the first approach, it serially steers the different initialization sequences to the different IP blocks from the primary inputs. This controller module will include an integrated FSM which determines the steering of the correct input sequences to a specific IP block. Multiplexors are used to steer initialization sequences to the IP blocks, or the primary inputs and internal signals during normal operation. The chip designer must modify the test-benches accordingly to perform block-level or chip-level logic simulations.

In the second approach, the initialization sequences is stored permanently on-chip in a ROM. In the beginning of operations, the controller module simply reads the different input sequences in parallel and sends them to the different IP blocks for initialization. The advantage of this approach is that the number of initialization cycles can be limited. However, additional overhead is incurred for storing the input sequences in an on-chip ROM. To increase the security of the scheme, the chip designer can arrange an instance-specific initialization sequence to be stored in an one-time programmable ROM. In that case, following the approach in [28], the activating patterns can be simple logic function (e.g. and XOR) of the patterns read from the ROM and the output of a *Physically Unclonable Function* (**PUF**) block. The patterns are written to the ROM post-manufacturing after receiving instructions from the chip designer, as suggested in [28]. Because the output of a PUF circuit is not predictable before manufacturing, it is not possible to have the same bits written into the programmable ROMs for each IC instance. Fig. 2.5 shows this scheme.

The manufacturing house manufactures the SoC from the design provided by the design house and passes it on to the test facility. If a PUF block has been used in the IC, the test engineer reports the output on the application of certain vectors back to the chip designer. The chip designer then calculates the specific bits required to be written in the one-time programmable ROM. The test engineer does so and blows off an one-time programmable fuse, so that the output of the PUF block is no longer

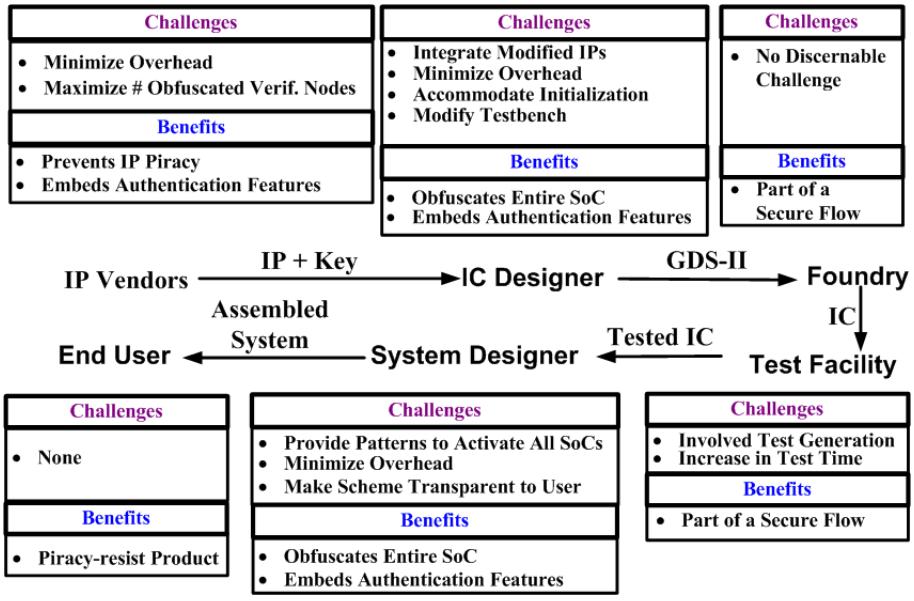


Fig. 2.6. Challenges and benefits of the *HARPOON* design methodology at different stages of a hardware IP life cycle.

visible at the output. The test engineer then performs post-manufacturing testing, using the set of test vectors provided by the design house. Ideally, all communication between parties associated with the design flow should be carried out in an encrypted form, using symmetric or asymmetric cryptographic algorithms such as Diffie-Hellman [8]. The tested ICs are passed to the system designer along with initialization sequence (again in an encrypted form) from the design house.

The system designer integrates the different ICs in the board-level design and arranges to apply the initialization patterns during "booting" or similar other initialization phase. Thus, the initialization patterns for the different SoCs need to be stored in Read Only Memory (ROM). In most ASICs composed of multiple IPs, several initialization cycles are typically needed at start-up to get into the "steady-stream" state, which requires accomplishing certain tasks such as initialization of specific registers [31]. The system designer can easily utilize this inherent latency to hide the additional cycles due to initialization sequences from the end user.

Finally, this secure system is used in the product for which it is meant. It provides the end-user with the assurance that the components have gone through a secure and piracy-proof design flow. Fig. 2.6 shows the challenges and benefits of the design flow from the perspectives of different parties associated with the flow. It is worth noting that the proposed design methodology remains valid for a SoC design house that uses custom logic blocks instead of reusable IPs. In this case, the designer can synthesize the constituent logic blocks using the proposed obfuscation methodology for protecting the SoC.

2.5 Results

In this section simulation results are presented to demonstrate the effectiveness of the proposed hardware obfuscation methodology for a set of ISCAS-89 benchmark circuits [32], as well as the AES encrypter/decrypter IP core [33].

2.5.1 Simulation Setup

All circuits were synthesized using Synopsys *Design Compiler* with optimization parameters set for minimum area and mapped to a LEDA 250 nm standard cell library. The flow was developed using the TCL scripting language and was directly integrated in the *Design Compiler* environment. All formal verification was carried out using Synopsys *Formality*. The verification nodes considered by Formality constituted of the inputs of state elements (e.g. flip-flops) and primary outputs.

2.5.2 Results for ISCAS-89 Benchmark Circuits

Choice of Scheme

A simple four-state FSM was designed for each of the benchmarks to achieve hardware and functional obfuscation. Initially, three choices were explored - the simple node modification scheme using only XOR gates (scheme 1), the theoretically

Table 2.1
Average Number of Failing Patterns for ISCAS-89 Benchmark Circuits for Different Modification Schemes

Benchmark	**Scheme-1**	**Scheme-2**	**Scheme-3**
s298	51	158	193
s344	215	1093	1233
s444	197	569	772
s526	146	485	1186
s641	598	2491	5135
s713	913	2928	3301
s838	382	1757	5106
s1196	2423	5382	9573
s1238	2552	5157	9511
s1423	6431	18120	28350
s1488	333	1816	1156
s5378	13311	29482	53066
s9234	13862	30385	53365

suggested modification scheme employing OR-ing of selected primary inputs (scheme 2) and lastly the low-overhead modification scheme employing random selection of internal nodes avoiding combinational loops (scheme 3). Then, the maximum number of modifiable nodes N_{max} for each benchmark circuit was determined considering four different area constraints (5%, 10%, 15% and 20%). For all the benchmarks, the number of nodes modified was less than 5% of the total number of nodes in the netlist. Table 2.1 shows the number of total *failing patterns* reported by *Formality* for the benchmarks, averaged over the different area constraints. From these results, it is clear that the random node selection algorithm (avoiding combinational loops) achieves the highest number of failing vectors for a given area overhead constraint. Hence, in the rest of this section, all the results presented would be for scheme-3.

Table 2.2
Number of Failing Patterns for ISCAS-89 Benchmark Circuits

Benchmark Circuit	Area Constraint			
	5%	10%	15%	20%
s298	176	184	202	210
s344	1010	1290	1228	1402
s444	364	638	1004	1082
s526	416	1216	1270	1842
s641	3723	3185	6045	7585
s713	3084	3143	3210	3765
s838	1742	5118	5466	8096
s1196	9631	10321	8931	9408
s1238	8442	9876	9780	9944
s1423	11868	25263	29007	47262
s1488	760	1096	1344	1422
s5378	48799	52059	55961	56638
s9234	13862	30385	53365	56638

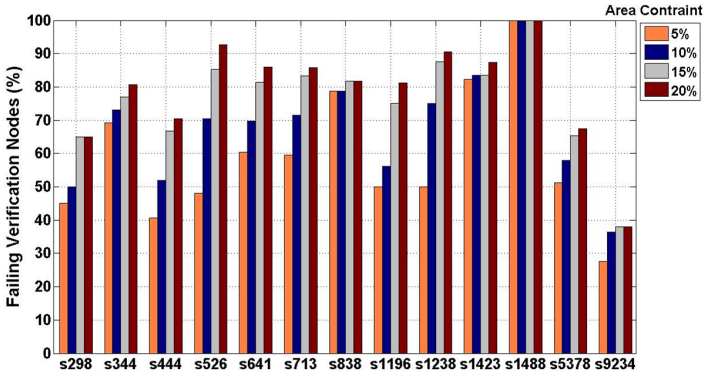
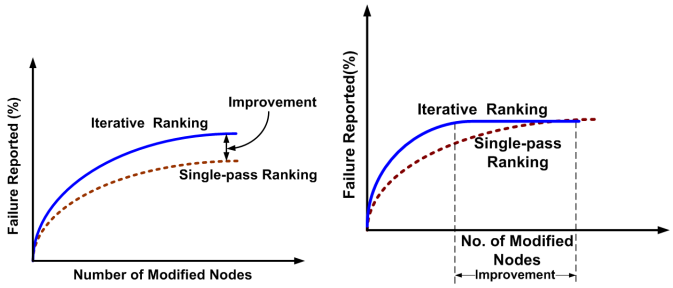


Fig. 2.7. Observed verification failures (with application of the *HARPOON* methodology) for ISCAS-89 circuits.

Obfuscation Effects

The benchmarks were then subjected to the hardware obfuscation design flow (including the iterative ranking algorithm). Table 2.2 shows the number of failing



(a) (b)

Fig. 2.8. Improvement with the iterative node ranking algorithm: (a) verification failure percentage and (b) number of nodes to be modified at iso-failure percentage.

patterns for the ISCAS-89 benchmark circuits for different projected area overheads. The modified and re-synthesized benchmarks were then subjected to Formal verification using Synopsys *Formality*. Fig. 2.7 shows the observed percentage of verification nodes failing verification reported by *Formality* for the different benchmark circuits.

Effect of the Iterative Ranking Algorithm

The improvement in the obfuscation methodology was also noted by adopting the multi-pass *iterative ranking* algorithm in place of a single-pass ranking algorithm. A general trend noticeable was that the percentage of nodes failing verification "saturated" with the increase in the number of modified nodes, irrespective of the ranking algorithm used. The iterative ranking algorithm primarily had two effects. The first effect (Fig. 2.8(a)) was the increase in the percentage of reported failures (at equal area overheads) for the iterative ranking algorithm compared to the single-pass algorithm. The second effect (Fig. 2.8(b)) was a decrease in the number of node modifications required to achieve the same maximum (saturated) failure percentage, so that one can identify the redundant node modifications and eliminate them, saving area overhead in the process. For example, from Fig. 2.7 for the s1488 benchmark,

Table 2.3
Design Overheads (%) for Different Area Constraints

Benchmark	5% area constraint			10% area constraint			15% area constraint			20% area constraint		
Circuit	**Area**	**Delay**	**Power**	**Area**	**Delay**	**Power**	**Area**	**Delay**	**Power**	**Area**	**Delay**	**Power**
s298	3.91	0.00	5.26	8.64	0.00	14.04	14.74	0.50	16.67	19.29	0.00	18.42
s344	3.45	-2.98	5.38	8.83	-8.96	9.87	14.89	-6.20	14.35	18.59	-7.90	18.83
s444	4.63	0.00	9.62	9.50	0.00	18.27	14.15	0.00	20.19	18.26	0.00	23.08
s526	3.64	0.00	7.06	8.12	0.00	16.17	14.89	-1.60	21.87	19.30	-0.78	25.28
s641	4.96	-3.66	8.20	9.74	-3.66	13.90	14.02	-4.60	19.59	19.63	-4.20	23.01
s713	4.06	-3.66	7.34	9.07	-3.66	14.69	14.43	-2.60	20.34	19.26	-2.60	22.88
s838	2.20	-2.39	6.92	9.00	-8.57	9.76	14.17	-5.50	12.93	19.13	0.00	14.00
s1196	3.96	0.00	6.04	6.06	0.00	16.09	14.45	-1.76	19.14	18.10	0.00	20.55
s1238	4.52	-0.42	6.52	9.99	-0.42	9.97	14.87	0.00	18.26	18.23	-0.90	23.79
s1423	4.70	-0.78	8.02	9.61	-2.64	14.91	14.97	-1.08	22.43	19.99	-2.42	26.19
s1488	3.27	-2.79	3.13	8.65	-0.93	8.33	13.19	0.00	10.49	18.17	0.00	13.62
s5378	4.34	0.00	8.91	9.87	0.00	13.80	13.84	0.00	20.43	19.93	0.00	23.70
s9234	4.74	0.00	5.80	8.82	3.60	12.37	8.82	3.50	15.52	14.29	3.80	19.72
Average	4.03	-1.28	8.83	8.92	-1.94	13.31	14.38	-1.49	17.81	19.06	-1.15	20.91

it is evident that one does not need to consider area overheads above 10%, because at higher area overheads, the same obfuscation level (in terms of the number of verification failures) is obtained.

Overheads Incurred

Table 2.3 shows the area, delay and power overheads of the re-synthesized benchmark circuits, following the application of the obfuscation scheme, for the projected area overheads of 5%, 10%, 15% and 20% respectively. From the table, it is clear that the actual area overheads were smaller than the imposed constraints in all cases, while the timing overhead was negative, i.e. the timing constraint was met with positive slack in most cases. The power overheads in all cases were within acceptable limits. The design overhead is caused both by the addition of combinational (in the form of the *modification cells*) and few sequential elements to implement the inserted

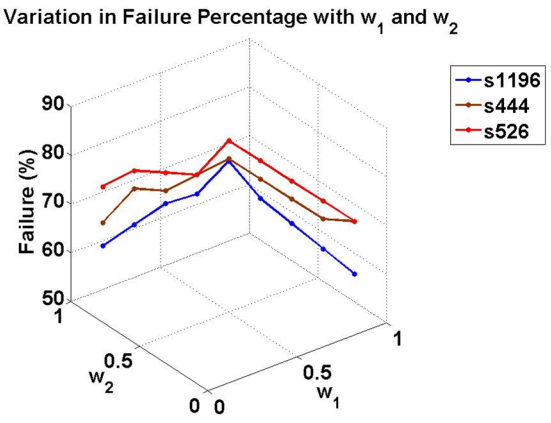


Fig. 2.9. Effect of weights w_1 and w_2 on verification failure percentage.

FSM. Although sequential memory elements do not scale as well as combinational ones, the percentage overhead is expected to remain unchanged for more advanced technology nodes because the combinational overhead forms the major fraction of the total overhead.

Optimal Choice of Parameters w_1 and w_2 in eqn. (2.7)

Different combinations of the parameters w_1 and w_2 in eqn. 2.7, were tried under the constraint $w_1 + w_2 = 1$. This was done to come up with a combination of w_1 and w_2 that will generate an optimal ranking of the nodes leading to maximum obfuscation. Fig. 2.9 shows the plot of the failure percentage for three benchmarks against different combinations of w_1 and w_2. From this plot, it is clear that a choice of $w_1 = w_2 = 0.5$ is optimal and leads to the maximum obfuscation. Such a trend was observed in all the benchmark circuits that were dealt with.

Table 2.4
Obfuscation Efficiency and Design Overheads (%) for the AES modules (for 5% area constraint)

AES Modules	Obfuscation Efficiency			Design Overhead		
	Nodes Modified (%)	Failing Verif. Points (%)	Failing Vectors	Area (%)	Delay (%)	Power (%)
Key Expand	0.95	78.3	4864	3.69	0.00	4.56
Sbox	0.95	100	236	3.62	2.42	5.31
Inverse Sbox	0.97	248	949	4.34	2.60	5.62

2.5.3 Results for the AES Encryption/Decryption Unit

Simulation results corresponding to the application of the *HARPOON* design methodology on the *Advanced Encryption Standard* (AES) encryption/decryption core are now presented. AES was chosen because it is widely deployed and is widely available from IP vendors as a RTL or gate-level IP. The open-source Verilog RTL implementations of 128-bit AES encrypter and decrypter cores available at [33] were used in the simulations. The cores were modified to integrate them in a single encrypter/decrypter core, the mode of operation being controlled by a *MODE_CONTROL* signal. The three main sub-modules of the design: the *Key Expand*, the *SBox* and the *Inverse SBox* were separately subjected to the design flow with an estimated area overhead of 5%.

The modules were designed with an integrated obfuscating finite state machine having 4 states and controlled by 10 inputs each. The scheme of Fig. 2.5 was used, with an integrated controller (which is itself a simple FSM) and a PUF module with 5 controlling inputs and 2 outputs. The controller is activated by a sequence of 3 inputs patterns applied at 10 selected primary inputs. On activation, the controller starts reading the bits stored on an on-chip ROM in sequence. Simultaneously, it applies a 5-bit pattern sequence to the on-chip PUF block. In response, the PUF block produces the 2-bit sequences. In the HDL simulations, the PUF module was modeled as a ROM. These output bits from the PUF are XOR-ed with the outputs

of the ROM and then applied to the different modules in parallel to activate them individually.

Design Overhead

Table 2.4 shows the performance and design overheads for the different modified modules, at an estimated 5% area overhead. Again a very high rate of verification failures was reported for the three modules, along with a large number of failing vectors. The actual area overhead was less than the estimated area overhead (5%) in all cases and the power and timing overheads were within acceptable limits. The unmodified synthesized design had a operating clock frequency of 230 MHz and a data rate of 2.45Gbit/s. The modified design after re-synthesis had a operating clock frequency of 220.1MHz and a corresponding bit-rate of 2.34Gbit/s. The controller FSM takes 3 clock cycles to get activated and then takes 3 clock cycles each to activate the *SBox* modules, the *Inverse SBox* modules and the *Key Expand* module. Note that all the *SBox* modules are activated in parallel and so are all the *Inverse SBox* modules. Hence, the latency of the system for initialization is 9 clock cycles, i.e, ∼41ns. The startup latency increases to 22 cycles from 13 cycles (the time taken to encrypt/decrypt one block of data) in the unmodified design. Note that this one-time increase in latency is incurred only at system start-up, which can be easily masked to the end-user. Table 2.5 shows the overall design overhead of the scheme. Again, all the overheads are less than 5%, with the overall delay overhead slightly higher than the individual module overheads.

2.5.4 Mode control through unreachable states

Addition of extra state elements to realize the mode-control FSM can be avoided by utilizing the *unreachable states* of a circuit. Complex sequential circuits would typically have a large number of unreachable states [34]. The state transition described in Fig. 2.2 can be achieved if a few unreachable states are identified and the

Table 2.5
Overall Design Overheads (%) for the AES core

Parameter	**Original**	**Modified**	**Overhead**
Area(μm^2)	2732060.0	2825636.6	3.43% increase
Power(mW)	565.0	586.6	3.82% increase
Start-up latency (cycles)	13	22	9 cycles increase
Op. Frequency(MHz)	230	220.1	4.30% decrease
Bit Rate(Gbps)	2.45	2.34	4.49% decrease

circuit is forced to go through these states during the initialization phase. By taking advantage of the unused state space of the original SEs, one can effectively reduce the design overhead and improve obfuscation level.

The required number of unreachable states would be equal to the number of patterns in the initialization sequence (e.g. 3-4). In case sufficient number of unreachable states are not identified, a combination of new states (due to extra state elements) and unreachable states can be used to realize the scheme. The unreachable states can be identified through simultaneous *sequential justification* of the state elements in the circuit. However, this is typically computationally intensive process. To avoid the difficulty of finding unreachable states for the entire set of state elements in the circuit, note that random selection of a small subset of state elements from the circuit and their sequential justification is sufficient to identify unreachable states for the circuit. It also reduces the computational complexity of the problem drastically. Once the unreachable states are identified comprising of a set of SEs, a *parallel state machine* (PSM) can be formed using these SEs to realize the state transitions during the initialization process and then folding the PSM into the original one during resynthesis process. Additionally, the *en* signal is generated as an output of the PSM and alters the behavior of the circuit in the *obfuscated* mode through modification cells as described earlier. Upon successful initialization, all state elements hold values corresponding to a valid state and the circuit enters *normal* mode.

Table 2.6
Design Overheads (%) at iso-delay for ISCAS-89 Circuits with an Embedded FSM Utilizing Unreachable States

Benchmark	**Overhead (%)**	
Circuit	**Area**	**Power**
s1196	18.44	15.88
s1238	16.31	15.83
s1423	6.11	9.65
s1488	12.81	6.22
s5378	14.45	23.84
s9234	6.53	9.04
Average	12.44	12.81

To check the effectiveness of using unreachable states, the technique was applied to several ISCAS-89 circuits by choosing six SEs at random. The states unreachable by these six state elements were found through sequential justification by *Synopsys TetraMax*, and the RTL for a four state PSM was generated from these unreachable states. The same number of modification cells as those used for the 10% area constraint were inserted. This RTL for the PSM and the modification cells was integrated with the gate level netlist and the circuit was re-synthesized under delay constraint. Table 2.6 shows the percentage area and power overheads at iso-delay for the benchmark circuits. The fraction of nodes failing verification and the number of failing vectors were comparable with or better than the results presented in Table 2.2. The overhead for some benchmarks such as *s1196* is high because the specific choice of six SEs and the state encoding using them resulted into large amount of extra logic in the synthesized netlist. By performing multiple iterations of random selection of SEs, it may be possible to find a different and/or smaller set of SEs which provides sufficient number of unreachable states. For example, a set of 5 and 4 SEs was found which led to an area overhead of 8.8% and 7.1%, respectively, both of which are considerably less than that obtained with additional SEs.

2.6 Summary

In this chapter, a netlist level hardware obfuscation based anti-piracy design flow for SoCs has been presented. It involves active participation of the IP vendor, the IC designer and the system designer and helps to preserve the rights of all of them. The scheme is based on modification of the gate-level netlist of a pre-synthesized IP core, followed by re-synthesis to obtain maximum functional and structural obfuscation at low design overhead. While providing obfuscation and authentication capabilities to the design at every stage of the design flow, the scheme does not affect the experience of an end-user. Simulation results with a set of ISCAS-89 benchmark circuits and the AES encryption/decryption core show that this scheme is capable of providing high levels of design obfuscation at nominal area overhead under delay constraint. The security of the design flow can be further enhanced by providing separate activation keys to each IP customer and using on-chip *Physically Unclonable Function* circuits. In the next chapter, we extend this technique for the protection of RTL design descriptions.

3. RTL HARDWARE IP PROTECTION THROUGH OBFUSCATION

3.1 Introduction

Hardware IP protection for gate-level designs has been explored in the last chapter. However, majority of the commercial hardware IPs come in the RTL ("soft") format, which offers better flexibility and portability by allowing design houses to use them in different technology platforms [35]. Hence, an IP protection approach which targets the security of RTL design descriptions, caters to a much larger section of the IP market.

In this chapter, we extend the fundamental hardware obfuscation concept presented in Chapter 2 to provide two RTL IP obfuscation solutions, which differ in level of protection and computational complexity. The first technique, referred to as the "STG modification approach", converts a RTL description to gate level; obfuscates the gate level netlist; and then de-compiles it back to RTL. The second technique, referred to as the "CDFG modification approach", avoids the forward compilation step and applies obfuscation in the register transfer level by modifying its control and data flow constructs, which is facilitated through generation of a CDFG of the RTL design. The first approach can provide higher level of obfuscation, but has greater hardware overhead and is computationally more expensive than the second one. We compare the two approaches both qualitatively and quantitatively – we derive appropriate metrics to quantify the obfuscation level for both the approaches. We show that the level of protection can be improved with minimal hardware overhead by increasing the length of the initialization key sequence. Finally, along with obfuscation, we show that the proposed approaches can be extended to embed a hard-to-remove "digital watermark" in the IP that can help to authenticate the IP in case of illegal usage.

The authentication capability comes at very little additional hardware overhead and thus helps to reduce the overall design overhead, while providing more security.

Although we focus on the protection of hardware IP modules primarily consisting of logic blocks and state elements, external memory has become an integral component of SoCs or embedded systems. The size of memory in a typical SoC or embedded device is ever increasing, allowing the user to harness greater functionality [36]. It is essential to ensure the security of both the memory interface hardware as well as the memory contents in a SoC or an embedded device. Hence, we also analyze obfuscation based solutions to these problems in this chapter.

The rest of the chapter is organized as follows. In Section 3.2, we describe the two proposed IP obfuscation schemes, and their relative pros and cons. In Section 3.3, we present theoretical analysis of the proposed obfuscation schemes to derive metrics to quantify the level of obfuscation achievable. Section 3.4 presents the automated design flows developed by us, and simulation results for several open-source IP cores, and quantitative comparison of the relative scalability with respect to security of the proposed schemes between themselves and with respect to the AES encryption system. In Section 3.5, we describe a technique to decrease the overhead by utilizing the normally unused states of the circuit; applicability of the proposed technique in providing protection against *hardware Trojans*, and how the obfuscation technique can be extended to the protection of memory interface hardware and memory contents. We conclude in Section 3.6.

3.2 Obfuscation Methodology

In this section, we describe the two proposed obfuscation-based RTL IP protection schemes, and compare their pros and cons. We also describe the enhanced design flow that we have developed by integrating them with the traditional SoC design and manufacturing flow described in Chapter 2. First, we describe the STG modification and de-compilation based IP protection technique [37], followed by the technique

based on CDFG modification [38]. We end the section with a comparison of the relative advantages and disadvantages of the two approaches.

3.2.1 STG Modification Approach

Methodology

This technique has three main steps:

- Logic synthesis of the RTL to an unmapped, unoptimized gate-level netlist, composed of generic Boolean gates.
- Functional obfuscation of the netlist by structural modifications following the principle outlined in Chapter 2, and,
- Subsequent de-compilation of the obfuscated netlist back to an obfuscated version of the original RTL.

The advantage of modifying the functionality at the gate-level is the relative incomprehensibility of such a circuit description compared to a RTL description. The modified gate-level design is then decompiled to regenerate the RTL of the code, without maintaining high level HDL constructs. Instead, the modified netlist is traversed recursively to reconstruct the Boolean equations for the primary output nodes and the state element inputs, expressed in terms of the primary inputs, the state-element outputs and a few selected high fanout internal nodes. The redundant internal nodes are then removed. This "partial flattening" effect hides all information about the modifications performed in the netlist. The obfuscation tool maintains a list of expected instances of library datapath elements, and whenever these are encountered in the netlist, their outputs are related through proper RTL constructs to their inputs. This ensures regeneration of the same datapath cells on re-synthesis of the RTL.

As an example, consider the simple Verilog module "*simple*" which performs addition or subtraction of two bits depending on the value of a free running one-bit

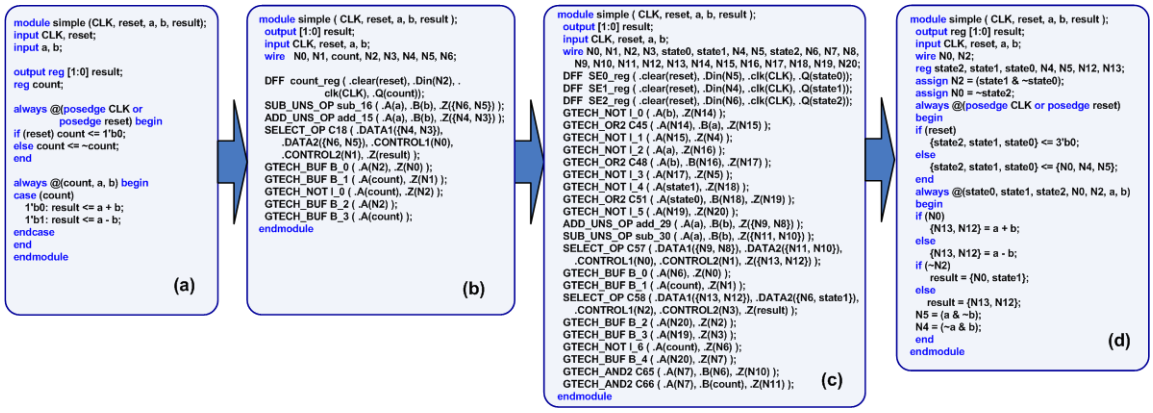


Fig. 3.1. Example of a Verilog RTL description and its obfuscated version [37]: a) original RTL; b) technology independent, unoptimized gate-level netlist obtained through RTL compilation; c) obfuscated gate-level netlist; d) decompiled obfuscated RTL.

counter, as shown in Fig. 3.1(a). Fig. 3.1(b)-(d) shows the transformation of the design through the proposed obfuscation process. The decompiled RTL in Fig. 3.1(d) shows that the modification cell and the extra state transition logic are effectively hidden and isolation of the correct initialization sequence can be difficult even for such a small design. Major semantic effect of obfuscation is the replacement of high level RTL constructs (such as `if...else`, `for`, `while`, `case`, `assign` etc.) in the original RTL with different such constructs and replacement of internal nodes and registers. Furthermore, internal register, net and instance names are changed to arbitrary identifiers to make the code less comprehensible.

After the gate-level modification, the modified netlist is de-compiled to produce a description of the circuit, which although being technically a RTL and functionally equivalent to the modified gate-level netlist, is extremely difficult to comprehend to a human reader. In addition, the modifications made to the original circuit remain well-hidden. A forward annotation file indicates relevant high-level HDL constructs and macros to be preserved through this transformation. These are maintained during the RTL compilation and de-compilation steps. From the unmapped gate-level

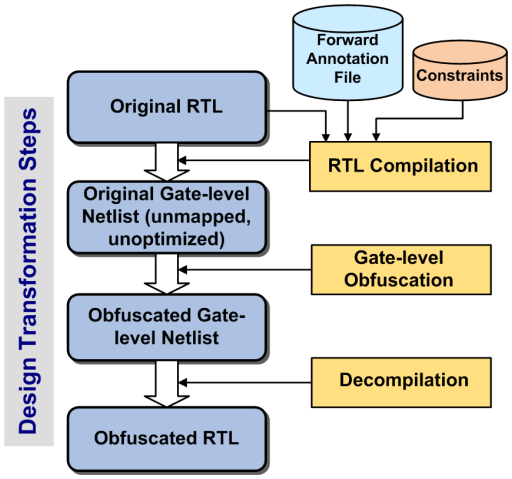


Fig. 3.2. Design transformation steps in course of the proposed RTL obfuscation process [37].

netlist, we look for specific generic gates, that can be decompiled to an equivalent RTL construct, e.g. multiplexor can be mapped to an equivalent `if...then...else` construct or a `case` construct. The datapath modules or macros are transformed into appropriate operands. The remaining netlist is traversed recursively to re-construct the Boolean equations for the primary output nodes and the state element inputs, expressed in terms of the primary inputs, the state element outputs and a few selected high fanout internal nodes. The redundant internal nodes are then removed. This "partial flattening" effect hides information about the modifications performed in the netlist. Once the logic equations are formed, specific signature in the equation is searched to map it to a suitable higher level RTL construct. For example, an equation $n1 = s1 \cdot d1 + s2 \cdot d2 + s3 \cdot d3$ can be mapped to a `case` construct. Fig. 3.2 shows the design transformation steps during the obfuscation process. We present an analysis of the security of this scheme in Section 3.3.

Embedding Authentication Features

To prevent against attacks from trusted parties with knowledge of the initialization sequence in the design flow, the designer can optionally embed a *signature* within the design which acts as a *digital watermark*. This can be done by another modification of the state transition function, as shown by the scheme of Fig. 2.3 of Chapter 2. Finding unreachable states for a large circuit with many state elements is computationally challenging; however as shown in Section 2.5.4 that by considering small groups of state elements in a given circuit and performing *sequential justification* using commercially available EDA tools such as *Tetramax*, it is feasible to derive unreachable states for large circuits in reasonable time.

3.2.2 CDFG Modification Approach

Methodology

Similar to the STG modification based scheme, the main idea of this approach is to efficiently integrate a key-enabled, *mode control FSM* into the design through judicious modification of control and data flow structures derived from the RTL, such that the design works in two different modes *obfsucated* and *normal*. The *mode control FSM* is integrated inside the CDFG derived from the RTL in a way that makes it extremely hard to isolate from the original IP. The FSM is realized in the RTL by expanding a judiciously selected set of registers, which we refer to as *host registers* and modifying their assignment conditions and values. Once the FSM has been integrated, both control and data flow statements are conditioned based on the mode control signals derived from this FSM. The proposed obfuscation scheme comprises of four major steps described below.

Parsing the RTL and Building CDFG: In this step, the given RTL is parsed and each concurrent block of RTL code is transformed into a CDFG data structure. Fig. 3.3 shows the transformation of an "always @()" block of a Verilog code to its

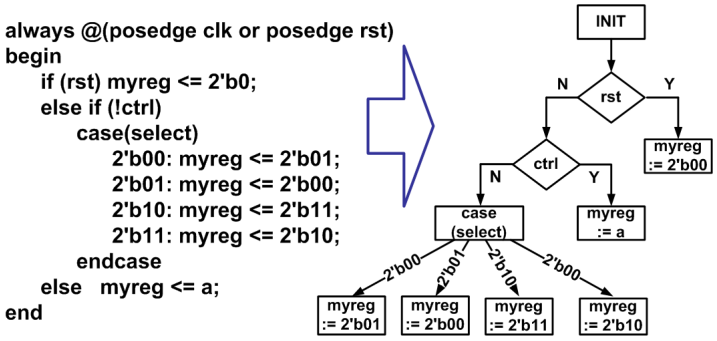


Fig. 3.3. Transformation of a block of RTL code into CDFG.

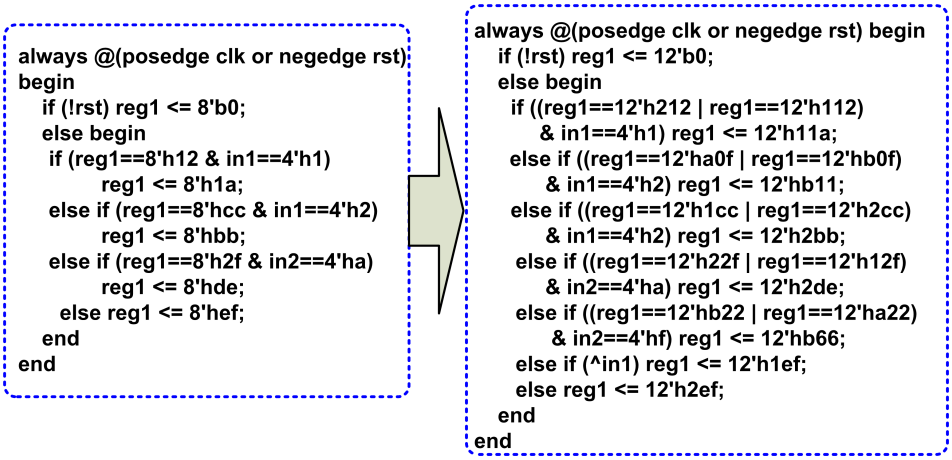


Fig. 3.4. Example of hosting the registers of the mode-control FSM.

corresponding CDFG. Next, small CDFGs are merged (whenever possible) to build larger combined CDFGs. For example, all CDFGs corresponding to non-blocking assignments to clocked registers can be combined together without any change of the functionality. This procedure creates larger CDFGs with substantially more number of nodes than the constituent CDFGs, which helps to obfuscate the *hosted* mode-control FSM better.

"Hosting" the Mode Control FSM: Instead of having a stand-alone mode control FSM as described in Chapter 2, the state elements of the mode-control FSM can be hosted in existing registers in the design to increase the level of obfuscation.

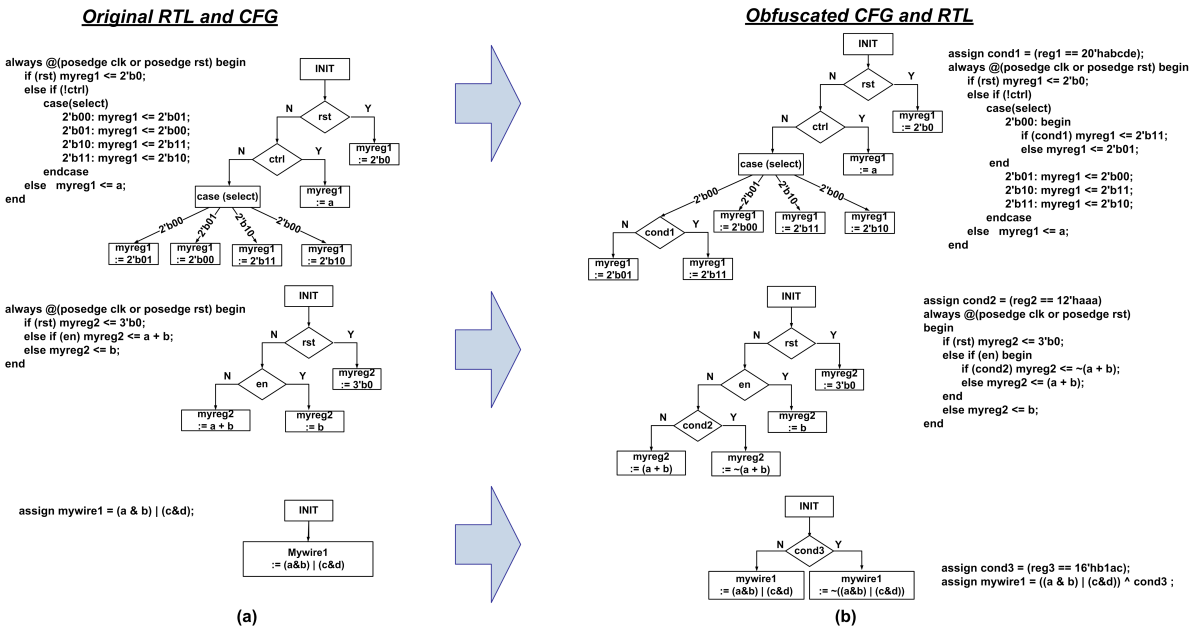


Fig. 3.5. Examples of control-flow obfuscation: (a) original RTL, CDFG; (b) obfuscated RTL, CDFG.

This way, the FSM becomes an integral part of the design, instead of controlling the circuit as a structurally isolated element. An example is shown in Fig. 3.4, where the 8-bit register *reg1*, referred as the "host register", has been expanded to 12-bits to host the mode-control FSM in its left 4-bits. When these 4-bits are set at values $4'h1$ or $4'h2$, the circuit is in its *normal* mode, while the circuit is in its *obfuscated mode* when they are at $4'ha$ or $4'hb$. Note that extra RTL statements have been added to make the circuit functionally equivalent in the *normal mode*. The obfuscation level is improved by distributing the mode-control FSM state elements in a non-contiguous manner inside one or more registers, if possible.

Modifying CDFG Branches: After the FSM has been hosted in a set of selected *host registers*, several CDFG nodes are modified using the control signals generated from this FSM. The nodes with large fanout cones are preferentially selected for modification, since this ensures maximum change in functional behavior at minimal design overhead. Three example modifications of the CDFGs and the corresponding

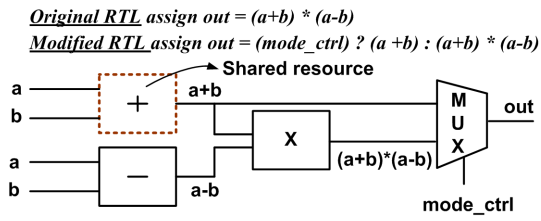
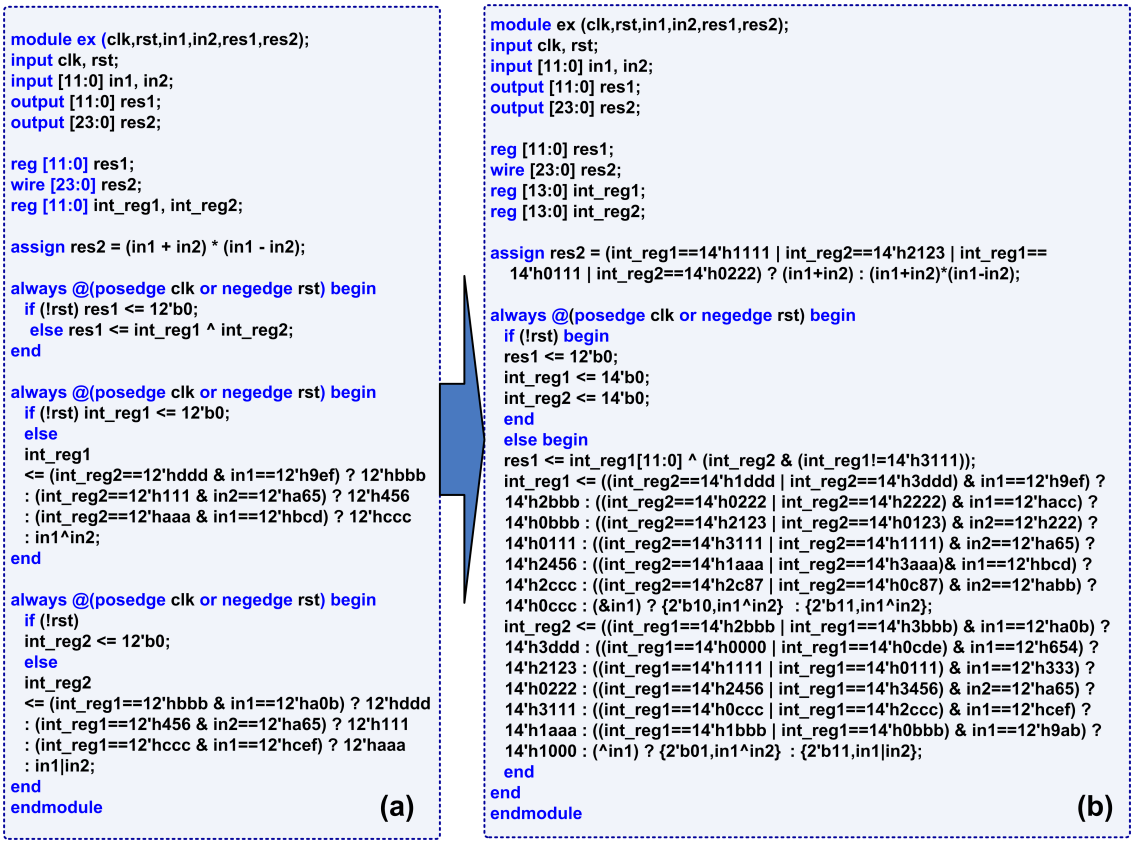


Fig. 3.6. Example of datapath obfuscation allowing resource sharing.

RTL statements are shown in Fig. 3.5. The registers *reg1*, *reg2* and *reg3* are the *host registers*. Three "case()", "if()" and "assign" statements in Fig. 3.5(a) are modified by the mode-control signals *cond1*, *cond2* and *cond3*, respectively. These signals evaluate to logic-1 only in the *obfuscation mode* because the conditions *reg1*=$20'habcde$, *reg2*=$12'haaa$ and *reg3*=$16'hb1ac$ correspond to states which are only reachable in the *obfuscation mode*. Fig. 3.5(b) shows the modified CDFGs and the corresponding CDFG statements.

Besides changing the control-flow of the circuit, functionality is also modified by introducing additional datapath components. However, such changes are done in a manner that ensures sharing of the additional resources during synthesis. This is important since datapath components usually incur large hardware overhead. An example is shown in Fig. 3.6, where the signal *out* originally computes $(a+b)\times(a-b)$. However, after modification of the RTL, it computes $(a+b)$ in the *obfuscated mode*, allowing the adder to be shared in the two modes and the outputs of the multiplier and the adder to be multiplexed.

Generating Obfuscated RTL: After the modifications have been preformed on the CDFG, the obfuscated RTL is generated from the modified CDFGs, by traversing each of them in a *depth-first* manner. Fig. 3.7(a) shows an example RTL code and Fig. 3.7(b) shows its corresponding obfuscated versions. A 4-bit FSM has been *hosted* in registers *int_reg1* and *int_reg2*. The conditions *int_reg1*[13:12]=$2'b00$, *int_reg1*[13:12]=$2'b01$, *int_reg2*[13:12]=$2'b00$ and *int_reg1*[13:12]=$2'b10$ occur only in the obfuscated mode. The initialization sequence is *in1*=$12'h654 \rightarrow$ *in2*=$12'h222$



```
module ex (clk,rst,in1,in2,res1,res2);
input clk, rst;
input [11:0] in1, in2;
output [11:0] res1;
output [23:0] res2;

reg [11:0] res1;
wire [23:0] res2;
reg [11:0] int_reg1, int_reg2;

assign res2 = (in1 + in2) * (in1 - in2);

always @(posedge clk or negedge rst) begin
  if (!rst) res1 <= 12'b0;
   else res1 <= int_reg1 ^ int_reg2;
end

always @(posedge clk or negedge rst) begin
  if (!rst) int_reg1 <= 12'b0;
  else
  int_reg1
  <= (int_reg2==12'hddd & in1==12'h9ef) ? 12'hbbb
  : (int_reg2==12'h111 & in2==12'ha65) ? 12'h456
  : (int_reg2==12'haaa & in1==12'hbcd) ? 12'hccc
  : in1^in2;
end

always @(posedge clk or negedge rst) begin
  if (!rst)
  int_reg2 <= 12'b0;
  else
  int_reg2
  <= (int_reg1==12'hbbb & in1==12'ha0b) ? 12'hddd
  : (int_reg1==12'h456 & in2==12'ha65) ? 12'h111
  : (int_reg1==12'hccc & in1==12'hcef) ? 12'haaa
  : in1|in2;
end
endmodule
```

(a)

```
module ex (clk,rst,in1,in2,res1,res2);
input clk, rst;
input [11:0] in1, in2;
output [11:0] res1;
output [23:0] res2;

reg [11:0] res1;
wire [23:0] res2;
reg [13:0] int_reg1;
reg [13:0] int_reg2;

assign res2 = (int_reg1==14'h1111 | int_reg2==14'h2123 | int_reg1==
   14'h0111 | int_reg2==14'h0222) ? (in1+in2) : (in1+in2)*(in1-in2);

always @(posedge clk or negedge rst) begin
  if (!rst) begin
  res1 <= 12'b0;
  int_reg1 <= 14'b0;
  int_reg2 <= 14'b0;
  end
  else begin
  res1 <= int_reg1[11:0] ^ (int_reg2 & (int_reg1!=14'h3111));
  int_reg1 <= ((int_reg2==14'h1ddd | int_reg2==14'h3ddd) & in1==12'h9ef) ?
  14'h2bbb : ((int_reg2==14'h0222 | int_reg2==14'h2222) & in1==12'hacc) ?
  14'h0bbb : ((int_reg2==14'h2123 | int_reg2==14'h0123) & in2==12'h222) ?
  14'h0111 : ((int_reg2==14'h3111 | int_reg2==14'h1111) & in2==12'ha65) ?
  14'h2456 : ((int_reg2==14'h1aaa | int_reg2==14'h3aaa)& in1==12'hbcd) ?
  14'h2ccc : ((int_reg2==14'h2c87 | int_reg2==14'h0c87) & in2==12'habb) ?
  14'h0ccc : (&in1) ? {2'b10,in1^in2}  : {2'b11,in1^in2};
  int_reg2 <= ((int_reg1==14'h2bbb | int_reg1==14'h3bbb) & in1==12'ha0b) ?
  14'h3ddd : ((int_reg1==14'h0000 | int_reg1==14'h0cde) & in1==12'h654) ?
  14'h2123 : ((int_reg1==14'h1111 | int_reg1==14'h0111) & in1==12'h333) ?
  14'h0222 : ((int_reg1==14'h2456 | int_reg1==14'h3456) & in2==12'ha65) ?
  14'h3111 : ((int_reg1==14'h0ccc | int_reg1==14'h2ccc) & in1==12'hcef) ?
  14'h1aaa : ((int_reg1==14'h1bbb | int_reg1==14'h0bbb) & in1==12'h9ab) ?
  14'h1000 : (^in1) ? {2'b01,in1^in2}  : {2'b11,in1|in2};
  end
end
endmodule
```

(b)

Fig. 3.7. Example of RTL obfuscation by CDFG modification: (a) original RTL; (b) obfuscated RTL.

$\rightarrow$ *in1*=$12'h333$ $\rightarrow$ *in2*=$12'hacc$ $\rightarrow$ *in1*=$12'h9ab$. Note the presence of *dummy state transitions* and out-of-order state transition RTL statements. The outputs *res1* and *res2* have been modified by two different *modification signals*. Instead of allowing the inputs to appear directly in the sensitivity list of the "if()" statements, it is possible to derive internal signals (similar to the ones shown in Fig. 3.5(b)) with complex Boolean expressions which are used to perform the modifications. The output *res1* has been modified following the datapath modification approach using resource sharing.

Table 3.1
Comparison of De-compilation based and CDFG based Approaches

Approach	Advantages	Disadvantages
STG Modfication based Approach	(a) Higher level of obfuscation	(a) Loses major RTL constructs (b) Greater hardware and computational overheads
CDFG Modification based Approach	(a) Works directly on RTL descriptions (b) Preserves RTL constructs	(a) Hiding modifications is more challenging

Embedding Authentication Features

Authentication features might be embedded in the RTL by the same principle as described in Section 3.2.1. RTL statements describing the state transitions of the *authentication FSM* can be integrated with the existing RTL in the same way the statements corresponding to the *obfuscation FSM* is hidden. In case the *unused states* are difficult to derive from the original RTL, it can be synthesized to a gate-level netlist and the same technique based on *sequential justification* as described in Section 3.2.1 might be applied. Note that authentication and obfuscation go hand-in-hand in ensuring security. While authentication acts as a second line of defense, obfuscation, through its defining characteristic of hiding the structure and functionality of the IP, helps to hide the embedded authentication features efficiently.

3.2.3 Comparison between the Two Approaches

Table 3.1 compares the relative advantages and disadvantages of the two proposed techniques. Although the de-compilation based approach potentially can hide the modifications better than the direct RTL modification based approach (as shown by our theoretical analysis of their obfuscation levels in Section 3.3 and by our simulation results), it also loses major RTL constructs and creates a description of the circuit which might result in an unoptimized implementation on re-synthesis. Hence, we provide the IP designer with a choice where either of the techniques might be chosen based on the designer's priority. For example, if the IP is going to released to a

untrustworthy SoC design houses with a prior record of practicing IP piracy, the STG modification system might be used. On the other hand if the IP is to released to a comparatively more trustable SoC design house where the design specifications are very aggressive, the CDFG modification based approach might be used.

3.2.4 Obfuscation-based Secure SoC Design Flow

The proposed obfuscation based techniques can be utilized to develop a piracy-proof SoC design and manufacturing flow, as described in Chapter 2. The SoC designer receives different obfuscated IPs from the same or different vendors, and then integrates them on the SoC. A special integrated controller module receives patterns from the primary inputs and controls the systematic initialization of the IP modules in the SoC. The system designer integrates several such SoCs on a board, and uses initialization sequences from a ROM to enable all the SoCs. Typically, the latency incurred in the initialization can be easily masked in the latency inherent in a "bootup" or a similar process. Thus, the end user remains oblivious to the embedded security measures in the SoCs. By supporting obfuscated IP cores in the design flow, all the parties (the IP vendor, the SoC designer and the system designer) are benefitted by being protected from piracy.

Next we derive theoretical quantitative metrics to estimate the security offered by the proposed schemes.

3.3 Measure of Obfuscation Level

3.3.1 Manual Attacks by Visual Inspection

To estimate the obfuscation level against a manual mode of attack, we propose a new metric called *semantic obfuscation metric* (M_{sem}), which depicts how many of

the original high level RTL constructs have been replaced by new ones. We define M_{sem} by:

$$M_{sem} = \frac{abs(N_{c,orig} + N_{w,orig} + N_{e,obfus} - N_{raw,obfus})}{max(\{N_{c,orig} + N_{w,orig} + N_{e,obfus}\}, N_{raw,obfus})} \tag{3.1}$$

where $N_{c,orig}$ is the total number of high-level RTL constructs in the original RTL; $N_{e,obfus}$ is the number of extra state elements included in the obfuscated design; $N_{w,orig}$ is the total number of internal `wire` declarations in the original RTL and $N_{raw,obfus}$ is the number of `reg`, `assign` and `wire` declarations in the obfuscated RTL. Note that $0 \leq M_{sem} \leq 1$, with a higher value implying better obfuscation. M_{sem} represents a measure of semantic difference between the obfuscated and the unobfuscated versions of the RTL, by taking into consideration the constructs introduced in the obfuscated code and the constructs removed from the original code. This is the weakest attack, with the adversary having very little chance of figuring out the obfuscation scheme for large RTLs which have undergone a complete change of the "look-and-feel".

3.3.2 Simulation-based Attacks

This attack was described and analyzed in Chapter 2. For a logic simulation based approach where random input vectors are sequentially applied to take the circuit to the *normal mode*, the probability of discovering the initialization key sequence is $\frac{1}{2^{M \cdot N}}$ for a circuit with M primary input ports and a length-N initialization key sequence. For example, in a circuit with $M = 64$ primary inputs and a length $N = 4$ initialization key sequence, this probability is $\sim 10^{-77}$. In practice, most IPs will have larger number of primary inputs and the length N can be made larger, resulting in smaller detection probability. Thus, we can claim that it is extremely challenging to break the scheme using simulation based reverse-engineering.

3.3.3 Structural Analysis based Attack

For the structural analysis based attack, the two proposed obfuscation schemes present different challenges to an adversary. For the STG modification scheme, the

adversary has to analyze the circuit in terms of the Boolean logic structure of the internal nodes, while in the CDFG modification based scheme, the adversary has to analyze the high-level RTL structure of the code. This is the strongest attack, to be acceptable, the proposed obfuscation approaches must provide adequate protection against this attack. We describe the complexity of the two analyses below.

Structural Analysis against STG Modification

Analysis based on structure of the internal nodes is most conveniently done by the construction and manipulation of Reduced Ordered Binary Decision Diagrams (ROBDD) [39] corresponding to the internal circuit nodes. To detect the node modification scheme, the adversary's algorithm must be able to solve several sub-problems in succession. We estimate the computational complexity of each of these sub-problems below to derive an estimate of the computational complexity of the entire problem.

Let the total number of primary outputs of the circuit be P, the total number of state elements in the original circuit be S and the total number of state elements inserted in the modified circuit be T. Then, it is sufficient to analyze the structures of these $(P + S + T)$ nodes between the original and the modified designs, out of which $(P + S)$ are also present in the original design. Suppose, the adversary finds F nodes out of these $(P + S + T)$ nodes to have contrasting logic structures by a ROBDD-based analysis. This dissimilarity is due to two reasons: (a) direct effect of the node modification scheme described in Section 3.2 on some of these nodes, and (b) indirect effect of these modified nodes on other nodes. Either way, from eqn. (2.1 in Chapter 2), the affected nodes would have their values inverted only if simultaneously $en = 1$ and $g = 1$. To isolate the inserted FSM, the adversary must detect this node modification scheme for each dissimilar node.

Finding the correct ROBDD representation of the modified nodes: To detect the effect of a particular en signal originating from the inserted state machine on a modified node, the adversary's algorithm should be able to represent the ROBDD

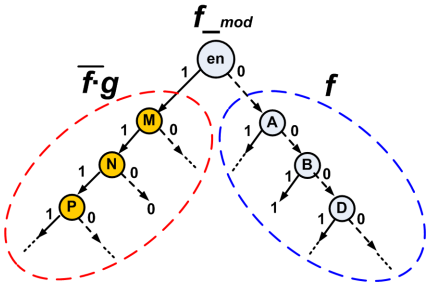


Fig. 3.8. Binary Decision Diagram (BDD) of a modified node.

of the modified node with the *en* signal as the root node, as shown in Fig. 3.8. Improving the variable ordering to minimize the size of a BDD is an *NP-complete* problem [40]. Hence, it follows that the computational complexity to find a particular representation of the ROBDD of the modified function which has *en* as the root node is also *NP-complete* with respect to the number of variables in the Boolean logic expression for the node. Hence, deciphering the modification scheme for a modified node with fanin cone size f_i has a computational complexity $O(2^{f_i})$.

Graph Isomorphism Comparison: After the ROBDD of the modified node has been expressed in the form shown in Fig. 3.8, each sub-graph below the node *en* should be compared with the ROBDD graph for f for isomorphism. Proving graph isomorphism is a problem with computational complexity between P and NP, with the best-known heuristic having complexity $2^{O(\sqrt{n \log n})}$ for a graph with n vertices [41]. Hence, establishing the equivalence for f through graph isomorphism has a computational complexity $2^{O(\sqrt{f_i \log f_i})}$ for a node with fanin cone size f_i. Let $\overline{f_i}$ be the average fanin cone size of the failing verification nodes. Hence, overall this sub-problem has a computational complexity $O(2^{\overline{f_i}} \cdot 2^{O(\sqrt{\overline{f_i} \log \overline{f_i}})})$, which must be solved for each of the F dissimilar nodes.

Compare Point Matching: So far we have assumed that the adversary would be able to associate the dissimilar nodes in the obfuscated design with the corresponding nodes in the original design and would then try to decipher the obfuscation scheme.

This is expected to be relatively easy for the primary output ports of the IP because the IP vendor must maintain a standard interface even for the obfuscated version of the IP, and hence the adversary can take advantage of this name matching. However, the names of the state elements can be changed arbitrarily by the IP vendor and hence, finding the corresponding state elements to compare is a *functional compare point matching* problem [42], which is extremely computationally challenging because of the requirement to search through $(S_N)!$ combinations, where S_N is the number of dissimilar state elements. Hence, we propose the following metric to quantify the level of protection of the proposed STG modification based obfuscation scheme in providing protection against structural analysis:

$$M_{str} = F \cdot 2^{\overline{f_i}} \cdot 2^{O(\sqrt{\overline{f_i} \log \overline{f_i}})} + (S_N)! \tag{3.2}$$

Observations from the Metric: From the above metric, the following can be observed which act as a guide to the IP designer to design a well-obfuscated hardware IP following the STG modification based scheme:

1. Those nodes which have larger fanin cones should be preferably modified because this would increase $\overline{f_i}$ in eqn. (3.2), thus increasing M_{str}.

2. An inserted FSM with larger number of flip-flops increases its obfuscation level because S_N increases. Also, as shown previously in this section, there is an exponential dependance of the probability of breaking the scheme by simulation based reverse-engineering on the length of the initialization key sequence. Hence, it is evident that FSM design and insertion to attain high levels of obfuscation incur greater design overhead. Thus the IP designer must trade-off between design overhead and the level of security achievable through obfuscation.

3. Modification of a larger number of nodes increases F, which in turn increases the level of obfuscation.

Structural Analysis Against CDFG Modification based Approach

The structural analysis of the CDFG modification based obfuscation scheme is estimated by the degree of difficulty faced by an adversary in discovering the *hosted* mode-control FSM and the modification signals. Consider a case where n mode-control FSM state-transition statements have been hosted in a RTL with N blocking/non-blocking assignment statements. However, the adversary does not know a-priori how many registers host the mode-control FSM. Then, the adversary must correctly figure out the hosted FSM state transition statements from one out of $\sum_{k=1}^{n} \binom{N}{k}$ possibilities. Again, each of these choices for a given value of k has $k!$ associated ways to arrange the state transitions (so that the *initialization key sequence* is applied in the correct order). Hence, the adversary must correctly identify one out of $\sum_{k=1}^{n} \left(\binom{N}{k} \cdot k! \right)$ possibilities. The other feature that needs to be deciphered to break the scheme are the mode control signals. Let M be the total number of blocking, non-blocking and dataflow assignments in the RTL, and let m be the size of the modification signal pool. Then, the adversary must correctly choose m signals out of M, which is one out of $\binom{M}{m}$ choices. Combining these two security features, we propose the following metric to estimate the complexity of the structural analysis problem for the CDFG modification based design:

$$M_{str} = \sum_{k=1}^{n} \left(\binom{N}{k} \cdot k! \right) \cdot \binom{M}{m} \tag{3.3}$$

A higher value of M_{str} indicates a greater obfuscation efficiency. As an example, consider a RTL with values $N = 30$, $M = 100$, in which a FSM with parameter $n = 3$ is hosted, and let $m = 20$. Then, $M_{obf} \approx 1.35 \times 10^{25}$. In other words, the probability of the hacker reverse-engineering the complete scheme is about 1 in 10^{25}. In practice, the values of n and M would be much higher in most cases, making M_{str} smaller and thus tougher for the hacker to reverse-engineer the obfuscation scheme.

3.3.4 Quantitative Comparison of the Two Proposed Approaches

The value of M_{str} is expected to be better in the STG modification based approach, because of the exponential dependance of this metric on the average fanin cone size of the modified nodes (eqn. 3.2), compared to the combinatorial dependance on the number of RTL statements for the CDFG modification based approach (eqn. 3.3). The value of M_{sem} is again expected to be superior in the STG modification based approach because the changes in the appearance of the RTL is more drastic in this case, whereas the CDFG modification based approach makes comparatively lesser changes to the high-level RTL constructs, as it is based on intelligent utilization of mostly existing RTL constructs. However, the run-time of the obfuscation algorithm is expected to be higher in the STG modification based approach, because it performs recursive backtracking to construct the logic equations of the internal nodes. All these predicted trends were supported by our simulation results presented in Section 3.4.

3.4 Results

3.4.1 Design Flow Automation

Fig. 3.9 shows the entire STG modification based RTL obfuscation design flow. The design flow starts with the *compilation* of RTL description of the IP core to a unmapped, unoptimized gate-level Verilog netlist. The maximum allowable area overhead is entered as a design constraint, from which the maximum number of modifiable nodes (N_{max}) is estimated. Additionally, the tool has a list of user-mentioned constructs and macros in a forward annotation file. These elements are preserved during the RTL compilation and de-compilation processes by treating them as *don't touch* modules. The N_{max} nodes to be modified are chosen based on the algorithm described in Chapter 2, which ensures maximum perturbation of the design. The modified netlist is re-synthesized and the resultant netlist is then decompiled to a

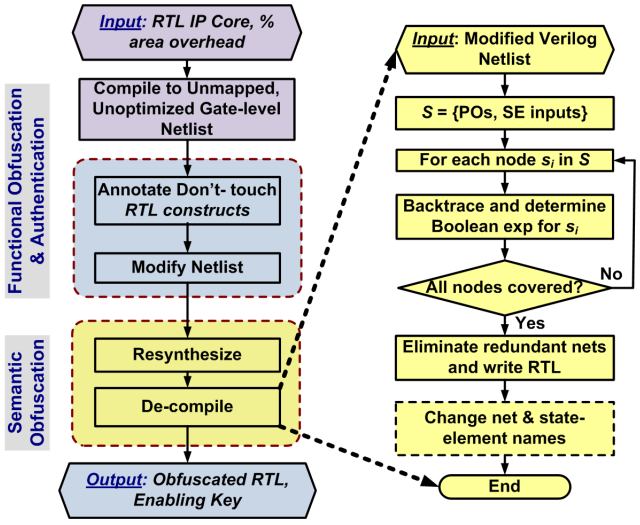


Fig. 3.9. Flow diagram for the proposed STG modification based RTL obfuscation methodology.

RTL code. The names of the internal nodes and instances are changed by a simple string substitution scheme.

Fig. 3.10 shows the steps of the proposed CDFG modification based design obfuscation methodology. The input to the flow is the original RTL, the desired obfuscation level represented by the obfuscation metrics (M_{sem} and M_{str}), and the maximum allowable area overhead. It starts with the design of the mode-control FSM based on the target M_{sem} and M_{str}. The output of this step are the specifications of the FSM which include its state transition graph, the state encoding, the pool of modification signals, and the *initialization key sequence*. Random state encoding and a random *initialization key sequence* are generated to increase the security. Note that in the STG modification based approach, we do not start with explicit target values of the metrics, because these two parameters cannot be predicted a-priori in this technique. However, the target area overhead is an indirect estimate of these parameters, and the de-compilation process automatically ensures a high value of M_{sem}, while an optimal node modification algorithm ensures a high value of M_{str}.

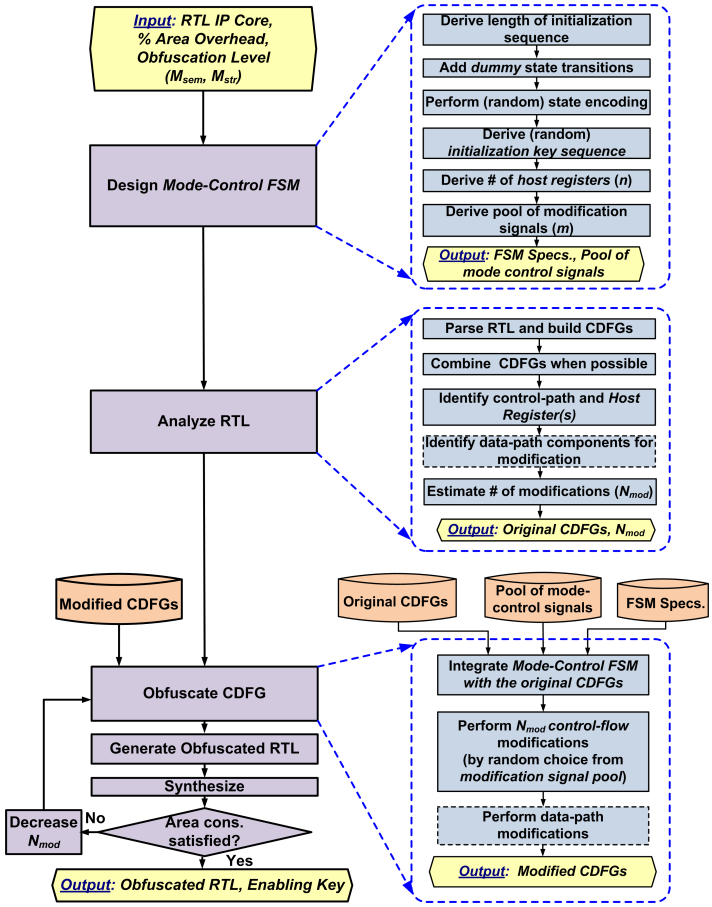


Fig. 3.10. Flow diagram for the proposed CDFG modification based RTL obfuscation methodology.

3.4.2 Simulation Verification

The above design flows were developed using C and the TCL scripting language and was directly integrated in the Synopsys *Design Compiler* environment. The state encoding of the inserted FSM was done using the state encoding tool *STAMINA* [43]. Synthesis was performed using Synopsys *Design Compiler*, using a LEDA 250 nm library. All Formal Verification was performed using Synopsys *Formality*. All work was performed on a Linux workstation with 2GB of main memory and a dual-core 1.5GHz processor

Table 3.2
Functional and Semantic Obfuscation Efficiency and Overheads for IP Cores for 10% area overhead target (CDFG Modification based Results at iso-delay)

STG Modification based Approach									
IP	**Sub-**	**Nodes**	**Obfuscation Efficiency**				**Design Overhead**		
Cores	**Modules**	**Modified (%)**	**Failing Verif. Nodes (%)**	M_{sem}	$\log_{10} M_{str}$	**Area (%)**	**Delay (%)**	**Power (%)**	**Run time (s)**
DES	key_sel	0.93	100.00	0.98	51.56	6.65	1.39	4.85	32
	crp	0.83	100.00	0.91	55.87	5.54	0.66	5.43	37
AES	Key Expand	0.95	90.30	0.92	43.17	5.29	0.00	4.56	28
	Sbox	0.95	100.00	0.96	45.78	4.95	2.42	5.31	29
	Inverse Sbox	0.97	85.25	0.94	46.22	5.51	2.60	5.62	35
FDCT	DCT	0.90	88.95	0.96	60.19	4.64	1.00	5.06	27
	ZIGZAG	0.95	100.00	0.92	52.12	5.74	0.88	5.67	30
CDFG Modification based Approach									
IP	**Sub-**	**# of**	**Obfuscation Efficiency**				**Design Overhead**		
Cores	**Modules**	**Modifications**	**Failing Verif. Nodes (%)**	M_{sem}	$\log_{10} M_{str}$	**Area (%)**	**Delay (%)**	**Power (%)**	**Run time (s)**
FPU	post_norm	20	98.16	0.69	36.81	8.22	0.00	9.14	27
	pre_norm	20	94.16	0.70	32.91	9.39	0.00	9.79	25
	pre_norm_fmul	20	90.00	0.77	23.13	8.30	0.00	9.69	20
	except	10	100.00	0.73	23.16	7.56	0.00	8.73	14
TCPU	control_wopc	20	92.79	0.75	42.12	8.74	0.00	8.97	29
	mem	10	97.62	0.71	19.69	8.29	0.00	9.76	15
	alu	10	97.62	0.81	15.01	9.59	0.00	9.88	15

A FSM with a length-4 initialization key sequence was designed for mode-control in both the schemes. In the STG modification based scheme, we chose high fan-in internal nodes as MKFs, as described in Chapter 2. The STG modification scheme was applied on three open-source Verilog IP cores, viz. "Data Encryption Standard" (DES), "Advanced Encryption Standard" (AES) and "Discrete Cosine Transform" (FDCT). The CDFG modification based obfuscation technique was applied for two Verilog IP cores - a single precision IEEE-754 compliant floating-point unit ("FPU"), and a 12-bit RISC CPU ("TCPU"), both collected from the public domain IP core collection at `http://www.opencores.org`. For the STG modification based approach, each gate-level modified design was verified using *Formality* with the corresponding de-compiled RTL to verify the correctness of the procedure.

Table 3.3
Overall Design Overheads for Obfuscated IP Cores (STG modification based results at iso-delay)

STG Modification based Approach			
IP Core	**Area (%)**	**Delay (%)**	**Power (%)**
DES	6.09	1.10	5.05
AES	5.25	2.00	5.45
FDCT	5.22	0.95	5.55
Average	5.52	1.35	5.35
CDFG Modification based Approach			
FDCT	8.48	9.36	0.00
TCPU	8.54	9.73	0.00
Average	8.51	9.55	0.00

Obfuscation Efficiency and Overheads

Table 3.2 shows the structural and semantic obfuscation metrics and design overheads for 10% target area overhead constraint for each module. Note that the value of the M_{str} metric is in a logarithmic scale. The *potency* and *resilience* of the schemes were very high, with most of the designs considered achieving close to 100% formal verification failure and having a very high value of the computational complexity metric M_{str}. However, the value of M_{str} is orders of magnitude higher for the STG modification based approach, as predicted in Section 3.3. The value of M_{sem} was very close to the ideal value of 1.0 for the STG modification based approach; however, it is closer to 0.75 on average for the CDFG modification based approach. Again, this is an expected trend as predicted in Section 3.3.

For all the individual circuit modules, the observed area overhead was less than 10%, the power and delay overheads were within acceptable limits (target delay overhead was set at 0% for the CDFG modification based scheme). The maximum runtimes of the obfuscation programs for the individual modules was 29 seconds for

the CDFG modification based approach, and 37 seconds for the STG modification based approach. Table 3.3 shows the overall design overheads after re-synthesis of the multi-module IP cores from the obfuscated RTL, which are again all within acceptable limits.

3.4.3 Effect of Key Length

The security offered by the two proposed approaches increases with the increase in the key length. Next, we investigate the following aspects of the proposed techniques (a) design and performance overheads to support multiple-length keys, and (b) effect of increasing key lengths on design and performance overheads. We compare the proposed techniques with hardware implementations the AES encryption/decryption algorithm, the global standard for secure information exchange.

Support for Multiple-length Keys

Most commercially available IP cores for AES can be operated in three different modes with three different input key lengths - 128 bit, 192 bit and 256 bit [44, 45]. This flexibility allows the SoC designers to trade-off between the available security (which increases with increase of key length) and performance (which decreases with increase of key length). Usually, a "*key_length*" input control signal determines the input key length. The same feature can be implemented in our proposed techniques, where the length of the initialization key sequence can be varied. Fig. 3.11 shows such a system which supports initialization key sequences of length 3, 4 or 5. However, in case of commercially available AES cores, this flexibility comes at a price - the multi-key IP core versions usually have greater area than the baseline designs supporting only a single key length [45].

Table 3.4 shows the area overhead effect of supporting multiple keys lengths on the proposed schemes for the IP modules presented in Table 3.3, compared to two versions of a commercially available AES core [45]. The key lengths for our proposed schemes

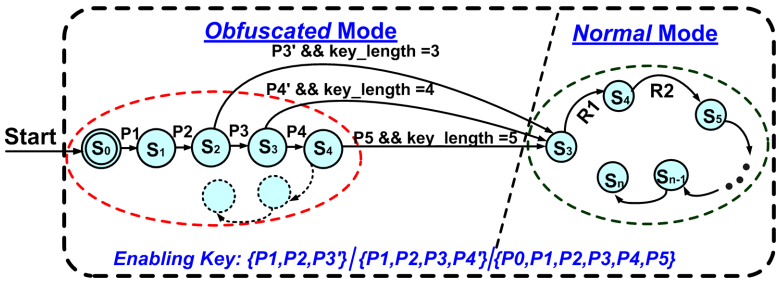


Fig. 3.11. Scheme with *initialization key sequences* of varying length (3, 4 or 5).

Table 3.4
Area Overhead for Multi-length Key Support

Scheme	Area Overhead (%)
STG Modification	2.26 (av.)
CDFG Modification	3.16 (av.)
Ref. [45] (AES 48-cycle core)	17.88
Ref. [45] (AES 96-cycle core)	24.59

were 4 (baseline), 6 (1.5X) and 8 (2X), while those for the AES implementations were 128 (baseline), 192 (1.5X) and 256 (2X). From this table, it is clearly evident that the proposed approaches are more scalable than the AES hardware implementations with respect to the increase in key length.

Effect of Increasing Key Length

For symmetric key cryptographic algorithms such as AES, in general the security increases with the length of the key, as the complexity of breaking the encryption is an exponential function of the key length. However, an increasing key length usually results in lower throughput. Similar trends are expected for the two proposed obfuscation schemes with respect to the length of the initialization key sequence. We

Table 3.5
Comparison of Throughput Scalability with Increasing Key Length

Scheme	**Decrease in Throughput (%)**	
	1.5X key length	**2X key length**
STG Modification	2.70 (av.)	3.96 (av.)
CDFG Modification	3.07 (av.)	4.72 (av.)
Ref. [45] (AES 48-cycle core)	14.10	24.83
Ref. [45] (AES 96-cycle core)	14.04	24.56

investigated the scalability of the two proposed techniques with respect to the increase in the length of the initialization key sequence for the proposed approaches vis-a-vis that for commercially available hardware implementations of AES with respect to the key length. Again, for the proposed obfuscation schemes we considered a a key length of 4 to the baseline case, while a 128 bit key for AES was considered baseline. Table 3.5 shows the decrease in throughput with the increase of key length. Once again, the simulation results showed that the proposed schemes had superior scalability of throughput than the hardware implementations of AES when the key length is increased.

3.5 Discussions

In this section, we describe a technique to decrease the hardware overhead by utilizing the normally "unused states" (states which are not reachable during normal operations). We also show how the proposed obfuscation techniques can provide protection against *hardware Trojans*.

Table 3.6
Design Overhead for ISCAS-89 Benchmarks Utilizing Unused States

Circuit	**# of Gates**	**% Ar. Ov.1**	**% Ar. Ov.2**	**%Po. Ov.1**	**%Po.Ov.2**
s1196	370	18.44	24.84	2.45	6.00
s1238	373	16.31	27.15	4.63	10.71
s1423	505	6.11	6.08	2.015	3.07
s1488	431	12.81	16.93	7.24	12.76
s5378	1102	14.45	18.69	8.76	25.78
s9234	5807	6.53	9.04	9.28	13.15
s13207	2488	7.93	8.17	7.54	16.28
s15850	2983	7.67	8.99	5.63	6.15
s35932	7966	1.34	1.955	6.93	9.49
s38417	8822	0.09	0.56	1.81	5.27
s38584	9019	0.85	3.03	5.10	10.77

3.5.1 Using Unreachable States during Initialization

Referring to Fig. 2.3, the states in the *initialization FSM* and the *obfuscation FSM* can also be encoded by using states which are unreachable during normal operations. By doing this, it is automatically ensured that the circuit would not operate in the correct mode prior to its initialization. This can help to eliminate the need to introduce a separate FSM to control the mode of operation, potentially decreasing the hardware overhead. We present simulation results for a method of obfuscation based on finding unused states for a suite of gate-level sequential circuits, and two open-source RTL IP cores.

Table 3.6 shows the design overhead in ISCAS-89 circuits following the de-compilation based methodology where normally unused states of the circuit are used to encode the states in the obfuscated mode. The unused states were found using *sequential justification* by *Synopsys Tetramax*. The results were taken without integrating any

Table 3.7
Design Overhead for IP Cores Utilizing Unused States

IP	Module	% Ar. Ov.1	% Ar. Ov.2	%Po. Ov.1	%Po.Ov.2
AES	sbox	5.95	7.46	15.19	17.67
	inv_sbox	6.99	8.25	8.98	15.27
	key_expand	3.93	5.47	13.80	15.93
	Overall	4.57	6.07	13.03	16.14
DES	key_sel	4.81	8.91	2.66	5.35
	crp	4.75	6.77	1.75	4.79
	Overall	4.77	7.69	2.03	4.97

mode-control FSM, considering 5-6 randomly chosen state-elements in the original circuit, having an initialization state space with 4 states, and (for result set "1") an authentication state space consisting of 4 states. State encoding with 6 state elements were required in cases where sufficient unused states are not available from 5 state elements. All results are at iso-delay, i.e. zero timing overhead. Table 3.7 shows the corresponding figures for two open-source IP cores.

3.5.2 Application of Obfuscation to Memory

As pointed out Section 3.1, protection of the memory module can serve two purposes: (a) protection of the memory interface hardware from IP piracy and Trojan infection, and (b) protection of the memory contents. Serving the second purpose is particularly important for embedded applications, where the program and the data are at the risk of unauthorized access and/or malicious alteration in unprotected memory, as detailed in Chapter 7. Obfuscation can be applied to simultaneously achieve the above two goals, as illustrated in Fig. 3.12. The memory control unit (including the memory address decoder) can be obfuscated using the proposed procedure. This protects the memory control unit from IP piracy and Trojan infection. In

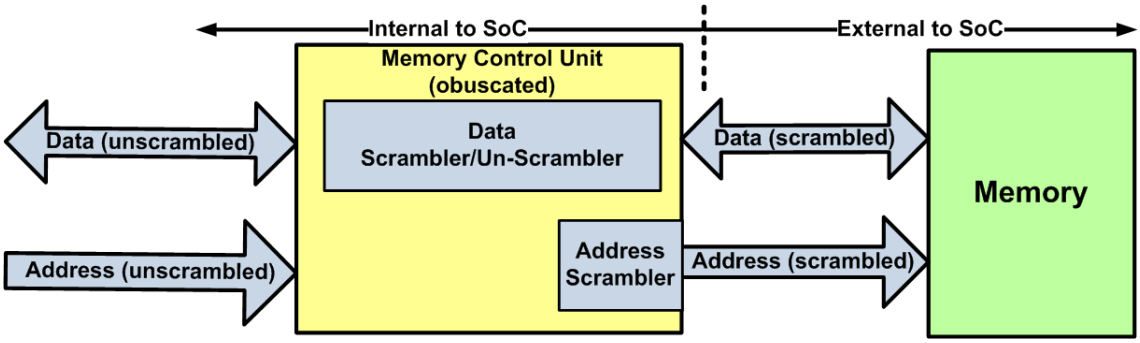


Fig. 3.12. Memory protection using hardware obfuscation and scrambling.

addition, data and address *scrambling* circuits are included in the obfuscated memory control unit. The *address scrambling circuit* re-maps any address generated by the SoC (or an embedded processor) to a different memory location, whereas the *data scrambling circuit* scrambles any data to be written in memory. A provision for data de-scrambling is also made to de-scramble any data read from memory before sending it to the processing units.

These scrambling/de-scrambling circuits are controlled by the outputs of the mode control FSM integrated in the memory control unit and they are operational in both the *normal* and the *obfuscated* modes, such that in the *obfuscated mode*, corrupted data is stored in or read from incorrect memory locations. For example, in the *obfuscated mode*, the address scrambler might map all addresses ending with four to a single memory location, thereby corrupting the memory content with high probability. It is important for the scrambling circuitry to have low hardware overhead and be structurally unrecognizable. Different low-overhead scrambling circuitry has been proposed for memory address and data obfuscation, including constant address shifters using adders, random address shifters using the output of controlled *linear feedback shift registers* (LFSRs), and controlled bit permuters which re-arrange the bits of the address or data [46]. Because adders and LFSRs have recognizable regularity in their structures (both at the gate-level and in RTL) which might be preserved

even after obfuscation, hence, controlled bit permuters composed of irregular combinational logic can be the most suitable choice among the approaches described here. The data de-scrambler should perform the reverse permutation to bring back the scrambled data to its original form. The behavior of the bit permutation circuit can be made arbitrarily complex, e.g. a scrambler that simultaneously permutes and shifts the address/data, thereby potentially increasing the difficulty faced by an adversary. However, because the data scrambler design requires a corresponding de-scrambler, the total hardware overhead of the scrambler and the de-scrambler should be taken into consideration before deciding on the scrambling algorithm.

In embedded processors with a simple load/store architecture, it can happen that there is not much logic circuitry between the register file holding or receiving the data and the memory (cache or external). In these scenarios, an inserted bit scrambling network between the data transfer path from the register file to the external memory can become extremely recognizable. Once it is recognized, an adversary in the fabrication facility can by-pass it or neutralize its effect. In such cases, a more secure option is to have the scrambling network at the input of the register file, and the processing unit has to be modified to have the ability to process scrambled data. The corresponding de-scrambling network in this case has to be at the I/O interface to the peripherals.

3.6 Summary

In this chapter, we have extended the role of key-based obfuscation in IP protection by describing two techniques applicable to RTL descriptions of IPs. The two different techniques provide the SoC designers with the to trade-off between attainable levels of security and complexity of the obfuscation process. The first technique operates by synthesizing the RTL to a gate-level netlist, modifying it and then de-compiling it to re-generate an obfuscated version of the RTL. The second technique operates on the RTL directly and makes the modification at judiciously chosen locations to

effectively hide them. We integrate a small FSM of special structure into the design to modify the STG (in the first technique) or the CDFG (in the second technique) of the circuit. The scheme includes additional hard-to-detect authentication features at low design overhead to increase the level of security. The proposed obfuscation techniques provide active defence mechanisms that can prevent IP infringement at different stages of SoC design and fabrication flow (similar to the scheme described in Chapter 2), while incurring low design and computational overhead and no impact on end user experience. The proposed technique is easily scalable in terms of increasing level of security using longer key sequence. They can also be applied for protection of memory content and memory interface hardware. Further improvement in hardware overhead is attainable by utilizing the normally unused states of the circuit.

The next chapter will examine the applicability of obfuscation in providing protection against hardware Trojans. Note that the security features based on obfuscation propagates through the down-stream stages of IC life-cycle and provides protection against hardware Trojans. Design of a stealthy and effective hardware Trojan requires detailed analysis of the circuit structure and functionality. However, if the circuit functionality is obfuscated by an initialization key based scheme, then this task can become significantly difficult for an adversary. This effect would be analyzed in detail in Chapter 4.

4. OBFUSCATION BASED PROTECTION AGAINST HARDWARE TROJAN

4.1 Introduction

As mentioned earlier, the issue of *Trust in Integrated Circuits* has become prominent recently due to widespread outsourcing of the IC manufacturing processes to offshore locations in order to reduce cost [10, 47–49]. A design can be tampered in an untrusted fabrication facility by the insertion of malicious circuitry that triggers a malfunction conditionally. Such malicious circuitry, referred to as a *hardware Trojan*, can activate during field operation condition and affect normal circuit operation, potentially with catastrophic consequences in critical application areas and public infrastructure. Such malicious circuitry can also be inserted by CAD automation tools obtained from untrusted third party vendors [50]. Due to the stealthy nature of hardware Trojan and inordinately large number of possible Trojan instances an adversary can exploit, detection of hardware Trojan by post-manufacturing test has emerged as an extremely challenging problem [10].

In this chapter [51], a novel application of design obfuscation is proposed to achieve security against hardware Trojan. The technique is motivated by the key-based state transition modification methodology for secure SoC design flow, as described in the last two chapters. It is shown that obfuscation can be extremely effective in protecting designs from malicious modifications. Furthermore, it can facilitate detection of inserted Trojan since it prevents an intelligent adversary from exploiting the true rare events in a circuit to design hard-to-detect Trojans. As mentioned in Chapter 2, the obfuscation scheme enables circuit operation in two distinct modes - (a) the *obfuscated mode*, when circuit functionality is different from normal one, and (b) the *normal mode*, when its behavior is identical to its non-obfuscated version. The mode

control is performed by the application of a specific sequence of input vectors on initialization, called an *initialization key*. Without the initialization key, an adversary fails to comprehend the intended functional behavior of the circuit. It can be argued that without proper knowledge of the circuit functionality, a Trojan inserted by an adversary will have high probability of either becoming functionally *benign*, or easily *detectable* by conventional logic testing. This technique is analogous to *software obfuscation* techniques which help in protecting against malicious modifications by hiding the functional behavior of a program, as discussed in Chapter 7. In particular, this work makes the following contributions:

- It analyzes the effectiveness of a key-based gate-level design obfuscation scheme in achieving security against hardware Trojans. It provides mathematical analysis to derive the impact of different design and obfuscation parameters on the degree of hardware protection achieved against Trojans. The security features propagate through the IC design flow to lower levels of design abstraction (such as GDS-II). The methodology ensures that no structural signature is introduced while obfuscating the functionality of the circuit by modifying the *state transition graph* (STG), while the *normal mode* circuit functionality is kept unaltered.

- It proposes two important modifications of the obfuscation process that helps in achieving high security against hardware Trojan at low design overhead. In particular, it proposes: (1) addition of extra state elements to "blow up" the state space exponentially and then use large number of these states in the *obfuscated mode* of operation, and (2) use of *functionally unreachable* states of the original state machine in the *obfuscated mode*. It develops an automated design flow to incorporate the above modifications while incurring low design overhead. It also proposes an integrated flow for evaluating the effectiveness of the proposed approach for complex gate-level netlists.

- Malicious CAD tools and automation scripts in automated design flows are potential sources of Trojan insertion [6, 50]. This threat is relevant at all stages of

the design flow, and is growing with increasing use of third-party CAD tools in IC design. We have shown how the proposed obfuscation based design methodology can provide protection against such malicious CAD tools in both the SoC and FPGA design flows. We also discuss how the proposed methodology can be extended to protect against different types of Trojans such as *information leakage* Trojans , which try to leak secret information from within an IC [52]. .

The rest of the chapter is organized as follows. In Section 4.2, we provide background on hardware Trojans and obfuscation for hardware security. In Section 4.3, we present mathematical analysis to show how an obfuscation-based IP protection technique can address the security threat posed by hardware Trojans. In Section 4.4, an automated design flow that implements the proposed methodology is described. Simulation results to validate the concept is presented in Section 4.5. In Section 4.6, a technique to decrease the design overhead and the effectiveness of this scheme in proving protection against malicious CAD tools and *information leaking* Trojans is discussed. Conclusions are summarized in Section 4.7.

4.2 Background

4.2.1 Hardware Trojan: Models and Examples

A hardware Trojan instance can typically be associated with two sets of internal nodes: the nodes which trigger malfunction by activating the Trojan (called the *trigger nodes*), and the nodes which are affected by the Trojan (called the *payload nodes*) [53]. Fig. 4.1(a) [10] shows a *combinational Trojan* where the payload node S has been modified to the node $S^{\star}$, and malfunction is triggered whenever the condition $a = 0, b = 1, c = 1$ is satisfied at the corresponding *trigger nodes*. The *sequential Trojan* shown in Fig. 4.1 (b), on the other hand, is a 3-bit counter which causes a malfunction at the payload node S on reaching a particular count, which is incremented each time the condition $a = 1, b = 0$ is satisfied.

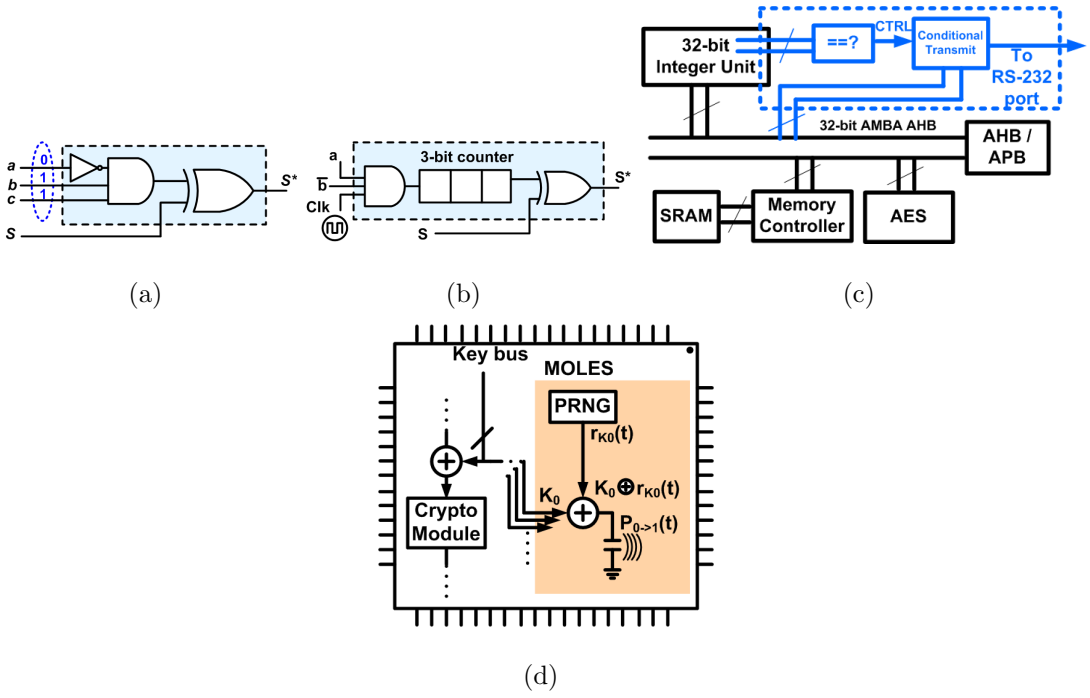


Fig. 4.1. Examples of (a) combinational and (b) sequential hardware Trojans that cause malfunction conditionally; Examples of Trojans leaking information (c) through logic values and (d) through side-channel parameter [52].

A Trojan can also exhibit its malicious effect by leaking information through covert communication channels [54]. An example Trojan that leaks logic information conditionally is shown in Fig. 4.1(c). The system is design for a cryptographic application with an integer unit and a cryto-core, which performs Advanced Encryption Standard (AES) based cipher operations. The Trojan consists of a comparator and a conditional data transmitter. The comparator compares the values at a few internal nodes of the integer unit with a constant value, and the transmitter sends out the contents of the data bus (which can be the key for the AES encryption) through the RS-232 port if the comparison succeeds. Another example is shown in Fig. 4.1(d) [52], where a bank of inserted capacitors is charged depending on the result of XOR-ing the output of a pseudo-random number generator (PRNG) with the contents of a

data bus connected to an AES module. Provided the adversary has a clock that is synchronized with the PRNG clock, it is possible to extract the AES key from the *side-channel information* consisting of supply current traces of the IC.

Note that an intelligent adversary is likely to choose rare conditions at internal circuit nodes as Trojan triggering condition [6, 53], which are unlikely to arise during test but can occur during long hours of field operation. Moreover, the inserted Trojan is likely to be designed such that its malicious effect at the payload is difficult to observe at the output ports. A Trojan designed with rare activation condition and/or rare observability of payloads would evade post-manufacturing test with high probability. It is extremely challenging to detect Trojan insertion using conventional test generation and application techniques due to the stealthy nature of the Trojan circuits as well as the inordinately large Trojan population space. The number of possible Trojan instances in a given circuit has a combinatorial dependence on the number of circuit nodes. For example, with an assumption of maximum four trigger nodes, a small ISCAS-85 circuit *c880* with 451 gates can have $\sim 10^9$ possible trigger conditions and $\sim 10^{11}$ single payload Trojan instances, respectively. Hence, it is not computationally feasible to enumerate all possible Trojan instances in a given circuit and generate deterministic test patterns for them.

Many existing approaches of Trojan detection rely on the measurement of side-channel parameters such as delay and power signature [47, 55–57]. However, these techniques can be extremely susceptible to measurement and process-variation induced noise. Moreover, they suffer from reduced detection sensitivity in detecting ultra-small Trojans consisting of few logic gates [47]. Design techniques have also been proposed that help to detect inserted Trojans [58, 59]; however, they often result in unacceptable design overhead. These short-comings of the existing approaches of Trojan detection constitute the main motivation behind the present work.

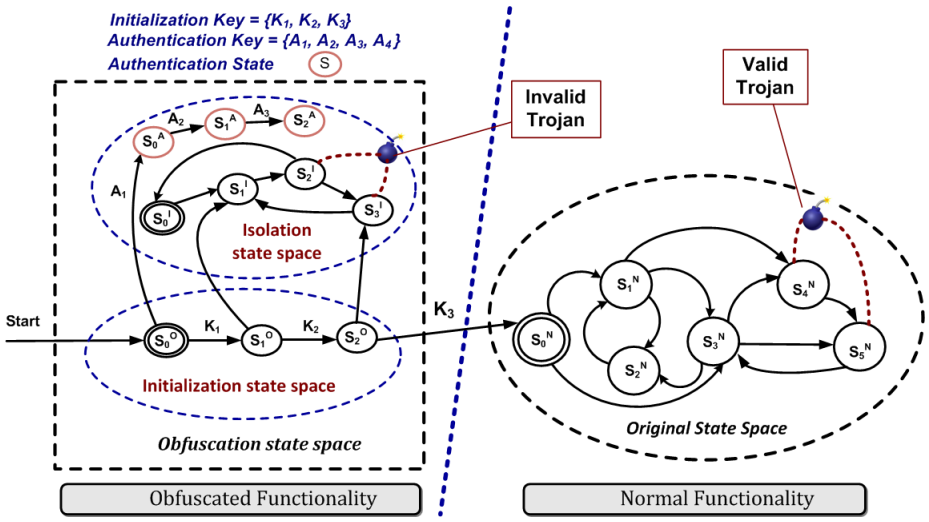
In the simulation results, the class of the Trojans that can be functionally represented by the examples shown in Figs. 4.1(a) and 4.1(b) have been considered. However, as shown in Section 4.6 that the proposed design methodology is also effec-

tive in protecting ICs against *information leakage* Trojans which are triggered by a specific set of digital values at selected circuit nodes (similar to the types shown in Fig. 4.1(c) and 4.1(d)), regardless of whether the information leaked by them is logic information or side-channel information.

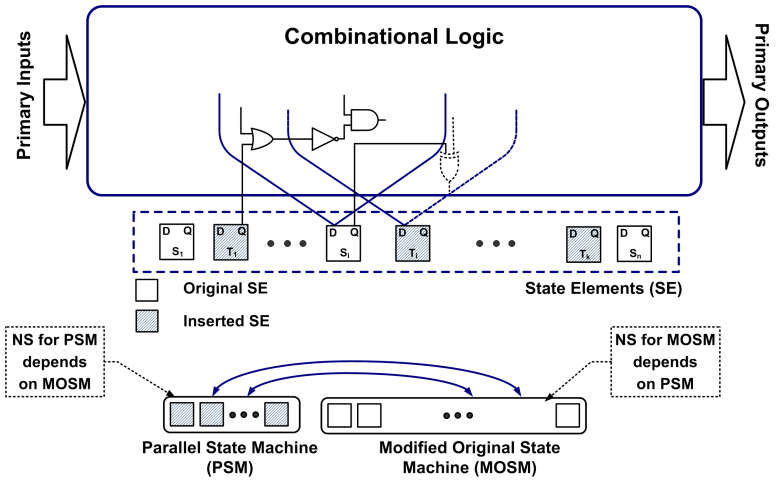
4.2.2 Obfuscation for Hardware IP Protection

The approaches for obfuscation based hardware IP protection can be divided into two main classes - (a) those that affect the comprehensibility of the description of IP core (usually in a hardware description language such as Verilog or VHDL), but keep the functionality unchanged [20, 60], and (b) those that obfuscate the functionality of the IP core [7]. In [20, 60] software tools have been described that can affect the human comprehensibility of a RTL by variable renaming, removal of comments, loop unrolling, etc. However, they do not address obfuscating low-level design descriptions (gate level or GDS-II) and hence cannot be used for preventing Trojan attack in foundry.

Functional obfuscation based approaches such as those proposed in [7], on the other hand, obfuscate the way the circuit operates and normal operation is enabled only after the application of a correct *initialization key sequence*. Such approaches work essentially by changing the state transition function of a design to define two distinct modes of operation: the *obfuscated mode* and the *normal mode*. Such a functional obfuscation approach is more promising with respect to security against Trojan because it prevents the adversary from fully understanding the circuit operations, thereby making an intelligent Trojan insertion extremely difficult. The design modifications become indistinguishable parts of the circuit structure, which makes it infeasible to reverse-engineer the circuit and isolate the security features. Note that key-based hardware protection techniques have earlier been investigated to prevent illegal manufacturing and circulation of ICs [8]. However, they do not address



(a) Modified state transition graph



(b) Modified circuit structure

Fig. 4.2. The proposed obfuscation scheme for protection against hardware Trojans.

obfuscating the functionality of the circuit or achieving protection against hardware Trojan.

4.3 Methodology

In this section, the design methodology based on obfuscation to protect designs against the threat of hardware Trojans is described. The obfuscation in the proposed scheme is achieved by two important modifications of the *state transition graph* (STG) of the circuit:

- The size of the reachable state-space is "blown up" by a large (exponential) factor using extra state elements.
- Certain states, which were *unreachable* in the original design are used and made reachable only in the *obfuscated mode* of operation.

These two modifications make it difficult for an adversary to design a functionally potent and well-hidden Trojan, as shown through the analysis presented in Sections (4.3.1)-(4.3.3). Fig. 4.2(a) shows the proposed obfuscation scheme based on the change in the STG of the circuit. On power-up, the circuit is initialized to a state (S_0^O) in the *obfuscated mode*. On the application of an input sequence $K_1 \rightarrow K_2 \rightarrow K_3$ in order, i.e. the *initialization key sequence*, the circuit reaches the state S_0^N, which is the reset state in the *normal state space*, allowing *normal mode* of operation. The states S_0^O, S_1^O and S_2^O constitute the *initialization state space*. The application of even a single incorrect input vector during the initialization process takes the circuit to states in the *isolation state space*, a *near-closed set* of states from which it is not possible to come back to the *initialization state space* or enter the *original state space*. The *initialization state space* and the *isolation state space* together constitute the *obfuscation state space*. All state encodings in the *obfuscation state space* are done using *unreachable* state bit combinations for selected state elements of the circuit. This ensures that the circuit cannot perform its normal functionality until the correct initialization key sequence has been applied. The initialization latency (typically < 10 clock cycles) can be easily hidden from the end-user by utilizing the inherent latency of most ASICs during a "boot-up" or similar procedure on power-ON [7].

To "blow up" the size of the *obfuscation state space*, a number of extra state elements are added depending on the allowable hardware overhead. The size of the *obfuscation state space* has an exponential dependence on the number of extra state elements. An inserted parallel finite state machine (PSM) defines the state transitions of the extra state elements. However, to hide possible structural signature formed by the inserted PSM, the state transitions in both modified state machine in the original circuit (MOSM) and the PSM depend on each other, as shown in Fig. 4.2(b). Next, description of the PSM is *folded* into the MOSM to generate an integrated state machine using a logic re-synthesis step. It requires considering the design constraints in terms of delay, area or power during flattening of the state machines and logic optimization. In effect, the circuit structures such as the input logic cones of the original state elements change significantly compared to the unobfuscated circuit, making reverse-engineering of the obfuscated design practically infeasible for a hacker. This effect is illustrated in Section 4.3.4 through an example benchmark circuit.

To increase the level of structural difference between the obfuscated and the original circuits, the designer can choose to insert *modification cells* as proposed in [7] at selected internal nodes. Furthermore, the level of obfuscation can be increased by using more states in the obfuscated state space. This can be achieved by: 1) adding more state elements to the design and/or 2) using more unreachable states from the original design. However, this can increase the design overhead substantially. In Section 4.6, a technique is described to reduce the design overhead in such cases.

Selected states in the *isolation state space* can also serve the purpose of authenticating the ownership of the design, as described in [7]. Authentication is usually performed by embedding a *digital watermark* in the design. A *digital watermark* is a unique characteristic of the design which is usually not part of the original specification and is known only to the designer. Fig. 4.2 shows such a scheme where the states S_0^A, S_1^A and S_2^A in the *isolation state space* and the corresponding output values of the circuit are used for the purposes of authenticating the design. The design goes through the state transition sequence $S_0^O \rightarrow S_0^A \rightarrow S_1^A \rightarrow S_2^A$ on the application of the

sequence $A_1 \rightarrow A_2 \rightarrow A_3$. Because these states are unreachable in the normal mode of operation, they and the corresponding circuit output values constitute a property that was not part of the original design. As shown in [7], the probability of determining such an embedded watermark and masking it is extremely small, thus establishing it as a robust watermarking scheme.

4.3.1 Effect of obfuscation on Trojan Insertion

As mentioned before, to design a functionally catastrophic but hard-to-detect Trojan, the hacker would try to select a "rare" event at selected internal "trigger nodes" to activate the Trojan. To select a sufficiently rare trigger condition for the Trojan to be inserted, the hacker would try to estimate the signal probability [61] at the circuit nodes by simulations. To do so with a certain degree of confidence, a minimum number of simulations with random starting states and random input vectors must be performed [62]. However, the hacker has no way to know whether the starting state of the simulations is in the *normal state space* or the *obfuscation state space*. If the initial state of the simulations lie in the *obfuscation state space*, there is a high probability that the simulations would remain confined in the *obfuscation state space*. This is because the random test generation algorithm of the hacker most likely would be unable to apply the correct input vector at the correct state to cause the state transition to the *normal state space*. Essentially, the STG of the obfuscated circuit has two *near-closed* (NC) set of states [63], which would make accurate estimation of the signal probabilities through a series of random simulations extremely challenging. An algorithm was proposed in [63] to detect the *NC sets* of a sequential circuit; however, the algorithm requires: (a) knowledge of the state transition matrix of the entire sequential circuit, which is not available to the hacker, and (b) a list of all the *reachable states* of the circuit, which is extremely computationally challenging to enumerate for a reasonably large sequential circuit. Hence, it can be assumed that

the hacker would be compelled to resort to a random simulation based method to estimate the signal probabilities at internal circuit nodes.

4.3.2 Effect of obfuscation on Trojan potency

To decrease the *potency* of the inserted Trojan, the designer of the obfuscated circuit must ensure that if the hacker starts simulating the circuit in the *obfuscation state space*, the probability of the circuit being driven to the *normal state space* is minimal. Consider a sequential circuit originally with N state elements and M *used states*, to which n state-elements are added to modify the STG to the form shown in Fig. 4.2(a). Let the number of states in the *obfuscation state space* be $S_i = f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right)$, with $f_1 < 1$ is a *utilization factor* reflecting the overhead constraint.

Let I denote the set of states in the *obfuscation state space*, U denote the set of states in the *normal state space*, and T denote the set of states actually attained during the simulations by the hacker. Let, p be the number of primary inputs (other than the clock and reset) where the initialization *key* sequence is applied, and let the length of the initialization key sequence be k. Then, it takes k correct input vectors in sequence to reach the *normal state space* from state S_0^O, $k-1$ correct input vectors from state S_1^O, and so on. Then, the probability that the simulation started in the *initialization state space* and was able to reach the *normal state space* by the application of random input vectors:

$$P\left(T \subseteq \left\{I \bigcup U\right\}\right) = \frac{k}{S_i + M} \cdot \left(\frac{1}{2^p} + \frac{1}{2^{2p}} + \cdots \frac{1}{2^{pk}}\right) \tag{4.1}$$

$$\approx \frac{k \cdot 2^{-p}}{\left(f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right) + M\right)\left(1 - 2^{-p}\right)} \tag{4.2}$$

assuming $2^{-pk} \ll 1$. Similarly, the probability that the simulations started in the *initialization state space* or the *isolation state space* and remained confined there:

$$P\left(T \subseteq \left\{I \bigcup U'\right\}\right) = \left[1 - \frac{k \cdot 2^{-p}}{\left(f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right) + M\right)\left(1 - 2^{-p}\right)}\right] \cdot \frac{f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right)}{f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right) + M} \tag{4.3}$$

where $U^{'}$ denotes the complement set of U. Again, the probability that the simulations started in the *normal state space*, and remained confined there is:

$$P\left(T \subseteq U\right) = \frac{M}{f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right) + M} \tag{4.4}$$

To maximize the probability of keeping the simulations confined in the *obfuscation state space*, the designer should ensure:

$$P\left(T \subseteq \left\{I \bigcup U^{'}\right\}\right) \gg P\left(T \subseteq U\right) + P\left(T \subseteq \left\{I \bigcup U\right\}\right) \tag{4.5}$$

Approximating $M \gg \frac{k \cdot 2^{-p}}{1-2^{-p}}$, and simplifying, this leads to:

$$f_1 \cdot 2^n \cdot \left(1 + 2^N - M\right) \gg M \tag{4.6}$$

This equation essentially implies the size of the *obfuscation state space* should be much larger compared to the size of the *normal state space*, a result that is intuitively expected. Two main observations are:

- The size of the *obfuscation state space* has an exponential dependence of the number of extra state elements added.
- In a circuit where the size of the used state space is small compared to the size of the unused state space, higher levels of obfuscation can be achieved at lower hardware overhead.

As an example, consider the ISCAS-89 benchmark circuit s1423 with 74 state elements (i.e. $N = 74$), and $> 2^{72}$ unused states (i.e., $2^N - M > 2^{72}$) [34]. Then, $M < 1.42 \times 10^{22}$, and considering 10 extra state elements added (i.e. $n = 10$), $f_1 > 0.0029$ for eqn. (4.6) to hold. Thus, expanding the state space in the modified circuit by about 3% of the available unused state space is sufficient in this case.

4.3.3 Effect of obfuscation on Trojan detectability

Consider a Trojan designed and inserted by the hacker with q trigger nodes, with *estimated* rare signal probabilities $p_1, p_2, \ldots p_q$, obtained by simulating the obfuscated

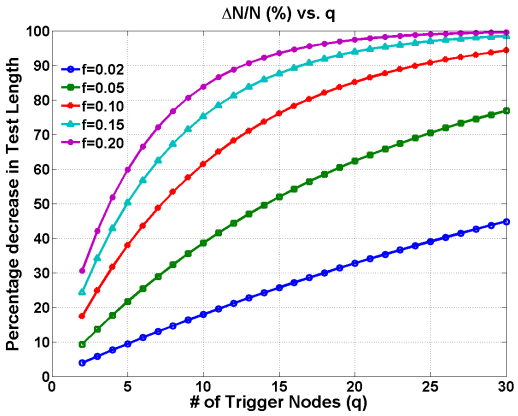


Fig. 4.3. Fractional change in average number of test vectors required to trigger a Trojan, for different values of average fractional mis-estimation of signal probability (f) and Trojan trigger nodes (q).

circuit. Then, assuming random input vectors, the hacker expects the Trojan to be activated once (on average) by the application of

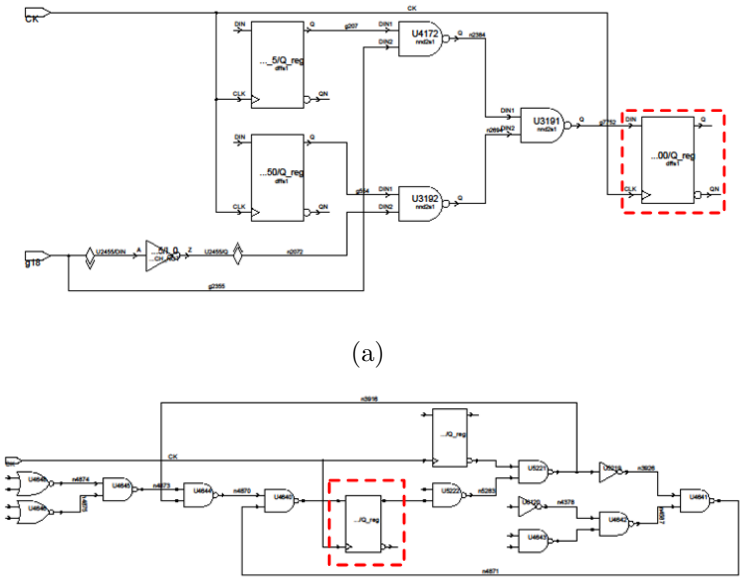
$$N = \frac{1}{\prod_{i=1}^{q} p_i} \tag{4.7}$$

test vectors. However, let the *actual* rare logic value probabilities of these internal nodes be $p_i + \Delta p_i$, for the i-th trigger node. Then, the Trojan would be *actually* activated once (on average) by:

$$N' = \frac{1}{\prod_{i=1}^{q}(p_i + \Delta p_i)} = \frac{N}{\prod_{i=1}^{q}(1 + \frac{\Delta p_i}{p_i})} \tag{4.8}$$

test vectors. The difference between the estimated and the actual number of test vectors before the Trojan is activated is $\Delta N = N - N'$, which leads to a percentage normalized difference:

$$\frac{\Delta N}{N}(\%) = \left(1 - \frac{1}{\prod_{i=1}^{q}(1 + \frac{\Delta p_i}{p_i})}\right) \times 100\% \tag{4.9}$$



(a)

(b)

Fig. 4.4. Comparison of input logic cones of a selected flip-flop in $s15850$: (a) Original design and (b) obfuscated Design.

To appreciate the effect that Δp and q has on this change on the average number of vectors that can activate the Trojan, assume $\frac{\Delta p_i}{p_i} = f \quad \forall i = 1, 2 \ldots q$; then Eqn. (4.9) can be simplified to:

$$\frac{\Delta N}{N}(\%) = \left(1 - \frac{1}{(1+f)^q}\right) \times 100\% \tag{4.10}$$

Fig. 4.3 shows this fractional change plotted vs. the number of trigger nodes (q) for different values of the fractional mis-estimation of the signal probability (f). From this plot and eqns. (4.9) and (4.10), it is evident that:

- The probability of the Trojan getting detected by logic testing increases as the number of Trojan trigger nodes (q) increases. However, it is unlikely that the hacker will have more than 10 trigger nodes, because otherwise as shown by simulations, it becomes extremely difficult to trigger the Trojans at all.

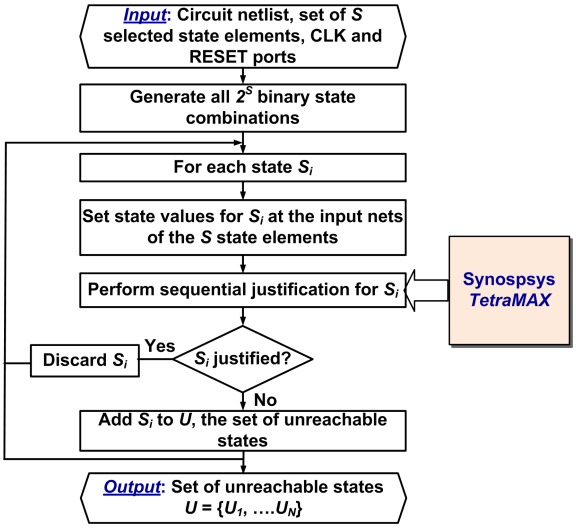


Fig. 4.5. Steps to find unreachable states for a given set of S state elements in a circuit.

- For values $2 \leq q \leq 10$, the number of random input patterns required to activate the trojan decreases sharply with q. The improvement is more pronounced at higher values of f. This observation validates the rationale behind an obfuscation-based design approach that resists the hacker from correctly estimating the signal probabilities at the internal circuit nodes.

4.3.4 Effect of obfuscation on circuit structure

The re-synthesis of the circuit after its integration with the RTL describing the *obfuscation state space* "flattens" the circuit into a single netlist, with drastic changes to the input logic cones of the primary outputs and the state elements. To appreciate this effect, consider the input logic cones (up to 4 levels) of a selected flip-flop in the gate level netlist of the $s15850$ ISCAS-89 benchmark, and its obfuscated version, shown in Fig. 4.4. As is evident from the difference in the structure of the logic cones in these figures, it is very difficult to identify these two nodes to be corresponding nodes by visual observation or automated structural analysis.

4.3.5 Determination of unreachable states

The construction of the *obfuscation state space* requires the determination of *unreachable states* in a given circuit. Fig. 4.5 shows the steps of determining the set of unreachable states for a selected set of S state elements in a given circuit. First, all the possible 2^S state bit combinations are generated for the S state elements. Then, each state of these 2^S states are subjected to *full sequential justification* at the inputs of the selected S state elements, using *Synopsys Tetramax*. The justified states are discarded, while the states which fail justification are collected to form the set U of structurally unreachable states.

4.3.6 Test generation for Trojan detection

Since deterministic test pattern generation for the Trojan population is practically infeasible due to the inordinately large number of possible Trojans, a statistical approach to sample and simulate a representative set of Trojan instances (10K-20K) from the total population of Trojans is adopted. First, the signal probabilities at the internal nodes of the circuit are estimated by the application of a large set of random vectors to the circuit. From the signal probability (S_p) of the internal nodes, which indicate the rareness of a logic-0 or logic-1 event in those nodes, a set of candidate trigger nodes with S_p less than a specified trigger threshold (θ) are selected. Next, starting from a large set of *weighted random vectors*, a smaller testset is generated to excite each of these candidate trigger nodes to its rare value at least N times, where N is a given parameter. This is done because excitation of each rare node individually to its corresponding rare value multiple times is likely to increase the probability of the Trojans triggered by them to get activated, as shown by the analysis in [10] and reproduced in Chapter 5. It was observed through extensive simulations on both combinational (ISCAS-85) and sequential (ISCAS-89) benchmark circuits, that such a statistical test generation methodology can achieve higher Trojan detection coverage than weighted random vector set, with 85% reduction in test length on average [10].

Note that for sequential circuits, a *full-scan* implementation is assumed. Sequential justification is applied to eliminate *false Trojans*, i.e. Trojans which cannot be triggered during the operation of the circuit. The above algorithm, referred to as **M**ultiple **E**xcitation of **R**are **O**ccurrence (*MERO*) test generation algorithm [10] is presented in Chapter 5 as Algorithm-2.

4.3.7 Determination of effectiveness

To determine the decrease in *potency* of the Trojans by the proposed scheme, a given vector set is reduced to eliminate those vectors with state values in the *obfuscation state space*. The circuit is then re-simulated with the reduced test set to determine the Trojan coverage. The decrease in the Trojan coverage obtained from the reduced test set indicates the Trojans which are activated or effective only in the *obfuscation state space* and, hence, become benign.

To determine the increase in *detectability* of the Trojans, the S_p values at the Trojan trigger nodes are compared between two cases: 1) a large set of random vectors, and 2) a modified set of random vectors which ensure operation of the obfuscated design in only *normal mode*. The increase in Trojan detectability is estimated by the percentage of circuit nodes for which the S_p values differ by a predefined threshold. The difference in estimated S_p prevents an adversary from exploiting the true rare events at the internal circuit nodes in order to design a hard-to-detect Trojan. On the other hand, true non-rare nodes may appear as rare in the obfuscated design, which potentially serve as *decoy* to the adversary. The above two effects are summed up by the increase in Trojan detection coverage due to the obfuscation. The coverage increase is estimated by comparing the respective coverage values obtained for the obfuscated and the original design for the same number of test patterns.

Algorithm 1 Procedure *OBFUSCATE*

Generate the obfuscated netlist from a given circuit netlist

Inputs: Circuit netlist, maximum area overhead (*max_area_overhead*), total number of states in the *obfuscation state space*, length of *initialization key sequence* (k)

Outputs: Obfuscated circuit netlist, *initialization key sequence*

```
1: Guess number of extra state elements to be added (n)
2: Guess number of original state elements to be used for state encoding (S)
3: repeat
4:     repeat
5:         Select S state elements randomly from circuit netlist
6:         Determine unreachable states for S state elements using sequential justification
7:     until sufficient unreachable states found
8:     Generate state encodings for the extra state elements
9:     Generate random state transitions for the extra state elements
10:    Generate random initialization key sequence of length k
11:    Generate RTL for obfuscation state space
12:    Integrate generated RTL with existing netlist
13:    Re-synthesize modified circuit
14:    Calculate area_overhead
15:    n ⇐ n − 1, S ⇐ S − 1
16: until area_overhead ≤ max_area_overhead
```

4.4 Integrated Framework for Obfuscation

Algorithm-1 shows the steps of the procedure ***OBFUSCATE***, which performs the obfuscation of a given gate-level circuit. The input arguments are the gate-level synthesized Verilog netlist of the given circuit, the maximum allowable area overhead, the total number of states in the *obfuscation state space*, and the length of the input key sequence (k). From the given area overhead constraint, an initial guess is made for the number (n) of extra state elements to be added and the number (S) of existing state elements to be used for state-encoding in the initialization state space. These S state-elements are randomly chosen for state-encoding in the *initialization state space*, and their unreachable states are determined by the method described in Section 4.3.5. If the number of unreachable states found is not sufficient for the required number of states in the *obfuscation state space*, another random selection of S state elements is

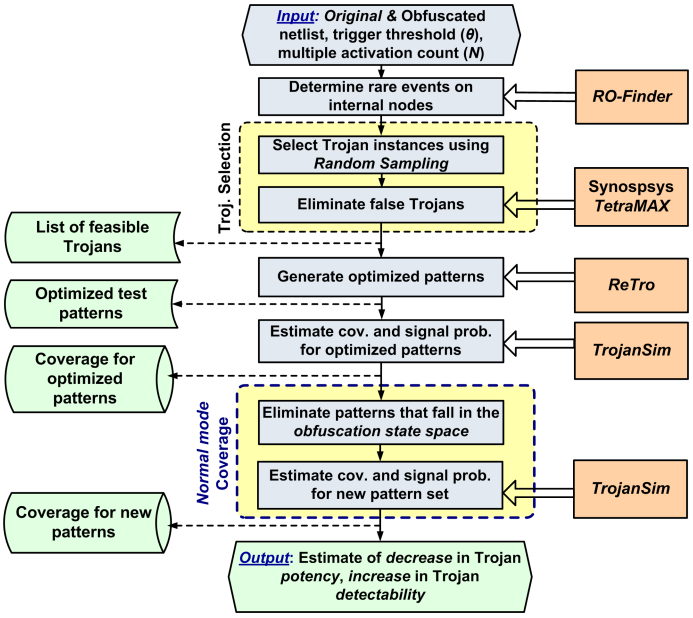


Fig. 4.6. Framework to estimate the effectiveness of the obfuscation scheme for protection against hardware Trojan attacks.

made and the process is continued until sufficient unreachable states are found. Once this is done, state encoding for the extra state elements and random state transitions for the 2^n states of the n extra state elements is generated. Next, an initialization key sequence of length k is selected randomly. The RTL of state transitions of the two separate set of flip-flops for the *initialization state space* is generated. As mentioned in Section 4.3, the RTL is constructed in a way that ensures that the chosen original state elements and the extra state elements act together as parts of the same FSM during the initialization phase. The RTL is then integrated with the original gate-level netlist, with appropriate control signals to enable the operation in the two different modes. The modified circuit is then re-synthesized under input design constraints using *Synopsys Design Compiler* to generate the obfuscated version of the circuit. If the area of the re-synthesized circuit is larger than the user-specified area overhead constraint, S and n are reduced and the process is repeated until the area constraint is satisfied for the obfuscated design.

The Trojan model shown in Fig. 4.1(a). Three C programs were written to estimate the effectiveness of the proposed obfuscation scheme for protection against Trojans. The computation of signal probabilities at the internal nodes is done by the program *RO-Finder* (**Rare O**ccurrence **Finder**). The testset for Trojan detection achieving multiple excitation of rare trigger conditions is performed by the program *ReTro* (**Re**duced pattern generator for **Tro**jans). The generation of the reduced pattern set by the elimination of the patterns with states in the *obfuscation state space* is performed by a TCL program. The decrease in the Trojan *potency* and the increase in the Trojan *detectability* are then estimated by a cycle-accurate simulation of the circuit by the simulator *TrojanSim* (**Trojan Sim**ulator). *TetraMax* is used for sequential justification of the Trojan triggering conditions. Fig. 4.6 shows the steps to estimate the effectiveness of the obfuscation scheme [10]. The entire flow was integrated with the Synopsys design environment using TCL scripts. A LEDA 250nm standard cell library was used for logic synthesis. All simulations, test generation and logic synthesis were carried out on a Hewlett-Packard Linux workstation with a 2GHz dual-core processor and 2GB RAM.

4.5 Results

To verify the trends predicted in Section 4.3.2, the effects of extra state elements (n) and unreachable states derived from variable number of existing state elements (S) on the level of protection against Trojans were investigated. Fig. 4.7 shows the variation in the percentage of Trojans rendered benign, percentage of internal nodes with false signal probability, and the percentage increase in detectability of Trojans for the *s1196* benchmark circuit. These plots clearly show the increasing level of protection against Trojans with the increasing size of the *obfuscation state space*, which matches the theoretical predictions in Section 4.3.2.

Table 4.1 and Table 4.2 show the effects of obfuscation on increasing the security against hardware Trojans for a set of ISCAS-89 benchmark circuits with 20,000

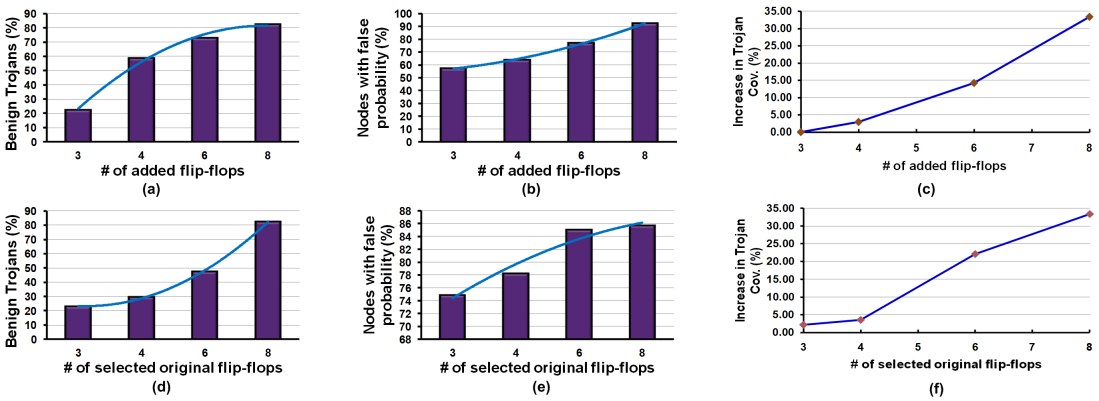


Fig. 4.7. Variation of protection against Trojans in *s1196* as a function of (a), (b) and (c): the number of added flip-flops in state encoding (S); (d), (e) and (f): the number of original state elements used in state encoding (n). For (a), (b) and (c), four original state elements were selected for state encoding, while for (d), (e) and (f), four extra state elements were added.

random instances of suspected Trojans, trigger threshold (θ) of 0.2, trigger nodes (q) 2 and 4, respectively. Optimized vector set was generated using N=1000. The same value of $n + S$ applies to both sets of results. The length of the initialization key sequence was 4 ($k = 4$) for all the benchmarks. The effect of obfuscation was estimated by three metrics: (a) the fraction of the total population of structurally justifiable Trojans becoming benign; (b) the difference between the signal probabilities at internal nodes of the obfuscated and original circuit, and (c) the improvement in the *functional Trojan coverage*, i.e. the increase in the percentage of valid Trojans detected by logic testing. Note that the number of structurally justifiable Trojans as determined by *TetraMax* decreases with the increase in the number of trigger nodes of the Trojan, and increasing size of the benchmark circuits. From the tables it is evident that the construction of the *obfuscation state space* with even a relatively small number of state elements (i.e. a relatively small value of $n + S$) still makes a significant fraction of the Trojans benign. Moreover, it obfuscates the true signal probabilities of a large number of nodes. The obfuscation scheme is more effective for

Table 4.1
Effect of Obfuscation on Security Against Trojans (100,000 random patterns, 20,000 Trojan instances, $q = 2$, $k = 4$, $\theta = 0.2$)

Benchmark Circuit	**Trojan Instances**	**Obfus. Flops (n + S)**	**Obfuscation Effects**		
			Benign Trojans (%)	**False Prob. Nodes** (%)	**Func. Troj. Cov. Incr.** (%)
s1488	192	8	38.46	63.69	0.00
s5378	2641	9	40.13	85.05	1.02
s9234	747	9	29.41	65.62	1.09
s13207	1190	10	36.45	83.59	0.56
s15850	1452	10	40.35	68.95	2.65
s38584	342	12	33.88	81.83	0.45
Average	**1094**	**≈10**	**36.45**	**74.79**	**0.96**

Table 4.2
Effect of Obfuscation on Security Against Trojans (100,000 random patterns, 20,000 Trojan instances, $q = 4$. $k = 4$, $\theta = 0.2$)

Benchmark **Circuit**	**Trojan Instances**	**Obfuscation Effects**		
		Benign Trojans (%)	**False Prob. Nodes (%)**	**Func. Troj. Cov. Incr. (%)**
s1488	98	60.53	71.02	12.12
s5378	331	70.28	85.05	15.00
s9234	20	62.50	65.62	25.00
s13207	36	80.77	83.59	20.00
s15850	124	77.78	79.58	18.75
s38584	11	71.43	77.21	50.00
Average	**≈103**	**70.55**	**77.01**	**23.48**

4-trigger node Trojans. This is expected since a Trojan with larger q is more likely to draw at least one trigger condition from the *obfuscation state space*.

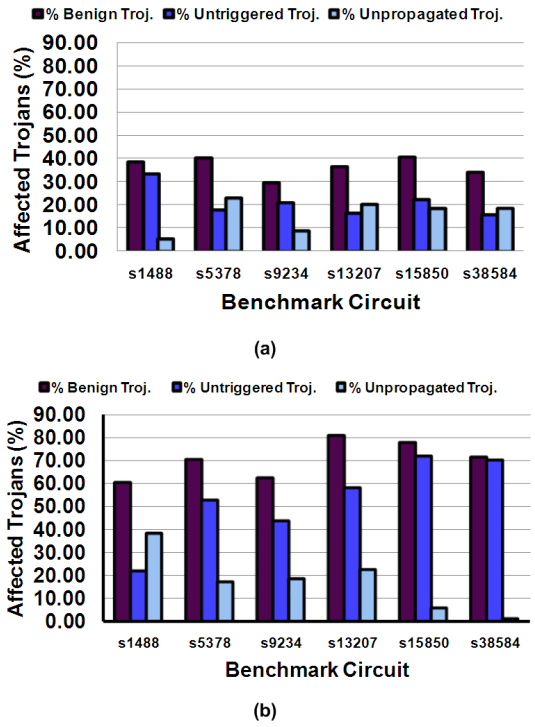


Fig. 4.8. Effect of obfuscation on Trojans: (a) 2-trigger node Trojans ($q = 2$), and (b) 4-trigger node Trojans ($q = 4$).

Fig. 4.8 shows the two different effects by which Trojans are rendered benign (as discussed in Section and 4.3.2) - i.e. some of them are triggered only in the *obfuscation state space*, while the effect of some are propagated to the primary output only in the *obfuscation state space*. In these plots, the greater effectiveness of the obfuscation approach for 4-trigger node Trojans is again evident.

Fig. 4.9 shows the improvement in Trojan detection coverage in the obfuscated design compared to the original design for the same number of random vectors. This plot illustrates the net effect of the proposed obfuscation scheme in increasing the level of protection against Trojans, with an average increase of 14.83% for for $q = 2$ and 20.24% for $q = 4$. The greater effectiveness for $q = 4$ agrees with the theoretical observation in Section 4.3.3.

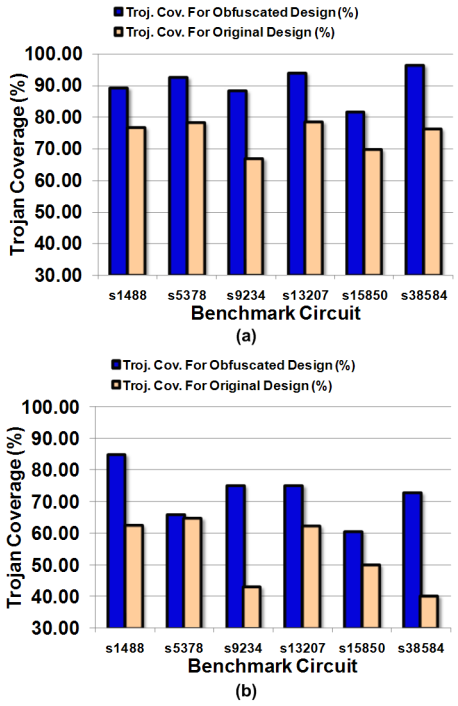


Fig. 4.9. Improvement of Trojan coverage in obfuscated design compared to the original design for (a) Trojans with 2 trigger nodes ($q = 2$) and (b) Trojans with 4 trigger nodes ($q = 4$).

Table 4.3 shows the design overheads (at iso-delay) and the run-time for the proposed obfuscation scheme. The proposed scheme incurs modest area and power overheads, and the design overhead decreases with increasing size of the circuit. As mentioned earlier, the level of protection against Trojan can be increased by choosing a larger $n + S$ value at the cost of greater design overhead. The run-time presented in the table is dominated by *TetraMax*, which takes about 90% of the total time for sequential justifications.

Table 4.3
Design Overhead (at iso-delay) and Run-time† for the Proposed Algorithm

Benchmark Circuit	**Overhead (%)**		**Run-time (mins.)**
	Area	**Power**	
s1488	20.09	12.58	31
s5378	13.13	17.66	186
s9234	11.84	15.11	1814
s13207	8.10	10.87	1041
s15850	7.04	9.22	1214
s38584	6.93	2.63	2769
Average	**11.19**	**11.34**	**1175.83**

†The run time includes the sequential justification time by Synopsys *Tetramax*.

4.6 Discussions

4.6.1 Protection against malicious CAD tools

Besides protecting a design in foundry, the proposed obfuscation methodology can provide effective defense against malicious modifications (manual or automated) during the IC design steps. As pointed out in Section 4.1, compromised CAD tools and automation scripts can also insert Trojans in a design [6, 50]. Obfuscation can prevent insertion of hard-to-detect Trojans by CAD tools due to similar reasons as applicable in a foundry. It prevents an automatic analysis tool from finding the true rare events, which can be potentially used as Trojan triggers or payloads. Moreover, since large number of states belong to the obfuscation state space, an automation tool is very likely to insert a Trojan randomly that is only effective in the obfuscation mode. Note that since the gate-level netlist is obfuscated, protection against CAD tools can be achieved during the design steps following logic synthesis (e.g. during physical synthesis and layout).

To increase the scope of protection by encompassing the logic synthesis step, a small modification in the obfuscation-based design flow is proposed. Fig. 4.10 compares a conventional IP-based SoC design flow with the proposed modified design flow. In the conventional design flow, the RTL is directly synthesized to a technology mapped gate-level netlist, and obfuscation is applied on this netlist. However, in the modified design flow, the RTL is first *compiled* to a technology independent (perhaps unoptimized) gate-level description, and obfuscation is applied on this netlist. Such a practice is quite common in the industry, and many commercial tools support such a compilation as a preliminary step to logic synthesis [64]. The obfuscated netlist is then optimized and technology mapped by a logic synthesis tool. Note that the logic synthesis step now operates on the obfuscated design, which protects the design from potential malicious operations during logic synthesis. Also, the RTL *compilation* (without logic optimization) is a comparatively simpler computational step for which the SoC design house can employ a trusted in-house tool. This option provides an extra level of protection.

This proposed obfuscation methodology also provides protection against malicious CAD tools in Field Programable Gate Array (FPGA) based design flows. As noted in [6], the main threat of Trojan insertion in such a flow comes from the CAD tools which convert the RTL description of a design to the FPGA device specific configuration *bitstream*. Typically, the fabric itself can be assumed to be Trojan-free [6]. Similar to the SoC design flow, a small modification is proposed to the FPGA design flow that maximizes the scope of protection against FPGA CAD tools. Fig. 4.11 shows the proposed design flow. The RTL corresponding to the circuit can be "compiled" to a unoptimized, technology-independent gate-level netlist. This netlist can then be obfuscated, and the obfuscated design can then be optimized and mapped by either third-party CAD tools or vendor-specific tools to a netlist in an intermediate format. This netlist is then converted to a vendor-specific bitstream format by the FPGA mapping tool to map the circuit to the FPGA. Note that once the design is obfuscated, the security against CAD tools flows down the design cycle.

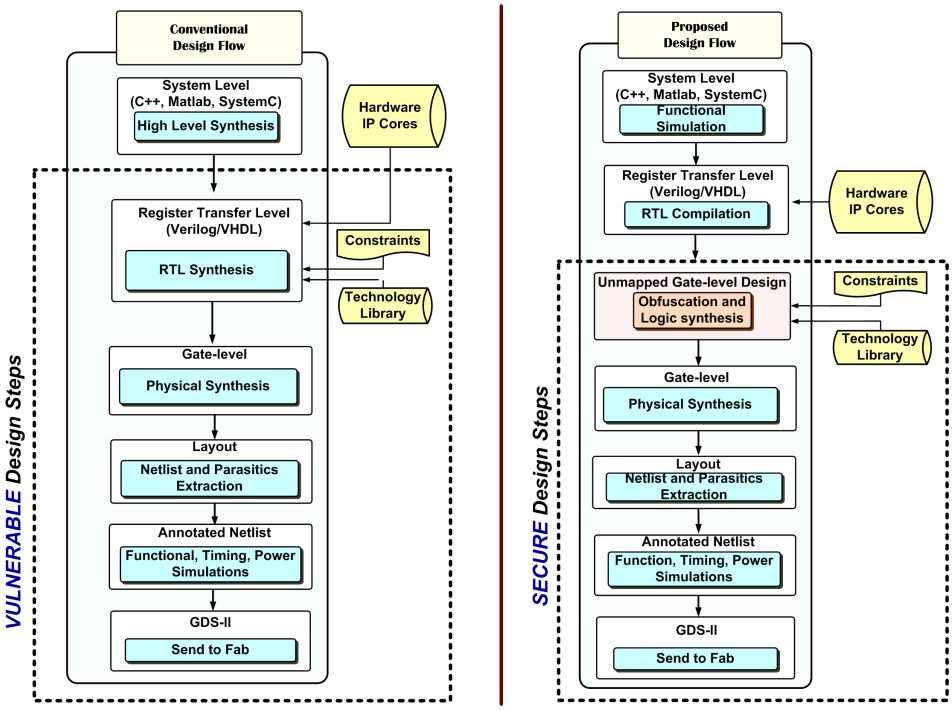


Fig. 4.10. Comparison of conventional and proposed SoC design flows. In the proposed design flow, protection against malicious modification by untrusted CAD tools can be achieved through obfuscation early in the design cycle.

4.6.2 Improving protection and design overhead

Eqn. 4.6 suggests that for large designs with a significantly large *original state space*, to attain satisfactory levels of design obfuscation, it is necessary to have the *obfuscation state space* much larger than *original state space*. This can be achieved by either: (a) addition of a large number of extra state elements, or (b) using a large number of unreachable states in the *obfuscation state space*. However, finding unreachable states through sequential justification for a large number of state elements in a large design is very expensive computationally. Even then, the generated RTL to describe the *obfuscation state space* would be complicated and not amenable to efficient logic synthesis, resulting in potentially unacceptable design overhead. To keep the problem computationally tractable and reduce the design overhead, the

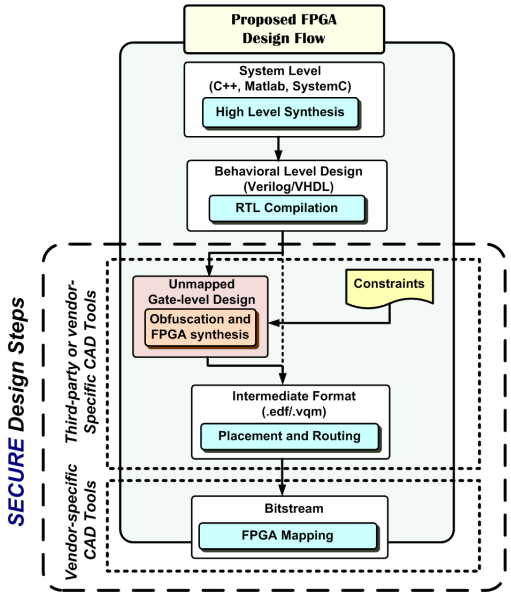


Fig. 4.11. Proposed FPGA design flow for protection against CAD tools.

scheme shown in Fig. 4.12 is proposed. The n extra state elements are grouped into p different groups to form parallel FSMs PSM_1 through PSM_p, and RTL for each of them is generated separately. Similarly, the S existing state elements used for state encoding in the *obfuscation state space* are grouped in q different groups $PSM_1^{'}$ through $PSM_q^{'}$. Sequential justification for each group is performed separately, and the RTL for each of the parallel FSMs $PSM_1^{'}$ through $PSM_q^{'}$ is generated separately based on the unreachable states. Such a scheme of having multiple parallel FSMs to design the *obfuscation state space* achieves the same design obfuscation effects, without the burden of high computational complexity and design overhead.

4.6.3 Application to other Trojan models

Although in the simulations the Trojans according to the model shown in Fig. 4.1(a) were considered, as pointed out in Section 4.2, the proposed methodology can also help to protect against Trojan attacks that aim at leaking secret information

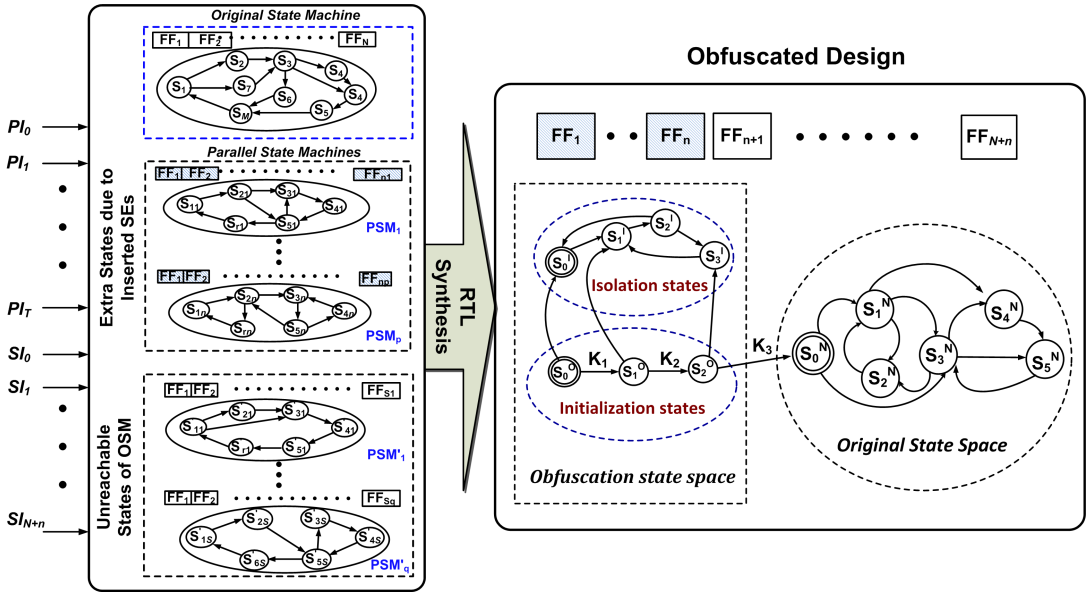


Fig. 4.12. Obfuscation for large designs can be efficiently realized using multiple parallel state machines which are constructed with new states due to additional state elements as well as unreachable states of original state machine.

about internal state of the circuit, either in the form of a data-stream (similar to Fig. 4.1(c)) or as side-channel signature (similar to Fig. 4.1(d)). Such a Trojan is shown in Fig. 4.13, where it transmits out a secret cryptographic key through a covert communication channel by "sniffing" the values on the communication bus. *Bus scrambling* or bus re-ordering is a simple technique to resist against this kind of an attack, so that the data transmitted out by the Trojan is also scrambled. To overcome this defense mechanism, the hacker has to figure out the actual order of bits in the scrambled bus to correctly interpret the collected data. Figuring out the actual order of the bits in a n-bit bus by simulations will require a search among $n!$ possibilities, e.g. $\sim 2 \times 10^{35}$ possibilities for a 32-bit data bus. However, since the attacker has access to the design, he/she is likely to perform structural analysis of the design to determine the order of bits in the bus. If the functional blocks are not obfuscated, one can employ equivalence checking between a functional block in the design and a

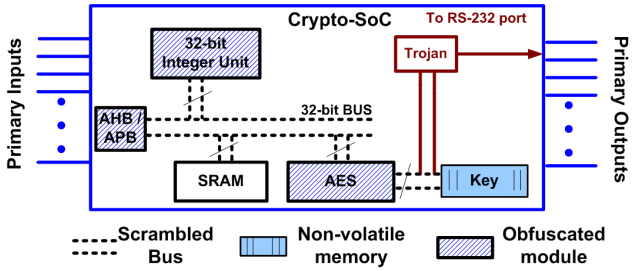


Fig. 4.13. Functional block diagram of a crypto-SoC showing possible Trojan attack to leak secret key stored inside the chip. Obfuscation coupled with bus scrambling can effectively prevent such attack.

corresponding reference design to identify the order of bits in both input/output bus for a module. For example, an attacker can perform formal verification between the integer unit in Fig. 4.13 and a functionally equivalent reference design to find the port association.

However, if all the modules in the given SoC are obfuscated using the proposed approach, it would be practically infeasible for a formal verification tool to establish structural equivalence [7]. The other choice left to the attacker is to simulate the circuit by applying input vectors. For simplicity, assume all modules in the SoC are initialized simultaneously and by the same initialization key sequence. Then, to reach the *normal mode* of operation, the hacker needs to first apply the correct *unknown* initialization vectors in correct order to enable normal operating mode of the IC. Only then the hacker would be able to establish the actual bus order through simulations, the complexity of which has already been shown to be extremely high. The probability of succeeding in reaching the normal mode by the application of random vectors to the primary input of a SoC with M primary inputs and an initialization key sequence length of N is $\frac{1}{2^{M \cdot N}}$. Assuming the SoC shown in Fig. 4.13 has 32 inputs, and assuming the length of the *initialization key sequence* to be 4, the probability of the hacker taking the obfuscated SoC to the *normal mode* is $\sim 10^{-39}$. The width of the data bus for the key is typically 128 or 256, which would increase the complexity

exponentially. A similar argument can be presented for Trojans of the type shown in Fig. 4.1(d).

4.7 Summary

Malicious modifications of integrated circuits in untrusted fabrication facilities has emerged as a serious security threat. Conventional logic test generation and application techniques cannot be readily extended to reliably detect hardware Trojans during post-manufacturing test. In this work, a novel application of design obfuscation to achieve effective protection against hardware Trojan has been presented. Obfuscation has earlier been employed to prevent hardware IP piracy and reverse engineering. In this chapter, we show that obfuscation can provide effective protection against Trojan attacks including defense against untrusted CAD tools, for both SoC and FPGA. We also show that the proposed approach can achieve protection against Trojans that tries to leak secret information from an IC. The level of obfuscation and hence the protection can be increased by state-space blow-up and use of unreachable states. The required design modifications can be easily automated and integrated with conventional design flow. Simulation results for a set of benchmark circuits show that a well-formulated obfuscation scheme can provide simultaneous protection against hardware Trojan and IP piracy at low design overhead. In the next chapter, we describe *MERO*, a statistical test generation approach to detect hardware Trojans.

5. A STATISTICAL APPROACH FOR HARDWARE TROJAN DETECTION

5.1 Introduction

In the previous chapters, we have proposed design techniques to achieve protection against various security threats, including hardware Trojan attacks. In addition to the obfuscation approach, we have explored another design methodology [59] where special circuitry was embedded in an IC to improve the controllability and observability of internal nodes, thereby facilitating the detection of inserted Trojans by logic testing. In this chapter, we propose a novel testing methodology, referred to as *MERO* (**M**ultiple **E**xcitation of **R**are **O**ccurence) protection against hardware Trojans. *MERO* comprises of a statistical test generation approach for Trojan detection and a coverage determination approach for quantifying the level of trust. The main objective of the proposed methodology is to derive a set of test patterns that is *compact* (minimizing test time and cost), while maximizing the Trojan detection coverage. The basic concept is to detect rare or low probability conditions at the internal nodes, and then derive an optimal set of vectors than can trigger each of the selected low probability nodes *individually to their rare logic values multiple times* (e.g. at least N times, where N is a given parameter). As analyzed in Section 5.2.1, this increases the probability of detection of an arbitrary Trojan instance. By increasing the toggling of nodes that are random-pattern resistant, it improves the probability of activating an unknown Trojan compared to purely random patterns. The proposed methodology is conceptually similar to the *N-detect test* [65, 66] used in stuck-at ATPG (automatic test pattern generation), where a test set is generated to detect each single stuck-at fault in a circuit by at least N different patterns to improve test quality and defect coverage [66]. In this chapter, we focus on digital Trojans [53], which can be inserted

into a design either in a design house (e.g. by untrusted CAD tool or IP) or in a foundry. We do not consider the Trojans where the triggering mechanism and/or effect are analog in nature (e.g. thermal fluctuations).

Since the proposed detection is based on functional validation using logic values, it is robust with respect to parameter variations and can reliably detect very small Trojans, e.g. the ones with few logic gates. Thus, the technique can be used as *complementary to the side-channel Trojan detection approaches* [47, 55–57], which are more effective in detecting large Trojans (e.g. ones with area $> 0.1\%$ of the total circuit area). In side-channel approaches existence of a Trojan is determined by noting its effect in a one or more physical side-channel parameters, such as current or delay. Besides, the *MERO* approach can be used to increase the detection sensitivity of many side-channel techniques such as the ones that monitor the power/current signature, by increasing the activity in a Trojan circuit [56]. Using an integrated Trojan coverage simulation and test generation flow, we validate the approach for a set of ISCAS combinational and sequential benchmark circuits. Simulation results show that the proposed test generation approach can be extremely effective for detecting arbitrary Trojan instances of small size, both combinational and sequential.

The rest of the chapter is organized as follows. Section 5.2 describes the mathematical justification of the *MERO* methodology, the steps of the *MERO* test generation algorithm and the Trojan detection coverage estimation. Section 5.3 describes the simulation setup and presents results for a set of ISCAS benchmark circuits with detailed analysis. Section 5.4 concludes the chapter.

5.2 Statistical Approach for Trojan Detection

As described in Section 5.1, the main concept of our test generation approach is based on generating test vectors that can excite candidate trigger nodes individually to their rare logic values multiple (at least N) times. In effect, the probability of activation of a Trojan by the simultaneous occurrence of the rare conditions at its

trigger nodes increases. As an example, consider the Trojan shown in Fig. 4.1(a). Assume that the conditions $a = 0$, $b = 1$ and $c = 1$ are very rare. Hence, if we can generate a set of test vectors that induce these rare conditions at these nodes individually N times where N is sufficiently large, then a Trojan with triggering condition composed jointly of these nodes is highly likely to be activated by the application of this test set. The concept can be extended to sequential Trojans, as shown in Fig. 4.1(b), where the inserted 3-bit counter is clocked on the simultaneous occurrence of the condition $ab' = 1$. If the test vectors can sensitize these nodes such that the condition $ab' = 1$ is satisfied at least 8 times (the maximum number of states of a 3-bit counter), then the Trojan would be activated. Next, we present a mathematical analysis to justify the concept.

5.2.1 Mathematical Analysis

Without loss of generality, assume that a Trojan is triggered by the rare logic values at two nodes A and B, with corresponding probability of occurrence p_1 and p_2. Assume T to be the total number of vectors applied to the circuit under test, such that both A and B have been individually excited to their rare values *at least* N times. Then, the expected number of occurrences of the rare logic values at nodes A and B are given by $E_A = T{\cdot}p_1 \geq N$ and $E_B = T{\cdot}p_2 \geq N$, which lead to:

$$T \geq \frac{N}{p_1} \qquad \text{and} \qquad T \geq \frac{N}{p_2} \tag{5.1}$$

Now, let p_j be the probability of simultaneous occurrence of the rare logic values at nodes A and B, an event that acts as the trigger condition for the Trojan. Then, the expected number of occurrences of this event when T vectors are applied is:

$$E_{AB} = p_j{\cdot}T \tag{5.2}$$

In the context of this problem, we can assume $p_j > 0$, because an adversary is unlikely to insert a Trojan which would never be triggered. Then, to ensure that the Trojan is

triggered at least once when T test vectors are applied, the following condition must be satisfied:

$$p_j \cdot T \geq 1 \tag{5.3}$$

From inequality (5.1), let us assume $T = c \cdot \frac{N}{p_1}$. where $c \geq 1$ is a constant depending on the actual test set applied. Inequality (5.3) can then be generalized as:

$$S = c \cdot \frac{p_j}{p_1} \cdot N \tag{5.4}$$

where S denotes the number of times the trigger condition is satisfied during the test procedure. From this equation, the following observations can be made about the interdependence of S and N:

1. For given parameters c, p_1 and p_j, S is proportional to N, i.e. the expected number of times the Trojan trigger condition is satisfied increases with the number of times the trigger nodes have been individually excited to their rare values. This observation forms the main motivation behind the *MERO* test generation approach for Trojan detection.

2. If there are q trigger nodes and if they are assumed to be mutually independent, then $p_j = p_1 \cdot p_2 \cdot p_3 \cdots p_q$, which leads to:

$$S = c \cdot N \cdot \prod_{i=2}^{q} p_i \tag{5.5}$$

   As $p_i < 1 \quad \forall i = 1, 2, \cdots q$, hence, with the increase in q, S decreases for a given c and N. In other words, with the increase in the number of trigger nodes, it becomes more difficult to satisfy the trigger condition of the inserted Trojan for a given N. Even if the nodes are not mutually independent, a similar dependence of S on q is expected.

3. The trigger nodes can be chosen such that $p_i \leq \theta \quad \forall i = 1, 2, \cdots q$, so that θ is defined as a *trigger threshold* probability. Then as θ increases, the corresponding selected rare node probabilities are also likely to increase. This will result in

Algorithm 2 Procedure *MERO*

Generate reduced test pattern set for Trojan detection

Inputs: Circuit netlist, list of rare nodes (L) with associated rare values, list of random patterns (V), number of times a rare condition should be satisfied (N)

Outputs: Reduced pattern set (R_V)

```
1: Read circuit and generate hypergraph
2: for all nodes in L do
3:     set number of times node satisfies rare value (A_R) to 0
4: end for
5: set R_V = Φ
6: for all random pattern in V do
7:     Propagate values
8:     Count the # of nodes (C_R) in L with their rare value satisfied
9: end for
10: Sort vectors in V in decreasing order of C_R
11: for all vector v_i in decreasing order of C_R do
12:     for all bit in v_i do
13:         Perturb the bit and re-compute # of satisfied rare values (C_R')
14:         if (C_R' > C_R) then
15:             Accept the perturbation and form v_i' from v_i
16:         end if
17:     end for
18:     Update A_R for all nodes in L due to vector v_i
19:     if v_i' increases A_R for at least one rare node then
20:         Add the modified vector v_i' to R_V
21:     end if
22:     if (A_R≥N) for all nodes in L then
23:         break
24:     end if
25: end for
```

an increase in S for a given T and N i.e. the probability of Trojan activation would increase if the individual nodes are more likely to get triggered to their rare values.

All of the above predicted trends were observed in our simulations, as shown in Section 5.3.

5.2.2 Test Generation

Algorithm 2 shows the major steps in the proposed reduced test set generation process for Trojan detection. We start with the golden circuit netlist (without any Trojan), a random pattern set (V), list of rare nodes (L) and number of times to activate each node to its rare value (N). First, the circuit netlist is read and mapped to a *hypergraph*. For each node in L, we initialize the number of times a node encounters a rare value (A_R) to 0. Next, for each random pattern v_i in V, we count the number of nodes (C_R) in L whose rare value is satisfied. We sort the random patterns in decreasing order of C_R. In the next step, we consider each vector in the sorted list and modify it by perturbing one bit at a time. If a modified test pattern increases the number of nodes satisfying their rare values, we accept the pattern in the reduced pattern list. In this step we consider only those rare nodes with $A_R < N$. The process repeats until each node in L satisfies its rare value at least N times. The output of the test generation process is a minimal test set that improves the coverage for both combinational and sequential Trojans compared to random patterns.

5.2.3 Coverage Estimation

Once the reduced test vector set has been obtained, computation of Trigger and Trojan coverage can be performed for a given *trigger threshold* (θ) (as defined in Section 5.2.1) and a given number of trigger nodes (q) using a random sampling approach. From the Trojan population, we randomly select a number of q-trigger Trojans, where each trigger node has signal probability less than equal θ. We assume that Trojans comprising of trigger nodes with higher signal probability than θ will be detected by conventional test. From the set of sampled Trojans, Trojans with false trigger conditions which cannot be justified with any input pattern are eliminated. Then, the circuit is simulated for each vector in the given vector set and checked whether the trigger condition is satisfied. For an activated Trojan, if its effect can be observed at the primary output or scan flip-flop input, the Trojan is considered

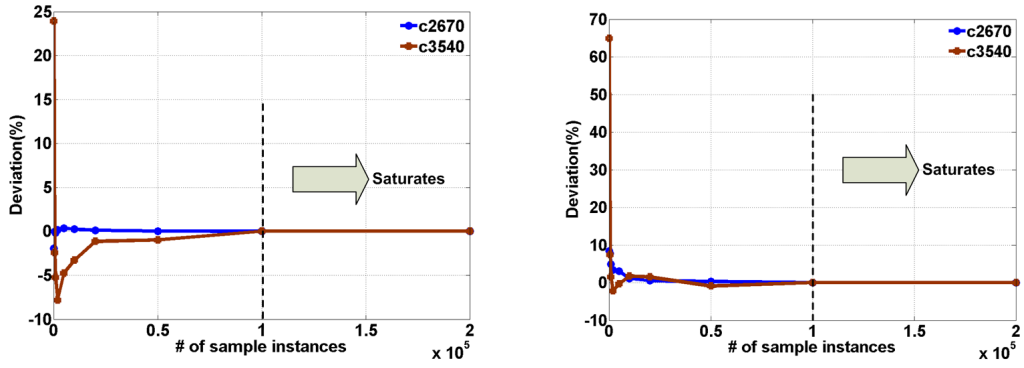
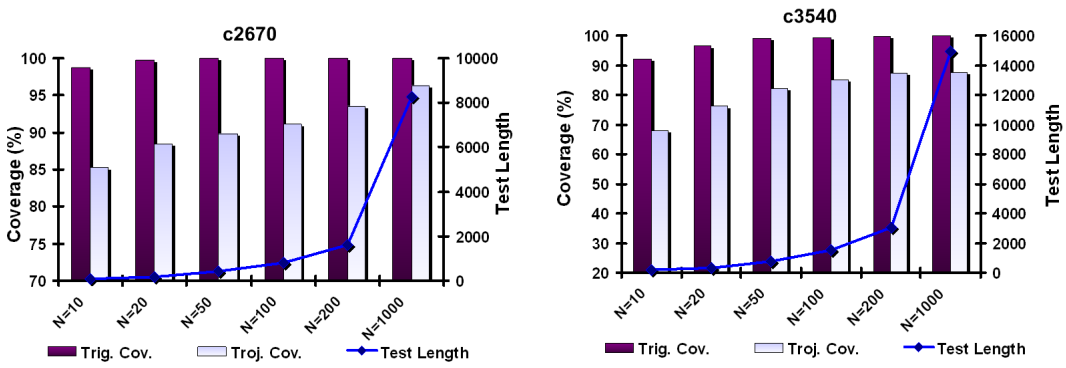


Fig. 5.1. Impact of sample size on trigger and Trojan coverage for benchmarks c2670 and c3540, $N = 1000$ and $q = 4$: (a) deviation of trigger coverage, and (b) deviation of Trojan coverage.

"covered", i.e. detected. The percentages of Trojans activated and detected constitute the *trigger coverage* and *Trojan coverage*, respectively.

5.2.4 Choice of Trojan Sample Size

In any random sampling process an important decision is to select the sample size in a manner that represents the population reasonably well. In the context of Trojan detection, it means further increase in sampled Trojans, renders negligible change in the estimated converge. Fig. 5.1 shows a plot of percentage deviation of Trigger and Trojan coverage ($q = 4$) from the asymptotic value for two benchmark circuits with varying Trojan sample size. From the plots, we observe that the coverage saturates with nearly 100,000 samples, as the percentage deviation tends to zero. To compromise between accuracy of estimated coverage and simulation time, we have selected a sample size of 100,000 in our simulations.



(a) (b)

Fig. 5.2. Impact of N (number of times a rare point satisfies its rare value) on the trigger/Trojan coverage and test length for benchmarks (a) c2670 and (b) c3540.

5.2.5 Choice of N

Fig. 5.2 shows the trigger and Trojan coverage for two ISCAS-85 benchmark circuits with increasing values of N, along with the lengths of the corresponding testset. From these plots it is clear that similar to *N-detect* tests for stuck-at fault where defect coverage typically improves with increasing N, the trigger and Trojan coverage obtained with the *MERO* approach also improves steadily with N, but then both saturate around $N = 200$ and remain nearly constant for larger values of N. As expected, the test size also increases with increasing N. We chose a value of $N = 1000$ for most of our experiments to reach a balance between coverage and test vector set size.

5.2.6 Improving Trojan Detection Coverage

As noted in previous sections, Trojan detection using logic testing involves simultaneous triggering of the Trojan and the propagation of its effect to output nodes. Although the proposed test generation algorithm increases the probability of Trojan

activation, it does not explicitly target increasing the probability of a malicious effect at payload being observable. *MERO* test patterns, however, achieves significant improvement in Trojan coverage compared to random patterns, as shown in Section 5.3. This is because the Trojan coverage has strong correlation with trigger coverage. To increase the Trojan coverage further, one can use the following low-overhead approaches:

1. *Improvement of test quality*: We can consider number of nodes observed along with number of nodes triggered for each vector during test generation. This means, at step 13-14 of Algorithm 2, a perturbation is accepted if the sum of triggered and observed nodes improves over previous value. This comes at extra computational cost to determine the number of observable nodes for each vector. We note that for a small ISCAS benchmark *c432* (an interrupt controller), we can improve the Trojan coverage by 6.5% with negligible reduction in trigger coverage using this approach.

2. *Observable test point insertion*: We note that insertion of very few observable test points can achieve significant improvement in Trojan coverage at the cost of small design overhead. Existing algorithm for selecting observable test points for stuck-at fault test [67] can be used here. Our simulation with *c432* resulted in about 4% improvement in Trojan coverage with 5 judiciously inserted observable points.

3. *Increasing N and/or increasing the controllability of the internal nodes*: Internal node controllability can be increased by judiciously inserting few controllable test points or increasing N. It is well-known in the context of stuck-at ATPG, that scan insertion improves both controllability and observability of internal nodes. Hence, the proposed approach can take advantage of low-overhead design modifications to increase the effectiveness of Trojan detection.

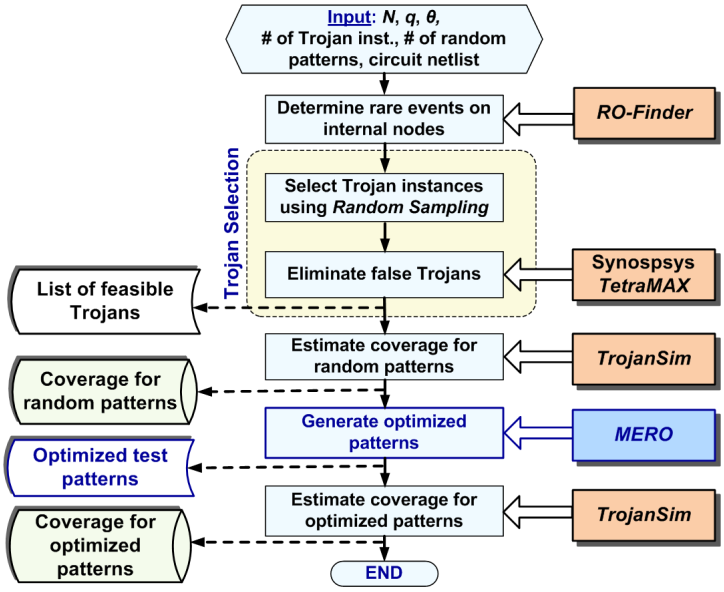


Fig. 5.3. Integrated framework for rare occurrence determination, test generation using *MERO* approach, and Trojan simulation.

5.3 Results

5.3.1 Simulation setup

We have implemented the test generation and the Trojan coverage determination in three separate C programs. All the three programs can read a Verilog netlist and create a *hypergraph* from the netlist description. The first program, named as *RO-Finder* (**R**are **O**ccurence **Finder**), is capable of functionally simulating a netlist for a given set of input patterns, computing the signal probability at each node and identifying nodes with low signal probability as rare nodes. The second program *MERO* implements algorithm-2 described in Section 5.2.2 to generate the reduced pattern set for Trojan detection. The third program, *TrojanSim* (**Trojan Sim**ulator), is capable of determining both Trigger and Trojan coverage for a given test set using random sample of Trojan instances. A q-trigger random Trojan instance is created by randomly selecting the trigger nodes from the list of rare nodes. We consider one

Table 5.1
Comparison of Trigger and Trojan coverage among ATPG patterns [68], Random (100K, input weights: 0.5), and *MERO* patterns for $q = 2$ and $q = 4$, $N = 1000$, $\theta = 0.2$

Circuit	Nodes (Rare/ Total)	ATPG patterns				Random (100K patterns)				*MERO* Patterns			
		q = 2		q = 4		q = 2		q = 4		q = 2		q = 4	
		Trigger Cov. (%)	Trojan Cov. (%)	Trigger Cov. (%)	Trojan Cov. (%)	Trigger Cov. (%)	Trojan Cov. (%)	Trigger Cov. (%)	Trojan Cov. (%)	Trigger Cov. (%)	Trojan Cov. (%)	Trigger Cov. (%)	Trojan Cov. (%)
c2670	297/1010	93.97	58.38	30.7	10.48	98.66	53.81	92.56	30.32	100.00	96.33	99.90	90.17
c3540	580/1184	77.87	52.09	16.07	8.78	99.61	86.5	90.46	69.48	99.81	86.14	87.34	64.88
c5315	817/2485	92.06	63.42	19.82	8.75	99.97	93.58	98.08	79.24	99.99	93.83	99.06	78.83
c6288	199/2448	55.16	50.32	3.28	2.92	100.00	98.95	99.91	97.81	100.00	98.94	92.50	89.88
c7552	1101/3720	82.92	66.59	20.14	11.72	98.25	94.69	91.83	83.45	99.38	96.01	95.01	84.47
s13207‡	865/2504	82.41	73.84	27.78	27.78	100	95.37	88.89	83.33	100.00	94.68	94.44	88.89
s15850‡	959/3004	25.06	20.46	3.80	2.53	94.20	88.75	48.10	37.98	95.91	92.41	79.75	68.35
s35932‡	970/6500	87.06	79.99	35.9	33.97	100.00	93.56	100.00	96.80	100.00	93.56	100.00	96.80
Avg.	**724/2857**	**74.56**	**58.14**	**19.69**	**13.37**	**98.84**	**88.15**	**88.73**	**72.30**	**99.39**	**93.99**	**93.50**	**82.78**

‡These sequential benchmarks were run with 10,000 random Trojan instances to reduce run time of *Tetramax*

randomly selected payload node for each Trojan. Fig. 5.3 shows the flow-chart for the *MERO* methodology. Synopsys *TetraMAX* was used to justify the trigger condition for each Trojan and eliminate the false Trojans. All simulations and test generation were carried out on a Hewlett-Packard Linux workstation with a 2GHz dual-core Intel processor and 2GB RAM.

5.3.2 Comparison with Random and ATPG Patterns

Table 5.1 lists the trigger and Trojan coverage results for a set of combinational (ISCAS-85) and sequential (ISCAS-89) benchmarks using stuck-at ATPG patterns (generated using the algorithm in [68]), weighted random patterns and *MERO* test patterns. It also lists the number of total nodes in the circuit and the number of rare nodes identified by *RO-Finder* tool based on signal probability. The signal probabilities were estimated through simulations with a set of 100,000 random vectors. For the sequential circuits, we assume full-scan implementation. We consider 100,000 random

Table 5.2
Reduction in test length with *MERO* approach compared to 100K random patterns along with runtime, $q = 2$, N=1000, θ=0.2

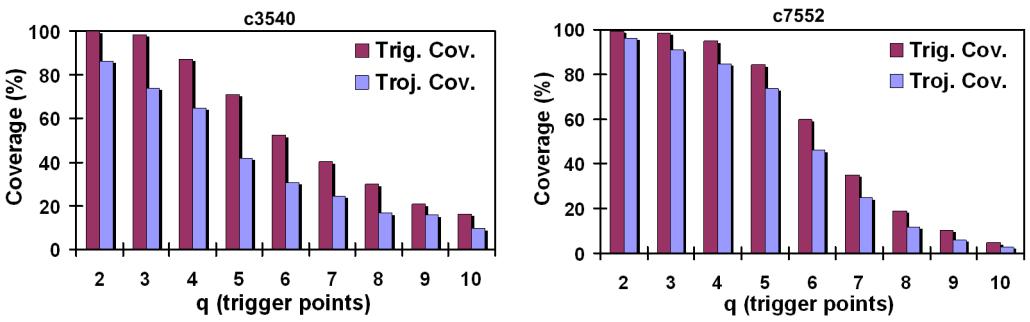
Ckt.	***MERO* test length**	**% Reduction**	**Run-time (s)**
c2670	8254	**91.75**	30051.53
c3540	14947	**85.05**	9403.11
c5315	10276	**89.72**	80241.52
c6288	5014	**94.99**	15716.42
c7552	12603	**87.40**	160783.37
s13207†	26926	**73.07**	23432.04
s15850†	32775	**67.23**	39689.63
s35932†	5480	**94.52**	29810.49
Avg.	**14534**	**85.47**	**48641.01**

†These sequential benchmarks were run with 10,000 random Trojan instances to reduce run time of *Tetramax*

instances of Trojans following the sampling policy described in Section 5.2.4, with one randomly selected payload node for each Trojan. Coverage results are provided in each case for two different trigger point count, $q = 2$ and $q = 4$, at $N = 1000$ and $\theta = 0.2$.

Table 5.2 compares reduction in the length of the testset generated by the *MERO* test generation method with 100,000 random patterns, along with the corresponding run-times for the test generation algorithm. This run-time includes the execution time for *Tetramax* to validate 100,000 random Trojan instances, as well as time to determine the coverage by logic simulation. We can make the following important observations from these two tables:

1. The stuck-at ATPG patterns provide poor trigger and Trojan coverage compared to *MERO* patterns. The increase in coverage between the ATPG and *MERO* patterns is more significant in case of higher number of trigger points.



(a) (b)

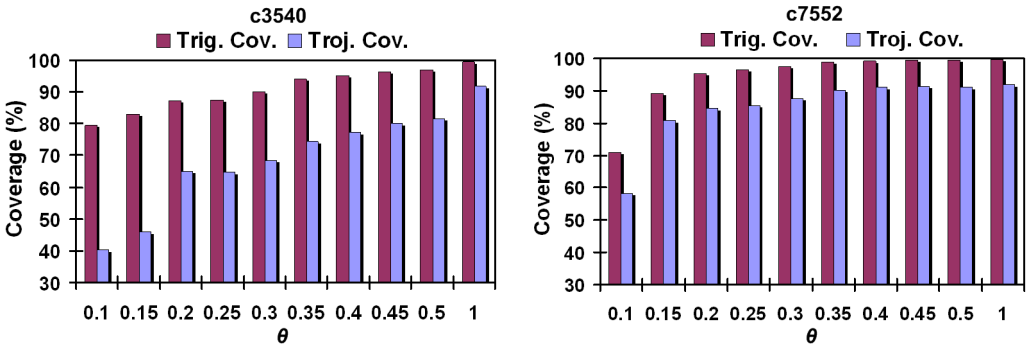
Fig. 5.4. Trigger and Trojan coverage with varying number of trigger points (q) for benchmarks (a) c3540 and (b) c7552, at $N = 1000$, $\theta = 0.2$.

2. From Table 5.2, it is evident that the reduced pattern with N=1000 and $\theta = 0.2$ provides comparable trigger coverage with significant reduction in test length. The average improvement in test length for the circuits considered is about 85%.

3. Trojan coverage is consistently smaller compared to trigger coverage. This is because in order to detect a Trojan by applying an input pattern, besides satisfying the trigger condition, one needs to propagate the logic error at the payload node to one or more primary outputs. In many cases although the trigger condition is satisfied, the malicious effect does not propagate to outputs. Hence, the Trojan remains triggered but undetected.

5.3.3 Effect of Number of Trigger Points (q)

The impact of q on coverage is evident from the Fig. 5.4, which shows the decreasing trigger and Trojan coverage with the increasing number of trigger points for two combinational benchmark circuits. This trend is expected from the analysis of Section 5.2.1. Our use of *TetraMAX* for justification and elimination of the false triggers helped to improve the Trojan coverage.



(a) (b)

Fig. 5.5. Trigger and Trojan coverage with *trigger threshold* (θ) for benchmarks (a) c3540 and (b) c7552, for $N = 1000$, $q = 4$.

5.3.4 Effect of Trigger Threshold (θ)

Fig. 5.5 plots the trigger and Trojan coverage with increasing θ for two ISCAS-85 benchmarks, at $N = 1000$ and $q = 4$. As we can observe, the coverage values improve steadily with increasing θ while saturating at a value above 0.20 in both the cases. The improvement in coverage with θ is again consistent with the conclusions from the analysis of Section 5.2.1.

5.3.5 Sequential Trojan Detection

To investigate the effectiveness of the *MERO* test generation methodology in detecting sequential Trojans, we designed and inserted sequential Trojans modeled following Fig. 4.1(b), with 0, 2, 4, 8, 16 and 32 states, respectively (the case with zero states refers to a combinational Trojan following the model of Fig. 4.1(a)). A cycle-accurate simulation was performed by our simulator *TrojanSim*, and the Trojan was considered detectable only when the output of the golden circuit and the infected circuit did not match. Table 5.3 presents the trigger and Trojan coverage respectively obtained by 100,000 randomly generated test vectors and the *MERO* approach for

Table 5.3
Comparison of sequential Trojan coverage between random (100K) and MERO patterns, $N = 1000$, $\theta = 0.2$, $q = 2$

Ckt.	Trigger Cov. for 100K Random Vectors (%)						Trigger Cov. for *MERO* Vectors (%)					
	Trojan State Count						Trojan State Count					
	0	2	4	8	16	32	0	2	4	8	16	32
s13207	100.00	100.00	99.77	99.31	99.07	98.38	100.00	100.00	99.54	99.54	98.84	97.92
s15850	94.20	91.99	86.79	76.64	61.13	48.59	95.91	95.31	94.03	91.90	87.72	79.80
s35932	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Avg.	**98.07**	**97.33**	**95.52**	**91.98**	**86.73**	**82.32**	**98.64**	**98.44**	**97.86**	**97.15**	**95.52**	**92.57**
Ckt.	**Trojan Cov. for 100K Random Vectors (%)**						**Trojan Cov. for *MERO* Vectors (%)**					
	Trojan State Count						**Trojan State Count**					
	0	2	4	8	16	32	0	2	4	8	16	32
s13207	95.37	95.37	95.14	94.91	94.68	93.98	94.68	94.68	94.21	94.21	93.52	92.82
s15850	88.75	86.53	81.67	72.89	58.4	46.97	92.41	91.99	90.62	88.75	84.23	76.73
s35932	93.56	93.56	93.56	93.56	93.56	93.56	93.56	93.56	93.56	93.56	93.56	93.56
Avg.	**92.56**	**91.82**	**90.12**	**87.12**	**82.21**	**78.17**	**93.55**	**93.41**	**92.80**	**92.17**	**90.44**	**87.70**

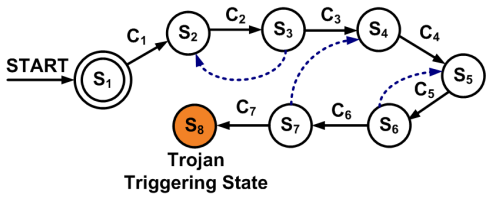


Fig. 5.6. FSM model with no loop in state transition graph.

three large ISCAS-89 benchmark circuits. The superiority of the *MERO* approach over the random test vector generation approach in detecting sequential Trojans is evident from this table.

Although these results have been presented for a specific type of sequential Trojans (counters which increase their count conditionally), they are representative of other sequential Trojans whose state transition graph (STG) has no "loop". The STG for such a FSM has been shown in Fig. 5.6. This is a 8-state FSM which changes its state only when a particular internal node condition C_i is satisfied at state S_i, and

the Trojan is triggered when the FSM reaches state S_8. The example Trojan shown in Fig. 4.1(b) is a special case of this model, where the conditions C_1 through C_8 are identical. If each of the conditions C_i is as rare as the condition $a = 1, b = 0$ required by the Trojan shown in Fig. 4.1(b), then there is no difference between these two Trojans as far as their rareness of getting triggered is concerned. Hence, we can expect similar coverage and test length results for other sequential Trojans of this type. However, the coverage may change if the FSM structure is changed (as shown with dotted line). In this case, the coverage can be controlled by changing N.

5.3.6 Application to Side-channel Analysis

As observed from the results presented in this section, the *MERO* approach can achieve high trigger coverage for both combinational and sequential Trojans. This essentially means that the *MERO* patterns will induce activity in the Trojan triggering circuitry with high probability. A minimal set of patterns that is highly likely to cause activity in a Trojan is attractive in power or current signature based side-channel approach to detect hardware Trojan. The detection sensitivity in these approaches depends on the induced activity in the Trojan circuit by applied test vector. It is particularly important to enhance sensitivity for the Trojans where the leakage contribution to power by the Trojan circuit can be easily masked by process or measurement noise. Hence, *MERO* approach can be extended to generate test vectors for side-channel analysis, which requires amplifying the Trojan impact on side-channel parameter such as power or current.

5.4 Summary

Conventional logic test generation techniques cannot be readily extended to detect hardware Trojans because of the inordinately large number of possible Trojan instances. In this chapter, we have presented a statistical Trojan detection approach using logic testing where the concept of multiple excitation of rare logic values at inter-

nal nodes is used to generate test patterns. Simulation results show that the proposed test generation approach achieves about 85% reduction in test length over random patterns for comparable or better Trojan detection coverage. The proposed detection approach can be extremely effective for small combinational and sequential Trojans with small number of trigger nodes, for which side-channel analysis approaches cannot work reliably. Hence, the proposed detection approach can be used as complementary to side-channel analysis based detection schemes. Future work will involve improving the test quality which will help in minimizing the test length and increasing Trojan coverage further.

6. A KEY-BASED SECURE SCAN DESIGN APPROACH

6.1 Introduction

In the previous chapters we have presented approaches for protecting ICs or hardware IPs against piracy, reverse-engineering and hardware Trojan attacks. In this chapter we will consider a different attack model, where an attacker tries to extract the secret key from a crypto-chip using scan chain. It is common to find crypto algorithms implemented in hardware to meet high throughput requirements of modern secure systems. This can be done by either implementing them as Application Specific Integrated Circuits (ASICs) [70] or as co-processors [71]. A common example is the "Advanced Encryption Standard" (AES) [72], which after being accepted as a standard by the National Institute of Standards and Technology (NIST) in 2001, is widely employed in applications ranging from low-end small mobile consumer products to high-end internet servers. Scan-chain based "Design for Testability" (DFT) is generally used in these secure chips to improve testability of the system at lower overhead. Their utility is enhanced by the fact that the scan chains can be connected to an external five-pin "Joint Test Action Group" (JTAG) interface to provide test and debug capability in the field. Such in-field test and debug capability is very important for many applications, e.g. microprocessors, which eases the maintenances and development of software.

However, the very nature of the scan-chain to provide improved test and debug capability is due to the fact that scan chains make the internal nodes of a system more controllable and observable [73]. Standard cryptographic algorithms such as "Data Encryption Standard" (DES), AES, "Rivest-Shamir-Adleman" (RSA), and "Hash-based Message Authentication Code" (HMAC) rely on a secret key built inside the

Work done in collaboration with Somnath Paul [69].

crypto chips. The security is provided by the crypto chip as long as the information regarding the secret key is inside the chip, both during normal mode as well as test mode of operation. A hacker with proper resources can access and analyze the scanned out information and thereby discover the secret key. The situation has reached such severe proportions that information is commonly available in websites on the internet which describe the step by step procedure to decrypt such "keys" for supposedly "secure" systems [74]. Hence, the designer is faced with a dilemma while designing the crypto system. On one hand, it must be ensured that the security of the system is not sacrificed, while on the other hand, the testability of the system both during manufacturing test and in the field cannot be compromised. Both security and testability factors have to be properly balanced when a design solution is considered. Thus, to protect the secret information while retaining the benefits of scan chain, additional hardware safety measures must be incorporated in the crypto chips.

To address this issue, in this chapter, we propose a new scan design methodology called *VIm-Scan* (**V**irtually **Im**pervious Scan). In *VIm-Scan*, all the advantages and simplicity of traditional scan-based testing is preserved, yet at the cost of small area and power overheads, the security of the crypto chip is improved tremendously which make the secure key virtually impervious to any unauthorized attempt to access them. The basic idea is to use the test stimulus itself to embed an N-bit key and to enable the scan-out process only when M different N-bit keys are matched in M successive test cycles during the test initiation process. Compared to the existing techniques for secure scan design [11, 73], the proposed method has the following salient benefits:

- It provides high security, making scan-based attack virtually impossible.
- It incurs extremely low design overhead. Simulation results show that we can achieve virtually infinite resistance to attack with only 68 logic gates, which is about 5X lower compared to existing techniques.
- Unlike existing techniques, the proposed method does not impact the scan insertion or test generation process. Moreover, except the first few test cycles,

which is used to enable the scan-out process, the proposed method does not affect the test application process.

- In cases, where a crypto system is vulnerable to scan-based controllability/observability attack, the proposed method can be easily extended to prevent these attacks by making the outputs of selected scan cells unavailable to combinational logic before actual test begins.

The rest of the chapter is organized as follows. In Section 6.2, we discuss previous work in this area on secure scan design in crypto chips. In Section 6.3, we present the proposed technique. In Section 6.4, simulation results are presented to illustrate the effectiveness of the proposed scheme. We extend the proposed method to enhance the robustness of the crypto chip against scan based controllability/observability attacks in Section 6.5. We conclude in Section 6.6.

6.2 Previous Work

Various methods have been proposed to prevent prospective hackers from accessing the secure information in a crypto chip [73], [11, 75–80]. In [11], a simple solution has been proposed based on use of the use of a second register, called a "Mirror Key Register (MKR)", which prevents any important data from entering the scan chain in the test mode. The MKR provides security to the secret key by isolating the data path and the control path performing the crypto algorithm. The authors distinguish between two distinct states of operation of the system under test: the "insecure" state and the "secure" state and the transitions between states are performed using a finite state machine. Although this scheme is conceptually simple, it affects the test application procedure considerably. It requires a two-step test procedure where part of the system independent of the key is tested during insecure mode and the rest (dependent on the key) is tested in secure mode, when scan is disabled. Since in a crypto chip, typically, a large part of the circuit operation depends on the key, testing this part with disabled scan chain adversely affects test time and/or coverage.

Moreover, the proposed method does not work for systems where the key is hardwired instead of being stored in a non-volatile memory (e.g. ROM). The secret key, that is built in as a ROM or a combinational logic cannot be tested by the structural scan-based test. Further, the method requires duplication of the entire key (typically 128 bit or more) and thus incurs a relatively high hardware overhead.

In [73], the authors propose a "low-cost secure scan" (LCSS) solution, where the test pattern is modified to include a special "key". These key bits are scanned into "dummy flip-flops" in the scan chain (which are not connected to the combinational circuits), and is matched by a "key checking logic" (KCL) block. If the key match, a "secure" signal is generated, which allows scanning out of the data from the scan chains; else, a randomly seeded LFSR "Random Bit Generator" (RBG) network randomizes the response to be scanned out. Thus, it makes any reverse engineering attempt based on the scan-out response very difficult. However, the scheme has several shortfalls. 1) The method requires extra dummy flip flops in the scan chain, which requires modification of the scan insertion process. 2) It adds the key to each test stimulus, thus for an example scan chain length of 500, an 80-bit key a dds as high as 16% overhead in test time. The overhead increases significantly for longer key or with decrease in scan length. 3) Finally, the expression for the total number of combinations reported in [11] is the maximum number of trials that the attacker has to undertake on order to correlate the scan input and output. It does not reflect the actual probability from the point of view of the hacker to be able to match the key while scanning in the test vector. Considering that the attacker chooses to set each scan-in bit randomly, the probability of matching any arbitrary k bits of the key still remains $\frac{1}{2^k}$, irrespective of the size of scan chain (S) and how these k bits are chosen from S positions in the scan chain. Thus, an 80-bit key renders a key-breaking probability of 2^{-80}, which might not be sufficiently low for practical purposes.

In [75], the proposed solution involves breaking the scan chain after the functionality of the chip has been validated. This is done by placing polysilicon fuses near the pin connections and blowing them after manufacturing testing or completely cutting

off the test interface with a wafer saw. However, this simple solution compromises the maintenance and debug capabilities of the chip in field. In [76], the flips flops that contain the secret key are isolated in a separate scan-chain, and this scan chain is made inaccessible to the end-user. However, this approach again limits in-field debug capability of the chips. This is also not too practical, as in stand-alone crypto chips, almost all flip-flops could contain information that can potentially reveal the secret key. In [77], the situation is circumvented by having "Built-In-Self-Test" (BIST) structures to test the entire design, while in [78], a mixed approach has been taken where BIST is used on sensitive portions and scan is used on the remainder. Although these solutions help preventing accessibility of secret key by potential attacker, they compromise fault coverage level with respect to scan and Automatic Test Pattern Generation (ATPG).

In [79], a scan scrambling technique was introduced which splits up the scan chain into sub-chains, and then uses logic circuits and random number generator to randomly reconnect the sub-chains together again, and thus internally scrambling the data. Although this method presents a high level of complexity to the prospective hacker, it incorporates a significant timing and area overhead. Moreover, the method is not foolproof, as statistical analysis of the data scanned out from the chip can still determine the scan-chain structure and the secret information. In [80], a "Lock and Key" technique has been proposed, in which the scan chain is divided into independent sub-chains and using a randomly seeded Linear-Feedback-Shift-Register (LFSR), one sub-chain at a time is randomly enabled to scan-in and scan-out, while the others remain unaffected. This technique has the problem of a large overhead for complex systems, where the number of subchains can be very large.

6.3 Proposed Methodology

In this section, we describe the proposed scan protection scheme, *VIm-Scan*, in details. The proposed scheme has very low probability for extraction of useful infor-

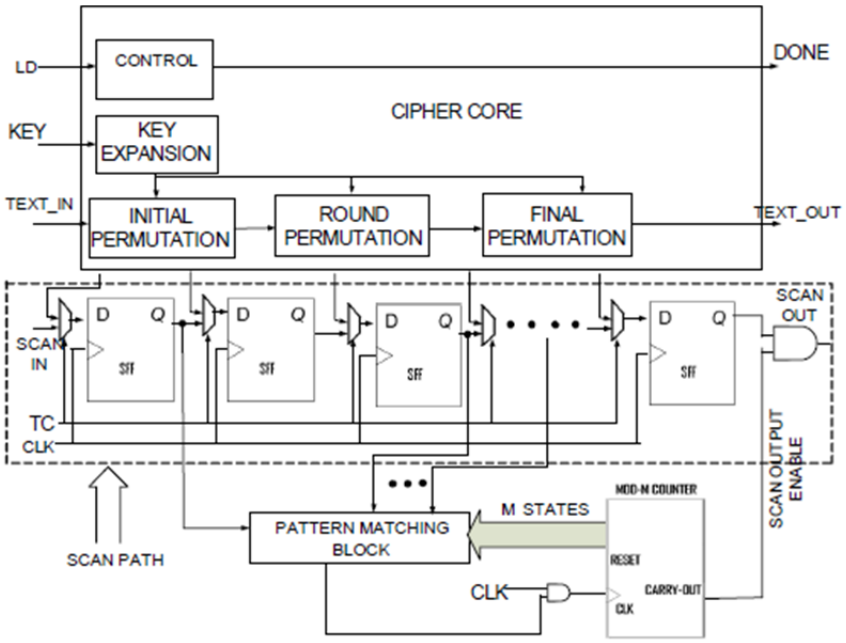


Fig. 6.1. Overall block diagram for *VIm-Scan*.

mation through the scan out process by an unintended user. The protection rests on the fact that in order to initiate the testing procedure by conventional scan, the user has to match M number of different keys of length N during a test initiation phase prior to actual test procedure. The keys should be embedded in the test vectors itself and they need to follow an order in the test vectors supplied. Only when all the keys have been matched in a specific order during the test initiation phase, the scan out process from the crypto chip is enabled.

The circuitry for detection of the keys from the test vector is built into the system. The choice of key is random and is determined at the time of system design. The approach is scalable in the sense that the designer can opt for a greater security (by increasing either the length or the number of keys) at the cost of increased design overhead. The approach also has the advantage that it seamlessly integrates with the synthesized scan chain without addition of any extra flip-flops or gates to the scan-chain itself.

The proposed protection scheme is illustrated in Fig. 6.1. The figure depicts a standard AES cipher core [72] along with the scan chain path. Let the scan path chosen be of length $L(> N)$. Since the key is N-bit long, during the design phase, N scan flip-flops are chosen randomly and their output is fed to a block used for matching the keys that is scanned-in during the test initiation phase. A mod-M counter counts the number of keys that have been matched during this phase. A count of M (indicated by the "carry out" signal) is used to enable the "AND" gate that provides the scan output. Until the scan-out is enabled the data available at the scan-out port will be all zero. A test vector containing the incorrect key or an incorrect number of mismatches does not allow any scan output and thus prevents useful information being leaked out.

6.3.1 Probability of Breaking the Key

As mentioned earlier, the user is not able to scan out any secure information unless M different N bit keywords are matched (from the vector loaded in the scan chain) in a specific order. Assuming the attacker is supplying a completely random binary pattern, the probability for each bit in the test vector (at any position) to be 1 or 0 is $\frac{1}{2}$. Thus the probability of N scan flip-flop outputs to match with a predefined keyword of length N is $\frac{1}{2^N}$. But in order to enable the scan output process, this match has to be performed M number of times, each time with a different keywords (each of length N bits). Since each trial is independent of others, the overall probability for a successful scan out enable is:

$$P_b = \frac{1}{2^{M \cdot N}} \tag{6.1}$$

A choice of even small values of M and N (e.g. M and N both 16) gives a very low probability of matching the keywords (10^{-78}). Hence, probabilistically, extremely large number of trials is required to match the above criterion for a potential attacker. The complexity of breaking the protection, obtained in the proposed scheme, provides virtually infinite resistance to attack, since, in reality, it requires an amount of time

more than the chip's lifetime to break the security. It is worth noting that the proposed scheme is scalable since the resistance to attack can be increased easily by increasing N and/or M.

6.3.2 Overhead Analysis

The proposed methodology entails significantly less overhead in terms of area and power. The extra hardware required are mainly due to two components: the "Pattern Matching Network" and the "modulo-M counter". The gating logic at the scan output port contributes negligible hardware overhead. The area overhead for the modulo-M counter is deterministic for a particular library and technology node, for a particular value of M. The area of the pattern matching network is however, is determined by the random keys that are to be matched. Thus, one can choose a set of keys that allow for a better area and power overhead, since their choice in no way compromises the security offered by the system as long as the length of each keyword is fixed. In terms of test time and test vectors, the overhead is simply M. This is because, in the proposed approach, we do not embed extra bits in the test stimulus unlike the method in [11]. The only overhead in test time results from the M number of additional vectors used in the test initiation phase to enable scan-out process. For a smaller value of M, the increase in test time is negligible. The test power overhead during the test initiation phase is only a small percentage of the total test power as described in the Section 6.4.

6.3.3 Functionality

A Power-On Reset is used to initialize the state elements of the modulo-M counter to start from a zero state. During the test initiation phase, the entire scan chain is loaded with the test vector before the intermediate flip-flop outputs are matched with pre-determined patterns. Note that the scan flip-flop outputs used in the Pattern Matching Network for matching each N-bit keyword are chosen from the N

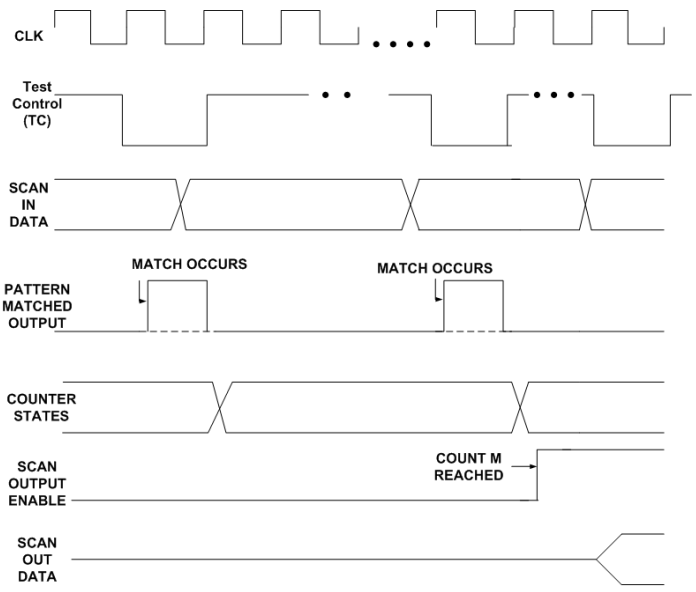


Fig. 6.2. Timing diagram for initiation phase before actual test application for $M = 2$.

predetermined positions. For the i-th match let the pattern to be matched is N_i. The keyword N_i to be matched for a particular vector is determined by the present state of the counter. If the scan flip-flop outputs match this N bit pattern, the count is incremented by one, else the counter holds its state. When a count of M is reached (indicating that M keys of N bits are matched), the scan output is enabled. Once enabled, the user is free to perform normal scan-in and scan-out operations. Fig. 6.2 shows the timing diagram for the operation of the protection scheme. As observed from this figure, the pattern matching occurs after a vector is scanned in and the scan output is enabled after the test initiation phase and before the actual application starts.

6.3.4 Testing the Proposed Scheme

The Pattern Matching Network and the modulo-M counter in the proposed scheme introduce additional hardware resources into the system and these resources need to

Table 6.1
Area, Power and Test time Overhead for the *VIm-Scan* Scheme

M	N	Proposed Scheme				Cipher Core + Proposed Scheme				Increase (%)			
		Area (μm^2)	Gate Count	Power μW	Test Time (# of Vectors)	Area (μm^2)	Gate Count	Power (μW)	Test Time (# of Vectors)	Area	Gate Count	Power	Test Time
256	1	11446	160	786	256	75581	11384	206.9	2078	1.77	1.67	0.87	14
128	2	9853	141	1179	128	75389	11359	207.2	1950	1.54	1.43	1.00	7
32	8	8617	119	1213	32	75344	11328	207.5	1854	1.46	1.16	1.16	1.7
16	16	8277	93	1601	16	75187	11305	207.8	1838	1.25	0.96	1.30	0.87
4	64	7340	102	1729	4	74944	11263	207.9	1826	0.93	0.58	1.35	0.21
1	256	7979	68	2879	1	74978	11255	208.2	1823	0.98	0.52	1.49	0.05

be tested as well for correct functionality during manufacturing test. The previous works on secure scan deign [11,73] do not provide the overhead in test time for testing the protection scheme inserted in the secure chip. In order to evaluate the additional complexity of testing these hardware resources, we have compared the test time for the stand alone cipher core with the core having the proposed security scheme. We note that any fault in the scan protection logic will be exhibited in two ways: 1) it will not enable scan out even if proper set of keys are scanned in and 2) scan out will be enabled with wrong keys. Detection of the first scenario can be simply accomplished by scanning in M vectors with N-bit keys embedded into it and checking if the scan out is enabled. To account for the second scenario, we used an automatic test pattern generation tool [81] to enumerate all possible faults and find out vectors to test them. The number of tests required for the cipher core alone is 1822 whereas the core with protection logic in *VIm-Scan* requires 1869 tests. Therefore, the increase in test time due to testing the protection scheme is merely 2.58%.

6.4 Results

The proposed method was implemented for the standard AES core [72]. We have only used the cipher block in our core, but the idea can be equally extended to the

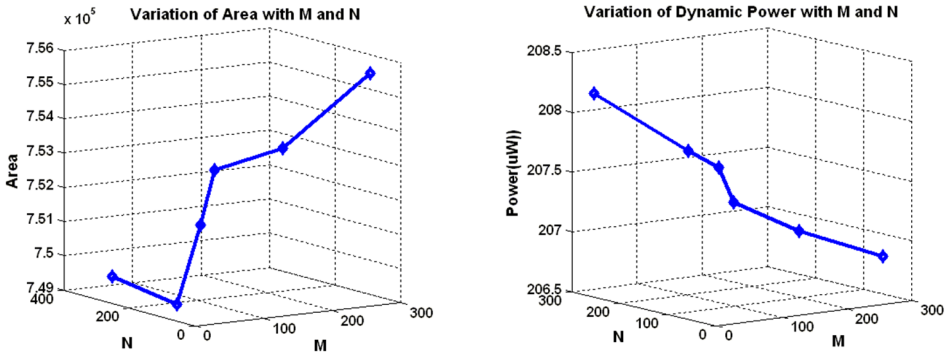
inverse cipher block as well. The keys chosen for our experiments were obtained using a random number generator with identical seed.

In our experiments, we use the probability of breaking the protection (as described in Section 6.3) as a measure of security. Different combinations of M and N were tried such that $M \cdot N = 256$, so that the probability of breaking the scheme by the application of random vectors is $\frac{1}{2^{256}}$. A higher value value of $M \cdot N$ will yield a lower probability of detection. The core and the protection scheme were synthesized using Synopsys Design Compiler with LEDA standard cell library using TSMC 250nm technology. First, the core and the protection scheme were synthesized separately and the area and power were noted. Finally, the designs were merged and re-synthesized. Table 6.1 summarizes the area, power and test time overhead for different values of M and N. The area, power and number of tests for the stand alone AES cipher core were obtained as 742439μm^2, 205.1mW and 1822, respectively for the same setup. The percentage increase in each of these parameters was calculated and has been listed in Table 6.1. Test time corresponds to the number of test stimuli required to test the design. The proposed method adds an overhead of M tests corresponding to the test initiation step. The power denoted is the dynamic power for the designs assuming 50% transition probability at the inputs. The power overhead in column 5, contributes to additional power to both test and normal mode. Although the overhead is very small (1.2% on average), we can eliminate any power overhead in normal mode using power-gating techniques (e.g. supply gating). From Table 6.1, we see that a minimum area overhead is obtained at a value of $M = 4$ and $N = 64$ with only a 0.21% increase in test time. Column 4 lists the number of additional gates required to implement the protection scheme. It can be noted that the additional gate count is as low as 68, which shows about 5X improvement compared to the technique in [11].

The proposed scheme has also been evaluated with a set of ISCAS-89 benchmarks. The area and test time overhead results for the ISCAS-89 Benchmarks are given in Table 6.2. The variation of the area and power of the merged design (AES cipher

Table 6.2
Area, Power and Test time Overhead for the *VIm-Scan* Scheme

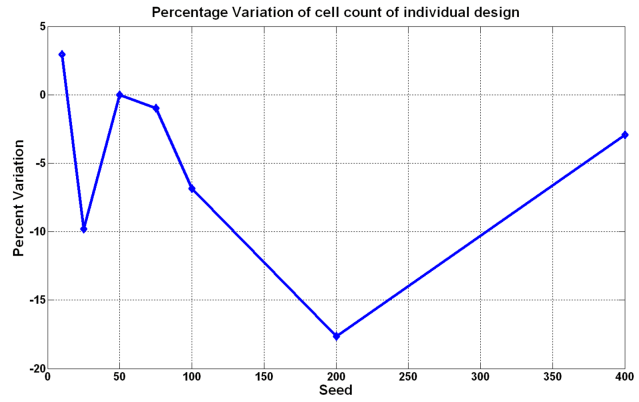
Circuit	Area (μm^2)	Area Increase (%)	Original Test Size (# of vectors)	Test Time Increase (%)
s13207	197929	3.70	497	0.8
s15850	182476	4.00	603	0.6
s35932	623481	1.17	4084	0.1
s38417	569401	1.28	1880	0.2
s38584	537559	1.36	1309	0.3



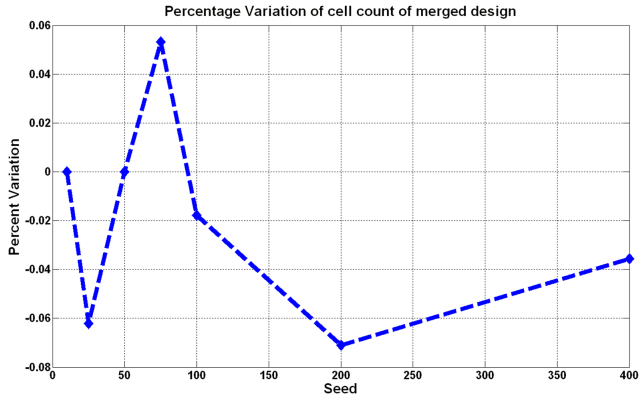
(a) (b)

Fig. 6.3. Variation of (a) area and (b) power for different values of M and N.

core + proposed scheme) with different values of M and N are shown in Fig. 6.3(a) and 6.3(b). Tests were also carried out to see the impact of different seeds on the design overhead for $M = 4$ and $N = 64$. Fig. 6.4(a) shows the impact of the seeds on the protection scheme. The percentage variation of the cell count is large due to the smaller number of cells. But after merging the protection scheme with the cipher core, the percentage variation of the total cell count becomes negligible for the different seeds, as observed in Fig. 6.4(b).



(a)



(b)

Fig. 6.4. Variation of cell count in synthesized design for different seeds for (a) original design and (b) merged design.

6.5 Improvement of Robustness against Attacks

The proposed method is resistant to both scan-based observability and scan-based controllability/observability attacks [73]. In the scan-based observability attack, the attacker applies a vector at the primary inputs and switches to the *Test Mode* in order to extract useful information about the position and content of critical registers on the scan chain. The proposed method prevents this possibility and only allows an authorized user to extract critical information through the scan out process. In

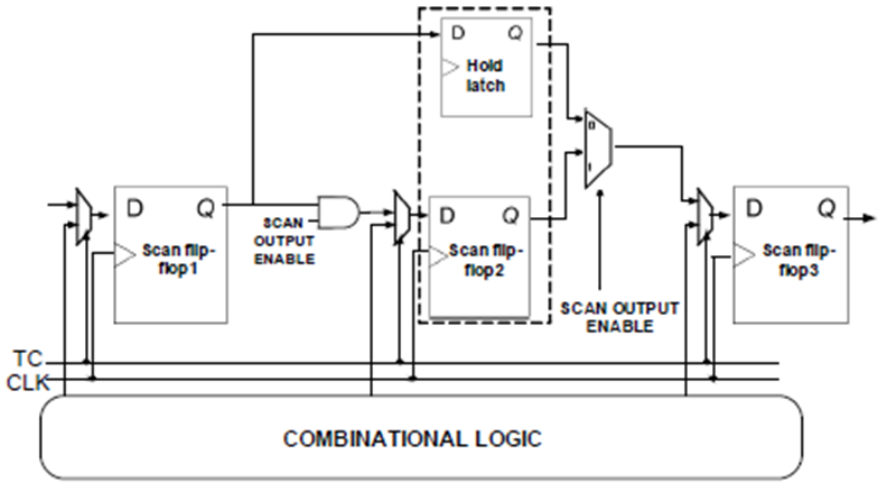


Fig. 6.5. Adaptation of scan chain to prevent scan-based controllability/observability attack

the scan-based controllability/observability attack, on the other hand, the attacker uses the scan-chain for loading the vector into the system (i.e. to control internal node states) and observes the primary outputs. However, to accomplish this, the attacker should accurately know the position of the critical registers in the scan-chain. This can be achieved using the scan based observability attack. *VIm-Scan* provides protection against scan-based observability attack and hence reduces the risk for scan-based controllability/observability attacks. However, the proposed method does not prevent any scan-in process by any unauthorized user and it is always possible to load a vector in the scan-chain. In certain scenarios, the attacker might have knowledge about the position of critical registers in the scan chain. For example, if the attacker has a behavioral description of the crypto core that is being implemented and the synthesis tool used for logic and test synthesis, it might be possible for one to know the position of the critical registers in the scan chain. With this knowledge, the attacker can load the critical registers of the design (using scan chain) with a "distinguishing sequence" (which reflects the secure key on the primary output response) and observe the response at the primary outputs to extract secret information.

The type of scan-based attack described above can be prevented by extending the proposed technique. This can be achieved by first identifying the critical registers in the design which contribute to the care bits in a distinguishing sequence and then bypassing them with a scan hold latch described as used in enhanced-scan-like delay testing [82,83]. Fig. 6.5 shows a possible implementation of this bypassing mechanism using the "Scan Output Enable" signal depicted in Figs. 6.1 and 6.2. "SL" denotes the shadow latch to hold the scanned-in value during test initiation phase. During the test initiation phase, the "Scan Output Enable" signal is used to bypass the second scan flip-flop, as shown in Fig. 6.5. The output from the hold latch can be used for matching the key in the initiation phase if the particular scan flip-flop output is chosen for pattern matching. During the actual test application, the system flip-flop is desirably in the scan path. The attacker is thus prevented from loading the important scan flip-flops (which contribute to create distinguishing sequence) by the scan-in operation during the test initiation phase. This prevents observing the effect of scanned-in vector in the primary outputs, thus eliminating the possibility of scan-based controllability/observability attack.

6.6 Summary

In this chapter, we have presented an effective and very low overhead scheme for preventing scan-based attack in secure chips. The proposed method has similarity with the key-based obfuscation approach in that it enables normal operation (i.e. scan in/out process) on application of an initialization key. The method provides security by utilizing the test stimulus to contain a keyword and then matching one/multiple keyword(s) during the test initiation phase to enable scan-out process. The method is capable of providing virtually infinite resistance to attack and can be easily integrated with existing design/test flow without affecting the scan insertion, test generation and test application steps. The proposed scheme comes with significantly lower overhead

(in terms of area and power) compared to existing techniques and can be easily extended to improve robustness against scan-based controllability/observability attack.

This chapter ends the description of the techniques dedicated to hardware protection, as explored in this thesis. In the next chapter, we explore the role of obfuscation in embedded software protection.

7. EMBEDDED SOFTWARE SECURITY THROUGH KEY-BASED OBFUSCATION

7.1 Introduction

The market share of embedded processors is ever-increasing, with more than 98% of the market share (in terms of unit sold) already occupied by them [84]. They can be found in almost every conceivable electronic application - from low-end household items such as microwave ovens to high-end 3G cell phones and PDAs. Combined with this trend is the increase in computing capabilities of embedded processors (with maximum operating frequencies of up to 2 GHz in 2010) rivalling that of mainstream microprocessors [85], as they are expected to run more computation-intensive software. An example is that cutting edge cellular devices are being increasingly used to surf the internet, play online games and perform "mobile commerce", functionalities that were more traditionally associated with personal computers. Software development for the mobile platform has advanced immensely, with users routinely downloading, installing and using both free and commercial software for their devices.

However, this trend has increased the security concerns encompassing data confidentiality and integrity, authentication, privacy, denial of service, nonrepudiation, and digital content protection [15], which were again relevant earlier only in the domain of commercial and personal computing. The threat is a two-edged sword - on one hand, malicious software installed in an embedded system harms the user; on the other hand, reverse-engineering of software causes loss of millions of dollars of intellectual property (IP) to the software vendors. Unfortunately, the traditional hardware or software security measures targeting personal computers are not directly applicable to embedded systems. The computational demands of secure processing often

Work done in collaboration with Seetharam Narasimhan.

overwhelm the computing capabilities of embedded processors, and physically the portable embedded systems are often severely constrained by form factor, resulting in limited battery capacities and memory [15].

In this chapter, we propose a novel technique of protecting embedded software by obfuscating its control-flow, based on a key validation mechanism that internally generates and compares a sequence of keys with their expected values loaded from memory. The keys are execution trace dependent, meaning thereby that for different input parameters to the program, the sequence and values of keys involved in the validation process are different. The normal functionality of the program is enabled only after a successful validation process, otherwise, the program produces incorrect output. We take advantage of existing instructions in the program to hide the instructions dedicated to program modifications that implement this validation process. Such an obfuscation has a two-fold effect on the security of the software:

- The obfuscation prevents an adversary from stealing the software intellectual property and using it as a "black-box" functional module, because without figuring out the correct key sequence and values for each correct input key, it is not possible to make the program function correctly.

- It also prevents malicious modification of the program, because to effectively modify a program in order to cause functional failure in a way that evades the existing protection mechanisms, an adversary (or an automated tool) needs to interpret the actual functionality of the program.

In our work we assume an attack model where the adversary only has access to the program, and does not have access to the hardware system which is successfully running such an obfuscated software. Although techniques for program obfuscation (in particular control-flow obfuscation) [86–88], program validation [89, 90], software tamper-proofing [91, 92], secure co-processors [93] and program monitoring for secure execution [94, 95] have been proposed before, most of these techniques are heavily dependent on supporting hardware, especially computationally expensive cryptographic

techniques. This makes them difficult to apply in severely resource-constrained embedded systems. Also, many of them assume the presence of an unbreachable secure memory acting as a "root-of-trust" where the secure keys are stored, a concept whose feasibility is questionable for embedded devices with small form factors. The technique proposed in this chapter is not limited by these factors. In addition, it provides a second level of defense (described in section 7.5) by which even if an adversary breaks the security scheme, the ownership of the software can be proven by an *authentication* mechanism based on a *digital watermark*. Although we have implemented the proposed obfuscation technique to obfuscate assembly language programs, the technique can be extended easily to handle binary programs, as shown in Section 7.5.

The rest of the chapter is organized as follows: in Section 7.2, we describe the previous work on this topic. In Section 7.3, we describe the proposed key-based control flow obfuscation methodology; derive a quantitative metric to estimate the effectiveness of the scheme; estimate the computational overhead of implementing the proposed scheme, and describe the automated flow to implement the methodology for a given MIPS assembly language program [96]. We chose MIPS because while being full-featured, the relative simplicity of the MIPS instruction set simplifies the implementation of the proposed obfuscation algorithm. We present the simulation results for a suite of MIPS programs in Section 7.4. In Section 7.5 we describe the authentication capabilities that can be incorporated with obfuscation as a second level of defense, extension of the technique to binary programs, and discuss the applicability of the proposed approach in providing protection against software infection. Finally, we draw conclusions and indicate future research directions in Section 7.6.

7.2 Previous Work

The issue of software security in general and the special features and requirements of embedded software security in particular have been surveyed in [15, 97]. Purely software based mechanisms of protection have been described in [87, 88, 90, 98]. In

[90], "software guards" distributed throughout the program were proposed that were essentially inter-linked small blocks of check-sum of the program binary to validate and "repair" a program against illegitimate modifications. In [87, 88], several code obfuscation and watermarking techniques that rely on insertion of dummy code, loop unrolling and addition of bogus jump instructions. Similar techniques have been proposed in [86] to hinder the static disassembly of executables, and in [98] control-flow obfuscation of Java programs has been proposed. However, note that all these techniques of software obfuscation conform to the definition of obfuscation as proposed in [13, 88], where the obfuscated program and the original program have the same "black-box functionality". Such obfuscation techniques make it difficult to protect against piracy attempts where the adversary simply uses the obfuscated software, without bothering to modify or reverse-engineer its black-box functionality. Also, the value of such techniques is the matter of debate because as mentioned before, it has been theoretically proven that software obfuscation in this traditional sense does not exist [14]. In contrast, we modify both the structure and the functionality of the software under question. Other purely software-based approaches include self-modifying code [99] (code that generates other code at run-time), self-decryption of partially encrypted code at run-time [91, 100], and code redundancy and voting to produce "tamper-tolerant software" (along similar lines of hardware redundancy for fault tolerance) [92]. A general shortcoming of these approaches is that they do not scale well in terms of memory footprint and performance as the size of the program (or the part of the program to be protected) increases [90, 101]. We have addressed the question of scalability for our technique in Section 7.3.4.

Hardware-assisted software protection techniques have also been widely researched. Fully secure co-processor based techniques have been proposed [93] and are commercially available [102] where sensitive data transfer between the processor and the memory is in encrypted form, and the cryptographic keys are stored in tamper-resistant secure memory inside the co-processor. A special "Trusted Platform Module" (TPM) IC to verify the validity of the system against malicious modifications has been com-

mercially implemented in many computers [103]. Architectural support for software security has been explored in [25, 89, 94, 95]. In [89], an "execute-only memory" was proposed where besides support for encrypted code, the machine was separated into compartments such that a process in one compartment is unable to read from the other. Runtime execution monitoring to detect tampering was proposed in [94, 95] where cryptographically hashed values of code-blocks were compared with the expected values during program execution, and execution was stalled on any mismatch. Hardware-assisted control-flow obfuscation in conjunction with encrypted code was proposed in [25] which performs memory-address re-mapping by having an extra level of memory between the processor cache and the main memory. However, these techniques all require extensive special hardware support which might not be cost-effective for widespread use in embedded systems.

7.3 Methodology

7.3.1 Obfuscation Technique

In the case of hardware obfuscation as described in Chapters 2 and 3, we modify the state transition function to enable normal operation based on a key. A software program can be viewed as a set of state transition rules based on the *Turing machine* representation [104]. This gives us the motivation to obfuscate software through modification of the state transition rules by embedding a finite state machine into the program code. The fundamental idea of the technique proposed in this chapter is to validate the code during execution using a "challenge-response validation" protocol. The correct execution of the program is achieved only after the correct application of a set of input values, which constitute the *validation key sequence*. Such a key sequence based validation protocol has been proposed by us in the context of hardware intellectual property (IP) protection before [7, 38], where normal functionality of the circuit is enabled only after the successful application of an *initialization key sequence* on power-up. Essentially, the successful application of the *initialization key sequence*

takes the circuit from an *obfuscated mode* to a *normal mode*. However, there are the following significant differences between the approach proposed in this work and the hardware obfuscation approaches:

- The validation procedure proposed in this work is not executed only at the start of program execution. Instead, steps of the validation process are distributed throughout the program and operates concurrently with the rest of the program which implements the original functionality. The weakness of the validation mechanism being concentrated at only a single point within the program has been analyzed before, and is known as the "single point of failure" issue. With sophisticated program debuggers, attackers might be able to trace targeted parts of the program, pinpoint the code they need to compromise, and easily apply changes to the program to bypass the defense mechanism [90]. The distribution of the security mechanism to multiple locations within the program instead of being "lumped" at a single location was suggested to be a desirable feature in [90, 91]. Also, the technique of hiding important information in a program by breaking up the information into smaller fragments, deriving the values of the fragments through procedural execution, and then combining the results to derive the main result was shown in [105] to have greater security.

- Unlike the hardware obfuscation technique, the technique proposed in this chapter does not employ the same set of *validation keys* for each run of the program. Instead, the *validation key sequence* depends on the input argument of the program. This, in conjunction with the "distributed validation" mechanism, increases the level of protection (as shown analytically later) by making it more difficult for an adversary to identify the complete modification scheme for all possible inputs.

The keys of the *validation key sequence* are fetched from pre-determined memory locations and compared with the expected "golden" values. If all the values match, the program execution follows the normal control flow. However, if even a single com-

Algorithm 3 Procedure *Enumerate_Paths_Depth_First*

Enumerate all possible control-flow paths of a given assembly language program.

Inputs: Directed Acyclic Graph G corresponding to given assembly language program, *instr_stack*, *current_node*, *last_node*

Outputs: Set of edges ($\mathbb{E}$) with corresponding number of paths on which each edge lies

```
1: if curr_node ≠ Φ then
2:     push_on_stack(instr_stack, curr_node)
3:     if curr_node == last_node then
4:         e.pathcount ← (e.pathcount + 1) ∀ edge e on current path
5:     end if
6:     Enumerate_Paths_Depth_First(G, instr_stack, curr_node → left_child, last_instruction)
7:     Enumerate_Paths_Depth_First(G, instr_stack, curr_node → right_child, last_instruction)
8:     pop_from_stack(instr_stack)
9: else
10:    return
11: end if
```

parison fails, the program executes incorrect instructions which produces an incorrect result. The main challenge in implementing this technique is the hiding of the instructions dedicated to the validation procedure in the program. Although pre-determined values are fetched from pre-determined memory locations, the key and memory location values are not hard-coded in the program. Rather, they are derived during program execution, and different sets of values are derived depending on the input argument. This makes static analysis of the code and "program profiling" to discover the validation mechanism extremely challenging, because *each and every* validation step in the obfuscated program must be identified and neutralized to ensure that the program operates properly in *every situation*. The requirement of the predicates and variables involved in obfuscation to be *opaque*, i.e. difficult to be deduced by static analysis was pointed out in [88].

The obfuscation algorithm proceeds by finding all possible control-flow paths in the program depending on the input values, and then making modifications at optimal locations in the program, such that with a given code-size (and run-time) overhead, the modifications would have maximum overall effect. Algorithm-3 shows the pseudo-

Algorithm 4 Procedure *Find_Optimal_Modifications*

Find the optimal modification locations for a set of given control-flow paths and given number of modifications.

Inputs: Set of edges $\mathbb{E}$, modification pool $\mathbb{M}$, required number of modifications (M), minimum modification radius (r_{mod})

Outputs: List of modification locations in the program

1: Sort $\mathbb{E}$ based on number of paths on which each edge $e \in \mathbb{E}$ lies (i.e. $e.pathcount$)
2: $num\_mods \leftarrow 0$
3: **for all** edge $e \in \mathbb{E}$ **do**
4: $e.modified \leftarrow FALSE$
5: **end for**
6: /*Iterate over the ordered edges and make modifications based on r_{mod} constraint*/
7: **for** $i = 1$ to $|\mathbb{E}|$ and $num\_mods < M$ **do**
8: Set $\mathbb{E}_r = \{e_j \in \mathbb{E} : |e_i - e_j| \leq r_{mod}\}$ /* $|e_i - e_j|$ stands for the physical separation of the two edges */
9: **if** $e.modified == FALSE\ \forall e \in \mathbb{E}_r$ **then**
10: Choose previously unchosen $m \in \mathbb{M}$
11: Insert m on e_i
12: $e_j.modified \leftarrow$ TRUE $\forall e_j \in \mathbb{E}_r$
13: $num\_mods \leftarrow num\_mods + 1$ /*Update number of modifications*/
14: **end if**
15: **end for**

code for the algorithm to enumerate the paths of the program. The procedure assumes that the given MIPS program has been modeled as a "Directed Acyclic Graph" (DAG), with the edges forming loops removed. Each instruction of the program forms a node of the graph, and each node has one child (the non-branch instructions) or two children (the non-loop branch instructions). For each node, one among the children is always the next instruction. Note that a return from a procedure call is not treated as being part of a loop, because the "directed acyclic" nature of the graph is still maintained even if the edge denoting the return from the procedure call is retained. The procedure performs a depth-first search with a stack to keep track of the path traversed, and updates the number of paths on which edges of the graph lie every time the terminal node of the graph (i.e. the last instruction) has been reached. The last information is essential in determining optimal locations to perform modifications in the program, as described next.

Algorithm-4 shows the procedure to find the optimal locations to make M modifications for a given program. At first, the edges of the graph are ranked in descending order in terms of the number of paths on which the edges lie. Then, M modifications chosen greedily from the pool of modifications are inserted on the top-ranked edges, with the constraint that the modified edges are situated at least a pre-defined "modification radius" r_{mod} distance away from each other. If any edge connects two vertices which do not represent consecutive instructions in the program, jump instructions are used to connect the modification code block to the two vertices on the edge. The following points should be noted about this algorithm:

- Choosing the top-ranked edges ensures maximum effect of a single modification on multiple paths, while the r_{mod} constraint ensures that the modifications are not inserted too close to each other.

- The constraint r_{mod} determines the *average number of modifications per path*:

$$M_{av} = \frac{\sum_{i=1}^{|\mathbb{P}|} M_i}{|\mathbb{P}|} \tag{7.1}$$

  where $|\mathbb{P}|$ denotes the total number of paths in the program, M_i denotes the number of modifications lying on the i-th path, and $1 < M_{av} \leq M$. An increase in the value of M_{av} can be thought of to signify an increase in the security of the system, because more successful validations are required on average per path to make the program run successfully. Another metric that is determined by r_{mod} is the *average distance between modifications*. Let $\mathbb{E}_{mod}$ be the list of modified edges, ordered by their positions in the program, and M be the total number of modifications inserted. Then the *average distance between modifications* is given (for $M > 1$) by:

$$D_{av} = \frac{\sum_{i=1}^{M-1} |e_{i+1} - e_i|}{M-1} \tag{7.2}$$

  for $e_i \in \mathbb{E}_{mod}$, with $r_{mod} \leq D_{av} < \frac{N}{M-1}$, where N is the number of instructions in the program. If r_{mod} is small, say $r_{mod} = 1$, the minimum value possible,

the top M ranked edges would be chosen which would increase the value of M_{av}. However, on the flip-side, the value of D_{av} might decrease, meaning that the modifications would be placed too close to each other which puts them at the risk of being more identifiable to an adversary. Also, a higher value of M_{av} implies higher security; however it also implies an increase in the average execution time of the obfuscated program with respect to the original program. Hence, the parameter r_{mod} provides a degree of freedom to balance between the quantitative metrics M_{av} and D_{av}, and the performance of the program.

- This algorithm inserts the modifications at "preferred pseudo-random" locations, with preference being given to locations that would affect the maximum possible number of paths, while being "pseudo-random" in the sense that the modification locations are distributed throughout the program, through the effect of r_{mod}.

- If a modification is inserted between two instructions which are part of a loop, then the key-validation step would be repeated as many times as the loop repeated, even if the validation is successful. To avoid this, the modification should be such that any successful validation is "remembered", so that the next time the loop is executed, the validation mechanism is not exercised. This can be implemented easily by having a "flag" register and local jumps in the modification. We have elucidated this point with an example in the next sub-section.

To increase the level of security, the operations dedicated to deriving and comparing the keys of a sequence do not not appear in the order in which the keys are compared. For example, it might happen during the course of modifications that the "load" instruction which fetches the 4th key of a sequence from memory appears before the instruction that fetches the 3rd key of the sequence from memory. In the next sub-section, we give a complete example to elucidate the two algorithms described above.

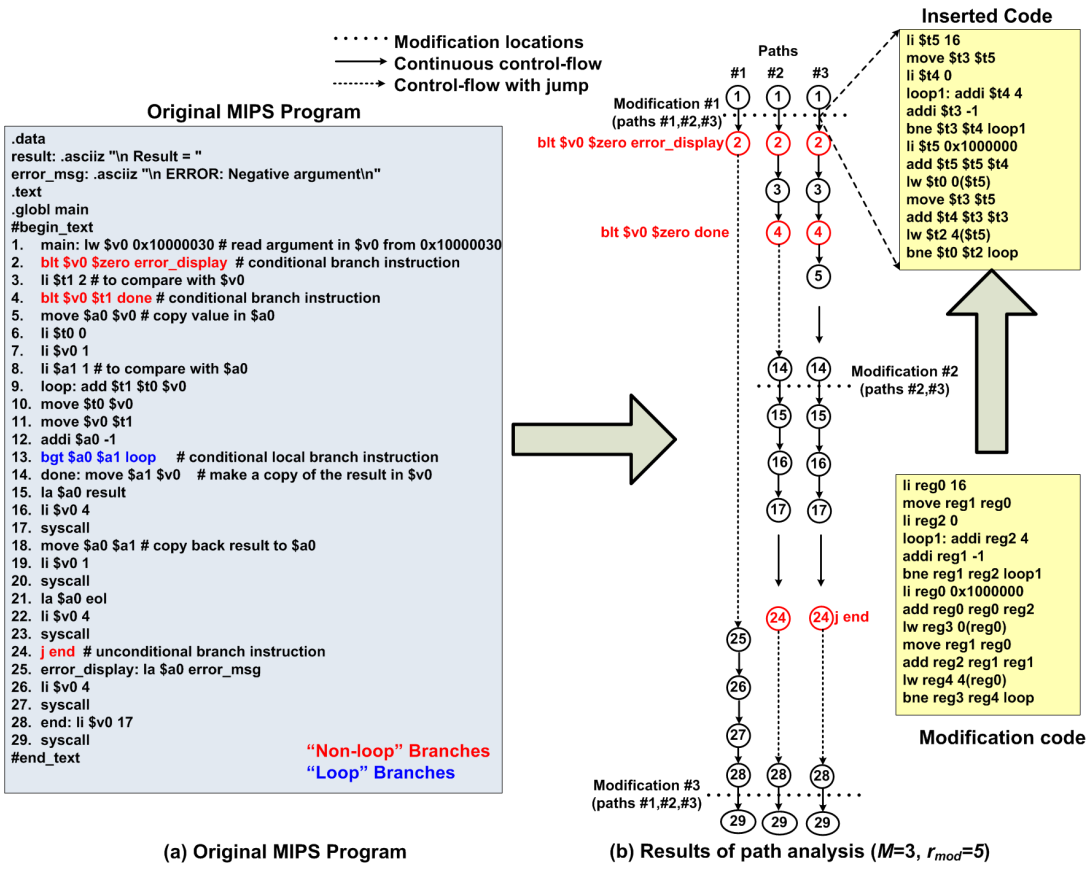


Fig. 7.1. Example of application of the proposed algorithm on a MIPS program to calculate the value of the n-th Fibonacci number for a given non-negative integer n.

7.3.2 Obfuscation Example

Fig. 7.1(a) shows an example MIPS assembly language program to calculate and display the value of the n-th Fibonacci number for a given non-negative integer n. The main part of the program to be modified occurs between the markers ***#begin_text*** and ***#end_text***, and the instructions between these two markers have been numbered for ease of understanding. Two types of branch instructions can be recognized - those which are not part of any loop in the program (instructions #2 and #4), which we term as the "non-loop branches", and those which are part of a loop (instruction

#13). The program is parsed between the two markers and transformed to a graph, with each instruction constituting a node in the graph. Only the non-loop branch instructions are recognized as true branch instructions, i.e. the loops are broken during analysis of the code. This makes it possible to model the control flow graph of the program as a directed acyclic graph, with each true branch instruction having at most two child nodes (two children for the conditional branches and one child for unconditional branches).

The feasible control paths of the program are then enumerated by analyzing the graph using Algorithm-3. The feasible paths for this program (paths #1, #2 and #3) are shown in Fig. 7.1(b), where each instruction has been represented by its serial number. Note that the different paths are followed depending on the value of the input argument n to the program - path-1 if $n < 0$, path-2 if $0 \leq n < 2$ and path-3 if $n \geq 2$. When Algorithm-4 is applied to find the optimal modification locations for $M = 3$ modifications and $r_{mod} = 5$, the modifications are placed between instructions 1 and 2 (modification #1), between 14 and 15 (modification #2) and between 28 and 29 (modification #3). Modification #1 and #3 affect all three paths, while modification #2 affects only paths 2 and 3. The average number of modifications is per path is thus $M_{av} = (3 + 3 + 2)/3 = 2.67$, which is less than the ideal value of $M_{av} = M = 3.00$. The average distance between modifications $D_{av} = 9.00$, while the ideal value is $\frac{N}{2} = \frac{29}{2} = 14.50$.

Note that Algorithm 4 implies that the first modification would always be inserted on one of the edges connecting the "root" node to the node corresponding to the first branch instruction in the program. This feature might make the first modification identifiable to an adversary performing static analysis. This issue can be handled by modifying the algorithm so that an exception is made about the position of the first modification, so that no modification appears between the "root" node and the first branch node.

An example modification has also been shown which is derived from the corresponding *modification pool* after binding the generic register names $reg0$, $reg1$ etc.

to actual resisters $\$t5$, $\$t3$, etc. As mentioned before, the register binding keeps the original functionality of the program functionally correct by a *lifespan analysis*. In the given case, registers $\$t0$ and $\$t2$ collect the input and golden values of the key from memory locations 0x10000040 and 0x10000044 respectively, and normal operation is allowed only if the fetched values match. In this particular case, incorrect operation is due to the fact that the register $\$t0$ contains an incorrect value (it should contain zero when the label *loop* is reached). Instead of an incorrect jump instruction on the failure of the comparison, incorrect dataflow operations followed by jumps might also be employed (e.g. putting an incorrect value to a register used later with its value not over-written in the original program). Dummy "register renaming" instructions might be utilized for the registers involved in the validation mechanism. This makes correct determination of the control flow dependent on analysis of the data flow, an operation whose complexity is *NP-hard* [101]. Similarly, dummy branch instructions based on opaque predicates might be added such that the branches always evaluate unidirectionally, to increase the complexity of static analysis of the program. Also, in case no registers are found free to be used bound to generic registers, the contents of these registers can be stored in memory, the registers used for validation purposes and after successful validation, the register contents can be restored. Note that as pointed out earlier, if any modification location had been chosen between instructions 9 and 13 (which form a loop), we would have to modify the program in such a way that after successful validation of the key, no further validation would be required till the loop finishes execution.

Next we qualitatively and quantitatively analyze the effectiveness of the proposed obfuscation scheme when applied to a given assembly language program.

7.3.3 Obfuscation Efficiency

Qualitative analysis

Qualitatively, the proposed obfuscation technique should be capable of resisting reverse-engineering attempts to discover the obfuscation scheme, thereby "filtering out" the original code from its obfuscated version. Reverse-engineering attempts are usually undertaken with the goal of discovering vulnerabilities, making unauthorized modifications, or stealing intellectual property [86]. Two main steps of reverse-engineering a binary executable can be recognized [86]:

- *disassembly* which attempts to produce equivalent assembly code from machine code, and,
- *decompilation*, which attempts to produce the equivalent high-level language program from the assembly code.

Each of the above two techniques comes in two different flavors - *static* and *dynamic*. The *static* techniques analyze the program without actually executing it, while *dynamic* techniques monitor and analyze the program during run-time. The previous work in preventing reverse-engineering as described in Section 7.2 attempts to increase the difficulty of either of these two steps. The novelty of the proposed technique is that it does not directly affect the difficulty of performing either of these two steps; instead, the principle of "execution trace dependent control-flow modification" ensures that even after successful *disassembly* and *decompilation*, the actual functionality of the program would be difficult to discover. The greater security of a variable key-based challenge-response process over one which employs a set of constant keys has been pointed out in [97].

Quantitative analysis

Although we have elucidated the qualitative effect of the proposed key-based obfuscation scheme through the above description and example, a theoretical analysis

to obtain a quantitative estimate of the security of the scheme is essential. We borrow the following metrics which have been previously proposed to estimate the success of a software obfuscation scheme [88]:

- *Potency*: the complexity in comprehending the obfuscated program compared to the unobfuscated one.
- *Resilience*: difficulty faced by an automatic de-obfuscator in breaking the obfuscation.
- *Stealth*: how well the obfuscated code blends in with the rest of the program, and
- *Cost*: how much computational overhead it adds to the obfuscated program.

A potent software obfuscation technique should provide high levels of *potency*, *resilience* and *stealth*, while incurring minimal *cost*. In particular, it should provide sufficient protection against both *dynamic* (i.e. run-time) and *static* program analyses. The technique automatically provides high levels of protection against dynamic analysis because of the fact that the particulars of the basic "challenge-response" mechanism of fetching the key from memory, comparing it with the golden key, and modifying the control-flow based on the result of the comparison, vary depending on the input arguments of the program. For example, it is highly unlikely that an adversary covertly observing the execution of an obfuscated code would observe the same control flow in two successive runs of the program. As a result, the number of "challenge-response" operation pairs, the memory access location, as well as the values against which the comparisons are made will vary from one run to the next. Because the input argument-space of most practical programs is larger beyond complete enumeration, hence, breaking the obfuscation scheme simply by observing the execution of the obfuscated program is practically infeasible. Hence, we concentrate on the protection provided by the proposed key-based obfuscation methodology against static code analysis efforts of an adversary.

Consider an assembly language program containing N instructions, to which n instructions are added to modify the control flow by the technique described above, as a result of which the code size increases to $(N + n)$. Let there be L "load" instructions in the original program, to which l "key load" instructions are added during modifications to increase the number of load instructions to $(L+l)$. Note that as pointed out earlier, these load instructions need not occur in the same order as the key comparison sequence. Similarly, let there be C "comparison-based branch" instructions in the original program to which c are added to bring the total number of branch instructions to $(C + c)$. To identify the modifications that have been made to the original program by static analysis of the obfuscated code, an adversary must perform the following steps:

- Identify the n instructions dedicated in modifying the original program, out of a total $(N + n)$ instructions in the obfuscated program. This is one out of $\binom{N+n}{n}$ possibilities.
- Identify the l "load" instructions dedicated to the obfuscation scheme out of the total $(L + l)$ "load" instructions, and from them determine the correct order in which the keys are collected from memory and compared to modify the control flow. Note that the adversary does not know a-priori the number of key comparisons for a given feasible control-flow path of a given program. Let M_{av} be the average number of modifications performed among all the feasible control-flow paths of the given program. Then, to break the scheme, the adversary has to make exactly one out of $\left[\sum_{i=1}^{\lceil M_{av} \rceil} \binom{L+l}{i} \times i!\right]$ choices to determine the correct number and sequence of keys to be applied.
- Identify the c "comparison-based branch instructions" dedicated in control-flow modification, from a total of $(C+c)$ such instructions in the obfuscated program.
- Identify the $(n - l - c)$ dataflow operations dedicated to obfuscate the code, from among the total $(N + n - L - C - l - c)$ in the obfuscated code.

Combining the three above factors, we propose the following quantitative metric to estimate the effectiveness of the proposed key-based obfuscation scheme:

$$M_{obf} = \frac{1}{\left[\sum_{i=1}^{\lceil M_{av} \rceil} \binom{L+l}{i} \times i!\right] \times \binom{C+c}{c} \times \binom{N+n-L-C-l-c}{n-l-c}} \tag{7.3}$$

Lower values of this metric implies higher levels of *potency*, *resilience* and *stealth*. To get an idea of the numerical order of this metric, consider the example shown in Fig. 7.1 and the portion of the code between the two markers ***#begin_text*** and ***#end_text***. Assuming the length of all modifications to be similar to the one shown, we have the values $\lceil M_{av} \rceil = 3$, $rmod = 5$, $N = 29$, $n = 3 \times 13 = 39$, $C = 3$, $c = 3 \times 2 = 6$, $L = 1$ and $l = 3 \times 2$. This gives the value $M_{obf} \approx 9.63 \times 10^{-20}$. In real-life applications, the value of this metric would be much smaller because of larger values of N and L, which in turn would allow larger values of n and l. Next we derive the computational overhead of implementing the proposed obfuscation scheme, which would estimate the *cost* aspect.

7.3.4 Computational Overhead of the Obfuscation Technique

Time complexity

The time complexity of the path enumeration step is essentially the time complexity of the dept-first traversal, which is $O(|\mathbb{V}| + |\mathbb{E}|)$, where $|\mathbb{V}|$ and $|\mathbb{E}|$ are the number of vertices and edges respectively in the graph [106]. However, note than in our particular case, $N - 1 \leq |\mathbb{E}| \leq 2N$, where $N = |\mathbb{V}|$ is the number of instructions in the block of the program to be obfuscated. The lower limit occurs when there is no non-loop branch instructions in the program, while the upper limit is because of the fact that no node in the graph has more than two children. However, note that an upper limit of $2N$ is overly pessimistic for real programs, because (approximately) only one in every seven instructions in real-life programs are branch instructions. Hence, the time complexity of the depth-first traversal step is $O(N)$. For the pro-

gram modification step, the time complexity is $O(|\mathbb{E}|)$, which because of the argument presented just now is $O(N)$. The time complexity of ranking the instructions based on the number of paths on which they lie is $O(N \log N)$, assuming an efficient sorting algorithms such as "Heapsort". Hence, the overall time-complexity of the obfuscation procedure is $O(N \log N)$.

To estimate the value of the average number of modifications made per path (M_{av}), it is essential to find the number of modifications made on every path individually, as well as the total number of paths. The total number of paths can be found during the first depth-first search. However, finding the number of modifications made individually on each path will require $O\left(\sum_{i=1}^{|\mathbb{P}|} |p_i|\right)$ steps, where $|\mathbb{P}|$ stands for the total number of paths, and $|p_i|$ is the length of the i-th path in the set of paths $\mathbb{P}$. If the total number of paths in a given program is unmanageably large, the program should be partitioned into smaller segments. Please note that this inconvenience is not part of the original algorithm - it is relevant only if somebody wishes to evaluate the parameter M_{av}.

Space complexity

The space complexity of the entire procedure is $O(N)$, the space required to store the information about edges in the program.

Next we describe the automated flow to obfuscate a given complete MIPS program by the application of the obfuscation methodology discussed above.

7.3.5 Automation of the Obfuscation Technique

The program obfuscation methodology described in Section 7.3 was implemented through an automated flow. The flow consists of the following components:

1. A Tool Command Language (TCL) script ***format_code*** that formats the input MIPS program to a form more amenable to the control-flow analysis.

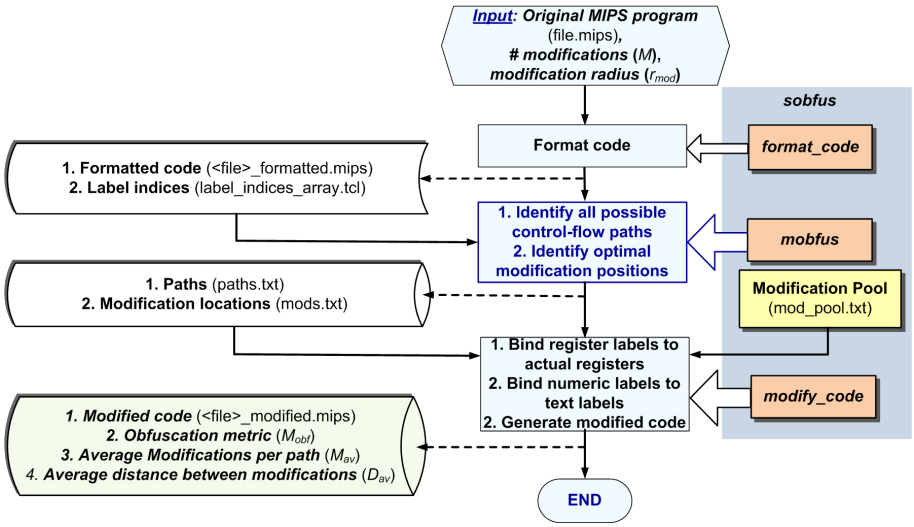


Fig. 7.2. Automation of the proposed obfuscation technique.

2. A C program ***mobfus*** that performs the control-flow analysis.

3. A TCL script ***modify_code*** that produces the modified MIPS program.

4. A top-level shell scripts ***sobfus*** that integrates all of the above programs.

Fig. 7.2 shows the automated flow. The top-level script ***sobfus*** accepts as input arguments the un-obfuscated MIPS program in a single file (let it be "file.mips"), the number of modifications (M) to be made and the *modification radius* (r_{mod}). M is estimated a-priori from the size of the modification code blocks in the modification pool, the size of the program, and the maximum code size overhead acceptable. ***sobfus*** invokes the TCL script ***format_code*** which formats the input code by removing all comments and blank lines and replacing all labels for branch instructions in the program by the corresponding destination line numbers. It produces a formatted version of the program in the file "file_formatted.mips", and a hash of the program labels and the corresponding line numbers in the file "label_indices_array.tcl". ***sobfus*** then calls the C program ***mobfus*** which enumerates all the possible control-flow paths in the program using Algorithm-3, and finds the optimal modification loca-

Table 7.1
Functionality of the Programs listed in Table 7.2

Program	**Functionality**
TokenQuest.mips	One player adventure game
hanoi.mips	Recursive solution of the "Tower of Hanoi" problem
MD5.mips	MD5 hashing of a given ASCII text file
connect4.mips	Two player "Four in a Line" game
DES.mips	Digital Encryption Standard (DES) encrypter/decypter (for ASCII text files)
sudoku.mips	*Sudoku* puzzle
ID3Ediror.mips	Reading and editing of ID3 tag information in MP3 music files
string.mips	MIPS implementation of the functions of the C standard header "string.h"
cipher.txt	Various cipher techniques for ASCII text
decoder.mips	MP3 music format decoder

tions using Algorithm-4. It reports the enumerated paths in the file "paths.txt" and the modification locations in the file "mods.txt". ***sobfus*** then invokes the TCL script ***modify_code*** which finally produces the obfuscated program in the file "file_obfuscated.mips" by using the modification code blocks provided in the file "mod_pool.txt", and binds the register mnemonics to registers available at a given point in the program (as described in Section 7.3.1 and elucidated in Section 7.3.2). It also produces an estimate of the obfuscation metric M_{obf} according to eqn. 7.3, and values for the metrics M_{av} and D_{av}.

In the next section, we present simulation results for the application of the above technique to a set of MIPS assembly language programs.

7.4 Results

7.4.1 Setup

The MIPS programs were collected from different MIPS programming projects performed by undergraduate students taking the EECS-314 course in the Electrical Engineering and Computer Science Department of Case Western Reserve University. The size of the programs varied 109 to 21024 MIPS assembly language instructions, and the programs were all interactive in nature. The functionality of the programs are listed in table 7.1. The functionality of the original and the obfuscated versions of all the two programs were verified using the *SPIM* simulator [107]. The program obfuscation methodology described in Section 7.3.5 was implemented and the programs were simulated on a Linux workstation with 4GB of main memory and a 2GHz quad-core processor.

7.4.2 Effect of Variation of the *Modification Radius* (r_{mod})

We investigated the effect of variation of the *modification radius* (r_{mod}) on the average modifications per path (M_{av}) and the average distance between modifications (D_{av}) for the $N = 270$ instruction program *connect4.mips*. The number of modifications (M) was set at 3, and r_{mod} was varied between 1 and 80. Fig. 7.3 shows the plots of M_{av} and D_{av} vs. r_{mod}. The values for M_{av} were normalized with respect to its value at $r_{mod} = 1$ (the minimum possible value of r_{mod}). The trends are as expected, with M_{av} decreasing with r_{mod} and D_{av} increasing with r_{mod}. Note that the metrics M_{av} and D_{av} satisfy the constraints $1 < M_{av} \leq M$ and $r_{mod} \leq D_{av} < \frac{N}{M-1}$, as stated in Section 7.3.1.

7.4.3 Obfuscation Results

Table 7.2 shows the effects of applying the proposed application technique on the MIPS program suite, at a *modification radius* ($r_{mod} = 50$), with a 10% target code-

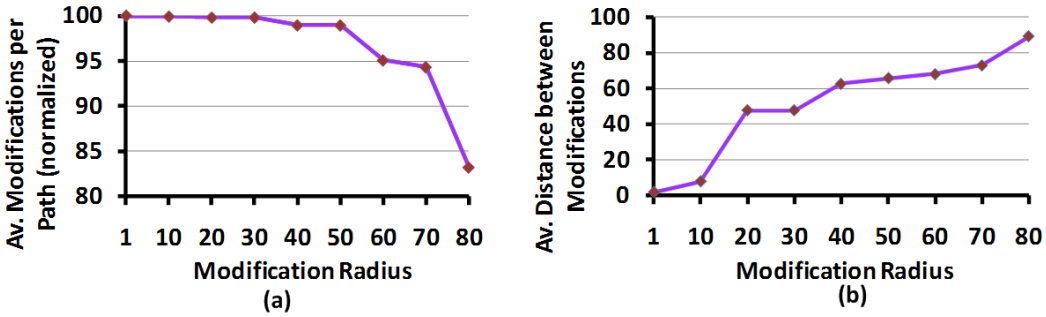


Fig. 7.3. Variation of (a) average modification per path (M_{av}) and (b) the average distance between modifications (D_{av}) vs. the modification radius (r_{mod}), in the program *connect4.mips*, for $M = 3$ modifications.

size overhead. For the largest program *decoder.mips*, only 1000 paths were considered to keep the memory requirement manageable, and r_{mod} was set to 500. As is evident from the obtained M_{obf} values, the proposed technique can provide high levels of protection at a nominal code-size overhead of 10%. Note that in larger programs and in programs with higher number of "load" and "branch" instructions, the effectiveness of the technique increases.

7.4.4 Overhead Results

Table 7.3 shows the code-size overhead of the obfuscated program (with respect to the original program), the CPU time and average increase in execution cycles to implement algorithms 3 and 4. The average increase in execution time was estimated by calculating the average increase in execution cycles per modification, and then multiplying the quantity with the average number of modifications per path. The CPU time has a strong correlation to the number of paths in the program, and a weaker correlation to the program size. These trends are consistent with the analysis of overhead requirements in Section 7.3.4.

Table 7.2
Program Obfuscation Efficiency for a Targeted 10% Code-size Overhead at a Modification Radius $r_{mod} = 50$

Program	Program Parameters†								Obfuscation Efficiency		
	N	C	L	$\lvert\mathbb{P}\rvert$	M	n	c	l	$\boldsymbol{M_{obf}}$	$\boldsymbol{M_{av}}$	$\boldsymbol{D_{av}}$
TokenQuest.mips	109	19	14	11	2	18	3	3	1.09e-20	1.55	95.0
hanoi.mips	132	20	40	169	2	16	3	3	1.43e-19	1.91	67.0
MD5.mips	250	41	35	114	4	26	5	5	6.33e-33	3.67	65.33
connect4.mips	270	72	37	4146	4	26	5	5	1.30e-33	3.47	89.33
DES.mips	372	43	64	5241	6	34	7	9	1.54e-40	5.31	68.00
sudoku.mips	436	110	43	111113	8	41	9	11	2.66e-49	6.76	58.29
ID3Editor.mips	878	160	134	98724	12	89	16	19	1.71e-106	5.66	79.45
string.mips	876	156	224	111075	12	89	16	19	4.42e-103	10.90	60.55
cipher.mips	1956	231	218	150129	27	188	35	43	1.65e-222	26.23	75.12
decoder.mips‡	21024	174	231	1000‡	27	188	35	43	$<10^{-400}$	13.50	502.00‡

†The meaning and significance of these parameters are as described in Sections 7.3.3 and 7.3.4.

‡Only 1000 paths were enumerated, and r_{mod} was set to 500.

7.5 Discussions

7.5.1 Authentication Capabilities

The proposed technique also has the capability of providing *authentication* capabilities as a second line of defense. This second life of defense becomes valuable if an adversary has been able to break the key-comparison based validation scheme. The original owner of the software might decide on the snapshot of the processor state for a particular input parameter and a particular mix of correct and incorrect keys as the *digital watermark* [87] to establish the original ownership of the program in the court of law. It is extremely difficult for an adversary to detect such a digital watermark

Table 7.3
Overheads for the Obfuscation Technique (with parameters of Table 7.2)

Program	Overheads		
	Code-size (%)	**CPU time (s)**	**Average Increase in Execution Cycles**
TokenQuest.mips	18.85	0.10	17.83
hanoi.mips	12.12	0.40	20.06
MD5.mips	10.40	0.90	31.20
connect4.mips	9.63	1.00	29.50
DES.mips	9.14	2.00	41.60
sudoku.mips	9.40	66.00	48.17
ID3Editor.mips	10.14	112.00	54.71
string.mips	10.16	217.00	105.37
cipher.txt	10.61	1474.00	241.90
decoder.mips	0.89%	1840.00	124.50

and change it, without the knowledge of exactly which processor state was chosen by the software owner as the watermark.

7.5.2 Application to Obfuscation of Program Binaries

Although our implementation of the obfuscation scheme is for MIPS assembly language programs, the technique can be easily extended to handle binary executables. To apply the obfuscation technique, one would need to disassemble the equivalent assembly language program from a given binary, apply the proposed obfuscation technique, and then again convert it back to the binary form. The details of the implementation would vary slightly depending on the instruction set of the processor for which the program was compiled; however, the algorithms would not change. Disassembly and de-compilation of binary code to assembly language code or to high-level programming language code is not very difficult, and free tools are available online [108] to serve the purpose.

7.5.3 Application of Software Protection in Protection against Malicious Modifications

Obfuscation can potentially play a major role in protecting software from malicious non-self-replicating modifications ("software Trojan horses") [101]. Any effective malicious code injection evading detection requires in-depth understanding of the behavior of the victim program. However, if the program to be infected is obfuscated, an effective virus insertion might be resisted. [101] explored the possible applicability of the "memory re-mapping" technique in providing protection against software Trojan horses. Similarly, the execution trace dependent control-flow obfuscation technique proposed in this chapter might also help to resist software Trojan horses. It might happen that failing to understand the actual control-flow of the program, the malicious code is inserted in a part of the program that is never executed if the key-sequence validation step is successful. In this case, the Trojan horse would be rendered "benign" in a legal copy of the program run with the proper validation keys, i.e. it would be unable to express its malicious effect during program execution in spite of being physically present.

7.5.4 Intelligent Attack Scenarios

The obfuscation metric (M_{obf}) described by eqn. (7.3) shows that it is extremely difficult for an adversary to reverse-engineer the program based on a "random-choice" based analysis. However, in reality, the adversary can perform input dependent control flow analysis of the obfuscated program similar to the technique that was used for to obfuscate the program in the first place. Implementing suitable defense mechanisms against these types of intelligent attacks is a major future work. One way by which the attacker's task can be made more difficult is to insert modification code blocks so that they result in numerous forward branches to increase the number of paths in the program. Similarly, having many dummy and "honey pot" type of instructions which will attract the adversarys attention would prove to be helpful. Note

that the dummy and "honey pot" instructions will only result in the increase of the program size; on successful key matching, they would never be executed and hence they would never adversely affect the performance of the program.

7.5.5 Automatic Generation of Modification Code

In this work, the modification code to be inserted have been chosen randomly from a given pre-designed pool of such blocks of code, and after the modifications have been made, the value of M_{obf} has been calculated. Instead, the technique can be made more sophisticated by having a mechanism, where, based on a target value of a suitable M_{obf} and the allowable code size overhead, the modification code would be generated automatically. The code generation mechanism should analyze the unobfuscated program structure to generate effective modification code that can be well-hidden in the program.

7.6 Summary

Increasing functionality of embedded systems is enabled by greater complexity of software that runs on them. However, embedded software are becoming increasingly vulnerable to piracy and malicious modifications. Severe hardware and energy resource constraints of embedded devices often limit the applicability of complex hardware and software protection approaches. We have proposed an "execution trace dependent control-flow obfuscation" technique which requires the application of an input-dependent set of validation keys to make the software function properly. The validation mechanism is implemented by distributing the validation code all over the program using an algorithm that balances the code overhead and proximity of the modifications. We have theoretically analyzed the level of security obtainable from this scheme, and the associated computational overhead. Application of the algorithm on a suite of randomly selected MIPS programs resulted in high levels of security at nominal code-size and acceptable computational overhead. The technique is highly

scalable and can be applied to arbitrarily large programs by appropriate program partitioning.

8. CONCLUSION

Economic reasons as well as the complexity of modern IC design process, dictate the participation of several external agents in modern IC design and manufacturing. IC design houses are becoming increasingly dependent on hardware IP modules and CAD software tools licensed from external vendors. Due to the spiralling cost of maintaining a fabrication facility, most semiconductor companies follow a "fabless" business model where the design is "taped-out" and the IC is manufactured in off-shore fabrication facilities by companies specializing in semiconductor manufacturing. These practices have reduced the level of control that IC design houses used to exercise over the ICs they design and manufacture. An IC can be maliciously modified at different stages of the design and manufacturing flow by the insertion of so-called "hardware Trojans". In addition, "clones" of an IC might be manufactured illegally in off-shore fabs resulting in loss of market share for the IC design house. Also, "Design for Testability" practices provide new opportunities for the adversaries to discover secret information from secure ICs. On the other hand, release of IPs to unscrupulous IC design houses makes the IP vendors vulnerable to IP piracy through illegal copying.

Obfuscation is a technique that makes comprehending and reverse-engineering a design difficult. In this research, we have developed novel hardware design techniques implementing key-based *design obfuscation* that effectively counter these threats. Side by side, an enhanced IC design methodology is developed where the security aspect is seamlessly integrated with the traditional design steps through EDA software tool support. Design obfuscation for both gate-level and RTL design descriptions has been applied successfully to develop a piracy-proof SoC design methodology. These obfuscation techniques have then been applied to resist Trojan insertion and to facilitate Trojan detection. A key-based secure scan chain design technique to prevent the leakage of secret information from secure ICs has also been proposed. Moreover, we have

presented possible extension of the proposed approach to embedded software obfuscation, thus providing an integrated low-cost security solutions to resource-constrained embedded systems.

We have shown that the proposed obfuscation procedure can also be used to realize a low-cost and robust authentication feature by introducing appropriate modification in the state transition function. Obfuscation and authentication go hand in hand; thus, effectiveness of an authentication technique can be substantially increased in presence of obfuscation. Obfuscation can also be used in combination with existing "design for security" approaches to enhance the overall security. However, obfuscation involves additional design overhead as well as modification of the existing design flow and requires application of an enabling key for normal operation. It is important to minimize the impact of obfuscation on designer, test engineer as well as end-user, while adopting the proposed approach for a target application. We have shown through simulation results that the obfuscation based techniques proposed in this research can be implemented at low hardware and/or computational overheads, with minimal impact on the end-user experience. To make the approach scalable, we have considered the issues associated with application of the proposed obfuscation process to complex SoCs.

Future work would explore the applicability of obfuscation techniques to higher levels of design abstraction such as "Electronic System Level" (ESL) and *SystemC*; protection of memory contents through possible obfuscation of address and data lines; design of hardware infrastructure to implement the "challenge-response" protocol for embedded software security, and ways to increase the security against *insider's attack*, i.e. attacks where an associate of the design house itself is the adversary. Analysis of the effect of design obfuscation on circuit and system testability and reliability can also be investigated.

REFERENCES

[1] Semiconductor Industry Association, "Industry factsheet." `http://www.sia-online.org/cs/industry_resources/industry_fact_sheet`, 2009.

[2] Semiconductor Industry Association, "Economy factsheet." `http://www.sia-online.org/cs/economy`, 2009.

[3] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 8, pp. 114–117, Apr. 1965.

[4] Global Semiconductor Alliance, "GSA members." `http://www.gsaglobal.org/membership/members/index.asp`, 2009.

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

[6] DARPA, "TRUST in Integrated Circuits (TIC) - Proposer Information Pamphlet." `http://www.darpa.mil/MTO/solicitations/baa07-24/index.html`, 2007.

[7] R. S. Chakraborty and S. Bhunia, "*HARPOON*: a SoC design methodology for hardware protection through netlist level obfuscation," *IEEE Transactions on CAD*, vol. 28, pp. 1493–1502, Oct. 2009.

[8] J. Roy, F. Koushanfar, and I. L.Markov, "*EPIC*: ending piracy of integrated circuits," in *DATE'08: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1069–1074, 2008.

[9] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware Trojan: threats and emerging solutions," in *HLDVT'09: Proceedings of the International High Level Design Validation and Test Workshop*, pp. 166–171, Nov. 2009.

[10] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "*MERO*: a statistical approach for hardware Trojan detection using logic testing," *Lecture Notes on Computer Science*, vol. 5737, pp. 396–410, Sept. 2009.

[11] B. Yang, K. Wu, and R. Karri, "*Secure Scan*: a design-for-test architecture for crypto chips," *IEEE Transactions on CAD*, vol. 25, pp. 2287–2293, Oct. 2006.

[12] A. Kerckhoff, "La cryptographie militaire," *Journal des Sciences Militaires*, vol. IX, pp. 5–38, Jan. 1883.

[13] B. Barak, "Can we obfuscate programs?." `http://www.math.ias.edu/~boaz/Papers/obf_informal.html`, 2009.

[14] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO '01: Proceedings of the Cryptology Conference on Advances in Cryptology*, pp. 1–18, 2001.

[15] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: design challenges," *ACM Transactions on Embedded Computing Systems*, vol. 3, pp. 461–491, Aug 2004.

[16] E. Castillo, U. Meyer-Baese, A. Garcia, L. Parilla, and A. Lloris, "IPP@HDL: efficient intellectual property protection scheme for IP cores," *IEEE Transactions on VLSI*, vol. 16, pp. 578–590, May 2007.

[17] D. C. Musker, "Protecting and exploiting intellectual property in electronics," in *IBC '98: Proceedings of the IBC Conference*, 1998.

[18] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Constraint-based watermarking techniques for design IP protection," *IEEE Transactions on CAD*, vol. 20, pp. 1236–1252, Oct. 2001.

[19] F. Koushanfar and G. Qu, "Hardware metering," in *DAC '01: Proceedings of the Design Automation Conference*, pp. 490–493, 2001.

[20] Semantic Designs, "Thicket^{TM} family of source code obfuscators." `http://www.semdesigns.com`, 2010.

[21] T. Batra, "Methodology for protection and licensing of HDL IP." `http://www.us.design-reuse.com/news/?id=12745\&print=yes`, 2005.

[22] R. Goering, "Synplicity initiative eases IP evaluation for FPGAs." `http://www.scdsource.com/article.php?id=170`, 2008.

[23] Xilinx Corporation, "Xilinx IP evaluation." `http://www.xilinx.com/ipcenter/ipevaluation/index.htm`, 2009.

[24] ARM, "ARM-the architecture for the digital world." `www.arm.com`, 2009.

[25] X. Zhuang, T. Z. Hsien-Hsin, S. Lee, and S. Pande, "Hardware assisted control flow obfuscation for embedded processors," in *Proceedings of the International Conference of Compilers, Architecture and Synthesis for Embedded Systems*, pp. 292–302, 2004.

[26] M. Wirthlin and B. McMurtrey, "IP delivery for FPGAs using Applets and JHDL," in *DAC'02: Proceedings of the Design Automation Conference*, pp. 2–7, 2002.

[27] M. Slater, F. Faggin, M. Shima, and R. Ungermann, "Zilog oral History panel on the founding of the company and the development of the Z80 microprocessor." `http://archive.computerhistory.org/resources/text/Oral_History/Zilog_Z80/102658073.05.01.pdf`, 2007.

[28] Y. Alkabani, F. Koushanfar, and M. Potkonjak, "Remote activation of ICs for piracy prevention and digital right management," in *ICCAD '07: Proceedings of the International Conference on CAD*, pp. 674–677, 2007.

[29] R. S. Chakraborty and S. Bhunia, "Hardware protection and authentication through netlist-level obfuscation," in *ICCAD '08: Proceedings of the International Conference on CAD*, pp. 674–677, 2008.

[30] F. Wang, "Formal verification of timed systems: a survey and perspective," *Proceedings of the IEEE*, vol. 92, pp. 1283–1305, Aug. 2004.

[31] W. Moore and P. Kayfes, "Us Patent 7213142 – system and method to initialize registers with an EEPROM stored boot sequence." `http://www.patentstorm.us/patents/7213142/description.html`, 2007.

[32] ISCAS, "The ISCAS-89 benchmark circuits." `http://www.fm.vslib.cz/~kes/asic/iscas/`, 2009.

[33] Opencores, "The 128-bit Advanced Encryption Standard IP core." `http://www.opencores.org/projects.cgi/web/aes_core/`, 2009.

[34] H. Yotsuyanagi and K. Kinoshita, "Undetectable fault removal of sequential circuits based on unreachable states," in *VTS '98: Proceedings of the VLSI Test Symposium*, pp. 176–281, 1998.

[35] M. Clendenin, "Chinese firms favoring soft ip over hard cores." `http://www.eetasia.com/ART_8800440032_480100_NT_ac94df1c.HTM`, 2009.

[36] D. Schwaderer and P. Martin, "Solving SoC shared memory resource challenges." `http://www.design-reuse.com/articles/5816/solving-soc-shared-memory-resource-challenges.html`, 2010.

[37] R. S. Chakraborty and S. Bhunia, "Security through obscurity: an approach for protecting Register Transfer Level hardware IP," in *HOST'09: Proceedings of the International Workshop on Hardware-oriented Security and Trust*, pp. 96–99, 2009.

[38] R. S. Chakraborty and S. Bhunia, "RTL hardware IP protection using key-based control and data flow obfuscation," in *VLSID'10: Proceedings of the International Conference on VLSI Design*, pp. 405–410, 2010.

[39] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[40] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993–1002, 1996.

[41] D. Johnson, "The NP-completeness column," *ACM Transactions on Algorithms*, vol. 1, no. 1, pp. 160–176, 2005.

[42] D. Anastasakis, R. Damiano, H. Ma, and T. Stanion, "A practical and efficient method for compare-point matching," in *DAC '02: Proceedings of the Design Automation Conference*, pp. 305–310, 2002.

[43] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, R. B. P.R. Stephan, and A. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, University of California - Berkeley Electrical Engineering and Computer Science Department, 1992.

[44] E. Electronic, "AES (Rijndael) IP-cores." `http://www.erst.ch/download/aes_standard_cores.pdf`, 2009.

[45] H. Technology, "Full datasheet AES-CCM core family for Actel FPGA." `http://www.actel.com/ipdocs/HelionCore_AES-CCM_8bit_Actel_DS.pdf`, 2009.

[46] S. Wong and S. Hock, "US Patent 5943283 – address scrambling in a semiconductor memory," August 1999.

[47] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *SP'07: Proceedings of the IEEE Symposium on Security and Privacy*, pp. 296–310, 2007.

[48] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSID'04: Proceedings of the International Conference on VLSI Design*, pp. 605–611, 2004.

[49] S. Adee, "The hunt for thr kill switch," *IEEE Spectrum*, vol. 45, pp. 34–39, May 2008.

[50] J. A. Roy, F. Kaushanfar, and I. L. Markov, "Extended abstract: circuit CAD tools as a security threat," in *HOST'08: Proceedings of the International Workshop on Hardware-oriented Security and Trust*, pp. 61–62, 2008.

[51] R. S. Chakraborty and S. Bhunia, "Security against hardware Trojan through a novel application of design obfuscation," in *ICCAD '09: Proceedings of the International Conference on CAD*, pp. 113–116, 2009.

[52] L. Lin, W. Burleson, and C. Parr, "*MOLES*: Malicious off-chip leakage enabled by side-channels," in *ICCAD'09: Proceedings of the International Conference on CAD*, pp. 117–122, 2009.

[53] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards Trojan-free trusted ICs: problem analysis and detection scheme,," in *DATE'08: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1362–1365, 2008.

[54] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *HOST'09: Proceedings of the International Workshop on Hardware-oriented Security and Trust*, pp. 50–57, 2009.

[55] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware Trojans," in *ICCAD'08: Proceedings of the International Conference on CAD*, pp. 632–639, 2008.

[56] M. Banga and M. S. Hsiao, "A region based approach for the identification of hardware Trojans," in *HOST'08: Proceedings of the International Workshop on Hardware-oriented Security and Trust*, pp. 40–47, 2008.

[57] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *HOST'08: Proceedings of the International Workshop on Hardware-oriented Security and Trust*, pp. 51–57, 2008.

[58] L. W. Kim, J. D. Villasenor, and C. K. Koc, "A trojan-resistant system-on-chip bus architecture," in *Proceedings of MILCOM'09*, 2009.

[59] R. S. Chakraborty, S. Paul, and S. Bhunia, "On-demand transparency for improving Hardware Trojan detectability," in *HOST'08: Proceedings of the International Workshop on Hardware Oriented Security and Trust*, pp. 48–50, 2008.

[60] M. Brzozowski and V. N. Yarmolik, "Obfuscation as intellectual rights protection in VHDL language," in *CISIM'07: Proceedings of the International Conference on Computer Information Systems and Industrial Management Applications*, pp. 337–340, 2007.

[61] F. N. Najm, "Transition Density: a new measure of activity in digital circuits," *IEEE Transactions on CAD*, vol. 14, pp. 310–323, Feb. 1993.

[62] M. G. Xakellis and F. N. Najm, "Statistical estimation of the switching activity in digital circuits," in *DAC '94: Proceedings of the Design Automation Conference*, pp. 728–733, 1994.

[63] T. Chou and K. Roy, "Accurate power estimation of CMOS sequential circuits," *IEEE Transactions on VLSI*, vol. 4, pp. 369–380, Sept. 1996.

[64] I. Systems, "Concorde – fast synthesis." `http://www.interrasystems.com/eda/eda_concorde.php`, 2009.

[65] M. E. Amyeen, S. Venkataraman, A. Ojha, and S. Lee, "Evaluation of the quality of N-detect scan ATPG patterns on a processor," in *ITC'04: Proceedings of the International Test Conference*, pp. 669–678, 2004.

[66] I. Pomeranz and S. M. Reddy, "A measure of quality for n-detection test sets," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1497–1503, 2004.

[67] M. Geuzebroek, J. T. van der Linden, and A. J. van de Goor, "Test point insertion that facilitates ATPG in reducing test time and data volume," in *ITC'02: Proceedings of the International Test Conference*, pp. 138–147, 2002.

[68] B. Mathew and D. G. Saab, "Combining multiple DFT schemes with test generation," *IEEE Transactions on CAD*, vol. 18, no. 6, pp. 685–696, 1999.

[69] S. Paul, R. S. Chakraborty, and S. Bhunia, "*VIm-Scan*: A low overhead scan design approach for protection of secret key in scan-based secure chips," in *VTS'07: Proceedings of the VLSI Test Symposium*, pp. 455–460, May 2007.

[70] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and performance testing of a 2.29-GB/s Rijndael processor," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 569–572, Mar 2003.

[71] J. Goodman and A. Chandrakasan, "An energy-efficient IEEE 1363-based reconfigurable public-key cryptography processor," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 1808–1820, Nov 2001.

[72] S. Mangard, M. Aigner, and S. Dominikus, "A highly regular and scalable AES hardware architecture," *IEEE Transactions on Computers*, vol. 52, pp. 483–491, Apr 2003.

[73] J. Lee, M. Tehranipoor, and J. Plusquellic, "A low-cost solution for protecting IPs against scan-based side-channel attacks," in *VTS'06: Proceedings of the VLSI Test Symposium*, pp. 94–99, 2006.

[74] Maestra, "Maestra comprehensive guide to satellite TV testing," 2002.

[75] O. Kömmerling and M. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proceedings of the USENIX Workshop on Smartcard Technology*, pp. 9–20, 1999.

[76] R. J. Easter, E. W. Chencinski, E. J. D'Avignon, S. R. Greenspan, W. A. Merz, and C. D. Norberg, "S/390 parallel enterprise server CMOS cryptographic coprocessor," *IBM Journal of Research and Development*, vol. 43, pp. 761–776, Sep 1999.

[77] K. Hafner, H. Ritter, T. M. Schwair, S. Wallstab, M. Deppermann, J. Gessner, S. Koesters, W. Moeller, and G. Sandweg, "Design and test of an integrated cryptochip," *IEEE Design and Test of Computers*, vol. 8, pp. 6–17, Oct 1991.

[78] R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner, "A 177 Mb/s VLSI implementation of the International Data Encryption Algorithm," *IEEE Journal of Solid-State Circuits*, vol. 29, pp. 303–307, Mar 1994.

[79] D. Hely, M.-L. Flottes, F. Bancel, B. Rouzeyre, N. Berard, and M. Renovell, "Scan design and secure chip," in *IOLTS'04: Proceeding of the International On-Line Testing Symposium*, pp. 219–224, 2004.

[80] J. Lee, M. Tehranipoor, C. Patel, and J. Plusquellic, "Securing scan design using lock and key technique," in *DFT'05: International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 51–62, 2005.

[81] D. Saab, Y. Saab, and J. Abraham, "Automatic test vector cultivation for sequential vlsi circuits using genetic algorithms," *IEEE Transactions on CAD*, vol. 15, pp. 1278–1285, Oct 1996.

[82] B. Arslan and A. Orailoglu, "CircularScan: a scan architecture for test cost reduction," in *DATE'04: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1290 – 1295 Vol.2, 2004.

[83] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing*. Kluwer, 2000.

[84] J. Turley, "The two percent solution." `http://www.embedded.com/story/OEG20021217S0039`, 2009.

[85] L. Gwennap and J. Byrne, *A Guide to High-Speed Embedded Processors*. The Linley Group, 2008.

[86] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *CCS'03: Proceedings of the Conference on Computer and Communications Security*, pp. 290–299, 2003.

[87] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation – tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, pp. 735–746, Aug 2002.

[88] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *POPL'98: Proceedings of the Symposium on Principles of Programming Languages*, pp. 184–196, 1998.

[89] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, pp. 168–177, Nov 2000.

[90] H. Chang and M. J. Atallah, "Protecting software code by guards," *Lecture Notes on Computer Science*, vol. 2320, pp. 160–175, 2002.

[91] D. Aucsmith, "Tamper resistant software: an implementation," *Lecture Notes on Computer Science*, vol. 5824, pp. 125–139, 2009.

[92] M. Jakubowski, C. Saw, and R. Venkatesan, "Tamper-tolerant software: modeling and implementation," *Lecture Notes on Computer Science*, vol. 5824, pp. 125–139, 2009.

[93] S. White and L. Comerford, "ABYSS: an architecture for software protection," *IEEE Transactions on Software Engineering*, vol. 16, pp. 619–629, Jun 1990.

[94] D. Arora, S. Ravi, A. Raghunathan, and N. Jha, "Hardware-assisted runtime monitoring for secure program execution on embedded processors," *IEEE Transactions on VLSI*, vol. 14, pp. 1295–1308, Dec 2006.

[95] A. Fiskiran and R. Lee, "Runtime execution monitoring (REM) to detect and prevent malicious code execution," in *ICCD'04: Proceedings of the International Conference on Computer Design*, pp. 452–457, 2004.

[96] D. Patterson and J. Hennessy, *Computer Organization and Design: the hardware/software interface*, ch. Appendix A. Morgan Kaufmann Publishers, 4th ed., 2009.

[97] R. Dube, *Hardware-based Computer Security Techniques to Defeat Hackers*, ch. 5. John Wiley and Sons, 2008.

[98] T. W. Hou, H. Y. Chen, and M. H. Tsai, "Three control flow obfuscation methods for Java software," *IEE Proceedings*, vol. 153, pp. 80–86, Apr 2006.

[99] H. Joepgen and S. Krauss, "Software by means of the protprog method," *Electronik*, vol. 42, pp. 52–56, Aug 1993.

[100] A. Schulman, "Examining the Windows AARD detection code," *Dr. Dobb's Journal*, vol. 18, pp. 42–89, Sep 1993.

[101] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey." `http://pages.cs.wisc.edu/~arinib/writeup.pdf`, 2009.

[102] D. Semiconductor, "Dallas DS5240 Secure Microcontroller." `http://datasheets.maxim-ic.com/en/ds/DS5240.pdf`, 2009.

[103] Trusted Computing Group, "Trusted Platform Module: Design Principles." `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2009.

[104] B. J. Copeland(ed.), *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life Plus the Secrets of Enigma*. Oxford University Press, 2004.

[105] C. Collberg, C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," in *Proceedings of the International Conference on Computer Languages*, pp. 28–38, 1998.

[106] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, ch. 21. MIT Press, 2nd ed., 2001.

[107] J. Larus, "SPIM: A MIPS32 simulator." `http://pages.cs.wisc.edu/~larus/spim.html`, 2009.

[108] T. Boomerang Decompiler Project, "Boomerang: a general, open source, retargetable decompiler of machine code programs." `http://boomerang.sourceforge.net`, 2009.